

**COGNITIVE SUPPORT IN SOFTWARE ENGINEERING TOOLS:
A DISTRIBUTED COGNITION FRAMEWORK**

by

Andrew Walenstein

B.Sc., University of Alberta, 1990

M.Sc., University of Alberta, 1992

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

© Andrew Walenstein 2002
SIMON FRASER UNIVERSITY
May 2002

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Andrew Walenstein
Degree: Doctor of Philosophy
Title of thesis: Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework

Examining Committee: Dr. Binay Bhattacharya
Chair

Dr. Robert D. Cameron, Senior Supervisor

Dr. Stella Atkins, Supervisor

Dr. Hausi A. Müller, Supervisor,
Professor of Computer Science,
University of Victoria

Dr. F. David Fracchia, Supervisor,
Vice President Software Development,
Mainframe Entertainment
Adjunct Professor, Simon Fraser University

Dr. Arthur Kirkpatrick, SFU Examiner

Dr. James D. Hollan, External Examiner,
Professor of Cognitive Science,
U. of California, San Diego

Date Approved: _____

Abstract

Software development remains mentally challenging despite the continual advancement of training, techniques, and tools. Because completely automating software development is currently impossible, it makes sense to seriously consider how tools can improve the mental activities of developers apart from automating them away. Such mental assistance can be called “cognitive support”. Understanding and developing cognitive support in software engineering tools is an important research issue but, unfortunately, at the moment our theoretical foundations for it are inadequately developed. Furthermore, much of the relevant research has occurred outside of the software engineering community, and is therefore not easily available to the researchers who typically develop software engineering tools. Tool evaluation, comparison, and development are consequently impaired. The present work introduces a theoretical framework intended to seed further systematic study of cognitive support in the field of software engineering tools. This theoretical framework, called RODS, imports ideas and methods from a field of cognitive science called “distributed cognition”. The crucial concept in RODS is that cognitive support can be understood and explained in terms of the computational advantages that are conferred when cognition is redistributed between software developer and their tools and environment. The name RODS, in fact, comes from the four cognitive support principles the framework describes. With RODS in hand, it is possible to interpret good design in terms of how cognition is beneficially rearranged. To make such analyses fruitful, a cognitive modeling framework called HASTI is also proposed. The main purpose of HASTI is to provide an analysis of ways of modifying developer cognition using RODS. RODS and HASTI can be used to convert previously tacit design knowledge into explicit and reusable knowledge. RODS and HASTI are evaluated analytically by using them to reconstruct rationales for two exemplar reverse engineering tools. A preliminary field study was also conducted to determine their potential for being inexpensively applied in realistic tool development settings. These studies are used to draw implications for research in software engineering and, more broadly, for the design of computer tools in cognitive work domains.

Acknowledgments

Funding for this research was provided by an NSERC scholarship, an BC Advanced Systems Institute scholarship, an NSERC grant, and the Consortium for Software Engineering Research (CSER). I wish to thank each of these funding bodies for their support.

This manuscript was prepared with the \LaTeX document preparation system and the associated \TeX utilities. All figures were created using the xFig system, with Figures 7.1 and 7.2 created with the help of xMaple . Bibliographic metadata was provided by the HCI Bibliography Project (<http://www.hcibib.org>). The instructions used in the field study is an adaptation of instructions written by Caroline Green and Ken Gilhooly [256].

Thank you to the nameless people who participated in my field study. I also wish to thank Hausi Müller, Janice Singer, and Anatol Kark for making my research visit to Ottawa possible, entertaining, and educational.

I would like to thank Rob Cameron for his careful and patient mentoring, and for the financial and emotional support he has provided me. I also wish to thank all of the members of my examining committee for their diligent reading of this dissertation. I am very grateful to my supervisory committee for their time and effort in reading all of the various documents and for their advice. I am also very indebted to each and every one of them as they have given me the freedom to pursue this somewhat unconventional and lengthy dissertation topic. Few students are so lucky!

I would also like to thank the many friends and colleagues who have helped make life during my PhD studies so much more enjoyable by discussing computing, humans, philosophy and life, and by just being great companions. Thanks to Michael Heinrichs, Georgina Regeczi, Adam Woodgaines, Sheelagh Carpendale, Philip Fong, Maria Lantin, Gabor Melli, Francisco Hererra, Robert MacDonald, and the many other friends I have made. Thank you all!

Finally, I am deeply grateful to my wonderful family who have supported me so thoroughly and unconditionally throughout my studies. Without them, I would never have been in a position to even start this project, little less manage to battle it through to the end. Thanks always to my marvellous and supportive wife Lorena.

Contents

Approval	ii
Abstract	iii
Acknowledgments	iv
List of Tables	x
List of Figures	xii
1 Introduction	1
1.1 Overview of dissertation contents	7
1.2 Overview of background sources	10
1.3 Overview of contributions	11
2 The Need for Cognitive Support Theories	14
2.1 A Case for Cognitive Support Research	15
2.1.1 Cognitive Support—the Other 90 Percent	16
2.1.2 Folk, Tacit, and Science Knowledge	21
2.2 Problems of A Theory-Thin Discipline	26
2.2.1 Evaluation Problems: Simple Comparison	28
2.2.2 Evaluation Problems: Undirected Observation	44
2.2.3 Evaluation Problems: Cognitive Modeling	49
2.2.4 Analysis Problems	51
2.2.5 Design Problems	54
2.2.6 Summary of Problems	55
2.3 Possibilities of Theory-Based Research	56
2.3.1 Leveraging Mechanical Support Theory	57
2.3.2 What Might Theory-Based Methods Look Like?	62
2.3.3 SE Research and Researchers	67
2.4 The Practical Art of Designing Theories	73
2.5 Summary	78

3	Cognitive Support Phenomena	80
3.1	Supportive Relationships	82
3.1.1	Embodiment and Strategic Artifact Use	83
3.1.2	External Memory and Internal Memory	87
3.1.3	External Resources and Structure	89
3.1.4	Reflective, Visual, and Intentional Thinking	91
3.1.5	Evolving Structures, Emergent Thought	92
3.1.6	Representation Effect	94
3.1.7	Automation	94
3.2	Descriptive Theories	95
3.2.1	Mediation and Reflective Media	96
3.2.2	Scaffolding	97
3.2.3	Augmentation, Extension, and Symbiosis	97
3.2.4	Fitness	98
3.3	Schools Of Cognitive Support	99
3.3.1	By Research Tradition	99
3.3.2	By Problem Domain	103
3.4	Summary and Conclusions	107
4	Strengthening the Foundations of Cognitive Support with RODS	110
4.1	DC Principles and Tenets	114
4.1.1	C0: Human Mind is a Cognitive Unit	114
4.1.2	C1: Cognition = Computation	115
4.1.3	C2: Cognitive Interpretation	115
4.1.4	D0: Distributed Functional Unit is a Cognitive Unit	116
4.1.5	D1: Cognition = Distributed Computation	116
4.1.6	D2: External Cognitive Interpretation	117
4.1.7	Summary of DC Tenets	121
4.2	RODS: Computational Principles of Cognitive Support	122
4.2.1	Task Reduction	124
4.2.2	Algorithmic Optimization	125
4.2.3	Distribution	131
4.2.4	Specialization	133
4.2.5	Summary of Principles	135
4.3	Analyzing Cognitive Support	135
4.3.1	The Design Space Induced by RODS	136
4.3.2	Distributed Cognitive Architectures	138
4.3.3	Virtual Architectures	141
4.3.4	Summary of Analysis Proposals	143
4.4	Requirements Check	144

4.5	Summary and Conclusions	145
5	HASTI: A DC Modeling Framework	147
5.1	Overview of HASTI	150
5.1.1	Framework Principles and Strategies	153
5.1.2	Structure Overview	157
5.2	Hardware Model: Cognitive Capacity Decomposition	160
5.3	Agent Model: Behavioural Decomposition	162
5.3.1	Problem (ends, operations, constraints)	164
5.3.2	Agenda (goals)	166
5.3.3	Control Panel (plans)	166
5.3.4	Progress (state, history)	167
5.3.5	The Agent Model and Its Mapping	167
5.3.6	Summary	170
5.4	Specialization Hierarchy: SRKM Strata	171
5.5	Task Decomposition: D2C2 Stratification	174
5.6	Interaction Decomposition: Virtual Architecture	176
5.7	Summary and Conclusions	177
6	CoSTH: A Hierarchy of Support Theories	179
6.1	Using HASTI and RODS To Formalize Tool Ideas	180
6.2	Distribution	185
6.2.1	Data Distribution	187
6.2.2	Processing Distribution	192
6.3	Specialization	205
6.4	Algorithmic Optimization	210
6.5	Composite Rearrangement	211
6.5.1	Revisiting Previous Examples	213
6.5.2	Reconstructing and Naming Happy Composites	215
6.6	Comparison to Related Work	219
6.7	Limitations	222
6.8	Summary, Commentary, and Implications	225
7	Building Theories Fit For Design	228
7.1	The Trouble with Theory	230
7.1.1	Navigating the Fitness Landscape	232
7.1.2	Crossing the Gulf of Synthesis	242
7.1.3	Theory for FP-reasoning	246
7.1.4	Summary and Implications for Designing Design	250
7.2	Cognitive Support Knowledge Fit For Design	251
7.2.1	Engineering Concepts and Vocabulary	253

7.2.2	DC Design Stances	256
7.2.3	Reifying Design Space	261
7.2.4	Summary of Design Ideas	265
7.3	Summary and Implications	266
8	Application: Where Craft and Science Meet	268
8.1	RMTool Example	272
8.1.1	Tool and Usage Description	272
8.1.2	Interpreting RMTool Using HASTI	276
8.1.3	Tool Analysis Scenario	278
8.1.4	Design Envisionment Scenario	281
8.1.5	Summary and Implications of RMTool Analyses	284
8.2	Rigi Example	285
8.2.1	Bottom-up Comprehension: BU-HASTI	285
8.2.2	CoSTH Analysis of BU-HASTI	286
8.2.3	Matching Features in Rigi	287
8.2.4	Summary of Rigi Analysis	288
8.3	Summary	288
9	A Field Study of Cognitive Support	292
9.1	Field Study Description	295
9.1.1	Motivation and Background	295
9.1.2	Study Design	300
9.1.3	Observation Methods	303
9.1.4	Test Run (Pilot)	307
9.1.5	First Stage Summary of Study	307
9.2	Context for Research Scenarios	308
9.2.1	Description of Participant Context	309
9.2.2	Distributed Planning in <i>Visual Café</i>	310
9.3	An Exploration of Data Analysis Techniques	313
9.3.1	Analysis Methods	313
9.3.2	Coding Scheme	315
9.3.3	Results	316
9.3.4	Discussions	319
9.4	Theory Application Scenarios	323
9.4.1	A Claims Check	323
9.4.2	Measurement Scenario	325
9.4.3	Summary and Discussion	326
9.5	Validity and Evaluation	327
9.6	Conclusions	329

10 Conclusions	331
10.1 Summary of Contributions	332
10.2 Future Work	336
10.3 Coda	338
A Invitation to Participate	339
B Research Description	340
C Questionnaire	342
D Instruction Card	343
E Instructions For Producing Verbal Reports	344
F Coding for Participant E	345

List of Tables

2.1	Summary of problems in SE research practice due to lack of cognitive support theories . . .	27
2.2	Comparison of methods used to evaluate tool ideas	29
2.3	Evaluation and testing problems compared to the promises of theory-based research	31
2.4	Theory requirements (lessons learned) and where they are addressed	62
2.5	Comparing problems of current idea evaluation practices to theory-based research	63
2.6	Matching problems of current design/analysis practices to theory-based research solutions	63
2.7	Arguments for a realistic pursuit of cognitive support theories	75
3.1	Descriptive theories for various cognitive support relationships	95
4.1	Key tenets of DC	114
5.1	Decompositions of phenomena and matching structures	158
5.2	Cognitive resources in the agent model	164
5.3	Overview of Rasmussen’s SRK categories of human adaptation	171
5.4	Summary of task taxonomy	176
6.1	HASTI structures that align with RODS substitution types	182
6.2	Summary of redistribution examples	186
6.3	Summary of specialization substitution types	206
6.4	Examples which are compositions of multiple support factors	213
7.1	Four types of reasoning and roles of theory for supporting them	248
8.1	Summary of references in Figure 8.5	290
9.1	Potential problems caused by dependency of cognitive support use on adaptations	299
9.2	Codes for event types in “shadowing” observation technique	306
9.3	Summary of data collected during the study	307
9.4	A list of cognitive support claims for Visual Café	311
9.5	Coding scheme for Visual Café example	315
9.6	Trace table for distributed planning activity in episodes V1 and V2	318
9.7	Description of goal and plan labels used in Figure 9.6	318

9.8	Frequencies of coded actions by type	319
10.1	Diagram of how contributions are spread across the chapters	333
F.1	Protocol and codes for first Visual Café episode (Episode V1)	347
F.2	Protocol and codes for second Visual Café episode (Episode V2)	348
F.3	Protocol and codes for third Visual Café episode (Episode V3)	348
F.4	Full Visual Café protocol for the three episodes	349

List of Figures

2.1	Knowledge explicitness and its relation to discipline maturation	22
2.2	Models of (a) pharmaceuticals and (b) possible cognitive support knowledge ecosystems	70
4.1	Summary of RODS computational advantage principles	124
4.2	ADT view of interfaces, and mappings to implementations	129
4.3	Virtual shared memory as an abstraction over complicated interaction	142
5.1	Summary of principles and strategies for modeling in HASTI	153
5.2	Simplified joint system model	160
5.3	Hardware level of description of a user	161
5.4	Problem solving metaphor of experts working around a blackboard	163
5.5	Schematic Agent model showing generic agents, panels, and panel data types	168
5.6	Mapping of Agent models to Hardware	169
5.7	Ordering of behaviour categories based on adaptation, preference, and fallback order	172
5.8	SRKM imposes stratification on Agent model	173
5.9	D2C2 task taxonomy partitions agents according to their task goals.	175
6.1	Hierarchy of support theories built from RODS+HASTI	183
6.2	Relationships between artifacts, examples, theories, and tool ideas.	185
6.3	Tentative leaves for processing distribution	199
7.1	A “fitness landscape” visualization	234
7.2	Problems of mindless search	235
7.3	Roles of theory for synthesis versus evaluation	239
7.4	The magician’s design method	245
7.5	Resource flow model of theory application in design	249
7.6	Hypothetical example of a tabular-form worksheet and its use	266
8.1	Simplified flow of RMT ₀₀₁ sessions	274
8.2	Illustrations of RMT ₀₀₁ ’s HLM and reflexion model outputs	274
8.3	Refining the Agent model with Brooks’ comprehension model	279
8.4	“Virtual blackboard” illustration of RMT ₀₀₁ processing	280

8.5	Timeline of main ideas and tools	291
9.1	Typical observation configuration	303
9.2	Snippet of coded field nodes (verbatim).	305
F.1	Fascimilie of Participant E's first error list	346
F.2	Fascimilie of Participant E's second error list	347
F.3	Fascimilie of Participant E's third error list	347

Chapter 1

Introduction

Our current ability to construct effective bridges across the chasm that separates our scientific understanding and the real world of user behavior and artifact design clearly falls well short of requirements. ... what is required is something that might carry a volume of traffic equivalent to an eight-lane cognitive highway. What is on offer is more akin to a unidirectional walkway constructed from a few strands of rope and some planks.

– Phil Barnard, “Bridging between Basic Theories and the Artifacts of Human-Computer Interaction” [28], pg. 107–108.

*H*ow and why are tools useful? How can a tool’s usefulness be explained? And how can tools be made more useful? There can hardly be more important questions to tool researchers. Another well-known and related research question is how tools can fail to be *usable*. They could be difficult to learn, for instance, or perhaps confusing, or inefficient to apply. But many researchers can be forgiven for being frequently more concerned with *usefulness* than with *usability*. Usefulness helps answer the question: regardless of its usability, why bother with the tool at all? Why go through the trouble of learning to use it? Why build it? The first rule of tool design is to make it useful; making it usable is necessarily second, even though it is a close second.

The issue of usefulness is certainly very relevant to researchers studying software development tools. Software development is notoriously difficult and costly, and consequently a wide variety of tools are sought after in order to make it less so. These tools vary in character from relatively automatic ones such as compilers and analyzers, to highly interactive ones such as editors, code browser, and visualizers. Moreover, most real software development environments mix aspects of automation and human interaction. Adequately explaining usefulness of these tools can be challenging due to their intrinsic nature. Whenever important software development tasks can be completely automated, the issue of usefulness seems relatively unproblematic. However wholesale automation of any non-trivial software development task is, in practice, currently the exception rather than the rule. Tools generally need to work in combination with human users.

Unfortunately, the issue of tool usefulness becomes problematic particularly when the tools are meant to work with humans. The reason for this is that their purpose becomes defined, at least in part, in terms of what they do for the user. In the context of SE, this is frequently related to mental work. Developing software is difficult, to a great extent, because it is mentally challenging. When tools are being interactively applied in such a context, their usefulness is ultimately dependent upon their utility relating to *cognition*: i.e., to thinking, reasoning, and creating. Assistance to such cognitive work can be called *cognitive support*. Explaining the usefulness of many software development tools consequently involves being able to explain cognitive support. A main point of these tools is to make the thinking parts of software development better, easier, faster.

Software comprehension tools, such as code browsers, are good examples of the sorts of tools in which cognitive support is important. The primary reason why these tools exist is to make it easier for developers to understand software, and, thus make software development easier. It stands to reason, therefore, that most of the important evaluations of such tools relate to how they affect cognition. Is mental effort reduced? Is developer knowledge improved? Are cognitively difficult problems made easier? Any suitable account of the usefulness of such tools *must not fail to explain* how the particular features of the tools give assistance to what are essentially cognitive activities. That is a key challenge: building credible explanations of cognitive support. How developers currently think, and what technological innovations a tool presents are either irrelevant issues, or are merely background considerations.

Being able to credibly explain cognitive support has many far reaching implications for the field. Our ability to adequately address many important research questions hinges on having such suitable accounts. These research questions go beyond simply scientific curiosity. The right sort of theories might allow us to engineer usefulness qualities of computer tools in a principled manner. They could allow us to evaluate and compare tools in terms of cognitive issues much better than we do now. They would likely be able to open up new avenues for rigorously testing and validating tools and tool ideas. And, of course, we may be able to use the theories to guide invention. Thus it is fair to say that an entire research area in SE depends on an understanding of cognitive support. Herein lies a peculiar conundrum. The SE research community is presently ill-equipped to tackle this research. Current research and development methods involve an uncomfortable amount of guesswork, and are poorly grounded in science.

The overall goal of this dissertation is to improve the research and design of cognitive support in SE tools by manufacturing some appropriate theory-based resources. These resources include theories and models, techniques for analyzing tools, methods for evaluating tools, and concepts, frameworks, and notations for designing new tools. A theory-based infrastructure, in essence. My conviction is that such theories and methods can presently be built, and that they can transform SE tools research from its current craft-like state (regarding cognitive support) into one that is more principled, engineering-oriented, and grounded in science. Unfortunately, this is a thesis that is reasonably proven only by the test of time.

A more specific and restricted thesis can, however, be proposed: that theories based on the principles of *distributed cognition* (DC) can form a broadly-applicable starting point that can be immediately applied, and thus make it possible to begin the process solidifying the theoretical foundations of SE tools research. This thesis, at least, can be supported by building some theories and trying them out to see if they are able to address a significant range of important cognitive support questions. That is the tactic taken in

this dissertation. I have constructed integrative theories (called “RODS”, “HASTI”, and “CoSTH”) that attempt to package critical knowledge in application-oriented forms. These are used to survey and classify the cognitive support in many common SE tool features, analyze existing SE tools, and empirically observe cognitive support in action.

It should be clear that any adequate explanation of cognitive support will carry significant psychological content—it will need to explain the reasons why thinking and problem solving during development is better with one tool than with another. The appropriate science knowledge is accumulating in cognitive science, psychology, and human–computer interaction (HCI) research. Such theories need to be imported and used.

It might be argued that human cognition is not yet understood well enough. Needless to say, a subject such as human psychology is far from perfectly understood. Real-world human cognition in a complicated domain, such as software development, is an extremely complex phenomenon. The need to understand the contributions of the complex tools and the development context seems to make the challenge even more insurmountable. Humans are complex; development tools are complex; the domain itself is complex. “Completely” understanding cognitive support in software development tools can only be considered, realistically speaking, to be a very long term goal at best. For the present time, we should expect at most relatively moderate, incremental progress in the science base.

We must not, however, blame all of our current woes on the general difficulty of the topic. The roots of our most immediate problems stem less from the height of the hill to climb, and more from the steps not yet taken. The lack of suitable explanations of cognitive support in software development is not primarily due to a lack of applicable theories. Instead, it is due primarily to (1) the general lack of appropriate research effort directed specifically towards the topic, (2) the poor theoretical foundation for building explanations of cognitive support, and (3) the wide gulf separating research in software development tools from research in psychology, cognitive science, and other disciplines that could contribute useful knowledge. In other words, cognitive support is poorly studied in SE research, there is little in the way of a suitable research framework much less an accumulation of actual support-oriented theory, and it is difficult for researchers from CS and SE to make headway in the area.

There are many reasons contributing to this state of affairs. Many of these are discussed in later chapters. A few of these reasons are clearly fundamental (e.g., cognitive support is really hard to understand and study), some are merely historical (e.g., attention has been elsewhere, such as on usability), and some are best described as “political” (e.g., some researchers are reluctant to pursue what are perceived as “softer” aspects of tool research). Whatever the reasons, the absence of a suitable theoretical grounding has a significant impact on SE research.

Suitable theoretical and empirical treatments of cognitive support may be vital for the long-term health of tools research. Many—perhaps most—of the significant claims that can be made about the usefulness of software development tools involve human-related considerations, be they psychological, organizational, or sociological. Pressure is mounting for researchers to approach tools—and the claims about their capabilities and qualities—more scientifically and empirically. There can be no doubt that a critical component of the systematic and scientific study of software development tools must include psychologically-based explanations of how tools are useful to developers. Consequently, the topic of cognitive support is surely

near the top of the list of issues deserving a scientific underpinning. Considering how much these theories are needed, the current state of knowledge within the field is not a matter of a small gap in our understanding, it is an ugly, gaping chasm. The present work is a directed attempt at helping fill this chasm. Doing so is expected to have multiple benefits for the field—there is nothing so useful, so the saying goes, as a good theory. If comprehensive and widely applicable theories, models, and techniques can be developed, then these could be employed to better analyze, test, and evaluate existing tools, and to develop improved cognitive support for future tools.

Efforts to develop comprehensive cognitive support theories are not premature; they are, on the contrary, much overdue. They are overdue because our current theoretical capabilities are greatly underutilized. To borrow a phrase from Norman [467, pg. 37], there isn't any realistic hope of developing "the" theory of cognitive support, at least not for a long time, but certainly we should be able to develop *approximate* theories. Psychology is not "solved", but we can hardly wait for human behaviour to be completely fathomed before starting the project. Approximated theories, if they are grounded in science, are still better than guesswork and folk psychology, if for no better reason than the various assumptions, hypotheses, and claims are made more explicit and therefore comparable and testable.

Furthermore, the raw materials for building suitable approximate theories are all there: for decades the fields of psychology and HCI have studied how people use tools, and the SE field itself has a wealth of practical experience with tools. We have many bits and pieces needed for making a full-fledged, coordinated attack at the theoretical void. As Barnard noted, what theorists have currently offered is little more than "a few strands of rope and some planks" [28]. The time is right for putting what we already know to better use. Now is hardly the right time to pick out some extraordinarily focused aspect of psychology or tool use in order to meticulously add a tiny, incremental piece of knowledge. At the moment the field has no pressing need for a long parade of tightly focused experiments, papers, and Ph.D. dissertations. The chasm will remain unfilled much too long that way. Now is the time to begin dumping in by the truckload whatever theoretical material can be found in hopes of developing a workable bridge spanning the chasm. We need to look at the big picture. Whatever we build now can be fixed up and refined later as science advances.

This is not to say that new and basic advances are not important. Indeed, for suitable theories to be developed, it may be very helpful to determine certain important facts, say, that programmers frequently switch attention between different sorts of information when understanding software (e.g., Vans' Ph.D. work [654]). Such findings *should* be able to inform theories of cognitive support. The point here, though, is that there is more than enough material available for building some initial—but comprehensive, and widely applicable—theories. We should be able to begin the construction of these approximate, applied theories even while the basic science research continues its slow, piecemeal process of accreting new knowledge about software development, about tool use, and about basic psychology. There is no reason why basic and applied research should not operate in parallel. In fact, the two research programmes should complement each other. Right now the applied stream concerning cognitive support in SE desperately needs a kickstart. It needs a broad, integrative treatment.

A DC approach to building cognitive support theories.

This work presents an initial attempt at building the needed applied theories of cognitive support, and at determining how to apply them to SE research. The approach taken is typical of applied sciences. First, a problem is identified, which in turn initiates a search within sister disciplines for existing and applicable theories, models, and methods that could shed light upon the problem. Second, the resources found are adapted in order to make them usable within the discipline. Next, an initial theory is constructed, typically as a translation, adaptation, and integration of prior theories. Attempts are then made to apply them to existing problems. The new theory is not an end unto itself—it is used as a seed for further research. Such theories are iteratively evaluated, augmented, and tuned. The initial foray into theory development, however, is essentially exploratory in character. That is what is presented here: an exploration into theories of cognitive support using resources gathered and adapted from prior relevant work. These raw materials are for the most part not new, but the particular collection and presentation of them is novel. The raw materials for this work come from prior research in cognitive science, HCI, psychology, and computing science.

The main focal point of the theory building is the construction of a general framework for making explanations of cognitive support. Because some sort of umbrella framework must be chosen, an approach termed “distributed cognition” (DC) was picked. Even though the details of the framework can be tricky, the core ideas are simple enough to state in a few, relatively short statements:

1. Cognition is usefully modeled as computation.
2. Computer tools are computational systems. Thus developers using tools can be modeled as joint multi-processing computational systems. They form distributed computational systems; they form distributed *cognitive* systems.
3. The cognitive support provided by a tool is the computational advantages that the tool provides. Cognitive support can therefore be understood entirely in computational terms: support is the provision of computational advantage.
4. All real-world cognitive systems are distributed. New tools reorganize the overall computations involved. The reorganization involves re-engineering the computations occurring between tools and humans. Design of cognitive support is computational systems re-engineering.

Collectively these statements can be said to form a high-level DC viewpoint for understanding and building cognitive support. Statement 2 is a theory of cognition: distributed cognition. But the key statements are actually numbers 3 and 4. Statement 3 is a theory of cognitive support, albeit an abstract one. Statement 4 is a theory of cognitive support design. The crux of this work is an elaboration of Statement 3, although Chapter 7 takes an initial stab at expanding Statement 4.

By themselves, of course, the above statements actually explain relatively little. What, specifically, are the sorts of computations involved in cognition? What cognitive processes are important in software development? What ways, specifically, are such processes distributed onto tools? How is cognition beneficially reengineered? The details are absolutely critical, and must be filled in. Models of cognitive processes

involved in software development need to be added. So do models of how such cognitive processes are distributed onto tools. These details are required to turn a general DC viewpoint into a framework with explanatory power.

The core part of this dissertation is an elaboration of these points in such a way that they can be applied to software development tools. Since software development is a very broad topic, a sub-task of development—software comprehension—will be singled out as a focal task. This focus should not unduly cripple the resulting theories by making them too task-specific. However it will definitely skew the presentation and reference list in a software comprehension-specific way. The DC viewpoint outlined above will consequently be fleshed out considerably by examining (1) models of cognition that can be applied to software comprehension, and (2) the ways in which cognition may be reengineered and distributed in order to support the comprehender.

Before continuing on, a simple and familiar example may help give the reader the basic flavour of the sort of analysis involved. The example is due to Norman [467, pg. 21]. He used a simple shopping list scenario to argue a cognitive view of tools and artifacts. He argued that artifacts which are normally considered simple and quite inert can be quite naturally understood to perform cognitive functions. Consider a scenario where you are going shopping for food. You might employ a shopping list to store the names of items that you need to get. Maybe you write the items down on the list as you find your household running out of them. You use this list when you go to the store in order to make sure you do not miss getting any items that you need. Perhaps you check off items as they are placed into your cart. From your point of view, you did not need to remember the items on the shopping list. In fact, if you checked off the items as you put them in the cart, you would not have even needed to mentally keep track of which items you had already collected. To figure out what to get next, you could merely scan down the list for unchecked items. The shopping list serves as an external aid—a cognitive support. If you did not make the shopping list, you would likely need to remember what items you require, and you would have to be able to recall them all while at the store. Basic psychology—and common sense—says that this is relatively hard to do. Many people prefer to use shopping lists instead of relying on their memory because not only is shopping easier with the list, it makes them better shoppers (see, for example, Block *et al.* [59]).

Even this simple shopping scenario illustrates many of the important issues in the DC view of cognitive support. First, note that the list serves a computational role: it is a memory device, an *external memory*. Using a shopping list means you do not have to remember the items, but the items still are remembered—the paper holding the list serves as a memory. External memory of many sorts occur in software development environments. Consider these examples: a book of design patterns [232] acts as a long-term shared memory; a history mechanism in a web browser acts like a personal short-term memory. Second, notice that a shopper and a shopping list together form a joint system. It is not just *you* doing the shopping, but you *in combination with* your external memory. If you lost your list you would likely lose much of your memory of what you need to buy. Third, observe how behaviour, performance, and capability are joint properties of the system as a whole. “You + list” perform (i.e., shop) differently than just “you”. The external memory extends your capability: it makes it possible for you to do things you could not otherwise do (e.g., remember more), or it makes these things easier or otherwise better than before.

In sum, the shopping list example serves to illustrate three sample concepts from the DC view of cognitive support: that support can be understood in computational terms, that the behaviour of *joint* human–artifact systems (especially human–computer systems) is the sort of thing that needs to be studied, and that the ways in which tools affect joint cognitive performance is a key issue. Notice the importance of the cognitive-level explanation of usefulness—the value of the shopping list is not fully captured by considering the features and qualities of the artifact (degree of automation, efficiency, number of user functions, etc.), nor by considering the misfeatures of the human (e.g., limited memory). Although the example is a simple one involving non-computerized tools, the character of the analyses for software development tools is directly analogous. The primary differences are in the complexity of the joint system models, especially concerning the particulars of the cognitive processes involved. Simply put, software development is significantly more diverse and cognitively challenging than shopping, and computerized tools offer richer distributions of cognition.

1.1 Overview of dissertation contents

The structure of the dissertation (minus introductions and conclusions) is as follows.

Chapter 2 (Motivation; Vision)

Chapter 2 describes the current state of affairs regarding how computing science and SE currently approaches the issues of cognitive support. Chapter 2 therefore motivates and orients the remaining chapters. It argues that the search for applied theories of cognitive support is not an idle quest to satisfy curiosity, but an important yet absent part of a rigorous SE research programme. It also provides a guiding vision for theory-based research, and a list of desired qualities of the needed cognitive support theories. Thus, Chapter 2 represents good engineering design: it acts to build a requirements analysis, a rough project plan, and a set of criteria for testing.

Chapter 3 (Cognitive Support Phenomena)

Chapter 3 surveys the phenomena to be studied, that is, it defines what cognitive support is, and characterizes several of its many forms. Chapter 3, in other words, describes the sorts of benefits that tools provide their users. These are the things which theories of cognitive support seek to explain. Simply describing the various types of cognitive support is a valuable first step. The various conceptions of cognitive support are widely scattered in the literature, so that some sort of review must be performed to bring them together and make them accessible. This review also makes a secondary contribution to theory building. It reviews many types of cognitive support important to SE. In this way it establishes lower bounds on the required breadth of coverage. Such a review is necessary to avoid two of the most pernicious problems associated with building applied theories: irrelevance and narrowness. A broad review helps the theory designer avoid both.

Chapter 4 (RODS)

Mere description of cognitive support phenomena is not sufficient—we need *explanations*. Chapter 4 presents a framework for building various explanations of cognitive support, that is, a comprehensive framework for researching and studying cognitive support. SE currently has no such framework to speak of. The framework is built from pre-existing ideas taken from DC and HCI. The key part of the framework is the proposal to understand cognitive support in purely computational terms. In particular, cognitive support is identified with computational advantages that artifacts engender, such that cognition is improved. A catalogue of four types of cognitive advantage is developed: task reduction, algorithmic optimization, distribution, and specialization. HASTI gets its name from these four principles. These four cognitive support principles are a central feature of an overall framework for studying cognitive support. This framework provides a general way of analyzing and comparing tools for their cognitive support, and describes the principles for abstracting and generalizing support arguments. Because of the centrality of the four support principles, the overarching framework is also named RODS. The result of the chapter is a widely applicable framework for (1) understanding cognitive support in computational terms, and (2) expressing and arguing claims of how tools support cognition. These two contributions are absolutely essential to an applied research programme on cognitive support.

Chapter 5 (HASTI)

RODS effectively produces an abstract “explanation” of cognitive support phenomena by providing a computational account of the underlying mechanics of cognitive support. However this explanation is at too abstract a level to be useful in SE. In order to systematically apply this computational account in tools research, more detailed models are expected to be frequently necessary. In particular, computational models of human–computer systems must be built in order for the support principles to be applied. The modeling framework developed in the chapter is called “HASTI”. RODS lists the ways of beneficially reorganizing cognition, but we need to know what cognition is being arranged and how. As can be expected, the most critical requirements of such models is that they capture important knowledge about user psychology—especially with respect to how tools are used. HASTI is tailored specifically to the needs of application-oriented researchers. They need abstractions and simplifications such that the important issues of cognitive support can be efficiently raised and addressed. They also need pre-built models that can be rapidly and widely applied to yield insight. The result is intended to be suitable for high-level design reasoning based on “quick and dirty” analysis. The name “HASTI” is an acronym for the modeling structures which decompose the model as a whole (hardware model, agent model, specialization hierarchy, task taxonomy, and interaction abstraction model). This decompositional structure is important for analyzing cognitive support. The HASTI modeling framework is therefore not simply a single model that can be used to understand particular human–computer interactions, but a modeling *framework* that exposes the computational principles on which RODS concepts can be applied to generate support explanations. The importance of being able to apply RODS concepts to models is outlined in Chapter 6.

Chapter 6 (CoSTH)

The number and type of cognitive support arguments an analyst is capable of making depends upon the power of the theoretical armaments used to build the explanations. Chapter 6 demonstrates that RODS and HASTI, in combination, can generate many diverse cognitive support arguments. This capability is shown by applying the various support principles of RODS to the modeling features of HASTI. This analysis generates a hierarchy of cognitive support arguments called “CoSTH”. Metaphorically speaking, the nodes of CoSTH are cognitive analogues to systematic variations of simple machines (lever, inclined plane, etc.). Like the simple machines, the types of cognitive support can be combined to make more complicated ones. These support arguments are shown to be able to theoretically reconstruct many common tool ideas from the literature. These support arguments act as generalized argumentation schemas that can be specialized according to specific instances of tools in order to create tool-specific arguments about the cognitive support they provide. Thus they are an initial attack at generating systematic, reusable design knowledge.

Chapter 7 (Design)

Chapters 4, 5, and 6 were focused primarily on theory building, and secondarily on tool analysis, understanding, and evaluation. Chapter 7 is focused squarely on *design*. One of the more enticing possibilities of cognitive support theories is their potential to guide tool design. This chapter begins with an analysis of the ways in which theory can guide design. From this analysis it is argued that the cognitive support framework built here is rather uniquely positioned to offer a particular form of design knowledge that, to this point, has been rare. Specifically, CoSTH offers the possibility of guiding “FP-reasoning”. FP-reasoning is essentially reasoning about features to add to tools to make them useful. This type of theory is contrasted with prior theoretical works that have “informed” design primarily by suggesting design goals or constraints. From this analysis, three different ways of repackaging RODS, HASTI, and CoSTH are proposed for making these theories more easily applied by practitioners during early design envisionment. Because these sorts of design theories are relatively rare in HCI, the chapter concludes with an overview of how this dissertation provides evidence that CoSTH can be useful as a source of practical designer resources.

Chapter 8 (Analytic Evaluation)

Chapter 8 begins the evaluation of the theories and modeling techniques proposed in the previous chapters. As these are meant to be application-oriented, the primary issues in their evaluation are whether they are useful enough, and whether they are broad enough in scope. These questions are answered primarily by demonstrations of the theoretical toolkit in action. Specifically, RODS, HASTI, and CoSTH are applied to two high-profile reverse engineering tools in order to analyze them for cognitive support. These “guinea pig” tools are strategically chosen in order to be topical for SE research, and to try to exercise much of the theoretical framework. The evaluations serve to demonstrate the applicability of the theories to questions relevant to current research interests in SE. They also demonstrate the ability of even this preliminary and skeletal toolkit to make interesting statements about tool features at abstract levels.

In the process, the theory-based toolkit is shown to be able to reconstruct several existing claims and experiences about the “guinea pigs”. The correspondence of these theory-derived arguments to craft-oriented knowledge builds confidence in the validity of both the theory and the craft knowledge.

Chapter 9 (Field Study)

Chapter 9 explores some of the possibilities for applying RODS, HASTI, and CoSTH in realistic research and development work. In the other chapters the primary concern is analytic power: the ability of the theories and models to explain or predict cognitive support, and their ability to generate useful design ideas. In this chapter, the issue is whether RODS, HASTI, and CoSTH can be employed in empirical studies in a lightweight manner. Being able to do so is expected to be a key enabler of practical theory-based tools development. All theories are limited. RODS, HASTI, and CoSTH provide a basic theoretical infrastructure for making interesting cognitive support arguments, but they cannot answer all questions. Sometimes user studies are needed. But the theoretical framework can kick start the user study by allowing the researcher to focus on answers to specific questions pertaining to cognitive support. This chapter explores this possibility by using a field study of software development as a testbed for evaluating various observational techniques. The aim is to provide some insight into how cognitive support theory can be applied in realistic tool development contexts. It provides an initial demonstration of how claims can be made about cognitive support, how that support can be observed and measured, and therefore how claims can be empirically validated.

1.2 Overview of background sources

This dissertation is focused on the “big picture” concerning cognitive support theories and their applications to real analysis and design. This necessitates drawing upon a very wide range of prior work. It would likely have been much simpler to target one particular, restricted cognitive support issue. Then a much smaller selection of prior works might be used as building materials. However simpler is not always better; integration is hard but necessary—especially if the results from other fields are to be made usable and palatable for SE. As a result, this work rests on the shoulders of a veritable academy of researchers from cognitive science, psychology, and HCI.

Four main sources or prior works are distinguished. The first is basic cognitive science and psychology, particularly DC work and work on the psychology of software developers. These substantially inform the modeling work of Chapters 4, 5, and 8. They also are used as the basis for the explorations in empirical methods from Chapter 9. The second main source is from applications of psychology to HCI and systems design. These are relied upon when extracting the basic foundations for understanding support in Chapter 4 and 7. The third main source of prior research is work on applied theory-building. Ideas from past theory-building activities used as a basis for Chapter 2 and 7. The final main source of prior work is the SE tool research literature. The literature contains a wealth of tool ideas which are used pervasively throughout to ground the discussion in the particular domain of application. The collected wisdom of

the field is relied upon in order to “calibrate” the theory building effort: it generates the research problems to solve, establishes the requirements for theory building and theory application, and is used as an initial “check” to make sure the applications are helpful and make sense. The cognitive support theories and applications methods would not exist without the prior work in cognitive science, HCI, and related disciplines. Likewise, the suitable application of such theories to SE problems could not have been made without the past work in SE.

1.3 Overview of contributions

The contributions of the dissertation can be categorized according to the chapter contents. In addition, the contributions are additionally classified into “core” contributions and “side effect” contributions. The core contributions reflect the dissertation’s main research questions. The side effects contributions are ones that needed to be made in order to answer those main research questions. It would have perhaps been preferable not to need to make these side effect contributions but, on the whole, the contributions all fit tightly together.

The contributions are broken down into theory building, theory application, and survey contributions. Each contribution is labelled with its status as a core (*C*) or side effect (*S*) contribution.

Theory Building

- (*C*) **Vision for tools research.** A vision of tool research is provided in Chapter 2. This vision suggests how tools research can be guided by cognitive support theories. Such theory-guided research may be able to resolve several significant inadequacies of current research methods, including the difficulties in comparing designs, constructing and evaluating claims about tools, and establishing what the proper burdens of proof should be for tools researchers in SE and CS.
- (*C*) **Integrated and simplified cognitive support framework.** In Chapter 4 and Chapter 5, a computational foundation is given for explaining cognitive support. This foundation is given in terms of DC theories. There are two critical parts of the framework. The first is a principled decomposition of the meaningful ways of performing comparative analysis of cognitive systems by way of evaluating differences according different support principles. The second is a method for using such an analysis for generating arguments about how artifacts provide cognitive support. Neither the ways of comparing joint cognitive systems, nor a principled way of generating arguments have been adequately articulated in prior work. Cohesive integration and simplification for non-specialists has been a particular problem in the past.
- (*C*) **Cognitive support arguments.** Chapter 6 presents an organized collection of theories of cognitive support. These theories are abstract and reusable (generalizable) arguments about how artifact features lead to cognitive efficiencies. Although all of these arguments can be found elsewhere in some form in the literature, no work has provided a framework for treating them in a uniform and integrated manner. Using only RODS and HASTI, many prior arguments can be directly generated.

This work shows that support can indeed be treated with a minimum of underlying theoretical apparatus. Consequently, this work makes a statement about cognitive support itself: it shows that the complexities of support reflect the complexities of the cognitive systems involved—that cognitive support is complicated because of the way that simple supports can be composed and arranged. Although similar theoretical frameworks and support analyses exist, I know of no work showing that such a wide variety of cognitive supports can be explained by appealing to a small set of fundamental principles. Secondly, this work demonstrates that the variety of cognitive support types can be generated by a few simple support principles in combination with models of cognition. That is, a broad range of specific cognitive support explanations can be generated from first principles—psychological and computational.

- (S) **Unified DC framework for cognitive support.** DC is a promising theoretical framework, but it is inhomogeneously presented, and it has a relatively weak tradition for modeling how cognition is distributed between human and artifact. That is, it is not yet in a “polished” enough form for wide use in SE. Chapter 4 extracts a focused portion of ideas from DC and related literature. Chapter 4 and Chapter 5 add to the existing DC modeling repertoire. Chapter 4 also takes the mechanisms used in traditional cognitive science to build generalizable models and expands them so that generalizable models of joint human–artifact cognition can be constructed.
- (S) **Unification and comparison of cognitive models.** Cognitive models abound, and few principled ways exist for unifying them. Chapter 5 provides one way of unifying several previously disparate model types. The key innovation is a way of separating out distinct modeling aspects and then mapping between them. HASTI is the decomposition and mapping approach in question, and it allows many different modeling traditions to be discussed in unison.

Theory Application

- (C) **Cognitive support analysis of software comprehension tools.** Many software comprehension tools are proposed and evaluated without adequate justification of their claims for support. This omission makes it impossible to engage in any principled analysis about what features offer which support. A lack of theory-based analysis hinders tool comparison, empirical validation of claims, and future application of the ideas embodied in the tools. Chapter 8 provides a theory-based analysis of cognitive support using the theories presented in other chapters. It fills in the missing support argumentation.
- (C) **Design implications from theory.** Generating good design implications directly from psychological theory is uncommon. Many design implications are derived indirectly by using theories to indicate common maladies (e.g., cognitive overload), and then using design *experience* to suggest possible resolutions. Chapter 7 attacks the problem of generating design implications by showing how the cognitive support framework provides: (1) a “scaffolding” resource for structuring early design, and (2) a way of reasoning forward from the *existing* to the *possible*, that is, from existing systems to new ones.

Survey

- (S) **Review of cognitive support research.** Research on cognitive support has been widely scattered, with little or no cross-discipline or cross-ideology unification and comparison. Chapter 3 adds a broad review of some of the core ideas about how artifacts can support cognition, a review of research paradigms which have been applied to understand cognitive support, and an overview of many research disciplines that have independently been researching cognitive support. This contribution demonstrates that there is a collection of common phenomena that can—even should—become the subject of a domain-independent research discipline which seeks these out, and attempts to explain them.

It may surprise some readers that part of Chapter 4, and most of Chapters 3 and 5, are considered peripheral to the main dissertation questions. One should note that although the side effect contributions do not directly address the dissertation's main questions, these contributions are neither unnecessary nor minor. Designating these contributions as side effects merely reflects the theoretical gap between the dissertation's pragmatic goals and the current state of the science of cognitive support. To build the desired application-oriented theoretical work it was necessary to first shore up the required theoretical background.

Chapter 2

The Need for Cognitive Support Theories

A successful system demonstrates nothing other than its own success, unless the possibly implicit psychological theory underlying the design is articulated.

– Alex Kirlik, “Requirements for Psychological Models to Support Design” [348], pg. 72.

The field of software engineering (SE) is in need of a research stream studying cognitive support in software development tools. SE, for the most part, just does not yet realize this. It is primarily this lack of awareness that makes this chapter necessary. Subsequent chapters will propose theory-based resources for studying cognitive support in software development tools. These resources are intended to be an initial seed for more thorough and ongoing studies of theories and models of cognitive support. Although this initial seed is a key contribution of this work, if the only issue was the technical (albeit difficult) problem of producing a decent initial theory, that theory could be presented immediately and little further discussion would be necessary. But the reality of the situation is that, within our field, little attention has been paid to the question of how to properly research and develop cognitive support in tools; many times the cognitive issues are ignored and avoided. If the importance of a cognitive support research stream is not fully realized, then no matter what theories are proposed, it seems likely that no actions will be taken to rectify the situation. So in addition to describing an initial seed of a theoretical framework, some elaboration is needed on the research context needed for this seed to grow and evolve.

There are many reasons for why cognitive support is so poorly respected. One of the main contributing reasons appears to be social and cultural. Specifically, it seems to be a result of the ways that tools are perceived by researchers and developers, and the way that cognition-related research is understood and conducted. On one side of the spectrum of opinions are technology-oriented researchers who have little background in—or patience with—psychology and cognitive science. Some of these view social science related work as “soft” and mostly irrelevant or even unscientific. Such researchers may not easily

be persuaded that knowledge of cognitive science and psychology could or should be integrated into the practice of SE and CS research. Sometimes it is assumed that tools can be adequately developed or evaluated in the absence of psychological knowledge. These attitudes were known at the dawn of modern HCI two decades ago [419] and they are prevalent to this day. Nearer the other end of the spectrum are those who are convinced that nearly any attempt at understanding the cognitive activities of developers will ultimately be useful and should be welcomed. Neither side has made much of an effort to further argue their positions (except to their peers), perhaps because of their blind convictions that psychological research will turn out to be either useless or almost unfailingly useful. In reality both positions hold some grains of truth and the topic deserves further debate and elaboration [494]. Exploring the debate has the capability to clarify our goals, to expand on the differences between various approaches to understanding cognitive support, and to provide a vision for the future of research in the field. A healthy field of study in cognitive support should be able to produce arguments to temper unrealistic optimism about the possibilities of understanding cognitive support and yet make a convincing case for its pursuit. What problems can theories of cognitive support address? What sorts of things need to be said? What sorts of issues are outside the area of concern? Who builds the theories and who uses them?

The remaining part of this chapter consists of a debate on the possibilities and roles of theories of cognitive support in SE and CS research. The debate is structured around a set of topic-defining questions: Is cognitive support that interesting a problem for SE? Do we need theories of cognitive support? Who should build them? What should theory-related research be like? These questions are asked and subsequently debated in separate sections. In Section 2.1.1, the focus is on evaluating current research practice. On the whole, many of the most problematic issues can be traced to a lack of good cognitive support research. In Section 2.2 and 2.3, the question of how to remedy these problems is addressed. From these debates the picture that emerges is of a discipline that is definitely concerned with cognitive support and which, for many reasons, has not been able to properly research cognitive support due to the lack of a suitable theory-guided research stream. Section 2.4 argues how theories of cognitive support might best be developed, thereby providing a philosophical vantage point for the remainder of the dissertation. Section 2.5 summarizes the debate.

2.1 A Case for Cognitive Support Research

Computer scientists make broad claims for the simplicity, naturalness, or ease-of-use of new computer languages or techniques, but do not take advantage of the opportunity for experimental confirmation.

– Ben Shneiderman, “Software Psychology” [582] (1980), p. xiii

Is cognitive support an important research focus for SE and CS? Should researchers be more concerned with understanding cognitive support? The following two subsections argue that the answer is “yes” on both accounts. Section 2.1.1 argues that the subject of cognitive support is a critical but under-appreciated

part of SE and CS research. Section 2.1.2 argues that the call for more explicit theoretical research on cognitive support is reasonable since it is simply a call to render our current practices more principled and scientific.

2.1.1 Cognitive Support—the Other 90 Percent

It is curious to observe how the authors in this field [programming logic], who in the formal aspects of their work require painstaking demonstration and proof, in the informal aspects are satisfied with subjective claims that have not the slightest support, neither in argument nor in verifiable evidence. Surely common sense will indicate that such a manner is scientifically unacceptable. ... The deplorable situation of programming logic outlined here is part of a much more widespread pattern of attitudes and manners prevailing in academic computing and mathematics, that tend to accept sales talk in the place of scientifically sound reasoning.

– Peter Naur, “The Place of Strictly Defined Notation in Human Insight” [440, pg. 477]

There is an old tester’s adage that goes as follows: *“The first 90% of software development consists of understanding what to program, writing the program, and documenting it. The last 90% involves testing it.”* For many years, software design and construction had received a great deal of attention at the expense of other critical aspects of software development like maintenance and testing. The gist of the adage is that the easily apparent difficulties of one particularly well-advertised part of a problem may obscure the difficulty of the other parts of the problem. To testers, the difficulties of designing and coding software is given too much emphasis. They might argue that developers need to change their attitude towards testing. Moreover, the saying issues a warning: woe to those who fool themselves into thinking the job is done when development ceases—the “remaining” task of testing may be the back breaker. Whether or not the adage is strictly true for testing, some similar sentiment certainly applies regarding how cognitive support is treated in SE.

In the world of software development tools, the lion’s share of the attention is focused on providing automation and clear formal precision. Computing science has developed numerous formalisms, logical frameworks, languages, and computational techniques that have been brought to bear on formalizing development problems (specifications, documentation, design, etc.), and on automating parts of the development. For instance, parsing, semantic analysis, and translation techniques have made possible the modern compilers that translate code written in “high-level” languages that humans write into the byte sequences that computers understand. Compilation is, in fact, one of our greatest achievements in automating software development. It has to a large degree eliminated the tedious and error-prone process of programming in assembly language, and thus has led to tremendous productivity gains [78]. However automation is essentially achievable only for those parts of software development that are within the reach of our current capabilities of formalization [695]. Unfortunately our reach does not extend far enough. True, many of the automatable tasks of software design are actually the tedious, mundane, and

repetitive parts. This is simply because the regular and well-structured tasks are the most easily automatable. This fact is quite fortuitous since those sorts of activities tend to be exactly the tasks that developers most wish to avoid. Of course, for many of the less easily formalized tasks, artificial intelligence (AI) techniques can be employed. But even with these, it is widely acknowledged that the tools do not work alone, and that significant intervention is required on the part of the developer (e.g., in reverse engineering [524,642]). Despite the strong focus on formalization and automation by researchers, most software development activities are not in immediate danger of being fully automated [78,378].

Even though automation and formalization are clearly important foci for tools research, it is entirely possible to concentrate too steadfastly on that goal. In fact, successful automation invariably exposes its own weaknesses. Brooks argued that good design removes inessential complexity [78]. By a similar argument, automation removes unproblematic work. Thus the development of better and better forms of automation slowly picks away at the more easily automatable parts of the problems—we would hardly expect anything else! Our capacity to formalize and then automate is mostly limited to the routine and predictable [371,695]. These limitations tend to ensure that, as Landauer put it, the “easily reached fruits have been picked” [371, pg. 6]. What is left for the developers? The mentally challenging, creative, and error-prone parts. Referring back to the old tester’s adage, the parts we should expect to automate will constitute only the first 90% of tool support issues. The developers are stuck with the other 90%. And this other 90% corresponds to the most difficult, most ill-defined, and most non-routine parts. These are the thinking parts of software development—the essentially *cognitive* work.

Consider again the example of automatic program translation (compilation). Compilers have almost completely eliminated the work of translating code from human-level programming languages to machine-level, but what tasks are still left for the developer to perform? Designing, programming, debugging, and testing. Successful automation etches away the routine and well-structured problems, leaving behind the ill-defined and poorly understood ones. Support for this ever-present residue is *cognitive support*. While automation hogs much of the spotlight, cognitive support lurks as a problem that may easily be its equal. And woe to those who ignore it. Automation-oriented research is not enough; cognitive support research is needed. Has this need been met?

Considering the significance of cognitive support for tools research, shamefully little explicit attention is given to it. It is fair to say that we have barely scratched the surface of that last 90%. An adequate understanding of cognitive support in software development requires, at minimum, attention to two¹ aspects of cognition² in software development:

1. COGNITIVE PROCESSES: The cognitive processes and resources that are important to software development must be understood. It is not adequate to treat these as unknowns or as mysterious quantities. After all, the thinking done by developers is, in practice, never totally eliminated, only

¹Some might complain that my dichotomy between cognitive *processes* and cognitive *support* is false, that is, that *all* cognition is supported by external artifacts, people, etc. This point is a good one but irrelevant: the categories describe research emphasis (developer psychology vs. support principles) rather than research topic (cognition).

²My attention here is focused on tool–user relationships that are related primarily to individual cognitive psychology, although obviously other aspects of cognition (e.g., organizational and social dimensions) are clearly relevant also (see e.g., Carroll [103]).

changed in form [470]. Thus no matter the tool, a human almost invariably operates it and so the cognitive processes of doing so must be appreciated.

2. COGNITIVE SUPPORT: The principles and means by which cognitive processes are supported or aided by tools must be understood. It is not enough to study cognitive processes without taking the next step of understanding how tools beneficially alter them (but not replace them). Learning only about existing development processes sheds little light on how to change them with tools.

Speaking informally, one could say that the study of cognitive processes is concerned with elaborating *requirements* for tools that can support cognition, while the study of cognitive support is concerned with the forms of the *solutions*, and with the ways of increasing the level of cognitive support.

Addressing *both* cognitive processing and support is challenging and, in the case of software development tools, exceedingly rare. Cognitive support, in particular, has not been given its fair share of attention. This contention may be argued by considering how infrequently (1) tools are thought of in terms of assisting the harder parts of development like human problem solving, (2) cognitive processes are explicitly considered or studied, and (3) cognitive support is explicitly considered or studied. On all three counts, attention is lacking.

Consider the following characterizations of software development tools:

The main [software engineering environment (SEE)] requirements from a software application developer's viewpoint are that: ...The SEE must be seen as providing help with many of the routine and mundane tasks associated with software development (for example, in producing documentation) ... The SEE must not remove or hinder the creativity and innovative aspects of software development, which are crucial to the enjoyment (or job satisfaction) of most software application developers. [82, pg. 33]

And

Software development tools are designed to assist software developers in producing quality products in minimum time. Such tools [provide assistance] by enabling their users to perform their creative intellectual activity under optimal conditions, by preventing or detecting human errors as they occur, and by relieving users of routine mental and physical activity associated with the productive process. [644, pg. 109]

These portrayals reveal certain common attitudes towards tools. Tools are not understood as actively *helping* developers in their “creative intellectual activity”; instead, they merely “enable” these activities. By the phrasing of these sentences one gets the feeling that what constitutes the “creative intellectual activity” is quite outside the area of concern for tool developers—that it is something sacred, or something to be left alone. For the sake of contrast, consider the following alternative characterization of software development tools:

Software development tools aid developers by participating in their thinking and work. This assistance can include helping them make complicated decisions in the context of uncertainty or restricted knowledge, helping them track their thought processes, assisting them in evaluating program quality...

The point here is not that either of the first two views are wrong, but that their definitions illustrate a common (but by no means universal) set of attitudes concerning the importance placed on understanding

how developers think, and the roles of tools in supporting these thought processes. In particular the issue of cognitive processes is marginalized, and the concept of support by tools is limited to be automation—especially of the mundane parts of development.

Although it is under-represented, there certainly does exist excellent work that considers the cognitive processes of software development. There has been a good deal of work on trying to determine “the” process of software comprehension, for example (e.g., see von Mayrhauser and Vans [674] as compared to Good [250]). However *even in cognitive-oriented work* it is entirely possible to methodically study cognition in software development and yet fail to adequately advance our understanding of how to support these processes. Some research, for instance, is focused on improving cognitive theories and models of software developers without regard to the tools involved. In these sorts of works the central issue is typically the “inner” environment of the comprehender’s mind rather than the “outer” tool context, or how the inner environment relates to the outer one [348]. In software comprehension and maintenance research the use of such “inner” cognitive models is modestly popular [675]. But these studies of cognition have failed to shed much light on the principles of cognitive support. It is difficult to see how an “inner” model or theory can shed light on the principles of cognitive support. What can be known about the impact of tools on cognition if the tools are assumed not to impact it?

Thus, the problem is not simply that cognition is unstudied, but that most of the cognition-oriented work in the field builds knowledge about developer behaviour or developer psychology, not knowledge of how tools mediate, modify, and support such behaviour. When models of developer behaviour are considered, they are typically believed to apply across many—if not all—development environments without significant modification. As a consequence, they have no power to make statements about cognitive support provided by the tools. To say something interesting about cognitive support, the models would need to indicate how a change in tools or development environment changes cognition. For a model to be able to do this, it is logically necessary to take into account some significant aspects of the developer’s environment. Too few actually do so. This complaint was strongly voiced by Bellamy and Gilmore when discussing their studies of planning and programming strategies:

To understand the psychology of complex tasks such as programming, we need to consider planning strategies in particular task contexts. Theories of planning and problem solving have spent too long with their heads in the sand, ignoring the role the external world plays in determining behaviour. If psychology is going to make significant contributions both in theoretical and applied areas of research, we need investigations of how features of the external world determine behavioural strategies. Only then will we be able to produce artifacts that support effective task strategies. [41, pg. 69–70]

Much more is said about this issue in later chapters. But for now, the goal is only to establish that the topic of cognitive support has often failed to receive due attention, so let us only consider an example and some of the implications.

A *raison d’être* of a cognitive model is to formalize knowledge about cognition. Cognitive models enable one to think and reason about cognition. A *raison d’être* of cognitive *support* theories is to formalize knowledge about how to *support* cognition. Cognitive models do not directly allow reasoning about support. Thus, the failure to systematically model and explain cognitive support should imply that

researchers must fall back on background knowledge and informal reasoning to understand how tools support cognition. As a case in point, let us consider the body of work by von Mayrhauser and Vans on the “information needs” of software comprehenders. von Mayrhauser and Vans developed a model of software comprehension [674,675], which they applied to analyze comprehension in a variety of tasks and environments [654,671–673,676,677,679,680]. They wished to use the knowledge gained in these sorts of studies towards developing better tools. This case is of interest not because the research is poor (it is cited quite widely and positively), but because it is an example of research on cognition that did not really study cognitive support. What kind of knowledge did they discover? Perhaps the main contribution is an analysis of “information needs” [654,668–670] for software comprehension, that is, a description of the sorts of knowledge that software comprehenders seek. This is what I would call “comprehension *process*” knowledge, not “comprehension *support*” knowledge. It makes a statement on the cognitive activities of comprehenders but it does not say how artifacts make these activities better in any way. This limitation stems from the fact that their model is an “inner” model only: it works as well for someone with a packet of printouts as for someone in front of a computer full of sophisticated software. They do go on to state that these information needs can be met by a variety of tools, and they do make many specific tool suggestions [668,670,681]. But at the moment, the question is whether they learned about *support* or about *comprehension*. What did their information needs analysis say about support? What was the role of their model and their studies? And what resources did they employ to reach their suggestions for tools?

Their information needs analysis said little about the possible means of supporting comprehension. In their earlier work [670], the sorts of tool suggestions they made were primarily along the lines of analysis, search, filtering, and display mechanisms for various types of program information (e.g., a hypertext browser). This is perhaps understandable given their focus on information seeking. But consider the following: how and why did they infer, in effect, that because software comprehenders are seeking particular pieces of knowledge then the main role of tools is to satisfy that knowledge (provide information)? If comprehension is to be made easier, maybe tools should seek to *eliminate* the needs for particular forms of knowledge, and in that way avoid comprehension problems? Perhaps comprehenders should also be *providing* knowledge to tools? The point here is that nothing in their models or studies prompted them—or prevented them—from making these sorts of suggestions. Their model and studies are neutral with respect to forms of cognitive support. One reasonable way of explaining their list of tool suggestions is that they used their background knowledge (see Section 2.1.2) about common forms of tool support and, to address the point of this section, their models did not provide them with an analysis of the support provided by their various suggestions, nor did it greatly help to generate their list. This complaint in no way suggests that the above sort of work is not valuable. It just points out that, even in cognition-oriented research, work in this field has for the most part failed to methodically investigate cognitive support. Even the most outstanding and methodical work on cognition can fail to adequately study cognitive support.

The pattern illustrated by von Mayrhauser and Vans’ information needs is common for many cognitive-oriented studies concerning development tools. The general form is to (1) recognize and model cognitive processes (e.g., common action sequences, or the knowledge used by comprehenders), and then (2) look elsewhere for support ideas, often with the intent of automating some portion of it. Weiser, for instance,

discovered a characteristic form of causal reasoning that is used during debugging [692] (backwards data-flow following) and this resulted in research on automated slicers [693]. Singer *et al.* [596] learned about and modeled activity patterns during software maintenance, but got their main design ideas through brainstorming sessions rather than directly from models of the activity patterns themselves [596] (see also Section 7.1.2). The overall pattern is one of studying cognition during software development, stopping short of actually studying cognitive support, and then having to fall back onto reasoning informally with background knowledge when suggesting or designing new tools.

Returning to the question this section asks, we have seen that cognitive support is, as a research topic, under-appreciated and insufficiently studied. Cognitive issues are often bypassed entirely; even when developer cognition is studied, the subject of cognitive support is typically ignored, or too frequently amounts to falling back on background knowledge and relatively simplistic design reasoning.

2.1.2 Folk, Tacit, and Science Knowledge

The intuitions of gifted and experienced designers typically play a crucial role [in artifact development]. By drawing on their experience, good designers become astute at interpreting user difficulties and relating them to problems in the design of the artefact and its interface. The tacit acquisition and application of knowledge may serve the purpose in some design contexts, but requires skill and judgment that is not easy to develop and to share. This motivates the search for supporting techniques and frameworks that can make the analysis of cognitive issues in artefact and interface design more systematic, and results of the analysis more accessible for recording, exploring and communicating.

– Benyon *et al.*, “Interactive Situation Models for Cognitive Aspects of User-Artefact Interaction” [45], pg. 357.

Research in our field calmly hums along without much consideration of explicit theories of cognitive support. Despite this, we have managed to produce tools that appear to have dramatically improved the cognitive work of developers. Historically speaking, this advancement is entirely unsurprising. It has been frequently noted in other applied domains that technology often advances without apparently needing a secure, explicitly theoretic scientific foundation [93, 102, 106, 455]. For example, bridge building was successfully practiced before many of the basic mathematical techniques of structural mechanics became mature [579]; a similar story holds for the steam engine [106]. As Sutcliffe *et al.* [626] note, “it is a typical pattern in HCI for new ideas to be first codified in exemplary artifacts and only later abstracted into explanations and principles” [626, pg. 214]. But even if it is *historically* unsurprising, it is surely still curious—amazing even—that we could have been so successful. Is it not a little like having painted the Mona Lisa without ever having sight? How did (do) we do it?

One of the first things that must be realized about tools research in SE is that we already have a great deal of knowledge about cognitive support—but that most of it is *tacit* or *folk* knowledge. That is, much of our knowledge is developed through long-term and intimate exposure to the problems of software

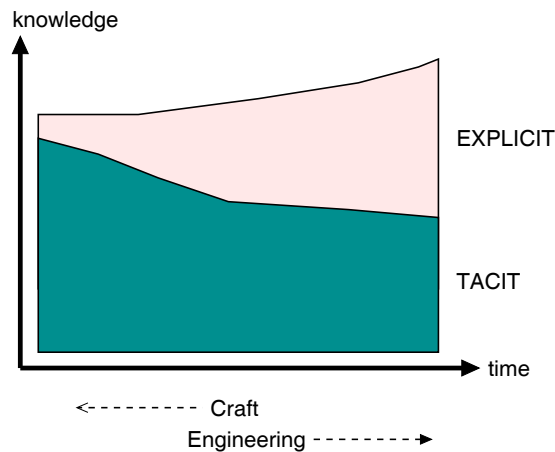


Figure 2.1: Knowledge explicitness and its relation to discipline maturation

development, and as such it remains to a great extent unarticulated, unformalized, and unsystematically developed [102, 579, 608]. Most of the researchers in the field have had extensive training in software development, are accomplished developers, and have many years of hands-on experience in both using and designing tools. Knowledge developed through this sort of extensive practice is often called “craft” knowledge. Enterprises that tend to have such inarticulable experience, and that tend to hand down that knowledge through long enculturation processes are called “craft disciplines” [391]. As craft disciplines mature they frequently are able to make the tacit, craft knowledge explicit and directly conveyable in the form of theories, models, and rules. Before such time, the wisdom tends to be predominantly folk or craft.

The basic progression from craft to engineering through the formalization of knowledge is illustrated in Figure 2.1. Knowledge quantity is represented by height of the bar and the makeup (explicit vs. tacit) of that knowledge is indicated by depicting a split between tacit and explicit knowledge. In the diagram, the tacit knowledge is reduced not by forgetting things, but by converting it into explicit knowledge. Science-based engineering disciplines are recognized as having significant explicit content [102, 391, 579], although practical, experience-based knowledge is always present and important [594, ch. 5]. Nevertheless, during the maturation process some of the tacit knowledge is converted into explicit, articulated knowledge. This knowledge is applied in increasingly controllable and predictable methods. As Long and Dowell say:

...Craft disciplines give way to engineering disciplines: personal experiential knowledge is replaced by design principles; “invent and test” practices (that is to say, trial-and-error) are replaced by “specify then implement” practices. [189, pg. 127]

Where along this progression is tools research in SE? To a significant extent, it is currently a craft discipline regarding our knowledge of cognitive support. Many of us are effectively engaging in applied amateur psychology and social science, whether or not we consciously think so. The tools we build carry significant psychological and sociological impact, altering how, for instance, problem solving in program development is being done.

How has this craft discipline evolved? In the past, when new tool ideas were being reported, simple sorts of argumentation and empirical evaluation were normally considered sufficient. This was partly

because the primary focus had been on the technological innovations presented by the tool's design [263, 640], and partly because little emphasis was placed on greater formality or rigour in either the argumentation or evidence [724]. Moreover, many authors were greatly helped by the simple fact that, after all, most tools are fundamentally more similar to existing tools than they are dissimilar. Almost all tools are complicated entities with features that are similar in hundreds of ways to dozens of other tools.³ Reviewers of papers that introduce new tools often would have used many similar tools and could therefore intuitively grasp many of their advantages even if these advantages are never fully articulated. Such situations greatly lessen the need for authors to fully articulate and test the reasons why their tools are beneficial: they can rely on the reader's familiarity with the tools and their cognition-related benefits. Consequently, in the past the main emphasis was naturally directed to the technical innovations rather than to the many other features for which the community had already achieved consensus as being beneficial. There was little impetus to develop clearly defined terms and concepts concerning cognitive support, and little emphasis was placed on more formal, rigorous testing of the basic support ideas. In addition, there was substantial cultural bias against social sciences-related issues in academic computing regarding tool design (e.g., see Curtis [158], Curtis *et al.* [159], or Green [263]).

Consider two simple but illustrative historical examples of the above principle. Early programming environments were batch-oriented rather than "interactive", so that programmers normally wrote programs on paper, submitted them to a keypunch operator, and then (much later) collected the output as a printout. Later on, when computing became "online" with the advent of cheaper time-sharing systems (and, subsequently, with personal workstations), new forms of interaction were made possible, such as direct manipulation environments using mice. Many of these interactivity innovations were related to making better human-computer interaction rather than merely computing more effectively. When these innovations were first introduced, their basic forms and their potential benefits were not well known, so authors promoting them sometimes made considerable efforts to describe their basic features and to relate these to the relatively "soft" issues of improving human-computer interaction. For instance, Baecker [19] needed to argue that the close interactivity of time-sharing systems permitted qualitatively different and better debugging since one could interactively examine the data as the program ran. Similarly Teitelman [636], writing in an era before modern windowing environments were commonplace, spent a considerable amount of time describing what a mouse was, and how multiple windows on one screen could be useful to a developer. Nowadays, to our ears these explanations may seem excessively detailed⁴—almost comical—and virtually nobody would waste time now in explaining such features in detail or in debating their possible advantages over batch-oriented, text-only interaction. The original papers arguing these basic concepts—which must surely be considered important bases for software development tool research—are rarely ever even cited. Even if niggling questions remain about superiority of the new tool features (e.g., are WYSIWYG editors *really* better? [95]), they are understood by the community as

³In this domain, if a tool appears simple (e.g., `grep`) it is almost surely because it is a tool embedded within a larger collection of tools (see Lethbridge *et al.* [381]).

⁴For example, Teitelman says "... the mouse is a small object (about 3" by 2" by 1") with three buttons on its top ... The user views his environment through a display consisting of several rectangular display "windows". Windows can be, and frequently are, overlapped on the screen." [636, pg. 159].

being generally beneficial and now our interests lie elsewhere. We have remembered the lessons—made them part of our craft knowledge—and have moved on. Similar things happened for other support ideas.

If we remember our lessons, how is this learning achieved? One way craft knowledge is *not* preserved, generally speaking, is through the development and propagation of explicit explanations of why tool features are useful. Instead, there are two other methods that do not involve explicit theories of cognitive support. The first is within the tacit understanding of researchers that is won through experience. It is exceedingly unlikely, for example, that any tools researcher in SE would not have had significant experience in using multi-tasking windowed environments. Many have even experienced great discomfort when forced to do without it, such as when being forced to switch to single-tasking environments (e.g., PC-DOS) that make it impossible to rapidly switch between applications. The value of effective multi-tasking environments seems hardly worth explaining. Other ideas are similarly accepted through exposure.

The second main store of cognitive support knowledge is actually within the tools themselves.⁵ As Carroll *et al.* noted [104,105], artifacts embody a type of wisdom about the psychology of their users and the activities they engage in. The properties of a multi-tasking windowing environment says something about the problem-solving methods of its users. For instance, the utility of having separate windows that make it easy to switch focus implies a need in the user (or their work context) to switch attention rapidly. Automated type checking in compilers make implications about the types of errors committed by programmers. The ubiquitous “undo” feature speaks volumes about slips, errors, and exploratory tool use. It is hard to overemphasize the importance of tools as storehouses of support knowledge. When designing new tools, designers frequently begin with a familiar (i.e., “exemplary” [626]) artifact form as a starting point and modify it to suit the situation. Singley and Carroll called this sort of designer reasoning “hillclimbing from predecessor artifacts” [598]. For example, when envisioning a tool’s features we might begin thinking of it as being “hypertext-like” (e.g., when developing a code browser [596]), and proceed by using our experiences with hypertext systems. When copying existing successful tools we are doing nothing less than reusing the knowledge about successful cognitive support that is tacitly encoded in the tools themselves.

If much of our knowledge is embodied in tools, how do we create it in the first place? One possibility is that it is to a great extent accidental: good tools can be built without knowing, *a priori*, if or why they are good (this thesis is expanded upon in Chapter 7). That is, the psychological and supportive knowledge embodied in a tool may not have been known to its designer in advance [110]. Iterative testing weeds out features that incorrectly understand the psychology of the user—it separates “nuggets of gold” from dirt [370]. This means that even pure guesswork (or, somewhat more charitably, a “lucky hunch” [370]) is occasionally rewarded [189]. Furthermore the “lucky hunches” need not be all that lucky: because tool design is normally highly iterative [251], big guesses can be broken up into a sequence of tiny guesses that can be checked before proceeding. Ignorance of the principles of cognitive support therefore presents no absolute barrier to development of better forms of tools, so long as good designs are preserved and accumulated within the communal storehouse of known tools (see Chapter 7). Consequently, better tacit knowledge about the principles of cognitive support *can* accumulate without much explicitly theoretical

⁵Or in secondary artifacts such as descriptions of tools or even patents [510].

understanding. And our main method of accumulating these principles currently is within tools. When these tools are copied, the good ideas are propagated.

Nevertheless, SE researchers may not wish to rely on lucky guesswork. Researchers can make *educated* guesses using whatever sources of knowledge are available to them. Good guessing results in faster convergence to good designs. To make better guesses, SE researchers can watch people work and try to understand their thinking processes, or try to introspect their own thinking processes and then reason about plausible tools to support them. They can perform case studies and visit or collaborate with working programmers in the field in order to observe developers at “authentic” work. In fact, this is what many tools researchers currently do. They are folk psychologists, weekend anthropologists, armchair sociologists. Case studies and expert opinion based on deep field experience is heavily valued within the community. Moreover, a form of experimentation is widely used. Researchers make changes to a prototype tool and observe how developers react to it, soliciting feedback reports from users. In short, SE researchers can and have used many of the tools of anthropologists, sociologists, and psychologists, however in a much more informal and a less rigorous or controlled manner. The distinction between an acknowledged science discipline and these sorts of craft disciplines can be, in some respects at least, surprisingly small [110].⁶ The claim that cognitive support is not studied *at all* is therefore quite incorrect, but the characteristics of that study is of a craft-like discipline, not a science-like discipline.

In summary, tools research in SE has historically managed to creep along as a craft discipline. Knowledge about good support ideas is created by intuition and guesswork, this knowledge is encoded and passed down in the form of tools, and the knowledge is reused by adapting existing tools during design. Knowledge about cognitive processes in software development—and how to support these—lies latent within these tools. An appreciation of these principles is gained through first-hand experience with the tools and in-person observation of their use. The craft discipline therefore resembles, in certain ways, an applied science (or a *design science* [110]), however the representations of knowledge are less explicit, and the methods lack proper rigour. The call for more explicit investigations of cognitive support is nothing more than a call to render our current activities more scientific—more principled.

⁶For an interesting take on the difference between science and non-science, see Sagan [556], particularly chapter 18.

2.2 Problems of A Theory-Thin Discipline

At a recent workshop, a software engineer insisted that his only criterion for assessing what psychologists had to offer was, would it save him money in development? To his ears, that was no doubt perfectly reasonable. To my ears, it was a case of double standards. The software engineering community sinks vast sums into systems such as CASE tools, based on collective delusions about the nature of the design process. The community ought to be interested in refining its beliefs and bringing them nearer reality. To ask an expert on thought processes to give a financial estimate of the value to be gained from refining software systems is obviously a piece of evasiveness.

– T.R.G. Green, “Why Software Engineers Don’t Listen to What Psychologists Don’t Tell Them Anyway” [263], pg. 330.

Even if cognitive support has not been studied well enough, does tools research in SE *really* need new theories of cognitive support? After all, we have managed to proceed quite successfully as a primarily craft discipline. So even if one were to grant that SE is fundamentally interested in cognitive support, it still is important to ask whether theories and models of cognitive support are really needed. Perhaps the status quo is fine?

A simple counter argument goes as follows: Why not try? There is very little to lose, and potentially so much to gain. To some it might be tempting to leave the argument at that point, but it is much more instructive to evaluate our existing research practices and consider some of the ways in which it is problematic—or even strictly inadequate. Constructive criticism provides a more informative motivation for improving the quality and character of our discipline. If we can identify our problems and trace them to a lack of appropriate theories, then we may find that instead of being an unheralded dark horse, a theoretical approach to cognitive support might be our most promising option.

Three points first need to be clarified in order to avoid confusions about the focus of this section. The topic of this section is the practice of SE research. The first point of clarification concerns the relationship between this work and HCI work. The field of SE tools research overlaps only partly with HCI (as well many other fields like math, logic, psychology, etc.). To readers with strong HCI backgrounds, many of the points brought up in the next section may seem familiar, but many of these points are not well known in SE research circles and thus bear some repetition here. Furthermore, this section is more than simply a review of HCI literature: I have attempted to directly relate the literature to SE research. Thus, although there is some subject overlap with HCI, the specific viewpoint of the SE researcher slants the issues. The second point is that it is desirable to distinguish between the *topic* or *content* of SE (or HCI), and the *research practices* of SE (or HCI), by which I mean the research and publication activities of the participants in the field. The subjects of HCI and SE may overlap even if most of the research practices do not. The final point is that it is also desirable to distinguish between SE researchers and SE practitioners [93]. The latter are normally more interested in making specific tools according to (as much as possible) known principles, whereas the former usually build tools in order to discover new principles. To reiterate, the following

is primarily concerned with the *research practice* of the *SE research* community. Much may be gained by further expanding the critique (e.g., to practitioners in SE or HCI), but that is left for others to do.

The key to evaluating research on cognitive support is to realize that the research programme in SE is one that routinely trades in ideas, claims, and explanations of cognitive support. As a group we are interested in the question of how to constructively aid in the hardest and most cognitively challenging parts of software development. We routinely strive to make software development easier while simultaneously recognizing that some of the main bottlenecks are conceptual and cognitive in nature. This concern for the cognitive aspects is clearly revealed in many of our research topics. Notations are evaluated as to whether they are clear and easy to understand; tools are claimed to be natural, intuitive, easy to learn. Languages are exalted as being easy to program in, or to comprehend [433]. Not surprisingly, many of our activities consequently involve talking and reasoning about psychological issues and cognitive support: when trying to develop new technology to aid software development, when comparing different tools, when evaluating and testing our ideas, and even when educating new legions of researchers. Part of our practice is, effectively, a dialogue concerning cognitive support. A push towards explicit theories is nothing less than a desire to increase the precision and formality of the very activities we already engage in [102]. Insisting on theories grounded in scientific research methods is essentially a demand for rigour [101,370].

If explicit and science-grounded theories are lacking, we would expect to see problems in our research programme that are symptomatic of a discipline that is missing the organizing influence that comes from a strong science practice of modeling and methodically investigating the issues at the discipline’s core. As the following sections argue, this is exactly what we find in the patterns of research in SE and CS concerning cognitive support in tools. It is beyond the scope of this work to thoroughly discuss the problems, but several important points will be touched upon. First the argument presented above—that research centred on theories of cognitive support is consonant with the goals of science research in computing—is expanded. Then some key difficulties in tool argumentation, evaluation, and design are surveyed, and these difficulties are then related to the lack of suitable theories that could guide research. This survey of problems in research practices is summarized in Table 2.1. The portrait that emerges from the survey is of a field that faces troubles stemming from its lack of a theory-guided research stream.

ACTIVITIES	RESEARCH PROBLEMS AND DIFFICULTIES
evaluation & testing	support claims are poorly articulated and tested
	“whole tool” testing is needed but is burdensome, problematic
	tools researchers are forced to do cognitive science
analysis	informal analysis suffers from concept- and lexicon-poverty
	deep knowledge of psychology or cognitive science is often needed
design	design is affected by theory too late, or not at all

Table 2.1: Summary of problems in SE research practice due to lack of cognitive support theories

2.2.1 Evaluation Problems: Simple Comparison

Although I could imagine a study to determine whether apples tasted better than oranges, I would not dream of conducting it because I am fairly certain that “it would depend.”

– John Karat, “The Fine Art of Comparing Apples and Oranges” [479], pg. 265.

Ideas about tools are generated and need to be evaluated and tested. It might be supposed, for instance, that a multi-focus “fisheye” interface would be beneficial for understanding software (e.g., SHriMP [621]). How do we evaluate such ideas? What problems do we face in evaluation and why? In the following, some of the most prominent empirical evaluation methods are explored through a sequence of scenarios interleaved with discussions of the problems they illustrate. A summary comparison of these practices is presented in Table 2.2, and the problems encountered in Table 2.3.

Scenario 0: Ad Hoc Evaluation

Elsie has an idea for a new program analyzer with a browsing interface for navigating the information the analyzer extracts (a tool similar, perhaps, to SNIFF [1] or Rigi [426]). She has designed this tool and built a prototype based on her belief that the tool can make it easier for programmers to understand their code. How does she know her idea is any good? She might begin by arguing that software comprehension is hard and expensive, that tool support is therefore needed, that her tool incorporates some of the functions (scrolling, search, etc.) which are clearly useful, and that her navigation interface is “natural”, intuitive, and easy to learn. As to how these features make program comprehension easier, she could suggest that the interface exploits the “powerful mental capabilities of the user,” and that the tool supports the user’s activities because they are functions needed by the user. She informally “tests” the tool on some sample problems in order to iteratively improve her prototype, and writes up an experience report.

Until fairly recently this level of justification might have been enough for Elsie, particularly because, as I argued in the previous section, most researchers were primarily interested in the technological advances, and there was a great degree of consensus over the value of the main parts of the overall tool and its environment. Recently, however, there has been increasing pressure to provide stronger theoretical and empirical evidence as to the quality of our ideas [640]. Elsie and her colleagues are wanting better evidence and proof. One approach Elsie could try is a tool comparison in the style of validation experimentation [724]. In this sort of experimentation one picks a performance measure of interest—total maintenance time, for instance—and compares developer performance with differing tools. The aim is to establish the relation between input variables (the tools) and output variables (measured performance). In this type of work, models (i.e., mathematical formulas) that predict the relation are called theories [724].

In Elsie’s case she could try to establish that her tool results in faster maintenance than some specific set of tools (perhaps a selection of the most common ones). This style of experimentation might be appealing to Elsie since (1) it has the potential to establish the superiority of her tool according to some measure (maintenance speed), and (2) it requires almost no understanding of the fine structure of the processes

of software maintenance, nor of the principles by which her tool actually caused the desirable output. It is a technique that makes it possible for her to establish *that* her tool helps, while maintaining maximal ignorance about the reasons for *why* it succeeds. Similar sorts of experiments could verify that adding yeast to your bread dough triples the height of your loaves even if you know nothing whatsoever about the biochemical action of yeast, and you have never even heard of carbon dioxide. Elsie can treat the user-tool unit as a *black box*: maintenance problems go in, and satisfyingly objective and quantitative data comes out. There is no need for pondering psychological effects, and no need for arguing theories of support. There is no question that these are formidable selling points for SE and CS researchers, but there

	INFORMAL TESTING	BLACK BOX COMPARISON	UNGUIDED OBSERVATION	COG. MODEL OBSERVATION	THEORY BASED
basic orientation	early feedback	adoption justification, evaluate tool promise	discovery, learning	behaviour discovery	generation & investigation of support hypotheses
empirical goals	informal feedback, serendipitous discovery	tool validation (find performance improvement)	discover key factors, understand use context	improve cognitive model	verify support hypothesis, measure support
observation focus	tool	output variable	whole scenario / task, user problems	human actor	tool effect & effect mechanism
study scope for tool	key features	whole program	whole program / environment	frequently ignored (!)	specific features
claim characteristics	informal	simplistic questions	to-be-discovered	informal	support mechanisms
perceived threats to method	folk beliefs	statistical validity, replicability	discover nothing new	ecological or statistical validity	incompleteness, irrelevance
claim articulation level	informal / ad hoc	N/A	informal / ad hoc	human-oriented model	human-tool model
benefit explanation type	descriptive, folk	avoided	observer dependent	informal	theory-based
explanation knowledge source	tacit expertise	N/A	observer's background	tacit / psychological training	science-grounded theory

Table 2.2: Comparison of methods used to evaluate tool ideas

are serious problems with this method.

Scenario 1a: Black Box Methods Part 1

Elsie decides to try a comparison, but she quickly runs into some troubles. Her tool is an unpolished prototype containing a few of what she thought were minor usability flaws. But she finds out that these flaws appear to have significant practical impact: her subjects are really affected by them. She feels she might be able to fix up those problems if she did a few rounds of usability testing and tool improvement first. But it might take months or even years before she gets it right, and she wonders if this is really worth the effort because there is a chance that her idea is fundamentally flawed. Despite the possibility that her prototype is a real gem-in-the-rough, she discovers that missing, inappropriate or even extra features entirely confound the effects of that gem on the experimental output. Furthermore, as a research tool it has enough novelties in its usage that it is hard to find many experts. Elsie watches her pilot subjects either ignore the novel features, or else clumsily struggle with them when the experimental design forces them to try these. She begins to suspect that much of the tool's potential is untapped by her subjects because they need months of long-term exposure to become proficient to the point where, like a pro golfer unhesitatingly pulling a 2 iron out of her bag in just the right circumstances, the special capabilities of the tool are integrated almost unconsciously in their strategies and problem solving. Meanwhile she watches as her control subjects skillfully use their habitual tools in highly inventive ways, and she wonders if her comparison experiment is entirely fair.

Scenario 1b: Black Box Methods Part 2

Despite her initial difficulties, Elsie fixes up her tool by adding functionality and cleaning up the interface, manages to recruit a handful of expert subjects, and runs the selected tools head to head. Unfortunately her results are somewhat disappointing: she sees a modest improvement, but the data exhibits uncomfortable levels of variability in performance. Maybe there are factors she has not controlled or randomized well enough? She fixes up the tool some more, runs another test, and gets marginally better performance. With some sort of performance improvements in hand, she can probably publish her results, but really she has more questions now than when she first began. Later when she adapts her tool to a new programming language, she is forced to repeat the whole process. Who is to say she did not destroy the benefits with her tool modifications? After submitting her results for publication, she tries the tool again on several different maintenance problems and discovers that the results vary considerably.

The practice of using validation-style experimentation for evaluating tools is quite widespread across the spectrum of SE research. Its use can be found in evaluating software visualization [67, 387] and reverse engineering tools [620]. It shares a long history with evaluations of different software development techniques, languages, and programming notations such as object-oriented language features [161] and program indentation and formatting techniques [694]. The aims of such methods are noble—to increase the scientific foundation of software engineering [724]. In the context of idea evaluation, however, the ways in which such methods are used undermine the effort.

PRACTICES	PROBLEM SOURCES	PROBLEMS ENCOUNTERED
informal evaluation	informality in methods and explanations	arguments and data are unconvincing
simple comparative validation (black-box methods)	simplistic questions	weak results
		fails to cumulate knowledge
	whole-tool evaluation	cannot use prototypes
		idea validation is late
	iterative prototype improvements	stopping criteria are unknown
		improvements weaken core arguments
explanation avoidance	support concepts are poorly articulated	
	support concepts are indirectly tested	
undirected observation	minimal theory being applied	heavy dependence on observer excellence
		analysis is made more difficult
	mismatched goals	learning is about cognition / support
		burdens of theory argumentation
	inappropriate methods	uncertainty of fishing expedition
tool shortcomings are highlighted over successes		
cognitive model based observation	model weaknesses	issue of support is not addressed
	mismatched goals	learning is about support mechanisms
		burdens of theory argumentation

Table 2.3: Evaluation and testing problems compared to the promises of theory-based research

In this discussion I am not primarily concerned with issues of methodological rigour in experimentation. Such concerns have been brought up many times in the past, such as whether we are studying the right populations of programmers [155], using toy problems [345], properly controlling variables [413], properly emphasizing effect size and reliability instead of statistical significance [238, 369], developing improper statistical tests [410], replicating results [35, 161, 162], and so on [68, 74, 238, 308, 413, 581]. Even ecological validity, which is a critical issue for HCI and SE work [369], is a problem that can potentially be addressed using this sort of experimentation. For instance, techniques from applied experimental cognitive science [402] or ecological psychology [221] have been developed to handle this style of experimentation in ecologically valid circumstances. Of course all of these methodological issues are important. But my concern here is not of proper execution, but of asking the wrong questions and applying otherwise reasonable techniques inappropriately.

There are three sources of problems: simplistic questions are being asked, whole tools are being tested instead of evaluating the tool ideas, and explanations of the benefits of the tools are neither developed nor tested. These problems are outlined below.

Simplistic Questions

On the strictly atheoretical extreme, an empirical study can be performed without a theoretical underpinning, but the utility of such a study is limited. In contrast, a theory provides guidance for targeting what behavior to study and for assessing a study's results.

– Karen N. Whitley, “Visual Programming Languages and the Empirical Evidence For and Against” [703], pg. 111.

Human activity in software development is very complicated, so it should surprise almost nobody that comparative evaluation of tools and techniques in software development are quite difficult and involved. Sheil pointed this out a long time ago when he said:

Such evaluations might, at first sight, seem to be a straightforward matter of comparing the performance of groups of similar programmers using different programming techniques. Unfortunately, the complexity of programming behavior makes the execution and interpretation of such comparisons anything but straightforward. [581, pg. 102]

One particular problem is the sort of questions that are posed in validation-style experiments. Frequently they are overly simplistic—especially in regard to the sorts of cause-effect relationships being sought. As Carroll once said of HCI, the field “suffers from a methodological bias for posing elegant either-or research questions that idealize away variables like task context” [99, pg. 88]. A decade ago Green reviewed the field and noted that:

Many of the early studies asked simplistic questions: ‘Are logical conditionals better than arithmetic ones?’, ‘Are flowcharts better than code?’, ‘Are nested conditionals better than GOTOs?’, etc. It is easy to see today that in general the answer is going to be ‘X is better than Y for some things, and worse for others’. [261, pg. 124]

This simple fact argues against using (simplistic forms of) comparison-style experiments to understand or even evaluate tools. There are just too many interrelated factors for such an approach to yield informative results. As the work by Storey *et al.* demonstrates [618,620], this fact is certainly borne out for comparing software visualization tools. Their work provided a demonstration that even though simple questions about the value of visualization tools can certainly be posed, the complexities of realistic software development can make simple questions seem like the bluntest of investigative instruments.

The thing is, we really *do* want to ask the complicated questions. We *need* the complicated answers. For hypertext, Wright noted that a

... binary judgment of either “Yes it works” or “No it doesn’t” may underestimate the importance of the interplay among the design decisions ... so the simple question “Does it work?” is too imprecise to be useful. [716, pg. 2]

A simplistic question gives only simple answers which leaves us too little substance to address the design issues we consider important. Two significant problems arise because of this. The first problem—alluded

to in Elsie's scenario above—is the somewhat notorious fact that comparative experimentations have results that are weak or equivocal [308]. The second problem is that it is difficult to cumulate knowledge gained by asking simplistic questions. These two problems are discussed in turn.

WEAK RESULTS

The weakness of many comparative experimental results has been noted in many different circumstances. In software development perhaps the most prominent examples are studies of program indentation and formatting [261, 581, 694], and of visual versus text-based programming languages [703]. But weak results for simple comparative experiments are pervasive. They have been found to occur in software visualization environments [508, 620], layout differences for graphs [519], software remodularization tools [359], and (despite a dizzying bevy of statistical tests) program inspection tools [411, 412]. Weak results also invade related comparative tests such as visual versus textual information [112, 332] (or analogical versus propositional representations [365]), and hypertext versus traditional book forms [182, 716]. Although there are many possible causes of weak results, two possibilities are related to the use of simplistic questions: the effect size is too small relative to the scale of the question being asked, and the human processes being observed are themselves just too unpredictable to answer the gross experimental hypotheses.

Weak results might be expected if the performance gains attributable to the tool (i.e., the input variable) are marginal due to the fact that the fraction of the overall problem it addresses is itself rather small. One cannot expect a 50% improvement in productivity if, after all, only 1% of the causal factors are addressed in the tool's design [82]. Software development tasks are not simple tasks like touching a button when a light turns on. For real software development tasks one should probably expect that the overall performance impact will actually be modest. Software development is a “wicked” [541] problem and it is unlikely that most tools [342] would qualify as a “silver bullet” [78] that can transform wicked problems into tame ones. If this is so, then experimental conditions may need to be exquisitely sensitive in order to reliably expose what is (at least relatively speaking) a minor effect. A whole host of other factors will, in combination, normally dominate the performance of software developers. Because of the sensitivity and control needed to show such a minor effect [238], some experimenters have been driven to use highly contrived experimental setups in order to make the effect of interest dominate the performance [103] (e.g., Davies' use of highly crippled editors [167]). While such contrived experiments are obviously useful for clearly exposing the phenomena of interest, the sensitivity needed during experiments bears witness to the relatively small *overall* effect on performance. The effects are easily lost to the error bars from the gross development tasks. Although this is perhaps not a fatal difficulty for comparative experiments, it clearly disadvantages them even without their other problems.

For “micro” tasks (e.g., making a menu selection, checking in sources to a repository) it seems reasonable that relatively large and easily measurable performance increases may be achieved. But making a dent on the *overall* domain task costs is often the consideration, and expectations for large measurable differences should probably be modest unless the tools being compared are enormously outmatched. There are at least two good reasons to offer why this should be so. First, making significant inroads on the mentally challenging problems in software development is hard and we have, realistically speaking, only just

started directly tackling the issue of cognitive support. Landauer [371] spent considerable time arguing essentially this point—that at the broadest task contexts we have measured only marginal increases in productivity precisely because automation is what we are good at, and that most of the easily automatable things already are well-automated. The easily automatable thus makes up only a small fraction of the work that is left. Second, for the broadest task contexts, the open-endedness and flexibility of human behaviour makes for even competition. Although effect size (and the related issue of effect reliability across individuals [238]) is important to applied disciplines [402], in practice the effects being searched for may be too easily drowned out in fairly matched comparisons. A case in point is typical Unix development tools. Practice and expertise can make up for many flaws in their design [468]. So the seemingly unremarkable Unix tools can make formidable opponents in comparison evaluations [380,632]. On the whole, then, simple comparisons run into a host of obstacles making their overall contribution either minor or hard to fairly measure and detect.

Another source for weak results is that the usefulness of a tool may be highly dependent upon various uncontrolled or unanticipated experimental variables. This possibility is hinted at by Wright's quotation above. Simplistic questions often belie a desire for simple answers that are not forthcoming. Are hypertexts better than traditional books? Are visual languages better than the more textual languages? Are apples better than oranges? These questions all presuppose a simple answer. But the answer typically is: "it depends". The dependencies are frequently uncontrolled for in experiments. Sheil cited this problem in reference to studies examining the usefulness of comments in programs:

Although the evidence for the utility of comments is equivocal, it is unclear what other pattern of results could have been expected. Clearly, at some level comments have to be useful. To believe otherwise would be to believe that the comprehensibility of a program is independent of how much information the reader might already have about it. However, it is equally clear that a comment is only useful if it tells the reader something she either does not already know or cannot infer immediately from the code. Exactly which propositions about a program should be included in the commentary is therefore a matter of matching the comments to the needs of the expected readers. This makes widely applicable results as to the desirable amount and type of commenting so highly unlikely that behavioral experimentation is of questionable value. [581, pg. 111]

Can comments be useful? Of course! But if the wherefores and hows are not understood beforehand, and not accounted for in the experimental setup, then we should expect our results to be equivocal at best. The same is surely true for any simple hypothesis testing concerning software development tools.

In terms of method appropriateness, however, the more important cause of weak results might actually *not* be either the relatively small effect size or the contextual dependencies—it might be the variable nature of the processes being observed. It has been widely noted that performance is highly variable between individuals and even between different performances for the same individual [158,197,655]. Such variability makes for messy statistical analysis. Individual performance differences appear to be partly a matter of individual psychology [185], but also partly a matter of proper training [158]. Even so, there is considerable subtlety to the issue of variability—individuals are not so variable when performing tasks

without significant problem solving. This was artfully illustrated by the careful word-processing experiments performed by Card *et al.* [94]. They ran different word-processors head-to-head in order to measure performance differences [94]. They were able to show significant performance increases on certain document editing problems. Because of the notable increase in performance, one might wish to shrug off their results as due mainly to the large effect size—an effect that large might have been hard to miss even with less meticulous experimental techniques. But a second explanation can be offered, however: Card *et al.* achieved good results because the process they were studying is reasonably predictable based on the experimental conditions that could be controlled. That is, the observed process was *well conditioned*. They had their users make changes to a document in order to get it to conform to a printed version of the document that had been marked-up with the changes. For experienced word processor users like their subjects, that task presents few cognitive challenges. Practiced physical and cognitive skills could therefore be applied. Indeed, this fact is precisely the reason Card *et al.* chose their tasks—they were concerned with modeling and predicting skilled performance.

Herein lies the problem of applying comparative methods in the domain of software development: the tasks of interest are not ones that can be performed using only skills-based methods. The so-called “device-level” tasks can be skilled and predictable [314, 370, 446]. The sorts of problems being tackled in software development—despite long years of training [158]—*never* become routine skills [361]. This means that when one is interested in tool support for software development, one is necessarily most interested in the so-called “higher-level” cognitive functions like learning and problem solving. To put it into context, how might Card *et al.*’s experiments have fared if the word-processors were the same, but the task chosen was to write an article, book, or dissertation? These are cognitively challenging tasks that frequently do make heavy use of a word processor (or “text processors”, if you prefer). But in these tasks, the overall processes are much more poorly conditioned. Individual knowledge and problem solving strategies begin to dominate. These are things the subjects bring with them into any experiment. True, word processing is a smaller part of the task of writing a book than it is of making simple edits to a document, but the fact is that even more sensitive and meticulous experimentation would not reveal the relatively small effect—it would only highlight the variability of the performances of individuals. True, if one restricts one’s study to only the lowest task levels like scrolling and deleting words, a reasonable level of predictability can be expected (perhaps—see Draper [191]). For example, in programming environments it is reasonable to try to compare the relative costs for the skilled execution of common low-level edits. Indeed, this has been done for syntax-directed editors for programmers [644]. But these works ignore the overall task context: the experimentation works only for the practiced parts of the development process, not the hard problem-solving parts.

The point to note about the above is that the difficulty for experimentation is not (only) that significant portions of processes like writing⁷ are *poorly understood*, but that they are poorly conditioned. Predicting such processes may thus be somewhat akin to weather forecasting: it is possible to understand the mechanisms underlying weather, but it is not possible to predict weather over time frames that are important

⁷Software development and design has often been likened to writing (e.g., [244, 499, 575, 603]).

to us. The software development processes of interest are much more like weather than simple editing.⁸ Whole-tool comparative experimentation in such highly cognitive domains is thus fundamentally problematic; unfortunately, the clean experimental results on well conditioned processes (like Card *et al.*'s) are red herrings that are unlikely to be repeated for comparative evaluations in circumstances that are interesting to investigators of cognitive support.

CUMULATION IMPOSSIBLE

A second problem raised with the method of asking simplistic questions in experiments is that, because of the form of their results, it is difficult to use them to build theories about tools which can adequately reflect the complexity of software development. The use of simplistic questions avoids understanding how the factors affecting performance interact, and why. It is therefore hard to know how to combine the answers one gets by asking such questions—even if one has a good idea of what the factors are. Say we manage to learn that *SHRIMP* is better than *Rigi* for certain tasks [620], and that *StarDiagram* is better than *vi* and *grep* for other tasks [64]. What now? With just this knowledge it is impossible to know how one result speaks to the other. Even if we manage to correctly guess that the major factors involved include the features of the tool, the task, and the user, the experimental results are better suited to tool adopters than designers. Designers want to know how to change their tools (see Chapter 7), yet the available results say only whether the existing tools might be any good. Returning to our scenario, Elsie might make a change to her program and re-run the experiment, but it would be far from clear how to know what to do with the results (no matter what they turn out to be). Her experimental setup does not allow her to find out what made the difference, and she must again resort to guesswork.

The above problems are symptomatic of experiments that seek to verify simple cause-effect relationships rather than seek to expose the mechanisms creating a causal chain. Each of these experiments asks a slightly different, but simply statable question. This is not a good recipe for accumulating knowledge. As Newell once remarked, “you can’t play 20 questions with nature and win” [444]. In his view, the problem is that knowledge just does not cumulate with experimentation based on simple dichotomous questions. He argued that building integrated models is a fundamentally more sound method of cumulating research knowledge [446]. His particular contention was that mechanistic models of causation are the best accumulators for science knowledge about cognition. Newell was talking specifically about cognitive models of individual psychology, but his argument seems equally applicable—perhaps even more applicable—to cognitive support. With cognitive support one wants to understand generalizable design principles, and it is hard to see how these could be constructed from simple tool comparisons. As Kirlik succinctly remarked:

Especially in HCI, a vast amount of research effort has been expended trying to answer questions comparing various interface technologies, for example, design options such as scrolling windows, hypermedia, and so on. This research is of dubious value ..., because the “it depends” answers produced by such efforts will only lead to a never ending series of technology-specific design principles, rather than

⁸This contention is somewhat contrary to what some works on cognition and human performance presume. Readers interested in debating this point can refer to Clark [136], Newell [446, ch 3.10] and, to a lesser extent, Landauer [370] and Carroll and Rosson [108].

a stable and generative theoretical account of human-environment interaction that can guide design in novel situations. [348, pg. 72–73]

Kirlik implied that the sort of knowledge needed by designers is difficult to collect and assemble into usable form from the individual results of simplistic comparative experimentation. Each experimental result is dependent upon myriad contingencies not accounted for by any cohesive theory. Without a theory that is able to abstract away these contingencies and relate the experimental results to one another, it is unclear how to integrate these “point-form” findings.

The fact is that, in the field of SE tools research, little or no work has tried to combine results from this style of experimentation, and it is currently difficult to imagine how it could be successfully done. Nonetheless, others have made serious suggestions as to different possible methods of cumulating scientific knowledge, so these must be investigated before it is presumed impossible. The need to integrate experimental results has, of course, been seriously studied for theories based on modeling mathematical or statistical relationships amongst experimental variables. Examples of this form of theorizing include various forms of meta-analysis of past experiments (e.g., on hypertext [120, 457], on the visual encoding of data [112]; also see the review by Miller [409]), the “sequential experimentation” method of Williges *et al.* [706], and the method advocated by Basili *et al.* [35] for disentangling and modeling performance relations (see also Basili [33], and von Mayrhauser *et al.* [678]). This last effort is perhaps the best example to consider since the other techniques mentioned tend to either compare the merits of competing theories (rather than build theories themselves), or else fail to actually integrate the results.

Basili *et al.* [35] suggested that knowledge be accumulated in the form of a cause-effect map of the factors affecting various qualities of performance. In this view, the process of experimentation not only provides evidence to support a given hypothesized relationship, it also serves to tease out a typology of the causal factors (e.g., development process type, design representation type, programmer experience, etc.) so that the effect of variations in the independent variables can be understood. The goal of experimentation is therefore: (1) to develop a list of input and output variables of importance, and (2) use empirical evidence to create a map of how they relate. The map, as a collection of hypotheses about cause-effect relationships, forms a theory. Basili *et al.* proposed their method for accumulating knowledge primarily as a way of counteracting unscientific beliefs about the effectiveness of various SE techniques. For instance, the technique was used to compare two different techniques for inspecting code [368]. Although these are not comparisons of “tools” like code visualizers, it is clearly possible to use such validation-style experiments in a similar way on such tools.

There is no way to know at present if the sort of knowledge building advocated by Basili *et al.* would succeed in creating suitable knowledge for understanding and engineering cognitive support (that is, *without* first supposing models of the mechanisms underlying performance). Yet there is enough evidence to raise doubts about the possibility. For one thing, the proposal relies on distinguishing and describing all significant variants on independent variables, including tool features. So, for instance, if Elsie made a novel modification to her tool she would need some way to consult the theory to determine if her tool would improve maintainer performance. At present this possibility seems unlikely. The cumulation, if it can be said to be occurring, is of facts that are in no principled way connected. Thus the accumulation suffers the fate that Kirlik noted of being a “never ending series of technology-specific” principles. However,

with an appropriate model-based theory, it seems at least plausible that one might input the tool features as a parameter and have the model *generate* a new prediction [446]. Perhaps a more direct threat to Basili *et al.*'s idea for the cumulation of knowledge is Newell's observation that simplistic questions are rarely ever convincingly decided one way or another [444]. The proposal of Basili *et al.* seems to depend heavily on being able to determine answers to just such questions. So even if Basili *et al.*'s process of accumulating knowledge is not strictly impossible, it seems more risky than its alternatives. In the past, similar methods have been tried for building "grand unified" theories in psychology (see e.g., Kjaer-Hansen [354]). Although they were not entirely without merit, the feeling now is that building actual computational models of thinking is a more fruitful tack.

On the whole, a seemingly more plausible position than that of Basili *et al.* is held by Green [259]. Like Basili *et al.*, Green envisions a way of decomposing the independent variables of interest, but unlike Basili *et al.*, he envisions a collection of relatively independent micro-theories that are kept unintegrated (see also Landauer [93,370] for a broadly similar proposal, and Rasmussen [526] for a different method of decomposition). For our purposes here, Green's proposal is inessentially different from Newell's because it adopts the premise that the causal mechanisms underlying performance should be understood and tested. Though he proposes an unintegrated melange of micro-theories, Green's focus on proposing models and evaluating them avoids asking simple dichotomous questions, and so makes reasonable the possibility of accumulating knowledge.

On a related note, it should be recognized that there are rather more immediate problems with generalizing simple comparative evaluations. What tools do you compare? Do you compare the "reigning champion" and, if so, how does one compare this champion to the other tools? Performing pairwise comparisons of all conceivable tools is obviously impossible. And unlike in medical experimentation, there are no valid baselines. The "white rat" that is called the "standard" Unix tool set has been developed by experts over decades. It is both sophisticated and honed by time. Developers have used it to build impressive systems, so it is demonstrably useful. Plus it is deeply embedded in many programming cultures and that often gives it the advantage of familiarity. Moreover, although the Unix toolset consists of some simple tools, the elegance and power of simple tools is frequently overlooked. Even paper and pen are sophisticated technologies with many psychological consequences [264,289,472]. The point is that tool evaluation is always comparative, and this simple fact raises the question of knowing what is being compared and why. It is hard to conceive of a way of answering that question without a suitable theory explaining why tools are advantageous.

Tools, Not Tool Ideas

The experimental work by Miller [409, 411] and Bowdidge *et al.* [64] provide an excellent study in the contrasts between what might be called “black-box” and “white-box” experimentation on tools. Miller developed a tool for assisting in software inspections. He used a black-box test to measure the difference in performance of tool-assisted inspection as compared to paper and pen. And, despite his tenacious use of sophisticated statistical analysis techniques, his experiment generated weak results (surprise?). After some deliberation, his recommendation was to iteratively design and test the tools. This is a recommendation well echoed by decades of HCI research, and is one practically guaranteed to have some degree of success no matter how poorly the design issues are understood (see Section 7.1.1). But simply testing out guesses is not the best use for experimentation [93]: how did Miller know how to redesign the tool? If we read between the lines in the paper, he used his own background knowledge in combination with the rather informal evaluations of the tools (such as responses to survey questions). It is debatable whether his black-box test results had any appreciable impact on the design process. The point to this example, however is the importance of being able to state and test ideas underlying tools. Miller did find some hints that his second tool is better (in ways) than his first. But despite the apparent soundness of his experiments, they say little about what features inside the black box lead to these differences. What design principles can be extracted?

Bowdidge and Griswold, in contrast, appreciated well the need to open up the black box and understand how the tools were being used. In particular, they knew they needed to observe their program restructuring tools in use in order to determine how (or if) they were good for the user:

[we] were unsure whether the technology in this tool should serve as the basis for similar tools designed to restructure large systems. To effectively use the ideas from this prototype to help develop production-quality restructuring tools, we need to understand how programmers use this tool, and how the organization and features of this tool influence how programmers perform maintenance. [64, pg. 222]

They also appreciated the need to understand the abstract principles of the support provided by the tools, rather than the interfering details of the particular prototype:

The purpose of observing programmers using a variety of tools was not to see which tool set was better. Indeed, our restructuring tools should prove better in certain ways simply because they are specifically designed to ease the task of restructuring, whereas the UNIX tools are not. The restructuring tools are also certainly inferior in other ways because they are prototypes. Rather, we looked at a variety of tools to help us generalize our observations and permit us to make comprehensive improvements to our tools rather than make narrow fixes to the few peculiarities observed in this study. [64, pg. 222-223]

They realized the limitations of black box comparisons for determining the value of their tool ideas. Moreover they realized that opening up the black box was the only way to circumvent the pernicious confounding effects of usability.

Usability is especially problematic for comparison-style research because usability (1) critically affects performance, and (2) is a quality that is inherently “non-linear” with respect to the tool’s functionality. What is meant here by “non-linear” is that small changes to a tool’s features can make dramatic impacts

on the overall usability. Performance is drastically affected in turn. A tool that regularly crashes the user's machine, for instance, is disastrously unusable. Yet the crashing version may be virtually indistinguishable, functionally, from a non-crashing version. True, a crashing program is perhaps just one extreme example. But myriad other misfeatures or omissions can make significant impacts on usability, from the omission of undo, file saving, search, or help facilities, to the colour assignments and command name choices. There are simply many, many more ways of making unusable tools than usable ones (see Section 7.1.1 for an expanded argument). The confounding misfeatures or omissions might be minor design details that do not fundamentally affect the basic idea of the tool, that is, the reasons why the tool is supportive. For example Storey *et al.* [622] found that the lack of a simple textual search function was a significant source of frustration to the users of their software visualization tool. Yet the search functionality was essentially parenthetical to their tool's essential design ideas, which concerned the utility of using fisheye views of software structure graphs. This raises the question: how can one evaluate one's tool ideas without needing to be "perfect" in all the other dimensions of its design?

The impact of usability might not cause problems for comparison experiments if the usability problems could be controlled and accounted for. However that is not usually possible, so experimenters are forced to use highly polished tools if they expect to obtain positive performance results. Note carefully that usability problems are not threats to experimental validity, only to flattering performance results. A clear example of this is illustrated by the SuperBook project [370,371]. As Landauer notes, SuperBook is one of the few hypertext systems that has compared favourably to conventional book technology. Thus it is a technology that could be said to be "validated" in the sense implied by Zelkowitz and Wallace [724]. But the initial version of SuperBook did not fare so well, and it actually took until the third major version before performance improvements were large enough to call the project a success. The first version would *fail* the evaluation criteria the later versions pass. In terms of developing productivity-enhancing tools the initial SuperBook "was a complete failure" [370, pg. 69] since, although it contained gems of ideas, it also contained enough flaws to ensure that using it would reduce productivity. In terms of developing tool *ideas*, however, the design of the initial SuperBook is entirely—perhaps spectacularly—successful. The initial SuperBook describes an application of fisheye views and full-text indexing to information retrieval problems (i.e., knowledge-seeking activity). The final version of SuperBook is substantially like the first SuperBook. Furthermore, if the design of the SuperBook is copied in the future, it is likely that many of the key ideas that define the first version of the SuperBook will be copied. Thus there are really two products to the SuperBook research: the tool, and the tool idea. Comparison-style experimentation validates tools, but seems unable to validate tool ideas except in the special case that they are implemented in a sufficiently polished and improved tool.

Overall, the fact that comparison-style experiments tend to evaluate whole tools rather than tool ideas causes several difficulties for SE researchers. Firstly, it is clear that in many circumstances prototypes cannot be used. This is certainly a hardship for many research projects since iterative development of products to a suitably polished state is costly, and the exclusion of prototypes will drastically limit the number of ideas that can be validated. Secondly, the validation comes too late. One of the main reasons for performing an evaluation is to establish evidence that the tool idea is good enough to spend the time

and effort to develop more polished versions. Thirdly, the stopping criteria for trying to improve a sub-standard tool are ill defined: maybe the basic idea is good, but nobody has yet figured out a way of fixing up the flaws? Finally, the iterative changes to the tool means that attribution of productivity improvement is more or less guesswork without specific hypothesis testing. What features should be credited for enhancing productivity? In the SuperBook example, for instance, who is to say that the enhancements made through iterative improvement did not merely make up for fundamental design flaws in the original version (i.e., maybe fisheye views were bad!⁹)? Without explicitly testing the hypotheses, therefore, attribution of benefit or failure is merely guesswork. Because of this fact, the iterative improvements serve to actually *weaken* the argument that the performance improvement is due to the core design ideas. That is, maybe the productivity gain is due to the subsequent improvements, not the core idea? If one needs to go through several rounds of tool improvement, it weakens the argument of the usefulness of the core idea.¹⁰ If that is the case, why not try to test these core design ideas in the first place? To do so requires testable hypotheses about the value of the tool. Without theories of cognitive support, in many cases it is hard to imagine how to generate such hypotheses without invoking guesswork.

Explanation Avoidance

Empirical demonstration and analysis of claims for usability and effectiveness is vital if interactive technology is to be built on principled grounds, rather than a mutually constructed and self-perpetuating folklore.

– Buckingham Shum & Hammond,

“Argumentation-Based Design Rationale: What Use at What Cost?” [85], pg. 42.

The need to ask simplistic questions in experimental methods reflects the inability or unwillingness to propose and test more explicit models and theories of cognitive support. It does not have to be this way: the same basic experimental methods that can be used to show that yeast raises bread dough can also establish the more specific theory that yeast raises bread dough by fermenting maltose to create carbon dioxide. The former type of experimentation establishes only that an unknown causal mechanism exists, while the latter one describes the mechanism and so produces a proper explanation. Simplistic questions avoid proper explanations—what Lewis called “inner theories” [384]. Two difficulties arising from this fact are that the key ideas underlying the tool’s design are neither articulated nor tested adequately [154]. A third problem arising from this fact is that, without testing a theory, there is little basis for generalizing the results of the experiment past the exact conditions of the experiment. This last point has been argued well in many places and experimental contexts (e.g., Greenberg *et al.* [276], Olson *et al.* [479], Foltz [227], Kirlik [348]). So let us concentrate here on the failure to articulate and test claims.

⁹In the SuperBook example, other observational data may have revealed that the fisheye views were an advantage rather than a liability, but the point is that *other* observational data is needed to establish this—it cannot be extrapolated from the performance data in simple comparative experimentation.

¹⁰To state it sardonically, if the original idea was so darned fantastic, why did the tool need to be fixed up so much?

Clear articulation of the cognitive support provided by a tool normally implies some method of formally expressing the mechanisms and methods by which cognitive support operate. To an adopter of technology, it hardly makes any difference if these support mechanisms are exposed since they are interested merely in whether the technology is “validated” [724]. With such a certification, they can determine whether it is wise to purchase and adopt it [82]. To many SE researchers, however, expressing succinct and explicit claims of the support embodied in a tool is important. They trade in support ideas and want to know, for instance, if their tools support development in essentially novel ways, or if the same ideas have been tried before, but in a slightly different guise. Without clear articulation, these ideas tend to remain at the folk level of description and analysis. The result is that evaluation and comparison tend to be ineffective for understanding the researchers’ ideas since the evaluations tend to focus on either (1) low-level features of the tool, or (2) measurable effect rather than cause.

Comparisons based on low-level features and effects can be widely found. Price *et al.* [516] developed a twelve factor taxonomy of features for comparing software visualization systems. With the exception of the “effectiveness” factor, this taxonomy considered only the features and capabilities of the systems: the graphical vocabulary used, the ability to produce multiple views, whether the system is scriptable, and the like. According to this factor decomposition, there is “effectiveness”, and there are the 11 tool feature types. One would suppose that the other 11 factors would somehow contribute to the “effectiveness” dimension, but it is impossible to determine what the causal influences would be. In a similar way, Kiper *et al.* [346] proposed to compare visual languages along the five dimensions: “visual nature”, “functionality” (meaning the sort of computations specifiable in the language), “ease of comprehension”, “paradigm support”, and “scalability”. Again, “ease of comprehension” is likely to be affected by each of the other four dimensions, but it is not known why. Likewise, one may compare reverse engineering tools by their surface or functional features (e.g., whether 3-dimensional views are generated or not), or their outputs (e.g., the accuracy of software representations generated). Bellay and Gall [42] did the former, and Gannod and Chen performed the latter [233]. Although there exist ways of evaluating these sorts of tools based on cognition-related proto-theories [272, 665], much of the emphasis in the field has been on tool features and measurable effect.

The problem with the above style of evaluation is that it tends to lead to what I like to call “so what?” and “but why?” questions. Visualization A provides multiple views, but visualization B does not. *So what?* Are multiple views vital—a sign of the ineffectiveness of individual views—or simply a distraction [407]? Are 3-D views better? When? Visual language U is demonstrated to be easier to comprehend than language V . *But why?* Is it because of feature difference \mathcal{F} or \mathcal{G} ? There is no doubt that feature comparisons are potentially useful. But without *some* type of theory about what the features are doing for the developer, these comparisons merely serve to chart out the presently known design space [81]. That is the reason why it is important to articulate tool ideas succinctly and then test them directly. *Of course* we have some ideas about why multiple views might be better for understanding software. We trade in such theories. But they are doomed to remain folk unless they are actually spat out and directly tested.

Blackwell termed the way we think about cognition as “meta-cognitive theories” [53]. He surveyed computing science literature on visual languages [53] in order to determine what sort of theoretical resources the field generally uses to argue the advantages of visual programming languages. He observed

widespread preference for simplistic argumentation about the superiority of visual representations over detailed understanding of the factors underlying their possible advantages. These included simple claims for visual languages' improved expressivity, intuitiveness, naturalness, and abstractness. Although some of these claims might well be grounded in psychology or other science disciplines, Blackwell's review strongly indicated that (1) we rarely take the time to actually back up our claims by grounding them in existing science, and that (2) many of these claims are, in fact, not easily justifiable according to existing research. Blackwell's thus argued effectively that as a group our understanding of the possibilities of visual languages is primarily folk and poorly grounded in basic science. Continued attention to simplistic explanations is not likely to change this fact, whether it is for visual programming languages, maintenance tools, or reverse engineering tools. We tend to avoid important explanations.

Investigating simplistic questions also means that the ideas underlying the tool are not directly tested. If we really are interested in expressivity, intuitiveness, or naturalness, then these claims should be put directly to the test instead of picking through the secondary evidence provided by productivity measurements. *Define* what we think naturalness is. Test the supposition. Measure it. Even supposing Elsie does have some ideas for why her tool supported development, her experiment does not test them. The problem is that the user and the user-tool interaction is treated as a "black box" [562]. Her ideas could be entirely off base and yet the tool might be demonstrably better in certain ways for certain tasks. There is a link between black box techniques and the tendency to keep our research discipline a craft discipline. Problems in evaluation are most likely in the case where the researcher's understanding of cognition and cognitive support are naive or folk. For example in, Blackwell's survey he noted that it is frequently believed that graphical representations are more "natural" than sentential representations. It has been suggested, for instance, that graphs of software structure are more readily understood without extensive training and effort [509]. Research on graph-based representations have instead indicated that graph-based representations require years of experience to read properly [506, 509]. Thus some graphical representations may be easy to read due to extended experience and familiarity, rather than due to inherent qualities of the graphical representation. Now imagine a tools researcher explaining their good performance as being a result of the natural superiority of graphical representations. If these specific claims are not being directly tested, then even if empirical evidence proves a performance difference, there is a decent chance that the explanation being held up is in fact wrong. It stands to reason that the likelihood of making valid claims about tool benefits is going to be proportional to the researcher's ability to make specific claims that are firmly grounded in the empirical data of basic psychological sciences. Although simplistic questions are sometimes the easiest to frame, by using them one risks keeping the field in the dark as to the true causes underlying the performance differences being noted.

The preceding analysis outlines many limitations of one of the most common experimental methods used in tools research. The method in question is to first propose a new tool or tool prototype, and then perform a direct comparison of user performance when using the new tool as compared to old ones (typically, only *one* old one). Although a favourable comparison adds fuel for further investigation, it provides virtually *no* measure of assurance that the ideas of the tool developer are justified. It also provides no statement of how the tool idea can be generalized outside the scope of the experiment. Simply put, even

the most carefully performed comparative experiments may be too uninformative if the experimental hypothesis is too simplistic. A hypothesis about the reasons and context for improvement must be directly put to the test. Frequently, the tool designers *have* such a hypothesis (we should probably be surprised if they did not!), but it is latent and often based on folk psychology. Nonetheless it is *never* tested adequately by simple, direct comparisons. Kirlik provides a forceful argument:

Assuming a particular prototype of a design concept is successful, any useful generalizations which emerge from creating the prototype will be at the level of the psychological assumptions underlying the design, rather than at the level of the particular technologies used to implement the design. ... it is incumbent upon the researcher to make explicit the psychological assumptions that contributed to the success of the prototype system. A successful system demonstrates nothing other than its own success, unless the possibly implicit psychological theory underlying the design is articulated. ... the hope for *generalizable* conclusions from such demonstrations [of success] surely rides on whether the researcher can ... identify the psychological hypotheses that were validated by the success of the prototype. [348, pg. 72-73] (emphasis original)

His argument is essentially that it is impossible to avoid the issue of stating and then testing specific hypotheses about psychological aspects of tools, and that simple tool comparisons effectively try to do that. Simplistic validation-style experimentation, however helpful it is for finding out whether one should adopt a given technology, is virtually useless as a method for validating design ideas. Unfortunately, simplistic comparison seems to be so entrenched and valued as an evaluation technique that it may be some time before their limitations are realized. Far from bringing rigour to the area of tool evaluation, they frequently have the opposite effect! They delay the required commitment to articulate the complicated, but necessary theories and hypotheses.

2.2.2 Evaluation Problems: Undirected Observation

Although there are clearly insightful individuals in every profession or occupation who are able to observe human behavior objectively, most specialists in domains other than psychology are unlikely to have the skill or knowledge to relate their observations to observations made in other domains.

– Ruven Brooks, “Comparative Task Analysis:
An Alternative Direction for Human-Computer Interaction Science” [77], pg. 51.

The previous scenario illustrated some of the problems inherent when human-computer interaction was treated as a black box, i.e., with maximal ignorance of what is happening when users sit down with the tool. The following scenario illustrates that opening up the black box introduces its own problems.

Scenario 2: Undirected Observation

Elsie's experiment left her with questions: what were the true causes of the performance improvements that she observed? How can she improve the tool? She notes her ideas about the tool's benefits went untested: was she right? She is now convinced that in order to determine if she was right or wrong, she is going to have to actually watch and understand maintainers and their comprehension activities. She knows she needs to collect and analyze observations about what her users are doing and how the tool helps. She hypothesizes that the tool's analysis functions extract useful information and that the browser's navigation facilities help the comprehenders traverse the code. She hopes to find data that supports these hypotheses. She instructs her subjects to produce a verbal report of their thoughts and collects videotapes and computer logs of several sessions—both with and without the tool. After obtaining these, she has to analyze the data somehow.

She begins by trying to watch the videotapes, pouring through dozens of hours. Many of the hours are uninteresting with respect to her tool's use. Moreover, when her users are the most proficient at using the tool she notices that they do not verbalize tool-related comments. A few ideas pop into her head as she watches. She notices that users switch attention between various aspects of the program when reading the code. She also notices that some users run into difficulties with her browser when trying to return to places they had been to, but no longer remember because of the intervening excursion. After spending time with the videotapes she begins to understand that her developers often stop what they are doing temporarily in order to browse elsewhere, and then try to return to the previous position and continue with what they were doing before. She notices that one user has a strategy of starting a new browser window for such excursions so that he can return to the old excursion by closing the new window. She hits on the idea that when the browser use is fluid, it is because the available browser actions (hypertext links, back button, close window) correspond to actions that bring in the immediately-needed information.

Once again, she seems able to publish. She learned that programmers switch their focus frequently, and that browsers need to support this activity. She can also propose that the way to support such activity is to make sure the browser can provide convenient access to the next-needed information. If only we knew what that next-needed information was at all times! Elsie also recognizes that she has new problems. She wonders if similar things about programmers have been noticed before. She notices she ignored most of the data and concentrated on a few critical incidents. Her analysis was quite informal, and she wonders about the generalizability of her findings.

In Scenario 2 Elsie engaged in what might be described as “unguided” or “undirected” observation because it is performed without explicit guidance from theory or hypothesis. It is essentially an exploratory attempt at understanding tool-user interactions. There are various methods for performing unguided observations, and there can be several levels of formality in the collection and analysis of the data. Unguided exploration ranges from developers simply watching people using their tools, to participant observation in the field by trained researchers, to ritualized and iterative coding and recoding of protocols during theory-building. Sometimes unguided observation is a response to a lack of understanding of the situations of tool use. Scenario 2 contains a portrait of exploratory verbal-protocol based observation. These

techniques have become relatively popular because verbal protocols have shown to be good at discovering new and unexpected facts, especially for the so-called “higher-level” cognitive activities [280]. Developers and researchers of all stripes have made use of observational techniques based in verbal-protocols (see e.g., Nielsen [458], Lang *et al.* [373], Suwa *et al.* [627]). Grudin noted this trend:

Studies of planning and interaction dialogue rely less on controlled experiments measuring time and errors, and more on recording the dialogue and analyzing transcripts. This includes videotaping users’ sessions, asking them to “think aloud,” logging their keystrokes, and engaging in “Wizard of Oz” studies...¹¹ [280, pg. 264]

This sort of unguided observation can be effective at exposing usability problems. Indeed, finding usability problems appears to be a *forté* of verbal protocol studies, for they tend to quickly reveal “usability catastrophes” [458]. Thus, such observation methods are helpful in practical endeavors. Furthermore, some form of exploratory observation is also indispensable for theory building. But, despite being an invaluable technique for both tool and theory building, its use is still arguably problematic for SE researchers such as Elsie. There are three facts that cause problems: little theory is explicitly applied, the discovery-orientation can be mismatched to the researcher’s goals, and the investigative methods being used can be inappropriate.

Theory Missing

Elsie did not directly apply any explicit theories of cognition or HCI, so she was more or less left to her own devices in her analysis. Successful analysis relies on her observational vigilance, her analytic capabilities, and a considerable amount of insight. It clearly would have helped if Elsie was a trained psychologist and could recognize various problem-solving behaviours. As it was, she had to rely on insight and background knowledge. The outcome of such unguided efforts are often highly variable, and dependent upon the observer. Would Elsie’s colleagues have come to the same conclusions? Are there things that Elsie missed which would have been obvious to a cognitive scientist? Furthermore, the analytic power that can be brought to bear in observation analysis depends upon the powers of the theories being wielded. At the very least, a theory can help out by providing a useful coding scheme. Elsie did not code the protocols according to a well-motivated coding scheme for the verbal protocols. Adopting or developing a coding scheme requires, at minimum, some hypothesis that certain actions are important, but she had only a vague idea of these to start with. Iterative coding, clustering, or pattern-matching techniques (e.g., behaviour summaries [542]) can be helpful in cases where coding schemes are not well developed *a priori*, but it is clear that starting with more powerful theoretical resources can kick-start the analytic engine. It seems unreasonable, for instance, to expect that SE researchers should need to devise entirely new coding schemes for each application of verbal protocol analysis [373].

¹¹Computer responses are simulated by humans.

Goal Mismatch

Even if exploratory observational methods are successful, they are often poorly matched to the researchers' goals. Elsie is a SE tools researcher, not a cognitive psychologist or an HCI theoretician. Her goal is to learn about the *tool* not about *cognition*. She wants to *use* theoretical knowledge not *create* it. She publishes in *SE* journals not *cognitive science* journals. Undirected observational methods are antagonistic to these goals since they are essentially learning- and discovery-oriented. This is not *necessarily* bad because Elsie may very well learn valuable things about her subjects and their use of tools. Nevertheless, Elsie's goals and the goals of unguided observation are still mismatched. One potential problem with the mismatch is that there is a reasonable likelihood that, instead of concentrating on important tool-related discoveries, she might rediscover something already well known. Elsie learned about the opportunistic switching behaviour of her users by watching them do it. This knowledge could possibly have been obtained from the literature on software comprehension or basic psychology. One might complain that the problem is that Elsie was not sufficiently well versed in the relevant literature. This complaint merely dances around the problem. *Nobody* can be expected to know all of the potentially relevant literature. But the inherently undirected nature of observational methods combines disastrously with the non-specialist knowledge base of SE researchers: SE researchers are *specifically* in the position of wanting to apply cognitive theories—not to create them.

Elsie's most basic problem, then, is that her activity was directed at *building* the theory she wanted to *use* to help her in her tool research. Data collected was employed to support a conjecture about user behaviour, and about possible tool support techniques. She had to notice the relation between browser state and the next-needed information and, if she was to go any farther in the analysis, she would have to further postulate models of cognitive processes in order to anticipate what sort of information is needed next at different points in the process. Statistical models of browser activity might be generated from her data (e.g., Tauscher *et al.* [633]), but that would still be discovery-oriented: she did not *apply* an existing statistical model. Theory-building is intellectually challenging, and the task of building them is exactly the opposite of her ambition to *use* theories. If she tries to publish her cognitive models and cognitive support theories, she encounters the demanding needs of the basic science researchers (i.e., cognitive psychologists, cognitive scientists, and HCI theoreticians). She would therefore face the burdens of making the supporting evidence scientifically acceptable to the appropriate audience, would need to know the existing literature, and would be exposed to the various controversies and disputes that rage on in fields such as cognitive science. For someone trying to build tools, these burdens are entirely unpalatable.

Wrong Methods

Unguided protocol-based observational methods are also problematic because (1) they fail to identify in advance what to look for, and (2) verbal protocol methods are strongly biased towards discovering usability problems. Elsie's hypotheses about her tool do guide her somewhat, but without theories with which to further focus her observations, she is naturally uncertain about which parts of the protocol are important. She is faced with coding and examining the whole of it. Furthermore, the observational goals are not clearly understood at the start, so it is hard to know when she should stop. Perhaps she has

not yet observed the critical sort of behaviour needed to understand the tool? The research process using unguided observations more closely resembles a “fishing expedition” than a focused search with a specific objective.

In addition to its uncertain nature, it has been observed that undirected observations made using verbal protocol techniques tend to highlight problems (errors, difficulties, etc.) more than successful and smooth performance. Consequently, casual observation is frequently excellent at helping find design problems such as usability blunders [458]. The unfortunate down side to this is that its ability to find problems is equalled by its spectacular weakness for understanding successes. For instance, verbal reports are not normally made for skilled and smooth action [207], yet copious reports tend to be made when trying to cope with usability problems. This means that even unguided observation has a good chance of discovering usability problems because they are conspicuously highlighted. Yet the causes of fluid and successful action is typically lost from view. As Vicente notes [657], the result is that verbal reports tend to clearly expose only what is called “breakdown knowledge” rather than the reasons for success. Yet understanding fluid action (sometimes called “throwness” [710]) seems crucial to understanding—and thus duplicating or designing for—successful work [26]. Simply put, verbal reports are better suited for investigating usability problems than good tool design and usefulness.

It is actually a curiously difficult problem to understand fluid action. The problem is that successful cognitive support removes conscious cognitive effort. Consequently, if one is to have any hope of understanding the cognitive support offered by a tool, one needs to be able to (1) sense the *absence* of problem-solving on the part of the user, or (2) directly appreciate the cognitive advantages of the tool (e.g., noticing a reduction in memory requirements). Suppose a new development tool provides a novel implementation of some type of cognitive support. Its users, when confronted with a problem as difficult as software maintenance, can be expected to use any spare cognitive resources the tool frees up to extend their problem-solving; thus we should always expect to see problem-solving behaviour in observational data—only the nature of the problems should change with cognitive support. Also, users can often be expected to always push improved tools to the point of exposing new limitations. These, once again, will become prominently highlighted by verbal reports. In combination, the continued prominence of problem solving behaviour, and the tendency of verbal protocols to emphasize tool limitations makes it much easier to discover the ever-present problems and limitations than to fully appreciate successes and their causes. This is perhaps not an insurmountable problem, but once again we find a method poorly suited to investigating cognitive support.

2.2.3 Evaluation Problems: Cognitive Modeling

If undirected observation suffers due to the lack of a guiding theory, then perhaps a cognitive model would help? This scenario highlights some potential problems with using many existing cognitive models.

Scenario 3: Cognitive-Model Based Observation

Elsie is unhappy with the results of her previous study. Even though she learned something about the comprehension processes of real developers, she felt that she ran into difficulties because she did not have a good model of how the comprehension processes work. She decides that in order to really “get into their heads,” she needs to adopt some kind of cognitive model, whether it is software comprehension-specific or not. She does some reading and finds the model of von Mayrhauser and Vans (vMV). She tries it out by recoding some of her collected data. The change is dramatic. The model allows her to interpret the many complicated sequences of activity (reading, browser manipulations, etc.) as essentially knowledge search and acquisition processes. Many of the classic features of the vMV model are exposed, including switching between different exploration strategies, and searching for knowledge at different abstraction levels. She reflects on the fact that some of the results of her previous study are not only well anticipated by the model, but that the model exposes additional details—like the difference between top-down and bottom-up strategies.

After a while, Elsie notices that the model has no appropriate coding categories for the actions immediately relevant to her tool. Users of her tool still search for data-dependency knowledge, but they can quickly follow the dependencies through hypertext links rather than follow flow in an editor. In addition, the trick of starting a new browser window with side excursions means the browser also provides support for opportunistic switching between search activities. The vMV model has no possibility of coding such support-related tool use. Elsie decides that in order to trace the support, it is necessary to augment the model—but how? After some thought, she comes up with a notion of “excursion states” so that she can code up the user’s activities with the tool as excursion state representation and manipulation. Starting up a new browser “pushes” the last excursion position on a stack and closing the browser “pops” that value back off. She realizes that the browser is being used as a memory extension during problem solving. She adds this activity to the model and recodes the data with the extension. Users who rely on the trick seem to depend upon being able to forget about the interrupted search and to rely on the state being stored in the browsing environment.

Once again, Elsie feels that she can now publish something since she came up with the insight about how the browser state aided exploration. She refined her ideas about the browser and its support, and she came up with data to support it. Still, she is a little frustrated because she had other hypotheses that eluded her.

Elsie’s activity in Scenario 3 is not nearly as common as informal evaluation, simple comparative experimentation, or unguided observation. In fact, the scenario may well be considered a little farfetched because this sort of theory use seems entirely absent in SE. But the scenario is plausible, and it illustrates the possibilities and limitations of observation using cognitive models (e.g., Storey’s work using the vMV model [618, 620], or Retkowsky’s work [535] using those of Detienne). The suggested procedure is to

start with a cognitive model that can code for mental actions, and then rely on *other* analytic resources to determine tool implications. The problems with this sort of work derive primarily from inadequacies of the available models. Essentially, they do not incorporate a way of explaining how external artifacts aid cognition. As a result, Elsie is left in a similar position as in the prior section; however this time instead of needing to import a model of *cognition*, she needs to import a model of *tool-supported cognition*. The result is that this type of research practice remains oriented towards discovery and theory-building rather than theory application.

Theory Building Again

Elsie wants to use the cognitive model *as-is* in order to derive an understanding or measure of how the comprehension process is aided by her tool. It is reasonable to say that with the model Elsie was able to understand much more about her users' comprehension activities. By starting with a strong model of the internal workings of the comprehension processes, the process of coding and analyzing the protocol is greatly enhanced. But the model was ultimately inadequate for her purposes. She needed some way of generating coding actions for external actions such as browser manipulation. Some way of integrating external and internal states or resources was needed so that the browser history could be incorporated into the analysis of behaviour. Many cognitive models fail to do this. Often they concern themselves only with "internal" cognition and do not following how external states evolve and therefore affect cognition [29, 41, 192, 414]. Furthermore, some way was needed to express the influences of the tool's capabilities on comprehension behaviour. For instance, one tool-dependent strategy is to "stack" excursions by spawning a new browser window for the excursion; such a strategy may emerge through long-term use of such tools, and then only in relatively complicated situations.¹² The strategy works only when the appropriate external memory mechanisms are available.

There do exist interaction models that can account for some of these issues. For instance there are models of tool-user interaction that consider evolving mental and tool states [192, 414]. Elsie could modify the vMV model with similar sorts of model features with an eye towards showing how such external state tracking interacts with high-level cognitive activities such as problem-solving. But, once again, the point is that discovery and theory building activities such as this are not Elsie's first concern. Most likely she would be perfectly happy to be given a modeling framework in which she could input her tool capabilities and then be able to code the user behaviour in ways that expose the tools' support for cognitive processes. Although we have some promising cognitive models, they are not the solution to answering many important tool evaluation problems.

¹²Anecdotal evidence suggests that such a window stacking strategy frequently develops spontaneously over time or is learned, perhaps, from others. A recent posting on the newsgroup `comp.human-factors` says:

With sidetrack [*i.e.*, *excursion*] links I'm much more likely to return to my current location. For example, I use a search engine to find information on a company and find a single press release article on a news site.

With mainstream links I tend to open in the same window, with sidetrack links I tend to open in new windows, view the information then close them. (Gary Bunker, `comp.human-factors`, Jan 12, 2000)

2.2.4 Analysis Problems

Analysis is generally the activity of coming to understand a situation or process, often by modeling it. Analysis activities range from informal musings to the generation of highly formal mechanical or mathematical models. For the present section, the analysis processes of interest are the ones that researchers use to come to understand how tools support (or *might* support) software development processes. Such analysis is often done in conjunction with, or prior to, the design of new software development tools. It is also done when writing research proposals and research papers, and when comparing the designs for different tools. The process of trying to understand observational data (e.g., from a case study) is also called analysis (e.g., verbal protocol analysis [207]). The concern of this chapter is the “other 90%”, so the primary focus is on analyzing the cognitive implications of tools. How do we analyze the cognitive support in software tools? What problems are encountered during analysis of cognitive support? To what extent are these problems caused by the lack of suitable cognitive support theories?

Earlier in this chapter a few analysis problems were already pointed out. In Section 2.2.1 it was noted that it is rare to find adequate articulations of cognitive support claims. Section 2.1.1 argued that much of our theory-based work suffers due to the lack of theories that could account for cognitive support, forcing analysts to rely on craft knowledge and intuition. These facts imply that we rarely are in a good position to analyze cognitive support in tools, compare them adequately, and so on. Rather than repeat these arguments here, attention will be directed to another common analysis problem: *word poverty*.

Green [262] pointed out three problems that plague theories in HCI. Two of these are related to the difficulty of making theories that can be readily applied. The third difficulty is that HCI knowledge is “word-poor”. By this Green meant to imply that researchers and developers must circumlocute when discussing common concepts. That is, instead of using a succinct word for the concept, they must often fall back onto drawn-out argumentation, applying analogies to similar tools, and arguing unproductively about the unimportant details. The following is an illustrative scenario concerning word poverty in discussing cognitive support. Although it is a made-up conversation, it follows other characterizations of real designer and analyst conversations [270,551]. Because it involves a design context, the scenario is also used in the next section when considering tool design problems.

Scenario 4: Communication and Understanding

Olga and Bart are discussing a design idea for an object-oriented (OO) software re-engineering tool they are working on. The conversation turns to some of the possible enhancements that Bart has come up with, specifically to what Bart has called a “method gathering clipboard”, or MGB. He argues that a MGB could be useful in OO re-engineering since a critical task in OO re-engineering is discerning which methods to define in a class hierarchy. Both Bart and Olga understand the MGB’s implementation, but Olga is not sure why Bart wants to add it:

O: It’s a pretty standard clipboard implementation and most windowing environments have them. What’s so special about this clipboard?

B: Well say an engineer decides that some unencapsulated lines of code really correspond to a method. He can select them with a mouse and drag them onto the MGB where they are copied.

- O: *Why doesn't he just make a new method from those lines then and there?*
- B: *Oh he could, but a class may not yet exist for it, and he still has to enter all the other stuff... [the tool has a dialogue for creating new methods]*
- O: *So? He has to do that **sometime**...it just seems like an extra step, especially with copy and paste...*
- B: *Yeah, but it takes a while to enter all the method information, and besides, he may not be sure about the method just yet.*
- O: *But he can just wait until he's sure..*
- B: *True, but he needs to remember the lines and might forget to do it later.*
- O: *True...*
- B: *Also he might decide later that he was wrong about the method, and he would have to undo the method addition. Besides, sometimes it takes a while to find all the lines that belong to a method.*
- O: *So its basically a temporary storage...*
- B: *Right! But it also lets him start collecting related lines together and comparing them with the rest of the code as he browses.*
- O: *Sort of like a very specific display window.*
- B: *Yeah, but the point is that he can modify it also—add more lines, delete lines, move them around, and modify them, perhaps making them more generic so they can be placed higher up in the class hierarchy. He may find several near-clones¹³, for instance.*
- O: *Like a collecting pieces to several small puzzles and rearranging them until they make sense.*
- B: *Sure...I never thought of it that way, but like a temporary puzzle board.*
- O: *And a scratchpad of other ideas in case some puzzles don't pan out.*
- B: *Yeah.*

The actual activity of arguing in detail about a proposed extension to a tool may be important for designers. Working through the argument may bring out new ideas or problems, for instance, or the actual conversation may serve to repair misunderstandings between the designers. But the absence of a good set of concepts and a supportive vocabulary can substantially hinder the process. Green puts it well when he says:

At present each application and its interface must be described in comparative detail, if anything useful is to be said, and each type of problem must be rediscovered anew. Absence of a conceptual vocabulary has well-known effects upon developing knowledge areas, especially in applied science: it prevents discussion about design choices because the repertoire of alternatives is not understood, it inhibits research or limits its application because fundamentally identical problems are not recognised in different contexts, and it constrains the focus of attention to a few aspects of a design when, in fact, the quality of a design depends on a complex tradeoff. [262, pg. 298]

¹³Near clones are highly similar snippets of code. This passage refers to difficulty of determining how to abstract repeated operations, a rather well known problem in reengineering (e.g., Kontogiannis [358], Bowdidge *et al.* [63]).

For cognitive support not only are *interface problems* not recognized in different context, but fundamentally identical *support concepts* may also be unrecognized in different contexts.

The conversation in Scenario 4 illustrates some of these difficulties. Olga and Bart grapple with the cognitive support they think is provided by the clipboard. Here, the correctness of their analysis is not the concern; instead notice (1) how the main topic of the conversation concerns cognition-related aspects of the tool and the reengineering process (method recognition, uncertainty, memory), and (2) the process of reaching an agreement makes use of background knowledge about the psychology of the user, and of the types of support exemplified by predecessor artifacts. On both counts, there are opportunities to improve the conversation with appropriate concepts and vocabulary. In particular, one would like to be able to introduce a widely-usable set of concepts and terms for cognitive aspects of the problem-solving processes of the user [257,262,272] (Chapter 5), and for the cognitive support offered by the tools (Chapter 6). These concepts and terms might have helped Olga and Bart to communicate more succinctly. They could short-circuit reiteration, cascading clarifications, and arguments over unimportant details.

The particular vocabulary and concepts one might introduce is a topic for argumentation elsewhere (see Chapter 7), but a simple example can help emphasize the point by hinting at the possibilities. Bart and Olga eventually came to agree that the clipboard contained certain functionality that supported some of the problem-solving of re-engineers. The support ideas included (1) the fact that the clipboard acted as a type of external memory so that details would not need to be remembered, (2) that this external memory was being used as a cost-effective place to temporarily hold uncertain and changeable parts of the problem solution, and (3) that the juxtaposition of problem parts within the clipboard may play a role in recognizing problem solutions. These possibilities all suggest supportive roles for the program features and we might for the moment agree to call these “external working memory” used to store an “externalized problem-solving heap”¹⁴ in order to take advantage of a “localization effect” that makes solution recognition more efficient.

The conversation might have gone much better if both Bart and Olga were familiar with the above concepts. They might not have had to appeal to analogies to other tools (display window, scratchpad). They had to come to agree about the value of the memory function of the clipboard and how its necessity relates to the tentative and backtracking nature of method recognition. If this was quickly acknowledged by describing it as an “externalized problem-solving heap”, attention might have turned instead to the tradeoffs between external and internal memory, or the relationship between problem-solving capabilities and working memory sizes. For example, such a characterization might have started a discussion of how an expanded problem-solving memory could make the user’s search for methods a more breadth-first, rather than depth-first. Designers are known to have a bias (e.g., Stacey *et al.* [615]) to prematurely focus on exploring sections of the solution space in depth (depth first) instead of exploring alternatives first (breadth first). Olga and Bart might therefore have effectively argued whether this external memory could help reduce this bias by providing an effectively expanded heap memory, therefore making breadth-first

¹⁴A heap is used for so-called “breadth-first” searches of problem spaces and is used to store a pool of unvisited nodes. Breadth-first oriented problem solving is frequently advantageous over “depth-first” work because one is less likely to be stuck searching deeply in unpromising portions of the solution space. However, excessive heap memory use is a notorious problem of breadth-first searches.

exploration cheaper.

Of course, these are just illustrative terms but the purpose of the scenario is to entice us to imagine the possibilities. The point is clear: a good vocabulary of cognitive support terms and concepts can help analysis. Without such a vocabulary, conversations about cognitive support are often muddled by oblique references to psychological effects, and by references to prior tools.

2.2.5 Design Problems

What is software engineering research? We have known for 25 years that our programming methods are inadequate for large projects. Research in software engineering, programming methodology, software design, etc., looks for better tools and methods. The common thrust of results in these fields is to reduce the amount that a programmer must remember when checking and changing a program.

– David Lorge Parnas,
“Software Aspects of Strategic Defense Systems” [487], pg. 1330.

Design is a *synthetic* activity. Designers of software development tools wish to understand the problems and difficulties of software development and then be able to come up with—synthesize—tools that support such development. What problems do researchers encounter when considering the design of software development tools? Clearly the list of potential design issues is enormously long, but in this section the concern is with the way that tool builders reason about how to assist thinking. Even if one does not entirely agree with the above summary by Parnas, his essential point must be heeded: to do good SE tools research often means to reduce cognitive challenges and burdens for software developers. Put another way, to assure quality in software development tools it is necessary to provide cognitive support. So the questions for this section become “how do SE and CS researchers engineer cognitive support?” and “what problems are encountered in this regard?”.

Several relevant points have already been covered regarding these questions. It should be reiterated that cognitive issues are frequently treated using craft knowledge and methods (Section 2.1.1). This generally means that they rely on the designer’s intuition, the reuse of existing successful designs, and the application of folk knowledge about psychology, sociology, or other relevant domains. The point of this section is to argue that keeping the practice craft-like is not desirable. What harm to design does not having an explicit cognitive support theory impart? There are actually two relevant aspects to this question. One aspect concerns design practices—how researchers build cognitive support—and the other concerns design theory—how researchers understand cognitive support. Some current problems with the latter were already covered in Section 2.1.1: when cognition is studied in software development, researchers are often forced to fall back upon craft-based design knowledge. Thus here I will concentrate on the former problem, that is, how tools researchers and developers from SE and CS think about and build cognitive support into tools.

In order to make a lucid argument, some concepts from design theory must be first introduced. In Chapter 7, these are defined in more detail, but for now a short introduction will suffice. Two of the key problems in design are knowing what is needed, and knowing how to design something to satisfy this need. These are the problems of *setting design goals* and then performing *design reasoning and synthesis*. For instance if it is known that software comprehension is cognitively challenging, one might adopt a design goal of reducing memory load, and then reason that one way of doing so is to add an external memory for browsed locations so that these locations need not be remembered (see e.g., von Mayrhauser *et al.* [682] or Singer *et al.* [596]). For the problem of setting design goals, there exist the beginnings of promising new theory-based research. A good example is the hierarchy of cognition-related design issues developed by Storey *et al.* [619]. Most of the design issues in that work immediately suggest design goals, but only a limited set of examples solutions are provided. Even so, SE researchers have shown that generally they are good at generalizing from examples, so collecting together design goals and prior solutions is a good step forward. However the second problem mentioned above has not yet been addressed as well, that is, the problem of generating design ideas given a design goal. This problem can be called the “gulf of synthesis” (see Section 7.1.2) because the designer must synthesize solutions from high-level goals.

In order to bridge this gulf of synthesis using theory, one must be able to *reason forward* about designs. Reasoning forward means that the designer is able to reason about design ideas or options when given a design goal and an applicable theory. Ideally, the theory would provide more than vague recommendations, and do more than merely validate design goals adopted through other means [93, 522, 605]. As pointed out before, most of our software comprehension models provide no account for how tools affect comprehension, so any design recommendations made by the model must be based on experience and common sense. Unfortunately that is currently by and large the state of theory in its application to design. Except for a few scattered results, most of the design advice has been quite vague and mostly derived from common sense or experience.

2.2.6 Summary of Problems

Some may argue that HCI does not need theory. I disagree. Any discipline that fails to make a principled explanation to justify its practice is building on sand.

– Alistair Sutcliffe, “On the Effective Use and Reuse of HCI Knowledge” [625], pg. 199.

Understanding software development tools seems to inevitably require understanding how tools affect and participate in human thinking and problem solving. Unfortunately we mobilize insufficient theoretical knowledge to tackle the issue of cognitive support. This basic fact, I suggested, is the root of many problems in tools research in SE and CS: we have a desire to address cognitive implications of our tools but we are unable to do so properly due to our under-developed theoretical research thread concerning cognitive support. To support this supposition, existing research problems were reviewed, and then each of these were traced to the lack of a theory-based research thread. These problems were divided into evaluation, analysis, and design problems.

Four evaluation techniques were reviewed: informal evaluation, simple comparative evaluation, unguided observation, and cognitive model-guided observation. Problems in evaluation were traced to the avoidance or absence of sufficient explanation and understanding of cognitive support. Moreover, when trying to back supposition up with empirical data, it is often the case that tools researchers are forced to essentially generate new theories of cognitive support. This fact also was a cause for several concerns, especially regarding how these efforts do not match the goals of tool builders. In hindsight, it might seem logical that when *evaluating* existing cognitive support ideas, it is not appropriate to discover new cognitive support concepts, or to build new cognitive support models. Explicit models of cognitive support are key input ingredients to evaluations.

Analysis and design of tools for cognitive work takes place partly in the abstract plane of cognitive support ideas. Our current state of practice was reviewed and shown to provide few ways of accessing this plane, and few ways of reasoning within it. One hindrance to access, it was argued, is the lack of a useful set of concepts and an appropriate vocabulary. Because of this absence, analysis and design tends to take place in the concrete plane of specific tools—it gets bogged down in details and is foiled by the indirectness of the designer’s conversation. Finally, the current state of theory was shown inadequate for design. It may be able to model and explain some parts of software comprehension, but it lacks the crucial ability to enable forward reasoning about how to support cognition.

2.3 Possibilities of Theory-Based Research

...it is likely that you have purchased a variety of artifacts, such as calendars, calculators, or computers, that were created primarily to support cognitive activities... The vast majority of these cognitive artifacts were designed not by the application of cognitive theory but instead by appeal to folk-psychological intuitions, trial and error, and the forces of the marketplace. Only recently have researchers begun to consider how theories and findings from cognitive science can be systematically applied to the design of the cognitive environment.

– Alex Kirlik, “Everyday Life Environments” [350], pg. 702.

Suppose we agree that cognitive support is an important issue for tools researchers, that it has been under-appreciated in the past, and that a lack of a suitable theoretical basis underscores many of the problems that SE researchers face in current practices. What alternatives are there to the status quo? What should theories of cognitive support look like? Who should be building them? How might SE research be changed if they became available? This section address these questions by presenting one vision of the possibilities of theory-based approaches to cognitive support. In this vision, theories and models of cognitive support are a foundation for research in designing new and better forms of cognitive support. SE researchers will be portrayed as consumers of theoretical advances from other disciplines, the focus of the SE thread of research will be set on applications of theories derived elsewhere, and progress on scientifically understanding cognitive support will be portrayed as a multi-disciplinary collaboration

mediated through theories exchanged between disciplines.

It is nearly impossible to convey any sense of a vision without an analogous success story to use as an intuition pump, and from which it is possible to draw an understanding of the reasons for the success. Two analogies are used here: one based on mechanical support to provide a vision of what theory-based methods of cognitive support research might become, and one based on medical research to paint a picture of how SE research on cognitive support relates to other research disciplines such as cognitive science.

The first analogy, based on mechanical support, is presented in Section 2.3.1. From it I will draw attention to the features of modern mechanical support knowledge that appear to be vital for successful explanation of mechanical support. By analyzing the successful case of mechanical support, we may develop a set of desiderata for new support theories, or, more strongly, a set of *requirements*. Later chapters will use this requirements analysis to guide theory development and to show that the resulting theories meet the anticipated requirements. For now, the mechanical support analogy will be used to argue that a theoretical understanding of cognitive support enables a theory-based approach to tool design and development. Such a theory-based approach could avoid many of the problems that were found in the survey of SE tools research practices. The mechanical support analogy also makes it possible to contrast a theory-based approach to existing practices. It also give a better indication of the potential advantages of theory-based methods.

The second analogy, based on medical research, is used to position SE research on cognitive support within the greater research milieu. SE tool researchers are compared in Section 2.3.3 to pharmaceutical researchers. The comparison is important to SE researchers because it limits the contributions that should be expected of SE researchers, and it provides a focus for future cognitive support research *and* SE research.

2.3.1 Leveraging Mechanical Support Theory

Its author said that machines were to be regarded as a part of man's own physical nature, being really nothing but extra-corporeal limbs. ... Observe a man digging with a spade; his right forearm has become artificially lengthened, and his hand has become a joint. The handle of the spade is like the knob at the end of the humerus; the shaft is the additional bone, and the oblong iron plate is the new form of the hand which enables its possessor to disturb the earth in a way to which his original hand was unequal.

– Samuel Butler, “Erewhon” [88], pg. 202.

Some of the earliest thinkers in HCI, such as Bush [87] and Engelbart [201], believed that computers (electro-mechanical machines in Bush’s case) could augment human thinking capabilities much in the same way that a shovel can augment innate digging capabilities [65, 124, 537]. The main difference is that shovels are for *physical* labour, whereas computers were envisioned to assist in *thinking* labour. Both physical labour and the advantages of mechanical devices have been studied for a long time. They are by now quite well understood. It might be possible to carefully examine how physical support is explained in order to appreciate the possibility of explaining cognitive support better. Here one of the simplest

physical supports is examined: a lever.

A Lever Story

Chris and Eric are hiking in the woods. Eric accidentally drops his car keys and they wind up in a hollow beneath a large fallen tree. After trying in vain to reach them, he and Chris try to move the tree by pushing it, but it is too heavy. They discuss various options for retrieving the keys and decide to try to use a lever to lift it enough for Chris to reach the keys. Eric finds a rock, which he moves beside the fallen tree, and then places on top of it a medium-sized branch that Chris found nearby. Chris presses down on one side of the branch. On the other side, the heavy tree is lifted enough so that Eric can retrieve his keys.

Although this example is simple, it serves to illustrate a number of important strengths and properties of our modern understanding of physical support. The first and foremost thing to note is that we can form an *explanation* of the physical support in question. In particular, the explanation exposes a detailed causal chain that expresses the mechanisms underlying the events. Such an explanation might go as follows:

1. The physical arrangement of the branch and boulder forms a *lever*.
2. The lever is supportive because it transforms Chris' downward force into a larger upward force, enabling her to lift the fallen tree. This force-amplification effect is called *leverage*. It is a form of *mechanical advantage*. It allows Chris to lift a tree larger than she can lift unaided.
3. The leverage arises because Chris' downward force is transmitted by the rigid branch onto the fulcrum, and since it does not move, the downward force is converted to torsional force which is in turn transmitted to the tree through the other end of the lever. Because the lever is longer on Chris' end, her downward force is exerted over a longer distance than on the tree's end. Since physical labour (work) is conserved, and the work is proportional to the force times the distance, the force on the tree's end is greater than on Chris' end.

Further details of such explanations are not important for this vision section. What is important is the sorts of lessons that can be drawn from the example, especially lessons for future cognitive support theories. A short summary of these appears in the left hand column of Table 2.4.

Lessons

MECHANISTIC EXPLANATION

The explanation of physical support is given a mechanistic explanation. No important mysteries are left about the powers of either Chris or the lever she uses. Nobody talks about amazing but mysterious properties of a lever, about it being more "natural", or about it evoking powerful and innate lifting abilities of Chris. Moreover, the explanation is not purely descriptive, as it would be if one had claimed that "the lever is pushed down, supporting the lifting of the tree," or that "levers mediate physical work." Neither is it tautological, as it would be if one claimed that "the lever helps lifting by supporting the up-pushing task."

- ▷ Cognitive support must be given “deep” explanations using mechanistic models. Most current explanations of cognitive support are extremely “shallow.” For instance a tool might be said to support comprehension because it offers search capabilities or browsing of program-dependency links. Such claims are not proper explanations.

SUPPORT VS. AUTOMATION

The lever did not *eliminate* work but it changed its characteristics. If Chris and Eric were hiking with a powerful robot, there might have been no need for a lever and no (physical) work would have had to be done by Chris. The robot’s power supply could have provided all the motivating force needed to move the tree. As it is, Chris needs to do the work, but it is supported.

- ▷ Cognitive support models need to explain how cognition may be assisted yet not completely automated [171]. To date, many explanations have been too shallow, often relying on arguing automation or referring vaguely to powers of the human mind (e.g., the power of visual systems) [53].

FUNCTIONAL, HIGH-LEVEL ONTOLOGY (RIGHT LEVEL)

The concept of leverage is a high-level one based on *functional relationships* between *high-level interpretations* of the basic components. Specifically, the low-level details are ignored: it matters little what the branch and boulder are made of. Subatomic physics, for instance, just do not enter into the discussion even if, ultimately, the action is reducible to such laws. A boulder is thought of as a “fulcrum” and a branch as a “pivot”. The abstraction is important because the same concepts could be used for many other configurations (e.g., a crowbar or a mechanic’s floor jack). The concepts used in the high-level interpretations comprise an “ontology”¹⁵: a list of the sorts of entities (fulcrums, pivots, etc.) and their relationships used in defining the elements of reality [653]. This ontology is a *function-related* one that is *relative* to the use and not “absolute”. A boulder is only a fulcrum if it is being used to form a lever.

Notice also that the words used in this ontology form the sort of vocabulary that is necessary to reason about mechanical support. Words like “leverage” are not only succinct, but they are defined in terms of mechanical models of work, and so they evoke other well-understood implications (e.g., the tradeoff between amount of mechanical advantage and the extra distance that Chris has to push).

For most purposes, the explanation based on leverage is at the “right level”. The analyst happily glosses over complications such as the fact that the branch is discontinuous (it is made up of tiny particles), and the fact that the “force” involved might be explained by more “fundamental” models (e.g., interaction of subatomic particles). At the same time, the mechanical explanation is important: many times it would not be sufficient to cut off the analysis at the point of noting that the lever is a force-transforming machine. In particular, if one is to build effective levers, one must understand the features of artifacts that can make good levers (tensile strength, smoothness, etc.).

¹⁵Philosophers of science will probably cringe at this use of term “ontology”, much as they cringe at the liberal use of the term “knowledge” in AI. In this work I follow the lead of knowledge engineering and use “ontology” roughly like “schema” [281] and “knowledge” roughly as “belief”. For the benefit of the purist, however, it *is* possible to employ the philosopher’s notion of “ontology” by proposing that the task-relevant properties (e.g., whether the rock is a fulcrum) are realist facts (see e.g., Dennett [176]), as is done in Gibsonian psychology [221, pg. 9].

- ▷ Cognitive support theories must be constructed at the appropriate level, using functional terminology and cognition-related interpretations. It is critical that this ontology be able to abstract away from implementation details, as the conception of a lever does. One possibility is to use “knowledge level” [445, 446] (or “cognition level” [523]) concepts and terminology, since they relate directly to cognition issues. They also abstract away implementation issues, that is, the uninteresting lower-level details. Then cognition-related aspects and relationships could be highlighted, and implementation-related ones deemphasized (or elided).

TRANSFORMATION AND WORK COMPARISON

With physical work there is a strong notion of conservation of labour. Of course, in the scenario a different solution path might have been found—for instance Chris could have picked up a small stick and fished out the keys with it, thereby avoiding the lifting of the tree. But assuming the tree needs to be lifted, a certain amount of work is unavoidable: the work needed to lift the tree the required distance. Conservation of energy laws state this is unavoidable. Using mathematical models of work, it is possible to compare the assisted and unassisted work; we can compare Chris’ lifting with the lever with what she would have to do (her power requirements) if she did not have a lever. In this comparison, it is the lever that changes the nature of the work that Chris does. True, she needs to push with less force than if she had no lever, but also she now pushes downwards on the branch rather than lifting up on the tree. The transformation makes strictly harder jobs possible, but at some cost (making the lever, pushing it a greater distance). It is important to remember the advantages in order to accept what may be unavoidable overheads.

- ▷ Cognitive support models need to explain how tools transform the tasks [470] to improve the cognitive ergonomics. Some method of comparing work done while using different tools must be generated [77]. Ways of establishing work equivalence seem important to making fair evaluations or comparisons of tools. Also required are ways of distinguishing between the main work being done from the overheads created.

USABILITY AND TESTING

Once a mechanical explanation of leverage is available, it can be studied with even the poorest of implementations. A lever made out of a pipe-cleaner and a wad of gum can be tested in a variety of ways, such as establishing the relationship between lever length, tensile strength, and possible load limits. As a working lever, however, the implementation may be completely unusable.

- ▷ As much as is possible, cognitive support modeling and evaluation should be independent of other suitability issues: it should be possible to evaluate the supportive nature of prototype tools (i.e., in the presence of usability problems).

TOOLS VS. SUPPORT CONCEPTS

Since leverage can be modeled and detected, once it is clear that leverage is needed, then a lever-builder’s challenge is restricted to showing that the lever is effective and useful for the tasks it is intended to be applied to. In more complicated tools, where levers are merely parts of the overall tool, the value of the lever can be established independent of the other functionality.

- ▷ Cognitive support models need to make it possible to separate the appropriateness of the tool's design from the correctness of the designer's reasoning about cognitive support. That is, support is independent of the overall design's rationalization.

TOOL DESIGN

Chris and Eric adapt the materials available to make a new tool. As tool designers, they require some knowledge of the task they are to perform and the materials available. But the key is to understand the implementation-independent concept of a lever and to be able to reason about how to build one (using available materials) to solve their problems.

- ▷ Cognitive support models should make it possible to reason forward about design from an analysis of the cognitive work that needs to be done. Cognitive support theories do not address task or user requirements, or the various implementations of supportive solutions.

UNIFIED EXPLANATORY MODELS & COMPOSITION FRAMEWORKS

Newtonian mechanics creates a framework within which it is possible to describe and explain many fundamental principles of mechanical advantage. Common simple machines such as levers, inclined planes and pulleys can be described as basic elements that can be combined in many ways to generate more complicated machines.

- ▷ Cognitive support theories would be most beneficial if most (or all) concepts of support could be decomposed into a collection of elemental support types. More importantly, the total collection of support types should be framed within a theory that can coherently unify them.

THEORY USE, NOT THEORY BUILDING

Once the principles of leverage are known, they can be applied with relative ease to understand the basic events in a given situation. If someone knowledgeable about such things were to have observed Chris and Eric, they would have little trouble recognizing and explaining the supportive nature of the lever. Because such an analysis rests on established concepts and theories, reasonable explanations are unlikely to be questioned. Even if they are, the correct response is to refer the skeptic to the science publications upon which they are based, and let the experimental data gathered elsewhere settle the case. It would be quite unrealistic to suppose that Newtonian physics would need to be first created and defended by such an observer. Only in highly exceptional design situations are new theories actually required to make progress in explaining phenomena (e.g. building transistors for the first time [110]). Good theoretical resources ensure that these situations are the exception rather than the rule.

- ▷ Cognitive support theories should strive to minimize the amount of specialist knowledge needed to that required for theory application. In particular, cognitive support theories should strive towards the state in which tool developers can almost casually recognize and discuss supportive mechanisms (like external memories) much as they might talk about a lever. Certainly, theory users should never need to *do* cognitive science. Proposing good cognitive support explanations may never be as simple as it is for mechanical support (much of cognition is not easily observable), but the importance of an established theory is still clearly analogous.

DESIGN INTENTIONALITY AND AD HOC TOOL BUILDING

A tool can be supportive even if it is not designed that way. The boulder and branch were not designed for the purpose of lifting. The support is related to the uses to which the artifacts are put.

- ▷ Cognitive support may be created in an *ad hoc* manner. It must be possible to generate explanations of cognitive support from observed or prospective user behaviour, not merely from designed-in mechanisms.

These lessons from the lever scenario greatly influence the remaining chapters. For reference, Table 2.4 lists where these desiderata are addressed in succeeding chapters.

LESSON REGARDING THEORY REQUIREMENTS	SECTION(S)
explanations of benefit must be mechanistic	4.1.2
explanations cover partial automation and cognitive assistance	6.2.2
ontology should be cognitivist (abstract, cognition related)	4.1.3
work equivalence with/without tool should be established	4.2.2, 5.5
definition of support should be independent from usability	6.7
definition of support should be independent from other functionality	6.7
forward reasoning about support should be possible / simple	7.1
good theories will have a small “vocabulary” of unifying support concepts	4.2, 6.5
need for deep psychology / cognitive science knowledge is minimal	7.2
theory should be applicable to <i>ad hoc</i> designs	—

Table 2.4: Theory requirements (lessons learned) and where they are addressed

2.3.2 What Might Theory-Based Methods Look Like?

The theory gives the answers, not the theorist. That is important, because humans also embody knowledge and can answer questions. ... What questions can be answered by a theory may well depend on the skill of the theorist, but the answer itself, if obtained, does not—it depends on the theory, not the theorist.

– Allen Newell, “Unified Theories of Cognition” [446], pg. 13-14.

In the previous section the example of a lever was used to drive an analysis of important features of theories and models of mechanical support. The present section asks the question: if theoretical models of cognitive support were to become as well-elaborated and, indeed, as mundane as those for mechanical support, what might SE tools research look like? The question is answered in two parts. First, a prospective scenario is suggested of how established cognitive support theories might be used in the future to perform theory-based tools research. This is a technique used previously by Card *et al.* in their landmark book “The Psychology of Human–Computer Interaction” [94]. In it, they tried to establish a vision of

how an applied psychology might be used during tool analysis. Second, after each scenario is presented, the problems of current practices will be revisited in order to generate suggestions of how theory-based methods might either avoid or solve them. These comparisons to theory-based methods are summarized in two tables: Table 2.5 for evaluation activities, and Table 2.6 for design and analysis activities. In order to make the example realistic enough, some genuine support ideas from the literature will be included in the scenario. For now, the reader is encouraged to ignore these details and concentrate instead on the general form of the scenario.

PRACTICES	PROBLEM SOURCES	THEORY-BASED APPROACH	ADVANTAGES
informal evaluation	informal methods and explanations	increased formality	results are more convincing
simple comparative validation	simplistic questions	explanation building	understanding cumulates
		explanation testing	validation is stronger
	whole-tool evaluation	specific feature evaluation	prototypes may be used feedback is directed feedback is early
(black-box methods)	explanation avoidance	explanation seeking	support idea is explicit ideas are directly tested
undirected observation	minimal theory is applied	support-relevant theories are imported	better coding schemes reduced proof burden
	mismatched goals (theory building)	matched goals (theory application)	learning is about tools, not psych/support support theory is imported, not argued/proved
	methods are inappropriate	experimentation is directed support mechanism exposed	avoids fishing expedition usability problems can be ignored
model-based observation	model weaknesses	support-capable models	mechanisms testable
	mismatched goals	matched goals	[see undirected obs. above]

Table 2.5: Comparing problems of current idea evaluation practices to theory-based research

PRACTICES	PROBLEM SOURCES	THEORY-BASED APPROACH	ADVANTAGES
analysis & design	word poverty	cognitivist vocabulary	support-related concepts
			abstraction over details
design	relies on craft knowledge	theory grounded design reasoning	better rationales
	backwards-only reasoning	forwards reasoning enabled	help designers cross the "gulf of synthesis"

Table 2.6: Matching problems of current design/analysis practices to theory-based research solutions

Scenario 7a: Design

Lorne is developing a new software reverse engineering tool that is intended to help reverse engineer design patterns from legacy code. He has a tool that contains a knowledge base of design patterns. He knows that these design patterns can be partially matched by an analysis engine he developed with his colleagues. The partiality of the matches is due to the fact that the analysis engine cannot make certain inferences about the correctness of matches, and partly because the patterns in the legacy code are subject to occasional violations of design rules. A small part of the overall HCI consists of the reverse engineer trying to complete partial matches of design patterns and finding and accounting for all pattern violations. The process of making these matches is a hard one—it is cognitively challenging to reverse engineers. Lorne wants to build in cognitive support for this process.

Lorne starts by thinking about the opportunistic problem-solving methods typical of reverse engineers. On his white-board he begins sketching the engineer and computer and the resources they have. He knows the computer can compute a partial match of a design pattern to a piece of code that implements it. Each match binds pattern features to code features, but these bindings may be erroneous, or may fail due to pattern violations in the code base. Lorne realizes that the engineer's task is to go through the matches and determine correct bindings to the legacy base, and also find pattern violations and determine how to possibly correct the legacy base. The tool essentially "kick starts" this process.

Based on his knowledge of tool-assisted problem-solving he realizes that the list of bindings the tool generates forms a shared "match completion plan" and the places where the pattern is violated is a shared "repair plan". That is, the tool and user act together in a distributed problem-solving manner, with the tool's initial work serving to develop a partial match, and to generate an initial plan for the user to establish a good binding to code. The plans provide a series of hints regarding the likely locations of good and bad bindings, and where pattern inconsistencies may exist.

Using his knowledge of cognitive support, Lorne realizes that a suitable support is an external planning environment that can ensure that the plan and its execution state are well-represented externally. He knows that if he can effectively distribute the execution states between user and tool, then the user can engage in display-based¹⁶ problem solving instead of internal planning-based problem solving. From cognitive support theories, he knows that this would free up cognitive resources, resulting in more systematic plan following, and it would make the process more tolerant of interruptions.

Armed with a general idea of what support is needed, Lorne's design task is to generate interface ideas to carry out the support. He begins sketching an interface that can (1) hold the (known) match completion plan steps and (2) accumulate the (unknown) repair plan. Support theories kick in to advise him to pay attention to several types of shared problem solving states. Based on these warnings he adds ways of tracking plan completion state and visually indicating action options relating to the problem goals. He then adds features to unroll decisions, and to enable backtracking. He also adds a heuristics-based prioritization algorithm to rank the bindings list so that the ones with the most confidence are listed first. After making many more detailed design decisions (e.g., involving how novel to make the

¹⁶A type of problem solving characterized by the use of external artifacts to represent problem-solving state and to cue future actions (e.g., see Larkin [374], Section 6.5.2).

proposed interface), he has a design for an initial prototype.

Lorne used a theory-guided approach to design reasoning. He analyzed the reverse engineering task in terms of problem-solving and reasoning that needed to be done. He then made use of cognitive support concepts to reason about how to distribute the problem solving process between the tool and user. In particular he began to reason about how to distribute the knowledge within the system (plan and state) in order to enable a particular form of problem-solving (display-based). Prior high-level ideas about cognitive support helped him make the transition from analysis to synthesis. The transition is critical and difficult; the potential for theoretical resources to assist designers in this regard is thus a powerful motivator for the pursuit of cognitive support theories.

Lorne was further aided by using an appropriate cognition-related ontology to analyze the potential benefits of the tools. The vocabulary and concepts concerning cognitive support were readily available to Lorne. He could abstract away from the low-level details (e.g., windows, list boxes) and think in cognition-related terms (e.g., planning, plan execution states). The ability to reason at such functional and high levels is absolutely critical. This ability is one of the most pervasive and well known differences between expert and novice performance (e.g., see Ormerod [481] or Chi *et al.* [123] for reviews). Experts tend to think at abstract, function-related levels rather than at lower implementation-related levels. Cognitive support related knowledge can act like a lightning rod for attention. For Lorne, the support-related concepts such as display-based problem solving may have helped bring to the fore relevant issues such as cost tradeoffs involved with externally-represented problem solving state. Lorne may well have been able to design a similar tool without theoretical backing (see, for instance, Koschke [359, ch. 9] for some similar ideas), however without being able to explicitly reason about why the tool supports cognition, Lorne may never think to empirically test some of these implications of his tool. But as can be seen, Lorne has a number of support claims that might be put to empirical test. For instance, he could test whether or not display-based problem solving is being performed, or whether the external representation plan execution state really offloads memory requirements. Without articulating the theoretical claim he might have been tempted to pursue only black-box methods. Thus theories are important to the analysis, evaluation, and redesign cycle.

Scenario 7b: Early Evaluation

Lorne wonders if the support mechanisms he proposed in his prototype are any good. His current concern is that maintaining an externalized plan-completion state is perhaps too costly. From the standard textbook on cognitive support, he learns that there is a multi-way tradeoff between externalized plan-completion state, the cost of manipulating the state, and resistance to interference effects (among other things, like user confidence in the correctness of the job, and the ability to support cooperative work by multiple users). He briefly thinks of trying to use a GOMS analysis¹⁷ to calculate state-manipulation costs, but decides to wait until later, thinking that state-manipulation costs may have only a minority effect. For now, he thinks the more important issue is whether the prioritization algorithm in the repair plan display destroys breadth-first problem space exploration. He thinks that users may adopt a more

¹⁷A task analysis applied to help determine the efficiency of low-level interface tasks [94].

focused, depth-first behaviour where they chase unpromising leads for too long before realizing their errors.

Lorne suspects there is no easy way to test his supposition, but he feels he might get reasonable initial feedback from the developers using his prototype. Lorne visits one of his test sites and videotapes developers using the tool. Back in the lab, he edits the videotapes to get rid of all observations except for cases where the step sorting is used. He watches as the users show a habit of always looking first at the highest priority binding. He might not know whether that strategy is good in all cases, but at least now he knows the prioritization does focus the search to options his algorithm considers most important.

The above sort of theory-based evaluation is significantly different in character than all of the non-theoretical methods surveyed in the last section. Unlike those other methods, it is highly focused on the specific supportive ideas designed into the tools. Detailed suppositions of the supportive nature are articulated. These details can generate a cascade of specific evaluation questions. Empirical work serves to directly validate or disconfirm Lorne's suppositions about the support rather than build a theory of the support or the cognitive processes involved. Data collection and observation are strongly directed. When Lorne analyzes the videotape he knows what sort of behaviour he is looking for. He also could use a model-based coding technique to code problem state transmission between internal and external representations (see Chapter 9); this would then help him evaluate the success of the display-based interface.

If Lorne wanted to, he could directly test some of the predicted performance benefits even with his prototype. For example, he could try adding various memory loads (like articulatory suppression [167]) to subjects to determine how their ability to track progress is affected. This allows Lorne to propose and evaluate interesting questions early in the iterative design phase, when such empiricism has the best chance of making the greatest impact. By being focusing on the cognitive support, he is able to appropriately ignore many usability issues (e.g., proper help systems, printing, interface consistency) during the time when changes to functionality are most critical. While he is designing and evolving new support, he is analyzing and modeling that support at the same time. His understanding of the tool's benefits therefore accumulates in the models he uses, and thus these directly reflect his design ideas. When it comes time to publish, the claims of support can be made crystal clear, and being able to appeal to standard theories of cognitive support grounds his work in a widely accepted science base.

These two scenarios of design and evaluation lead to a vision of theory-guided research where researchers can work directly in the domain of cognitive support. They can fully articulate their ideas and by doing so engage in a tight feedback loop of proposing support implementation concepts, directly testing them, and then modifying their designs. Cognitive models and theories act as strong allies by helping them do their intended work—developing computational implementations of support—rather than exposing them to the full burdens of building theories of cognition and of cognitive support.

2.3.3 SE Research and Researchers

Engineering relies on codifying scientific knowledge about a technological problem domain in a form that is directly useful to the practitioner, thereby providing answers for questions that commonly occur in practice. Engineers of ordinary talent can then apply this knowledge to solve problems far faster than they otherwise could.

– Mary Shaw, “Prospects for an Engineering Discipline of Software” [579], pg. 16

What does research on cognitive support for software development entail? What is the role of SE researchers? I have already portrayed SE researchers as future *users* of cognitive support theories. These theories are not going to magically appear—they must be developed. Who develops them? How are they evolved? It is unreasonable to claim to be able to predict answers to these questions, but by looking into the past history of another domain it might be possible to glimpse the future history of our own. Here, I take a cue from Lewis [384], and suggest that SE research might best exhibit many parallels with modern medical and pharmaceuticals research.

From Quackery to Modern Medicine

Early medicine was frequently *highly* unscientific. In America and England in the 18th and 19th century, quackery prevailed.¹⁸ Our view back on this quackery is sometimes romantic: charismatic hucksters crawling from town to town with a cargo of small bottles filled with mysterious potions. The salesmen would speak to crowds and extol the virtues of their wares. These snake-oil salesmen promised that the mysterious concoctions contained in their bottles could cure a multitude of ailments. The recipes for these concoctions frequently were created by mixing together various herbs and spirits which were believed to have healing “powers”. There was little available from medical science to either understand the ailments, or the possible mechanisms by which the cures could be effective. Consequently, little emphasis was put on such explanations: if the body’s complicated mechanisms were poorly understood, what confidence could be placed on any explanations offered? Potential clients were just interested in whether or not the potions *worked*. Of course, “explanations” for why these miracle cures worked were pronounced, but without the sciences of biochemistry and genetics, all explaining had to be simplistic. Sometimes the healing ability were attributed to mystical properties of the ingredients; sometimes to the potions’ capability to tap into the powerful—and often unnamed—healing capabilities of the human body. Sales depended on endorsements, reputation, gullibility, and snappy brand names.

Since the time of the traveling snake-oil salesmen, medical sciences and pharmacology have blossomed and matured to the point where, not only do we have marvelous new drugs, but we also understand to a much better degree the causes of diseases, how they progress, and how various drugs, organisms, and radiations chemically and physically interact with the human body. The contrast between the snake-oil

¹⁸See e.g., “The Golden Age of Quackery” [310], and “Peddling Snake Oil” [453]. Actually, quackery is still alive and well. Both quackery and anti-quackery sites litter the web.

era and modern medicine is stark. Now, we have a certain level of sophistication concerning pharmaceuticals. We know much more about the causes of ailments. The knowledge of underlying causes, and the capability to be more demanding in showing evidence go hand in hand. Now we insist on having clinical evidence showing drug safety and effectiveness. But we also want to be able to explain *why* the drugs work. Appeals to mystical properties of drugs or of humans are no longer acceptable. Good *explanations* are important. Unfortunately, the explanations come at a high cost—one must have theories and models out of which to create the explanations. Chemistry is needed to decompose pharmaceuticals into active ingredients and fillers. Biochemistry has to be developed before it is possible to explain how something like the birth control pill works. A great deal of science knowledge is needed to support modern pharmaceuticals research. Fortunately for the pharmaceuticals developer, a great deal of knowledge is available.

It may just be possible to glimpse part of the future of SE tools research from the history of pharmaceuticals and medical interventions. The state of SE tools today is arguably much better than medical interventions were in the “golden age” of quackery. But the analogy to the snake oil and quackery times still draws some uncomfortable parallels. Trade literature today routinely tout wares in ways reminiscent of the snake oil salesmen. One web page, for instance, claims that “Even if a software project involves millions of lines of code, high-end application developers can use SNIFF+ tools to quickly and easily comprehend, navigate and analyze source code.”¹⁹ The “easily comprehend” claim is vague yet it is clearly disputable. Of course, web sites can still be found for such medically dubious products as cellulite removing creams, but the point is that the current marketing of software development tools too much resembles the snake oil era when it comes to cognitive support. Much emphasis is currently made on product endorsements by prominent clients, and on favourable hands-on reviews. The subtle implication is that there are few expectations of better forms of evidence. Of course, trade literature is not the same thing as the academic literature on SE tools. Still, it hardly can be argued that the academic research community have done enough to distance themselves from criticism [724]. The point is that much more needs to be done than simply “validating” tools in ways similar to the way drugs are validated in clinical trials. Validation is not enough: “deep” explanations of the actual tools are required. For the case of pharmaceuticals research, producing better evidence required many improvements in scientific knowledge about the causes of maladies, and the ways in which the pharmaceuticals work.

Gaining this scientific understanding is far from trivial. It is, in fact, much more than one researcher can effectively handle. The typical way to deal with subject complexity is to decompose the problems into well-partitioned subproblems, and then set up subdisciplines to study them. As Plato is thought to have said, science should “carve Nature at its joints” [317, pg. 7]. Where these joints should rightly be raised some concerns [26]. But wherever they are set, it is impossible to take on the whole at once, so together the subdisciplines (and sister disciplines) set up a *knowledge ecosystem*. In such an ecosystem, knowledge is produced by one subdiscipline and consumed by another. Modern medicine split itself up (quite successfully) in exactly this manner, and has developed a roughly defined set of roles for medical research. The knowledge ecosystem in medicine is now very complex, with many astonishingly narrow specialties.

¹⁹http://www.windriver.com/products/html/embed_dev_tools.html, retrieved on 2001/01/10.

Luckily, a rough generalization is enough to make an illustrative analogy. Medical practitioners diagnose medical conditions and determine interventions to solve the medical problems they discover. The practitioners know about drug capabilities and side effects, and prescribe and administer drugs after matching problem diagnosis to an appropriate intervention. They need to know the efficacy and risks associated with the drugs, and they depend upon the fact that the drugs are validated to ensure their efficacy and to establish their associated side-effects and risks.

Pharmaceuticals research itself is divided into (1) the research and clinical trials that establish efficacy and risks, and (2) the research that develops new drugs and drug ideas. These latter pharmaceuticals researchers frequently use their understanding of biochemistry, genetics, and human physiology to develop the new drugs. In particular, they rely on an understanding of how drugs can produce their effects (delivery mechanisms, interaction with the immune systems, barriers to absorption, etc.). In general, they do not make up the problems they are solving, nor do they build the underlying science upon which they base their solutions. They rely on more basic science researchers such as protein structure researchers, geneticists, and cell chemistry researchers. They especially rely upon those researchers who turn basic research into applicable techniques. These latter researchers include those that make gene splicing or protein synthesis practical. Pharmaceuticals researchers import this knowledge, and they frequently export their set of unsolved problems for which a solution would be desirable. Over time, the medical professions have tackled the early unscientific basis for building medical interventions to human ailments with an overall approach involving a separation of basic sciences research from applied science, and design sciences from testing. Clearly, in order for this ecosystem to be successful the pharmaceuticals researcher must be able to effectively use knowledge gained in the basic sciences. In this scheme the applied theories and methods carry this knowledge.

The above pharmaceuticals research ecosystem is depicted in Figure 2.2(a). I have ignored many details, such regulatory agencies and pharmaceuticals packaging, but even though the overall picture is a crude characterization, it is sufficient for drawing parallels to cognitive support research. One way is with the following mapping: drugs are like tools or tool concepts, medical practitioners are akin to software developers (i.e., tool adopters), pharmaceuticals testing is similar to tool validation, pharmaceuticals development is like tools research and development, and the applied scientists that bring basic science to practice are like cognitive support theory developers. In this scheme, the basic principles of cognition and cognitive support are akin to the basic sciences underlying modern medicine. This scheme is diagrammed in Figure 2.2(b). This dissertation fits squarely into the “cognitive support adapter” role in that diagram. My target audience for the knowledge developed is the “tool innovator”; I draw upon the basic sciences of psychology and cognitive science. My goal, therefore, is to make knowledge from the basic sciences available to the tool innovator.

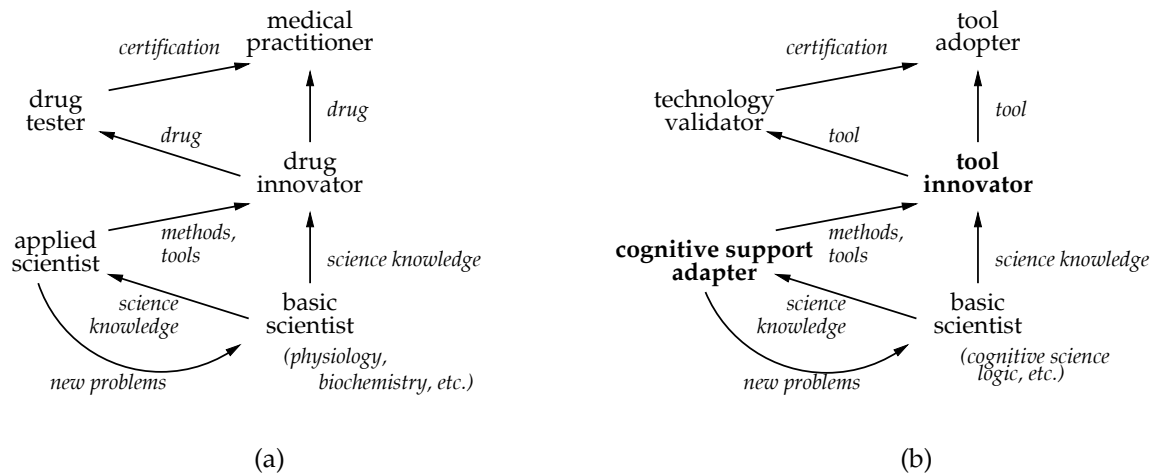


Figure 2.2: Models of (a) pharmaceuticals and (b) possible cognitive support knowledge ecosystems

Three Alternative Visions

The above scheme of the ecosystem for software development tools research establishes a clear distinction between users and developers of cognitive support theories and models, and users and developers of the basic science underlying these theories. It is the applied theories that tie basic science to its application. This scheme is not without its potential critics. Three different sorts of proposals have been at times advocated which blur the distinctions between specialists. Although they do not have standard names, the proposals might well be called the “scientist as tool builder”, “tools researcher as scientist”, and the “group knowledge” approaches. Each approach makes different suggestions as to how the appropriate specialist knowledge should be applied.

The “scientist as tool builder” approach advocates that basic science is done through tool building. For instance Anderson [12], Landauer [369], Norman [468], and many others (e.g., [169, 186, 447, 448, 494]) envision research programmes in which scientists such as cognitive psychologists create basic science knowledge by building tools. This vision of research collapses the roles of basic and applied scientists with the roles of tool developers. Several reasons have been given for trying this approach [350]. Norman argued that applications tend to drive science rather than the other way around [468], so that the basic science researcher should be developing tools. Other reasons given have included the belief that theories would become more relevant to application, and the fact that novel tools would rely on new theories. Most of these reasons cite benefits for the basic scientist, not the tool developer. Be that as it may, even if none of these reasons are borne out, it is certainly true that wearing the dual hats of tool developer and basic science researchers does provide good assurances that at least some knowledge from the underlying sciences can be applied by the tool builder during research and development.

The “tools researcher as scientist” approach essentially suggests that tools researchers do the basic science. Commonly the argument advanced to support this approach is that the scientific methods used by the basic scientists can be adopted and wielded by tool builders during design. Tool designers would need to establish the validity of the principles inherent in their designs. Numerous suggestions have been made in this vein for using methods from other specialist disciplines during software development. These

include applications of methods from psychology [369–371,402,458] and anthropology [24,60,311]. Sometimes there are no expectations that the software developers become specialist experts (as in Nielsen’s suggestion that computing scientists use protocol analysis [458]). In other cases it is clear that tools researchers are expected to acquire and maintain highly specialist skills and knowledge such as from cognitive science. For instance the proposal by Williges *et al.* [706] assume that fundamentally theory-building experimental methods be used during design. The general thrust of this approach is that the tools researcher is not so much using principles and theories originally constructed elsewhere, as they are experimentally establishing them (once again) in their particular design contexts.

The “group knowledge” approach argues that specialist knowledge be imported in the form of human experts. Thus psychologists, anthropologists, and HCI specialists, it is suggested, should be part of software design teams, and of research teams as well. This imports the required knowledge in the form a human specialist. It consequently side-steps the problem of making basic science knowledge available to non-specialists. This raises a number of other problems (e.g., scientists tend to specialize in narrow areas, so it is an open question as to how many experts should be included), but it seems to assure some measure of availability for specialist knowledge.

The above three proposals all raise serious concerns for the practice of SE research. The “group knowledge” approach is in principle quite sound and has been practiced quite successfully in the past (see, e.g., Carroll *et al.* [111], Landauer [371, ch. 5], and Green *et al.* [271]). But (1) the approach does not avoid the need for good cognitive support theories for the specialists (which is still poorly developed anyway), and (2) it does not really solve the problems for SE researchers, who are left with a critical issue they cannot properly address without recruiting knowledgeable colleagues from other disciplines. Should all grant proposals in the field include funding requests for a psychologist as a research associate?²⁰ The other two approaches are problematic because of their implication that SE researchers wear the two hats of basic science researcher and tools researcher.

Some arguments certainly can be made in support of the two-hats approaches. History has shown that it certainly *is* possible for one scientist to successfully wear the two hats. In fact, it has been frequently pointed out that the seemingly clean split between basic and applied research quickly dissolves under close inspection (e.g., Nickerson [455], Naur [439], Kirlik [348]). So perhaps the split between applied and basic scientist in Figure 2.2(b) is never quite so clean anyway. In addition, it is common for researchers like psychologists and sociologists [156] to study software development as a particularly interesting domain for study and application. As a result, several works studying software development have tried to advance theoretical understanding and simultaneously build better tools (e.g., Adelson *et al.* [6]). Researchers from other domains *can* do good software development and tools research. Sharing a common problem domain has been nothing but mutually beneficial. And there is certainly nothing stopping SE researchers from becoming more knowledgeable in cognitive science or sociology. An excellent example of the results possible is the work done by Redmiles *et al.* [532] to weave together applied theory and tool development in work on the Explainer [533] and Argo [544] tools. But are the close ties really necessary?

²⁰The question was rhetorical. Those psychologists and HCI specialists currently looking for work need not email me to argue the affirmative.

A Necessary Marriage?

Laboratory experimentation to test systems is an interim procedure, to approximate the benefits of models, while we wait for our science to mature to the point that such models are complete.

– Bonnie E. John, Panel statement on “The Role of Laboratory Experiments in HCI” [711],
pg. 266.

Even if each of the three alternative views have some merit, the question still remains: does the tool innovator *need* to be, effectively, a trained scientist in a contributing research discipline? It is still reasonable to question whether the tight couplings that are being proposed (and which exist in current practice) will really be necessary in the future. There are several troubling issues. First, it has yet to be proven that advances in basic sciences are necessary to build better tools. It is certainly unclear that we currently employ what we know to full effect. Consequently, research in cognitive support should not, as I have argued, invariably require extensions to the underlying science base. So doing basic science research is certainly not an absolute and universal necessity. Second, the requirements made of research in the basic sciences are frequently at odds with the aims of tools researchers. For instance, as Green has argued [259,270], the sorts of research demanded by parent disciplines too frequently generates results of the wrong sort, emphasizes statistical significance over effect size, and favours simplified models that can be easily verified over models that integrate numerous phenomena and can therefore address design tradeoffs. The basic sciences simply haul in too much baggage from their respective research paradigms. Third, without accessible cognitive support theories, SE researchers are faced with the difficult task of finding, interpreting, integrating, and applying basic science research. As the applied work by Redmiles [532] shows, many of the experimental problems described in Section 2.2 crop up, such as the burdens of building theories.

The alternative vision presented in Figure 2.2(b) is one where the basic ideas and models of cognitive support are quite well established, and the main challenge presented to tools researchers is to use these theories and models in establishing new supportive technologies. The scientific basis of tools research is improved when tools researchers use applied models that are ultimately grounded in basic research. The vision creates a clear distinction between tool-oriented research that *uses* basic science, and support-oriented research that *generates* knowledge about cognition and cognitive support. Knowing more about cognition and cognitive support may still be a requirement for the future tool-oriented researcher, but the value of a mature theoretical framework that can be imported for this purpose is inestimable. Furthermore, a permanent position is created for research that generates applied models from more basic sciences. The GOMS models created by Card *et al.* are the results of one well-known prior effort in generating applied theories for tool researchers [94]. The next few chapters of this dissertation are another attempt.

2.4 The Practical Art of Designing Theories

So far in this section I have argued that the topic of cognitive support is important to our research community, that we need to pursue cognitive support theories, that we have largely neglected them to our own detriment, and that they appear to be a resource that, in the long term, we hope to *use* to build good tools with, rather than having to continually build ourselves. To continue the ongoing debate, then, the questions become: how should we begin building them? *Can* they be built? It was already mentioned above that there is some debate as to whether or how an appropriate cognitive support theory could be built. The debate appears to hinge upon what we expect cognitive support theories to be, and what we expect them to do for us. I will argue here that if our expectations are relatively modest and thoroughly pragmatic, there should be no real difficulty in developing useful cognitive support theories. In my view, the theories are tools—artifacts—and we should think of theory building as a design activity.

Setting Proper Expectations for Theory

Opinions about the usefulness of psychology-related sciences differ greatly. Some people are highly dismissive; others feel that psychological sciences are obviously relevant and have much potential. It is unwise to avoid this debate. By studying one's critics, one can identify and therefore avoid potential pitfalls. More importantly, debating the issue requires preparedness. To answer the critics of psychology requires one to either demonstrate appropriately useful theories, or else put forward a reasonably convincing and realistic plan for building them. The following presents a pragmatic plan for building SE-relevant cognitive support theories. Many potential criticisms against the approach are considered one by one, and then countered with pragmatic replies. Afterwards, an approach to iteratively building theories is outlined.

Being able to present such a plan hinges on first settling on an appropriate definition of the term “theory”. The position taken in this work is essentially the same as Newell's in his heroic assault on “Unified Theories of Cognition” [446]. Newell said:

To state it positively and in general, let there be some body of explicit knowledge, from which answers can be obtained to questions by inquiries. Some answers might be predictions, some might be explanations, some might be prescriptions for control. If this body of knowledge yields answers to those questions for you, you can call it a theory. There is little sense worrying that one body of knowledge is just a collection of facts (such as a data base of people's heights and ages) and another is the fundamental axioms of a subject matter (such as Newton's three laws plus supporting explication). The difference, of course, is important. But it is clear what is going on—they answer different sorts of questions and have different scope and have different potential for further development. That is, they are indeed different bodies of knowledge. But both play a role in providing needed answers... [446, pg. 13]

Theories are considered here as ways of generating answers to questions. Naturally, the traditional conceptions of scientific and empirical justification still apply, and are considered desirable. Given this definition, a key aim of this work is to define theories that are useful during tool analysis and design, and to work towards their scientific and empirical justification. The remaining question is how to do so.

The answer offered here is to consider seriously the idea that application-oriented theories are *artifacts* that need to be *designed*—just like any other artifact—with the needs and characteristics of its users in mind. This stance will be expanded upon somewhat in Chapter 7, but a few comments can be made here. The approach treats theories as artifacts—as things to design.

The purpose of the theory-as-artifact approach is to take seriously the idea that theories need to be built to suit their intended application [84]. The overall approach is not new. Barnard [28,30], for instance, pointed out that perhaps one of the reasons so few good design-relevant cognitive theories exist is because nobody has really ensured their usability:

... those seeking to apply cognitive theory in HCI have, in fact, behaved just like early system designers. Like glass teletypes, basic psychological technologies were under-developed for their new purpose and, like the early design of command languages, the interfaces (application representations and engineering models) were heuristically constructed by the theorists around what they could provide rather than by any deep analysis of requirements, extensive study of their target users (designers), or of the actual contexts in which design occurs. [30, pg. 106]

Taking this view into account, the outlook pursued here is that the artifacts (theories, models, examples, etc.) being produced are supposed to be early prototypes. They are tools for analysts and designers—tools to answer questions. But the first ones will be “alpha” versions. They will necessarily seem hacked together, will be missing functionality, and be only partially usable. But it is reasonable to expect them to go through the improvement cycles common to other artifacts. A few researchers may try them and then improve them. New capabilities will be added. More and different theories will join them. Perhaps “user manuals” and tutorials will be written on how to use the theories, as has been done for the Cognitive Dimensions framework [269]. This theory-as-artifact viewpoint provides the way of answering possible criticisms (i.e., potential pitfalls). This shows up in the point-counter point debate below. The debate is summarized in Table 2.7.

Point, Counter Point, and Approach

Pylyshyn took the approach of debating with himself about the possibility of theory that is actually applicable to design [522], and this tactic seems to work here also. The tactic is used below to raise and counter arguments which may be put forward for why useful cognition-related theories cannot be built.

Behaviour is too complex.

Point: Psychology has been seriously studied for a century and we seem to barely have made a dent [370]. Thus, whatever approach we use we cannot expect the basic sciences to provide many good answers.

Counter: Good approximations are still useful and we should be able to generate some good approximations [467]. Furthermore, sometimes the approximations are what we want. For instance, Newtonian mechanics, though approximate, is frequently just fine for ordinary mechanical engineering. Moreover, it has been argued that relatively little deep scientific knowledge is needed to make useful observations. For instance, Hutchins [320], in his work on DC, argued that little of the “internal” psychology of the

Useful Theory is Impossible

POINT	COUNTER POINT	MY APPROACH
behaviour is too complex	should be able to approximate	iteratively refine / deepen theories
	shallow models are still useful	build “broad-brush” theories
there is no hope for “the” theory	approximation is better than intuition	use theory when available

Theories Are Not Really Needed

POINT	COUNTER POINT	MY APPROACH
design precedes theory	craft and folk knowledge is used instead	try to add rigour and formality
	analysis and evaluation is still important	improve theory basis for evaluation
ignored by practitioners / SE community	problem is their usability	use familiar computational ideas
SE researchers are not psychologists	theories/models carry the psychology	build knowledge into models
SE research is not psychological research	import psychology/HCI, not export it	build theories from existing knowledge

Current Theory is Too Poor

POINT	COUNTER POINT	MY APPROACH
theories are not applicable	only <i>some</i> is not applicable	seek related and usable research
theories are irrelevant	used wrong base / tackled wrong issues	focus on problem, not available theory
theories are immature	exploratory work invigorates research	begin with broad high-level theories
theories are unintegrated, fragmentary	can still try to integrate	pioneer integrative frameworks

Table 2.7: Arguments for a realistic pursuit of cognitive support theories

individual was needed. Thus it may be enough to produce decent approximations of relatively few psychological phenomena. **Approach:** Start with relatively gross approximations and build “broad-brush” theories [257] initially. Then iteratively refine the theories. Detail can be added if or when it is required; the application-oriented theories can follow advances in the basic sciences when they occur.

There is no hope for *the* theory.

Point: There is no hope that complete and accurate theories can be built which will be able to handle our analysis and design needs. As Singley *et al.* said, “Certainly now, and probably always, psychological principles will under determine design. Deducing a design from first principles is too much to ask from any psychological theory.” [599, pg. 197]. So even if psychology were solved, design theory will never be. **Counter:** Like in the previous counter point, the answer to impossible perfection is to happily approximate. Even rather extreme approximation is useful since right now we have practically no scientifically based cognitive support theories. **Approach:** The SE researcher can make use of whatever theory can be

provided, and simply supplement it with their own craft skills and intuition. The aim is to use whatever resources we have now to the best of our abilities.

Design precedes theory. _____

Point: It is often the case that we can build good tools without the theory to explain why they are good. Thus theories are not needed. **Counter:** So what? Theory construction does not *have* to postdate design [384], and designers can still use theory whenever it is available [106]. If we actually had the theory, nothing would stop us from using it. Furthermore, even if good design precedes theory, we need the right explanations during analysis and evaluation (see Section 2.2, above). How can we rationally compare tools otherwise? How do we know what parts of a tool to copy? What lessons generalize? **Approach:** It is worthwhile pursuing more rigorous foundations to tool building. Chapter 8 illustrates that understanding our existing tools better is in fact a worthwhile goal even if no truly novel design knowledge comes from it.

Ignored by practitioners / SE community. _____

Point: Tool developers in SE and CS for the most part ignore HCI and psychology [263]. Cognitive support theories will therefore have little practical impact other than for psychologists, or possibly HCI specialists. **Counter:** The problems of making theories usable to non-specialists are indeed severe, but nobody has proven the problems are insurmountable. Modern undergraduate HCI courses teach a modicum of theory. If what we do is simple enough, we could reasonably expect it to be added to such a curriculum and in that way slowly make further inroads. **Approach:** In this dissertation, I have striven to use concepts and terminology familiar to researchers with traditional computing science backgrounds. For the most part familiar intuitions about computation transfer to the topic of cognitive support. This should help increase the acceptability of the results to computing scientists, although obviously there are no guarantees.

SE people are not psychologists. _____

Point: Past approaches in HCI have required a rather considerable amount of psychological sophistication in order for them to be effective [657]. Building theories for specialists with deep knowledge is not inherently wrong, but what about the average SE researcher who wants to do SE research? They will not have the right background and skill set. **Counter:** The straightforward answer is that models carry psychological knowledge. So the necessary psychology can be built into the models being used. Although a limited number of useful models can be built in, the number is surely nonzero. **Approach:** Focus on important items that can be addressed using broad-brush theories. They will probably have the most “bang for the buck”. Some basic psychology is included in the models of Chapter 5 (e.g., limited memory capacity). Learning the models entails learning a little of the relevant psychology. More importantly, if the models are good, the psychology learned is precisely that which is needed most.

SE research is not psychological research.

Point: SE researchers want to build novel technologies and computing science theories, not psychological theories. To build good cognitive support theories, one needs better cognitive science or other scientific foundations. **Counter:** We do not have to do the psychological research if we can import it. Even right now, there is plenty of useful research to start the process, and much of it has never been adequately explored in the context of software comprehension and software development. **Approach:** Instead of trying to add a small bit of detail to the tapestry of basic science knowledge (e.g., validating a model of comprehension [654]), the focus will be on collecting and integrating existing knowledge.

Existing research is not applicable.

Point: Most of the basic science sources we can consult fail to be directly applicable. They may be too low-level, explain highly esoteric and isolated phenomena, and fail to be ecologically valid [26,155]. Thus we are left with practically no applicable science base. **Counter:** Not *all* of the prior work is inapplicable. True, a good fraction of the applicable work seems somewhat underdeveloped, but I would claim that we have more than enough basic science to work with to build good initial application-oriented theories. **Approach:** The plan is to seek out useful research, for it is unlikely to be entirely absent. A start to the required survey work appears in Chapters 3, 4, and 5.

Prior work has been irrelevant.

Point: Even when one finds potentially applicable theory in the existing literature, it is mostly irrelevant to the important issues in software development. If one wishes to understand the role of tools for programmers, studies which restrict the use of tools, or which model “disembodied” thinking are not going to be especially useful [41,472,493]. Even if you succeed in modeling real-world thinking, it may not help you understand cognitive *support*. **Counter:** Prior work has not concentrated on cognitive support in part because there has been little awareness of the issue. In addition, much of the work in this field has adopted methods and techniques which are inadequate for addressing cognitive support. But that does not mean the existing knowledge is entirely useless. What if it could be suitably augmented? **Approach:** The solution is to let the problems drive the theory, rather than to start with one’s favorite theory (or whatever one is available) and try to squeeze as much out of them as possible. Gaps in the science base will be highlighted and filled in as best as possible. This can feed new problems back into the basic sciences. Thus over the long term, being problem-driven should help to steer research towards buttressing the theories in the needed ways.

HCI research is too immature.

Point: Even if you used the right science base, our understanding of HCI and cognitive support is very immature. What could you possibly produce from it? **Counter:** True, it is in ways immature, but we have used hardly any of it yet! We can try make better use of what we have, and fill in the rest tentatively. **Approach:** Given the current state, the most reasonable course of action is to start with very

“broad-brush” [257] theories. This goal is adopted in Chapters 4 and 5 when building theoretical frameworks and models. And, of course, these theories are expected to be iteratively refined and deepened in time.

Cognitive support theory is unintegrated, fragmentary. _____

Point: Artifact-supported cognition seems to have frequently taken a back seat to studies of the “unaided” mind. Where research exists, it is scattered in multiple problem domains and research disciplines, and it is unorganized and unintegrated. Efforts to build application-oriented theories seem, in this light, to be premature. **Counter:** So let us start gathering the stuff together in earnest. We do not need perfection the first time. Try and then update. **Approach:** In order to get anywhere, a broad survey of cognitive support phenomena and theories needed to be generated. This survey is presented, collectively, in Chapters 3, 4, and 6.

2.5 Summary

...like others in human factors, we have been faced with sometimes critical questions of human behavior in technical systems and, having gone to our cache of theoretical weapons, have found the cupboard uncomfortably bare.

– Flach *et al.*, in preface chapter [220], pg. xii.

For the last four decades or so, SE and CS researchers have pondered the psychological aspects of software development [308]. But this has been a fringe area, and not many traditional SE and CS researchers have delved into it. And probably rightly so. Mainstream SE and CS research regarding tools has dealt primarily with computational theories and technological innovation, not with performing psychology or sociology. Even so, the topic of supporting cognition is still one of the central issues of the discipline! The cognitive and social issues, it is widely agreed, are some of the most pressing problems of software development, and are probably the most difficult to solve. The endeavor of building tools for software development has psychology written all over it. Thus although the majority of SE researchers should probably not *do* psychological research, neither should psychology be ignored. Nor should it be marginalized in the way it has been in the past. A particularly vexing problem is that *even* when developer cognition is studied, the distinction between studying cognition and cognitive support is frequently not made, and the supportive nature of tools are not fully realized. All of this needs to change—the sooner, the better.

SE research must more seriously address how development tools aid cognition. It needs an applied science of cognitive support, and an engineering discipline built around that applied science. This is not a plea for more basic psychological research, but an argument to establish an applied science that imports the basic research. Really, nothing of this sort currently exists. An applied science of cognitive support would be nearly as important to SE tools research as are parsing and automata theory, logical frameworks,

or models of formal semantics. The difference is that applied cognitive support theories will be born from disciplines like cognitive science, cognitive anthropology, and psychology, whereas the latter have come from the historically closer siblings of mathematics, logic, and theoretical computing science. In a way, this chapter has aimed to advertise the clear need for an applied science of cognitive support, and thus serve as a “call to arms” to both appreciate the need for applied cognitive support theories, and to join in their development.

This chapter was organized as a debate about the role of cognitive support in SE research. My hope is that this debate helps put the importance of the topic of cognitive support into sharp focus. In the past, too many SE researchers have been able to justify to themselves, in one way or another, that it is alright to maintain a blissful ignorance of the psychological aspects of their tools. On the flip side, too many cognition-oriented works in the field effectively avoid the question of applicability. Both of these viewpoints are real threats to positive change. I hope the extended debate in this chapter can pick apart these roadblocks to a more rigorous research discipline.

The debate consisted of a progression of questions and arguments in response. First, it was argued that cognitive support is, as a subject, important for SE research. It was pointed out that we have been studying cognitive support all along, but it has been informal, unprincipled, and unsystematic. A call for research into cognitive support theories is therefore “merely” a call for increased rigour, and for more sound scientific principles. Second, several serious problems with existing research practices were noted, and these were traced to the lack of suitable cognitive support theories. Third, the desirable form of such theories was explored by examining the mature example provided by our knowledge mechanical support. In addition, the way SE research related to cognitive support research was elaborated upon in order to set realistic goals and expectations for SE researchers. Finally, a programme for building suitable theories was outlined.

The programme is typical of applied sciences research. In applied sciences, one starts with a problem, looks amongst existing research for applicable theory, massages a candidate theory into shape, and then tries it out to see if it is useful. The work by Rasmussen *et al.* on “ecological interface design” [531] is an outstanding example of the sort of research that is involved in this sort of programme. Vicente summed up their efforts as follows:

First, we conducted a literature review to identify empirical findings that might be pertinent to the aspects of interface design in which we were most interested. Second, we used the SRK taxonomy²¹ as an “umbrella” framework for integrating, under a common language, the variety of research results that were encountered. Third, we then used the theoretical constructs of SRK to deduce from these findings a set of three principles for interface design [657, ch. 11].

Except for the particular theories and findings involved, the process described in this quotation is closely followed the remaining chapters.

²¹A theoretical construct that will be covered in later chapters.

Chapter 3

Cognitive Support Phenomena

The power of the unaided mind is highly overrated. Without external aids, memory, thought, and reasoning are all constrained. But human intelligence is highly flexible and adaptive, superb at inventing procedures and objects that overcome its own limits. The real powers come from devising external aids that enhance cognitive abilities. How have we increased memory, thought, and reasoning? By the invention of external aids: It is things that make us smart.

– Donald Norman, “Things That Make Us Smart” [472, ch. 3]

The very idea of cognitive support would be quite absurd without some way for the world outside the head to make some kind of difference to the world inside, that is, to human thinking and problem solving. If tools are claimed to support cognitive activity, it behooves us to provide some sort of explanation and proof of how in the world such support can possibly work. To really *explain* cognitive support, the way that the external world is integrated into thinking needs to be modeled, and the supportive nature of this integration needs to be understood. If cognitive support is to be *designed*, some way of reasoning about how to introduce and tune these interventions needs to be elaborated. The attempt to deliver design reasoning is saved for Chapter 7. This chapter, and the two following it, are concerned with the first issue, that is, with simply being able to explain how the external world can impact cognition, especially the way artifacts like software development tools can support cognition in developers. Here we are in search of cognitive support theories.

The approach taken is the theory *design* approach outlined in Section 2.4. The overall plan for theory construction is to determine what explanations are needed, to then find or make resources from which to build suitable explanations, and to then create theories and models from these raw materials in a way that makes them valuable to their eventual users. From a scientist’s point of view, this roughly corresponds to elaborating the phenomena of interest and what sort of statements one wishes to make about it, setting the theoretical stance and researching prior related work, and then generating an initial theory accounting for the phenomena. From a theory designer’s point of view, the process looks instead like performing

requirements analysis, exploring the design space, and then elaborating the design once an overall approach is settled on. Viewing the situation from this top-most level it is clear what the general plan of this chapter and the four following will be: first survey and expose the types of cognitive support that are relevant to software development (this chapter), next scour existing DC theory for applicable resources (Chapter 4 and 5), and then customize or build appropriate models and theories (Chapters 5, 6, and 7). The first step cannot be skipped. Cognitive support is currently rather enigmatic. It is hard to understand cognitive support simply because cognition and human behaviour is hard to understand. But there are other complications.

For one thing the essential nature of cognitive support does not seem to be a unitary and homogeneous thing. Simply put, there seem to be different types of support. Consider, for instance, three tool sets: paper and pen, a typical compiler that checks a program's partial semantics (e.g., type checking), and the search tool `grep`. These are all clearly important for many types of software development. Each seems to support the thinking work of developers in some way. But the support seems different in each case. Is it, and if so, how? If we cannot answer this question there is a real danger of choosing a focus that is too narrow, and consequently adopting a conception of support that is far too limiting. The first challenge, then, is to make sure that we cast the net wide enough to encompass the important variations, and yet still be able to recognize the similarities and differences among them.

A further complication is the fact that the research concerning cognitive support lies scattered in many small pieces, and that it sometimes spans the boundaries of traditional research disciplines. Sometimes it is investigated by anthropologists, sometimes by cognitive scientists. Sometimes it is studied in educational contexts, but sometimes in industrial ones. Not surprisingly, the basic descriptions of the *phenomena* of cognitive support are diffused and variable. For example, one researcher might talk about artifacts being "external media" that act to enhance "reflective thought", while another may speak about "augmenting" human intellectual capabilities. Cognitive support has many guises and aliases. The challenge this presents is thus the problem of synthesis: the various ideas must be gathered together and coherently composed. Comparing and collecting past knowledge into a coherent whole is itself a difficult challenge that exists independent of the problem of actually *explaining* these phenomena by proposing models and theories that account for them.

Making this last obstacle more difficult is the fact that the *explanatory frameworks* used to understand cognitive support have been fairly incompatible and varied. Several of the potentially applicable theoretical schools are widely known to be difficult to reconcile [103,434,546]. Later chapters will settle on one particular school called DC, but its "competitors" should first be discussed and briefly assessed.

The above challenges are met in this chapter with a short review of selected research pertaining to cognitive support. The review is necessarily incomplete. Perhaps thousands of thinkers throughout history have contemplated the relationship between human thought and artifact. Psychologists have considered it off and on for over a century [454]. A complete or even decently representative review of this history is thoroughly beyond the scope of this work. My goals in this particular chapter are correspondingly modest: I wish to help bring into the general SE zeitgeist a number of common observations about the nature of intelligent human activity, and to enumerate some of the supportive relationships that artifacts have on these. Like flares shot over an unfamiliar field to illuminate the general lay of the terrain, I hope these

brief summaries can outline many of the basic types of cognitive support issues that may be of particular interest to SE. In addition, I hope that this review, and the resulting collection of citations, can be of aid to those researchers wanting to continue on in this line of work, and to SE researchers desiring additional reading. I also wish to take the opportunity to relate cognitive support in SE research to studies in other disciplines. Some of these disciplines are well-related by topic and method, and yet somehow the papers they generate never seem to be found sitting next to each other in bibliographies. Relating them to one another may thereby help build better bridges into these sibling disciplines, so that they might in the future become conduits of knowledge useful to SE.

Organization

The survey is organized along three main dimensions: the “supportive relationship”, “descriptive theory”, and “schools” dimensions. The “supportive relationship” concerns the characteristics of human thinking, and how artifacts and tools relate to this thinking. This dimension addresses the question: “generally speaking, how can artifacts aid thought?” The “descriptive theory” concerns the ways that researchers and philosophers have tried to argue the purpose and significance of artifacts in human activity (e.g., the belief that certain interactions with artifacts are crucial to developing so-called “higher” mental capacities). This dimension generally addresses the question: “why are artifacts important for human thinking?” Finally, the “schools” dimension is concerned with identifying the sort of research traditions and problem domains that have seriously studied cognitive support. Each of these dimensions is tackled in a separate subsection in the following. After this chapter, the stage will be set for building models and theories that seek to explain these phenomena.

3.1 Supportive Relationships

When they are solving problems, human beings use both internal representations, stored in their brains, and external representations, recorded on a paper, on a blackboard, or on some other medium.

– Larkin and Simon, “Why a Diagram is (Sometimes) Worth Ten Thousand Words” [375], pg. 66.

Aristotle speculated that one of the reasons that the base-10 numbering system is so prevalent is because our 10 fingers are so useful for counting [588]. We now know that many different number systems have been developed throughout history which are based on counting other physical artifacts, and thus are not base-10 [629]. But the point is that we use our hands, bodies, and other artifacts to support our thinking, and consequently these significantly affect it. Certainly it takes only a few moments of reflection to realize some of the many ways human thinking and human artifacts are inextricably codependent.¹

¹Yes, *co*-dependent. Our cognitive culture depend on them, and the cognitive artifacts owe their existence and form to our cognitive needs [320].

Where would we be without written language, without the wealth of books that fill our libraries, or, in light of technology updates, the flurry of emails that many of us rely on in daily work? There can be little doubt that artifacts significantly impact our mental lives.

The thinking work of software developers is no exception. Would software developers work nearly as well without scrap paper, a white board, or the seemingly ubiquitous restaurant napkin? The software developer's intellectual world differs from the more mundane only in the fact that some of the artifacts they think with are special and, perhaps, somewhat more complicated than in many other everyday activities. Software designers build and "play" with prototypes as a critical part of design; requirements analysis draw diagrams and write formal models; programmers run programs in order to understand figure out how (or if) they work. Thinking, creating, and problem solving takes place within the context of such artifacts, and is thus impacted by them. In particular they have a supportive relation to mental activity.

There are several important classes of such supportive relationships that are surveyed below. The categories below do not represent completely orthogonal dimensions of support: there is significant overlap amongst them. But I have attempted to draw together common "themes" in understanding these relationships. Within each theme, support concepts will be described, some salient research will be reviewed, and some of the implications for SE tools research will be drawn.

3.1.1 Embodiment and Strategic Artifact Use

One key to understanding the psychology of complete tasks is to understand how different task environments affect peoples strategies, as Newell and Simon (1972) argued long ago.

– Stephen J. Payne,

"On Mental Models and Cognitive Artefacts" [493], pg. 104.

We live in a complicated world and, since we are quite smart, we should expect that people will use artifacts in their environment to full advantage. This world includes our own bodies and the artifacts that surround ourselves. This might be called the "external" world—the world outside the head.² Because of being embedded in the physical world, we can make use of this external world as an extension of our internal world. Moreover, one could consider the use of the external world *in place of* the internal world. For example we might use our fingers when counting coins [352], or while following data flow lines on a visual program [272]. One way of looking at these external devices is that they are "resources for action" [719]. If these external resources were not available, internal resources (e.g., memory) would need to be used instead. Besides our own bodies, we also make use other artifacts in nearly endless variety. Well known examples include the use of the abacus for calculation, and the "breadcrumbing" or "bookmarking" techniques that are used for tracking progress. Without these sorts of external markers, we would have to perform mental arithmetic or mentally track our progress in our tasks. Consider, for

²Like many other authors do, I will use the term "internal" to refer to things "in the head" [465,469] or mind, and "external" to all other things (external memory, etc.). Assorted nitpickers and philosophers may object to these definitions (are eyes internal or external?) but they should suffice for our purposes.

example, how “long division” is frequently accomplished with paper and pencil. Having the world hold partial results is crucial to the process.

One particularly important facet of embodiment is that with developing experience, people often begin to *strategically use* external resources to make their work less mentally challenging. In the words of Kirlik, people become “ecological experts” [349]—they become adept at making effective and strategic use of the work settings [48]. Two delightful examples of this strategic use of external resources comes from Norman [472] and Kirlik [348]. Norman tells a story of how one of his colleagues uses her office workplace to organize her activity [472, ch. 7]. She files her papers in a three-level system, uses multiple calendars to mark different categories of appointments, and sprinkles post-it notes about the office to serve as reminders and reinforcements. The picture painted is of a researcher that keeps the current state of her work embedded in the state of her office. Kirlik’s example is from a rather different domain: that of short-order cooks. He observed that the cooks make strategic use of their environments to simplify their mental work. They “may organize the placement of meats in order of doneness, [and] may lay out dishes or plates to serve as a temporary external memory of orders to be prepared” [348, pg. 84] (also see Kirlik [349, 350]).

Neither Norman’s nor Kirlik’s example are in any way unusual. Like Norman’s colleague, there are several other published examples of such inventive and creative use of artifice to organize thinking and research (see, for instance, Harnad [289] and Engelbart [200]). The lesson is that humans adapt their environment to their activities and adopt strategies to make use of this fact. Lave’s early work [376] on “everyday math” was one of the earlier and more influential sources of similar observations of situated strategic use.

This relationship between thought and artifact is hardly new or obscure. McKim, focused on “visual thinking”, provides as good an overview as any:

Consider the sculptor who thinks in clay, the chemist who thinks by manipulating three-dimensional molecular models, or the designer who thinks by assembling and rearranging cardboard mockups. All are thinking by seeing, touching, and moving materials, by externalizing their mental processes in physical objects. Many contemporary thinkers, in science and engineering as well as art and design, respect the fertility of this venerable form of visual thought. ... Do not be confused by the similarity between externalized visual thinking and the expression of visual thought. A chemist who is advancing his or her thinking by playing with a molecular model is not involved in the same process as a chemist who is using a molecular model to communicate a fully formed idea to another person. Externalized thinking involves actively manipulating an actual structure, much as one would manipulate that structure mentally. [404, pg. 44]

In sum, there are two points to embodiment. The first is that various parts of external world (including our own bodies) may stand in place of objects of thought. Manipulations of such objects can also replace mental work. A recently popular phrase is that “the world is it’s own best representation” [136, pg. 46]—a reference to the fact that internal representations of external artifacts are not always necessary if the external ones can “stand in” for themselves. A corollary to the saying might be “try to use the world instead of the head” in reference to the fact that, using the right strategy, artifacts can serve representational, calculational, and inferential roles that might otherwise have required mental capacity. This corollary raises

the second point to embodiment: that there is a relationship between the capability of external artifacts to support thought and our *ad hoc* and habitual strategies for employing them in that capacity. It is important to know not only *that* artifacts can be used strategically, but also *how* people actually do it [311].

Studies

Using artifacts strategically to reduce mental effort is familiar—even mundane—and can frequently be found in studies of cognitive development (e.g., Piaget’s celebrated studies of children manipulating artifacts to make difficult problems easier), and what is sometimes called “everyday cognition” [350]. Studies of everyday cognition have been performed for quite some time, but there has been rather little overlap between these studies and many other streams of cognitive science and cognitive psychology. Nevertheless, some examples are well known in the cognitive science circles. Norman’s popular “Psychology of Everyday Things” [469] can also be categorized as a work on how everyday cognition is facilitated by cognitive artifacts. Another relatively well known example of strategic artifact use is de la Rocha’s study of how dieters turned to physical objects to solve math problems. De la Rocha observed one dieter physically manipulate cottage cheese to calculate fractional amounts instead of trying to do it mentally (e.g., see Greeno [277, pg. 287]). Another important example is Larkin’s analysis of coffee-making [374]. Larkin explained that in the normal course of events, the state of the world helps out by making plain the current state of the problem, and by providing clues as to the next course of action (see also Section 6.5.2). Larkin’s analysis suggests that these external resources reduce the need for memory and planning to such a degree that the “problem” of coffee making is made facile enough that “many of us can solve [it] reliably even with the diminished capacities of early morning” [374, pg. 320]. All of these examples serve to illustrate ways in which people naturally incorporate their environment to reduce cognitive burdens.

Relevance to Software Development

Although the examples noted above tend towards the simple and familiar, the general rule they illustrate carries well into even highly complicated domains of software development. Software developers also strategically use the resources available to them—however some of the tasks and artifacts in software development are more complex. One example of such use comes to us from Cardelli [96]. Cardelli noted that “experienced programmers adopt a coding style that causes some logical errors to show up as type-checking errors (For example, by changing the name of a field when its invariants change even though its type remains the same, so as to get error reports on all its old uses.)” [96, pg. 2240]. The type checker can be employed to do, essentially, simple *change impact analysis*.³ This is a trick of getting the world to help do thinking work. Automated typechecking is not explicitly designed to do impact analysis, but it can be employed to do so. The significant point is that without the typechecking ability of the compiler, some other (most likely tedious) method must be used to find all of the uses of the field.

There are a few cognition-oriented studies of similar artifact use in software development—but not many. One of the most interesting is the study done by Bowdidge *et al.* [64], which investigated how maintainers reacted to different toolsets when performing remodularization (a maintenance task) on C

³Analyzing the changes to dependent software features, see e.g., Arnold *et al.* [16].

programs. In this study, they observed how the maintainers made strategic use of external resources in order to make their work easier and more systematic. By observing several teams of maintainers, they revealed some of the variations in “ecological expertise” that practiced maintainers bring to tasks. They noted:

Keeping track of the state of the overall restructuring task as well as the state of specific restructuring modifications—what might be called bookkeeping—is a crucial activity. Bookkeeping occurs at many levels in the process: completely performing a specific restructuring change, evaluating progress during a set of changes, and overall sequencing of restructuring activities. *Each team exploited structure* implicit in the tools (e.g., cursors) and the program representation (e.g., the ordering of lines in a file) to keep track of information regarding the current state of the activity. ... The *methods employed by the programmers vary widely*, although they share the underlying similarity of trying to achieve certain properties of completeness and consistency of a change. Each method of exploiting structure decreases the possibility of some class of oversights (e.g., missing a required change), but does not address others (e.g., formulating a flawed design), hence requiring additional integrity checks. In general, these tactics amount to maintaining to-do lists of data or design considerations that have yet to be processed. [64, pg. 224] (emphasis added)

This is a glimpse into the ecological expertise of maintainers. From these observations, the authors go on to more directly state a general rule of strategic use that has implications going far beyond their study’s context of modularization:

when faced with a complex design task, a programmer will use tool features in a way that allows the tool to store information conveniently for the programmer. Moreover, if the tool cannot conveniently store the information, the programmer will order subtasks in a way so that the information is immediately used and can then be forgotten. *ibid.*, pg. 225.

A variety of other studies complement this one. Flor and Hutchins [224] observed how paired software maintainers would strategically use the program correctness checking capabilities of their compiler to get feedback on their work, and to gain a sense of closure during incremental modification. Bellamy [39] studied the inventive use of Smalltalk browsers by experienced Smalltalk programmers. She observed them use a trick of creating dummy classes to perform category-based searches for reusable classes by exploiting reference-following capabilities of the browser.

It is interesting to note that these sorts of strategic and flexibly *ad hoc* uses of software development tools are frequently overlooked when evaluating the usefulness of software development tools (but see Lethbridge *et al.* [380,596]), and when studying cognition in software development. This raises a question regarding the proper evaluation of software tools. If one advantage of software tools is to employ them strategically in ways for which they are not overtly designed (e.g., using `grep` for dependency analysis and generally as a utility knife [380,590,696]) then how do you determine what tasks to evaluate them on? In addition, there is a risk of evaluating tools without acknowledging the burden of learning good strategies. For studying SE tools, the appropriate research slogan might be “don’t study how software developers think, but how they think *through tools*.”

3.1.2 External Memory and Internal Memory

Man has now many extra-corporeal members, which are of more importance to him than a good deal of his hair, or at any rate than his whiskers. His memory goes in his pocket-book.

– Samuel Butler, “Erewhon” [88], pg. 203.

The world remembers things for us, just by being there.

– Donald A. Norman, “Things that Make Us Smart” [472].

External memory generally refers to the storage of knowledge or mental states on artifacts (externally) rather than within internal memory.⁴ Schönplflug and Esser [569] recount the intriguing fact that Roman senators once used educated slaves to store appointments, to remember facts and figures, and to remind them of important points during conversations and arguments. They pointed out that there is fundamentally little difference between our modern use of memory devices like personal digital assistants (PDAs) or online databases, and these more ancient uses of external memory systems (apart from, of course, the enormous socio-political dimensions, and the underlying technology differences). In general, external memory devices are used in a variety of settings, and for a variety of purposes [326,472], from simple reminders stored on post-it notes, to the list of items kept on a shopping list, to a phone number written temporarily on the palm of one’s hand, to long-term reference works that systematically and permanently record knowledge.

Even though external memories have access costs [567] and use costs [569], they are employed because they can make up for our own mental memory limitations and processing shortcomings [325,469]. As any textbook on HCI is sure to point out, our own memory is often short-lived, poorly accessible, and effortful to update. External memories can help overcome these shortcomings [472]. In addition to the inherent cost of external memories, effectively using them often requires specialized or ritualized strategies [114].

Studies

Despite the obvious importance of external memory there is a general reluctance to study external memory on par with internal memory [325]. However the tides are slowly turning in favour of a balanced treatment on how both types of memory are utilized during activity [50,304]. A mixed bag of researchers have contributed to this overall mosaic of how individuals use external memory aids (e.g., Norman [472], Intons-Peterson [325,326], Schönplflug [567–569], Hunter [318], Card [92,95], and Davies [166,168,212]).

⁴Of course, there are conditions that must be met to be counted as external memory: not all artifacts will qualify [137], and not all representations can be said to be knowledge or mental state. These details will be discussed later.

External memory has been more extensively studied in the context of group, social, and cultural memories [318,320]. For instance “organizational memory” appears to have a relatively higher research profile [2].

Relevance to Software Development

Software development is memory intensive, so external memory use is, as one might expect, pervasive. Individual, personal external memory use is common. External memory can be found in something as simple as command line histories (e.g., Greenberg [275]) or browser histories (e.g., Tauscher *et al.* [633]). In these, the memory acts as a type of external *working memory* [95]. The study by Bowdidge *et al.* [64], already mentioned in the section on strategic artifact use, also provides good examples of the use of external working memories in the form of “bookkeeping”. External memory use can also be found in longer-term versions such as in developer diaries (e.g., Naur [438]) or scavenged code (e.g., Flor *et al.* [224]). Organizational memory also has a place, such as in code libraries and repositories (e.g., Retkowsky [534]), or for collecting other development- and design-related knowledge (e.g., Zimmermann *et al.* [728]). As a final example, Arunachalam *et al.* [17] studied how the use of external memories interacted with software comprehension and redocumentation processes. In their study, for instance, the act of externalizing memory was shown to have an influence on the degree to which the recorded material is understood, a fact noted in other research on external memory [325].

Because it is such a crucial resource in software development, models of development that can account for external memory use may turn out to be important for understanding developer behaviour. For instance, a characteristic of expert developers is that they often retain high-level knowledge of a piece of software rather than the low-level information that novices tend to develop [4,498]. One way of explaining this preference is that experts make extensive use of external memory instead of needing to remember the program details, and because of this they need small, efficient, and therefore high-level *index* knowledge in order to be able to retrieve from this external memory [569]. The observations by Singer *et al.* [595] regarding the use of a “just-in-time comprehension” strategy is consistent with this conjecture: a developer relying on externally stored knowledge would seem to “page in” this knowledge on an as-needed basis, but would still need index information to know where to look in order to page it in. Besides understanding software comprehension better, research on external memory has the potential to address the problem of developing computer support in many different ways, from the design of short-term external memory in browser histories to improving the effectiveness of long-term memory code repositories.

3.1.3 External Resources and Structure

... the user interface itself can stimulate and initiate cognitive activity ... a good user interface helps organise and direct cognition—it is not a passive receptacle for thoughts emanating from an internal model, but plays an active role in the problem solving process.

– Nardi and Znarmer, “Beyond Models and Metaphors” [437].

Most difficult tasks can accurately be called “ill-structured”: the properties of the solution are only partly understood at the start, no fixed or routine methods exist to solve them, and the criteria for success are only weakly defined. As a result, ill-structured problems inherently involve guesswork, backtracking, and reflection. And consequently one role that artifacts can play is to help organize and direct problem solving in ill-structured tasks. That is, they can help *structure* the ill-structured. Structuring problem solving inherently involves sharing control of the cognitive activity, at least to some degree. Simple examples of structuring artifacts are checklists, shopping lists, or written plans for action.⁵ These are resources that one can consult during an activity. The above are examples of rather “passive” structuring artifacts. A more “active” example is a “wizard” [210]—a software agent that guides the problem solver. Structuring resources can be pre-designed, or built by the problem solvers themselves. Solving ill-structured problems often requires the problem solvers to perform *problem structuring* (e.g., Goel *et al.* [244], de Vries *et al.* [172]), *problem discovery* (e.g., Carroll [103]), or *problem setting* (e.g., Gedenryd [235]). In this work, these activities will all be termed “problem structuring”. Problem structuring often yields artifacts (e.g., plans) that structure future action.

Studies and Literature

One of the classic works frequently cited to illustrate the capacity of artifacts to structure human behaviour is Simon’s parable of an ant walking on a beach [594]. In this parable, the actual path that the ant takes is to a great degree dependent on the shape of the dunes rather than some complicated problem solving mechanism within the ant. Simon used the analogy to argue that human behaviour can be characterized as similar sorts of structure following. However unlike the ant, humans think using two sources of structure: the outer environment and the inner.⁶

⁵Notice that the shopping list example came up again—it was also used as an example of an external memory. This is because a single artifact can exhibit more than one type of support, but also remember that the sections of this summary do not necessarily represent orthogonal notions of support. The issue of orthogonality of cognitive support types is visited in more depth in Section 6.5.

⁶Some have mistaken Simon’s intentions, suggesting he is arguing that human behaviour may be structured *more* by the external environment than the inner [395]. But this stretches his point somewhat: Simon was primarily likening human memory to the ant’s beach so that he could ignore the structuring influences of both memory and environment, and concentrate instead on modeling the (presumably) simple and generalizable cognitive processing that drives the navigation of these structures. Elsewhere in that same work, Simon argues that human adaptations to their environment means that many machinations of the mind will not be revealed by analyzing behaviour. This might also be seen as an argument that human action is conditioned primarily by external structures rather than internal ones. But in reality his argument is correctly seen as only limiting the sorts of validations possible on “inner” models (see e.g., Rasmussen [526, pg. 264]), barring some way of tapping into those inner processes (e.g., by protocol analysis [207]).

Generally speaking, the notion of activity structuring tends to crop up in at least three contexts: (1) when problem solvers do not have the appropriate knowledge needed to solve their problem, (2) when problem solvers do not wish to retain structuring representations internally, and (3) when problem solvers structure their own action by constructing artifacts or manipulating the environment. Examples of the first case include the design of wizards for novice application users [210] and the construction of learning environments for students [172] (see Section 3.3.2). An example of the second case is the “precomputation” performed in creating checklists [321]. Examples of the third case include the office organizing activities of Norman’s colleague, and the cooking organization done by Kirlik’s short order cooks (see Section 3.1.1).

Relevance to Software Engineering

Most software development activities—including reverse engineering [331]—are ill-structured. It should therefore likely surprise nobody to find widespread use of artifacts to structure software development processes. All of the three classes of structuring contexts described above will apply, and are discussed in turn.

1. *Delivering problem solving knowledge.* Providing structures for novices is relevant in software development: (1) when novices are learning software development techniques, (2) when expert developers are relative novices on a particular tool, and (3) when expert developers are novice to a domain or software system. Examples of these cases include the structuring of lessons to students learning data reverse engineering techniques [56], the provision of wizards to execute source code queries for engineers unfamiliar with tool capabilities [324], and the structuring of lessons for new immigrants to software projects [132].
2. *Pre-structuring.* One of the potential advantages of CASE tools that is regularly cited is the ability to enforce a particular development process (e.g., Brown *et al.* [82]). This type of constraint on action is one of the more active forms of structuring. Hutchins’ classic “precomputation” form of structuring using checklists is found in the case of software inspections [208]. Inspections often use checklists of defect types (e.g., Miller *et al.* [409]) to ensure a systematic evaluation. Of course, the checklists do not have to be so passive: tools can enforce attendance to the checklists to varying degrees [412].
3. *Self-built structuring artifacts.* The third type of structuring is perhaps the most well known in software development. Because development problems are so difficult, planning and problem structuring activities are common. It could be argued, for instance, that flow charts represent one of the earliest uses of external structuring mechanisms in programming. Flowcharts were used by developers to work out the form of a program (i.e., problem structuring) from which a program could be more-or-less directly transcribed (e.g., see Knuth [356], Sheil [581]). Once produced, the flowchart structures (but does not dictate) the coding process. Similar arguments can be made for the modern descendants of this technique, such as coding OO systems from UML diagrams. But problem structuring is not limited to only coding, it is desirable for any difficult task in software development. An excellent example of this is revealed by the previously cited work by Bowdidge *et al.* on software

restructuring [64]. They observed maintainers building and manipulating “to-do lists” which are used to structure their subsequent activities.

3.1.4 Reflective, Visual, and Intentional Thinking

Externalizations (1) create a record of our mental efforts, one that is “outside us” rather than vaguely in memory, and (2) represent artifacts that can talk back to us ... and form the basis for critique and negotiation.

– Arias *et al.*, “Transcending the Individual Human Mind” [15], pg. 88.

Reflective thinking is sometimes contrasted with experience-based or “rote” thinking [472]: it is described as thinking about one’s own activities, knowledge, or mental processes and strategies. Reflective thought seems important when considering situations that are in some way out of the ordinary [469, ch. 5], and therefore not amenable to highly practiced or routine thinking [564]. It has long been observed that artifacts can play important roles in such reflective thought [180]. One way of better understanding this role is to highlight the metaphor implied in the term “reflective”—that the artifacts reflect one’s own thoughts or actions much as a mirror reflects one’s image. In this sense artifacts can make the abstract world of thinking concrete, that is, they can “reify”⁷ mental constructs like ideas and process descriptions or models. By doing so, the artifacts encourage or facilitate reflective thought, making it potentially easier for one to think about one’s own thought or actions.

A phenomenon related to reflective thinking is what might be called “intentional thinking.” Intentional thinking involves consciously using learned ways of extending memory, solving problems, thinking creatively, or thinking critically. A familiar example is the technique of “brainstorming,” a “train-of-thought-following” method of writing down words and concepts as they are thought of, in the hopes of discovering unappreciated connections when structuring them later. Many of these intentional thinking techniques inherently make use of artifacts like drawings, charts, and checklists. Some are organized around trying to make use of visual and perceptual capabilities (e.g., McKim [404]).

Studies

Some of the main points of literature on reflective thinking will be covered later in Section 3.2.1, so for now let us concentrate on intentional thinking. There exists a *great* deal of non-academic trade press in the “self-help” and “pulp psychology” sections of many modern Western bookstores. Many of these books advertise that they can improve the mental work in such fields as management, marketing, and mathematics (e.g., Polya [513]), or claim to be able to improve creativity and idea development in creative writing and design (e.g., McKim [404], Flower [225]; see Allen [9, ch 1] for a small review). Although a few of these works are highly regarded, most of them are lean on scientific respectability and, for the most

⁷Reify: to regard (something abstract) as a material or concrete thing, *Merriam-Webster Online Encyclopedia*.

part, academic psychology has avoided these techniques and publications [121].⁸ Nonetheless, there has been considerable interest in a few of these sorts of techniques in contemporary education and educational psychology literature (e.g., Perkins [500], Dunson *et al.* [193]), especially constructivist educational literature (see also Section 3.3.2). Certain idea processors [121], like “concept mapping” programs [231], are perhaps the most widely known examples.

Relevance to Software Development

When confronting the cognitively challenging problems in software, developers will naturally reflect on their own thinking and attempt to self-consciously apply problem solving strategies. For instance, intentional strategies for problem decomposition and design are commonplace in software development. They are called a variety of names such as “analysis methods” or “engineering methodologies”. Classic examples include proposals to solve design problems by focusing first on persistent data, or on functional decomposition. The main implications for SE tools research are derived from the relationships between reflective and intentional thought, and the artifacts that support them. On one side of the coin are the properties of the tools that are needed to effectively enable reflection, and to make intentional strategies work well (e.g., Sugiyama *et al.* [624], Fischer *et al.* [216]). On the other side are the creation and teaching of the thinking methods that enable effective use of the tools available.

3.1.5 Evolving Structures, Emergent Thought

Complicated artifacts are often created incrementally and iteratively instead of by thinking things through completely beforehand. Writing a research paper is a familiar example. The final form of a research paper may resemble only superficially the writer’s original vision. During the process of writing the paper, the author might revise and edit the structure, flow, passages, and individual wording. Sections may be rewritten or deleted entirely. While writing, the author may think of new things to say. She may re-read what was already written in order to re-orient herself, and to inspire new passages. The paper is an artifact that is constructed piece by piece and grows incrementally but “non-monotonically”.

From one point of view, such a writing process is unstructured and inefficient: if only the paper could have been planned beforehand so that it could simply be typed letter for letter without having to make changes! An “ideal” planner [41] could “write” the paper in the head, hierarchically decomposing the topic into sections, and then simply transcribe [261] the results onto paper. Although such a process might be considered “optimal” it is almost never observed in practice for anything but the most trivial or ritualized construction problems. Instead, construction frequently consists of an incremental and interactive process of interleaving writing with reading and evaluating. Throughout the process, uncertainty, discovery, backtracking, and much updating of previous results occur.

⁸There are some interesting exceptions to this rule though: psychologists have studied certain mental tricks and techniques such as mnemonic coding techniques for memory and calculational shortcuts for arithmetic problems (e.g., Ericsson and Chase [206]). These have tended to be artifact-free methods that work on well-understood problems.

Recently, these sorts of problem-solving processes have all been characterized as variants of *design* [244]. Carroll and Rosson describe design as being

non-hierarchical, neither strictly bottom-up nor strictly top-down. ... [as] radically transformational, involving the development of partial and interim solutions which may ultimately play no role in the final design. Design intrinsically involves the discovery of new goals. [108, pg. 27]

Design problem solving is an activity that occurs in a broad spectrum of activities, including engineering design [514,661,662], paper writing [575], software design [103,283,663], programming [267] or software construction [488], redocumentation [17], and reverse engineering [139,331]. A number of reasons have been suggested for its prevalence: the limited planning capacity of human designers [267,662], the fact that goals (or “requirements”) frequently change during design [488], and the fact that, ultimately, designers must learn about the problem and solution and will therefore make poor decisions due to the presence of uncertainty, and the absence of foresight [79,488].

To return to the point of this section, though, design processes are ones where thought and artifact are intimately tied. Complex artifacts like sketches, documentation, specifications, and programs are normally created and iteratively evolved during design. The artifacts themselves become part of the designer’s environment and situation [627]. Designers make extensive use of this environment for evaluating current progress, for managing focus and attention, and for receiving ideas back in return (see Section 3.2.1 on “backtalk”). Design therefore involves a *feedback loop* where design moves impact the future. It is difficult to attribute causal roles in systems with such a feedback loop because of the circular causality. Brooks described this problem of circular causality when he said that “it is sometimes hard to point to one event or place within a system and say that is why some external action was manifested.” [70, pg. 572]. As a result, the designed artifact is seen to *emerge* from the iterated application of local changes which feed back into decision making—design is an *emergent* process (see e.g., Poon *et al.* [514]). The incrementally evolved artifacts play a crucial but complicated role in this emergent process. They partly function as external memories (alleviating the need to remember the partial form of the solution), they encourage reflective thought about one’s own progress, and they serve to partially structure further work. To say that the designer alone is responsible for the design would be to miss the contribution of artifacts: they are an inseparable part of the design process.

Before proceeding, a small point needs to be covered. It might be thought that certain abstract representations (e.g., UML diagrams [554]) can eliminate some of the backtracking involved, making the development process more structured. Clearly though, using such techniques merely changes the medium and form of the design. Instead of incrementally constructing a program more or less directly in the medium of a programming language, a planning stage is added in which the incremental and iterative process is that of developing the abstract representations (like UML diagrams). The point of adding this extra step is to take advantage of an environment where planning (especially exploring and backtracking from unpromising decisions) is less costly [40,575,709].

Studies

See Sections 3.2.1 and 3.3.2 below on reflective media and design studies.

Relevance to Software Development

The iterative and evolutionary process of artifact and design update has several implications for the design of supporting languages and environments, and for engineer training [64,257,283,284,331,544,663]. The design of such tools needs to take the characteristics of design processes into consideration. This is true for tools designed for software design [5,544], maintenance [64], redocumentation [17], reverse engineering [331], and even software comprehension (see Section 8.1). These tools need to take into account the emergence of solutions resulting from the cyclic interaction of developer and environment.

3.1.6 Representation Effect

It has long been observed that differing representations which are logically equivalent (according to some measure) can have remarkably different psychological implications, and be significantly different in their ease of use. A problem represented in one form may make certain inferences almost automatic, whereas a different representation of the same problem can make the problem incredibly difficult. Norman calls this phenomenon the “representational effect” [472], which “refers to the phenomenon that different isomorphic representations of a common formal structure can cause dramatically different cognitive behaviors.” [727, pg. 90]. A familiar example is the difference between a line graph and a table of values. A line graph can make it easy to read off certain information (e.g., finding a maximum value). Certain visualizations and displays also can make some features “pop out” [687]. But the representation effect is not solely concerned with fast inferencing, quick recognition, or rapid perception. The form of the representation can also help mental manipulations and problem solving. Norman gave the example of Arabic numerals [472], which facilitate multiplication much more than Roman numerals do. In addition, the design of a representation can make it possible to nearly simultaneously answer multiple questions by giving certain inferences “free rides” [31]: once one inference is made, the result of another is seemingly immediately apparent.

Studies

The representation effect is one of the most widely studied cognitive support-related phenomena. Sample overviews include Norman [472] and Scaife *et al.* [562].

Relevance to Software Development

Representation effects are one of the primary motivations for pursuing software visualization research. A further area of application is the design of textual or visual programming languages and environments.

3.1.7 Automation

Lest we forget, some problems are amenable to more or less completely automated solution through the processing capabilities of computers. These problems can involve mathematical manipulation, or symbolic manipulation such as inferencing, pattern matching, and search. Complete automation of any task represents the extreme case of cognitive support: when a task is automated, mental effort is absent.

3.2 Descriptive Theories

There is no denying that some people have important and valuable insights, both into human nature and into the design of technologies. But we should not confuse wisdom and insightfulness with having a scientific theory.

– Zenon W. Pylyshyn, “Some Remarks on the Theory-Practice Gap” [522], pg. 43.

Once it is realized that artifacts are important parts of human thought and culture, then attention frequently turns towards characterizing and describing the relationship between the two. Various theories are proposed—some only rough sketches, some more thoroughly developed. These theories are often primarily *descriptive* [522]: they introduce concepts and terms to describe the phenomena of thinking with artifacts (and its importance) without producing causal explanations of the mechanics of such thought.⁹ Sometimes these theories are developed by basic science researchers (e.g., psychologists) and find strong voices in certain domains of practice (like education). Other times the theories originate in those practice domains. Any of these theories could conceivably be applied to the various supportive phenomena noted in the previous section. For instance, the concept of *mediation* (Section 3.2.1) could conceivably be invoked to help understand external memory. The following short review surveys several such descriptive theories, and the domains in which they are frequently found. Table 3.1 summarizes these. In the table, column one lists entities from this section, column two from Section 3.1, and three from Section 3.3.

CHARACTERIZATIONS AND DESCRIPTIVE THEORIES		
THEORIES / CONCEPTS	ASSOCIATED SUPPORT RELATIONSHIPS	COMMON STUDY DOMAINS
Schön's reflective practitioner	reflective thinking, emergent thought, strategic artifact use	software design, education, writing
Activity Theory (mediation)	embodiment, intentional thought	education
augmentation / symbiosis	external memory, structure, automation	writing / hypertext
fitness	representation effect, visual thought	programming, decision making

Table 3.1: Descriptive theories for various cognitive support relationships

⁹In this admittedly brief review of these theories I do not mean to imply that they are without analytic power, or that research is not underway to more fully elaborate the models of causation. Neither of these things are true. The point is, like behavioural or gestalt psychology before the emergence of cognitive science, most of the discussed schools have not developed strong mechanical explanations, and this section tries make a distinction between mechanistic theories and these others.

3.2.1 Mediation and Reflective Media

If thought products and processes can be represented externally, it seems natural to say that artifacts are *media* for thought much as clay is a medium for sculpture, or newspaper is a medium for communication. If external artifacts structure and control activity, one might say that they *mediate* the thinking processes of the artifact users. The themes of mediation and reflective thinking with external artifacts have been woven into diverse schools of thought for nearly a century [563]. A few instances are noted here.

The notion of reflective media can be found in the work of Donald Schön in his work on designer thinking and design education [564,566]. He describes design as a “conversation with materials,” where designers work out parts of the solution using external media and models, and that these then provide “backtalk” to the practitioner. This explanation relies on the idea that the practitioner engages in distinct “modes” of thought [389]. In the course of design, the designer frequently makes use of her expertise and skill to be able to act with a minimum amount of reflection, often making decisions and gaining insight fluidly and with little (conscious) effort. These activities can be said to evoke “skilled” or “practiced” responses, and the designer may not even know how she knows how to do these things. Nevertheless, in challenging design settings, this fluid design activity is inevitably interrupted when poorly understood or surprising evidence is encountered. Such “breakdowns” in skilled execution cause designers to reflect upon their actions. In such circumstances the designers enter a different “mode” of thought in which they use artifacts to help them reflect on their own actions and on the design problem. The interaction with artifacts is portrayed figuratively as a conversation—a conversation with materials. Schön’s work is quite influential in the “design studies” or “design science” world as a way of understanding the practice of industrial designers, architects, and software designers and developers (e.g., [130,216,544,566,638,650]). Schön’s work, especially his later work on designer education [565], has also found a strong following in education research, including the design of learning environments and educational materials (e.g., Lehrer [379]).

Mediation is also an important concept in the approach called “Activity Theory” [142], a school of sociocultural psychology that traces its history to Soviet psychologists, particularly Vygotsky [203,435]. Activity Theory has a reputation¹⁰ for being difficult to understand by many Western thinkers [52], but a taste of some of the ideas is enough for our present purposes. In Activity Theory¹¹ tools and artifacts are understood to stand, metaphorically speaking, between a thinker (the subject) and the thing being thought about (the object). This interposition of tools mediates the activities and, what is more, the mediation can radically change the very nature of the activities engaged in. One especially relevant notion in Activity Theory is the idea of a “functional organ”—a joint system composed of a human working in close combination with artifacts. For example, eyes and glasses can form a functional organ that produces sharp vision [339]. Activity Theory is of interest here because it is often posed as an alternative theoretical framework in HCI [52,103,339,366,616], particularly as an alternative to DC [367,434].

¹⁰Well deserved!

¹¹“Activity Theory” is written capitalized. The acronym CHAT for “cultural–historical activity theory” is now also used [393].

3.2.2 Scaffolding

“Scaffolding” is an evocative term that can be used to understand the roles of artifacts or other people or agents in helping people think or learn. The metaphorical reference is to a temporary rigging during construction. The rigging makes it possible to build a structure that would fall down without the scaffolding, but after being built it is strong enough to stand without it. The roots of scaffolding, as it is frequently used now (see e.g., Palincsar [483], Soloway *et al.* [612], Goldman *et al.* [247]), can be traced back to so-called “constructivist” and socio-cultural education theory and psychology [483]. The idea of scaffolding is commonly applied to artifacts such as idea processors, learning environments [329], simulations (or “micro-worlds”), and other writing tools that help establish thinking processes and enable learning (e.g., [107, 246, 247]).

The meaning of the term “scaffolding” has sometimes been expanded to include practically any external structure that can be strategically used to perform tasks [136, pg. 46]. The concept is thus extremely versatile, if not precisely defined. Jackson *et al.* [329] defined three classes of scaffolding: “supportive” scaffolding, “reflective” scaffolding, and “intrinsic” scaffolding. The exact definitions do not especially concern us here, but each of these three types has close analogues with other concepts in this review of cognitive support. For instance the notion of reflective scaffolding echoes the “reflective” media of the previous subsection. When used in a very general manner, scaffolding is very similar to “support”, but it adds certain wrinkles. In particular, the issue of learning while using scaffolding is critical, and the idea is added that scaffolds can “fade”, that is, be removed after learning is accomplished.

3.2.3 Augmentation, Extension, and Symbiosis

The idea of computers augmenting humans comes primarily from Bush [87], who brought the term into computing, and from Engelbart [201], who carried the torch of augmentation.¹² The basic ideas are far older than either of these pioneers, however, since they have been used to understand physical or mechanical augmentation for centuries [515] (see Butler’s shovel on page 57). The concept of augmentation has enjoyed some popularity in the distant computing past (e.g., see the 1964 Symposium on “Computer Augmentation of Human Reasoning” [560]). The main idea underlying augmentation is that basic human capabilities can be added to (augmented) by computer capabilities. So to take Bush’s Memex [87] as an example, computers can augment human memory by being storehouses of knowledge. Thus, like the way that telescopes augment vision and thus make it possible to see farther than with unaugmented vision, computers can augment human memory and make it possible to remember better and think better. Extension (as in “extended memory”) is essentially the same idea, but it may carry an additional connotation that the extension can apply to existing human capabilities only—not to complementary ones.

Licklider (who worked with Engelbart at SRI [199]) proposed that computers should be related to human activity in terms of symbiont systems [385]. In Licklider’s view, humans and computers coordinate closely to bring their different talents into play during complicated problem solving. Licklider attempted to distance the idea of symbiosis from the notion of augmentation by suggesting that the relationship

¹²See the “Bootstrap Institute” homepage, www.bootstrap.org, or Rheingold [537].

between computers and humans is more involved than simply adding memory to human capabilities. In particular, he claimed that some of the thinking and problem solving is taken over by the computer in symbiont systems. For instance, AI programs might perform pattern matching or suggest alternatives to problem solvers. But his distinction may be superficial rather than fundamental: the difference between augmentation and symbiosis is that in symbiosis human thinking capabilities (cognitive processing) are being augmented *in addition* to human memory capabilities. Later authors have noted this possibility; for instance, it seemingly corresponds to Perkins' "distribution of the executive function" [501]. Besides augmenting thought, the other connotation of symbiosis not shared by augmentation is of the mutually beneficial co-evolution of the symbionts (see also Butler [88] and Mackay [394]). But this distinction seems to be unimportant with regard to explaining how cognitive support works.

The notions of augmentation, extension, and symbiosis are all powerful and evocative terms describing the ways of aiding human thinking and problem solving. They are therefore useful for envisioning possibilities and setting goals. Several more modern threads of work on human-machine cooperation can trace their ideas back to these roots [65, 124, 306]. But although they are highly descriptive, they are *just* that: descriptive, not generative or predictive. Engelbart was well aware of this limitation, and he carefully cast some of his main writings as a "conceptual framework" intended to outline more of a research paradigm rather than to generate specific tool ideas and requirements [199]. What is really needed by SE research, though, is explanatory, predictive, and generative theories. The authors promoting the above sorts of perspectives provided no models of exactly how augmentation, extension, and symbiosis might actually work, and how they should be built. Consequently only the smallest illumination is cast by them on the mechanics of how artifact supported thinking works.

3.2.4 Fitness

... the notion that fitting the representation to the task is beneficial for problem-solving is not a new one ... However, although the idea may even seem intuitively obvious, it has little explanatory or predictive power when expressed at this level of abstraction and must therefore be defined in operational terms.

– Judith Good, "The Right Tool for the Task" [249], pg. 91.

Some artifacts are better than others for certain tasks or in certain circumstances. This relationship can be named and described without necessarily explaining it. We can call it "fitness". The notion of fitness can be invoked to describe the suitability of an artifact with respect to some aspect of its intended use context (see Section 7.1.1). That is, tools need to fit their ecology of use. So, for instance, one may hear that one tool is "fit" for expert programmer use when performing impact analysis, but that it is "unfit" for the novice performing program restructuring.

Naturally, the definition of fitness is as complicated as the environments in which tools are found. A tool's fitness will depend upon its users, their work environment, their work activities, and so on. In other words, all of those things that affect performance and behaviour (see e.g., Storey *et al.* [619], Basili [33],

or Teasley [635]) will have a stake in the tool's fitness. It should therefore not be a surprise that the notion of fitness has been applied to tools in various ways. Examples include the notion of fitness to task [187, 188, 399, 596], fitness to the psychology of the user [124, 469, 473, 689], fitness for purpose [237], cognitive fitness [601, 610, 656], and fitness to the work system [530, 657]. Similar concepts are called "congruence" by Gilmore [239] and Good [249], and "match-mismatch" by Green and Gilmore [240]. If one should identify support with fitness, then cognitive support becomes cognitive fitness.

3.3 Schools Of Cognitive Support

In their everyday lives, people seldom undertake complex reasoning or decision-making tasks without turning to pencil and paper, yet in the scientist's laboratory, how many subjects have been permitted such luxuries? What explains this state of affairs? There seems to exist an unspoken argument underlying traditional approaches to cognitive skill: only by looking at unaided skill can we see the mind in action. If we let people use tools, like pencil and paper, then our view of the mind will be confused by contingencies and circumstances that are outside the scope of psychology. But this argument is surely fallacious. ... You can't realize the crucial role of locomotion in vision unless you let your subjects move, and you can't realize the crucial role of artefacts in cognition unless you let your subjects use them.

– Stephen J. Payne,

"On Mental Models and Cognitive Artefacts" [493], pg. 105.

The relationships between artifact and human thought and action has been a research focus for certain groups of people. Sometimes research cliques form around a particular problem domain (education, architecture, etc.) and sometimes they form because they are studying some particular phenomena in a common way but without specific domain or problem orientation (short term memory, social norms, etc.). We might say these latter have a *research paradigm, school, or tradition*, whereas we might say the former have a common *problem type*. Here are a few of the salient clusters of cognitive support research are surveyed and grouped according to tradition and problem type.

3.3.1 By Research Tradition

Cognitive support is a subject that can potentially apply to any problem domain involving thinking and problem solving. Consequently, it is possible to develop research paradigms to study cognitive support in a domain-independent way. As a result, it is studied by various schools (or sub-schools) of psychology, and to a lesser extent other schools like social psychology, sociology, cognitive anthropology, linguistics, and so on. Some sort of research tradition—a set of beliefs and commonly understood ways of doing research—must be adopted to study cognitive support no matter which school the researchers belong to, and no matter which particular methods (e.g., ethnography versus laboratory study) are used.

In reviewing these research traditions it is exceedingly difficult (and not worthwhile) to identify strict boundaries between traditions, so I shall be understandably brusque in my categorizations. Because later chapters be labouring under the umbrella of DC, the following review starts with traditions linked most closely to DC, and then works outwards. The main focus shall be on matters close at hand—that is, work that relates well to cognitive science in the context of artifact use in real-world situations like software development. There are four reasonably distinct clusters of activity closely tied to DC: traditional cognitive science, socially distributed cognition, situated or embodied action, and ecological psychology.

Traditional Cognitive Science

From a functional viewpoint, the STM [short term memory] should be defined, not as an internal memory, but as the combination of (1) the internal STM and (2) the part of the visual display that is in the subject's foveal view ...

– Newell and Simon, “Human Problem Solving” [449], (1972).

The research paradigm that might reasonably be called “traditional” cognitive science tends to centre its research around cognitive models with high-level symbolic representations and centralized processing. Newell and Simon’s seminal work “Human Problem Solving” [449] is paradigmatic. Contrary to what is sometimes claimed, in such work, external representations are often crucial to the explanation of problem solving and performance. For example, within this school it is possible to find work where intermediate results are viewed as being stored externally during algebra problem solving. To explain cognition in these contexts, the memorial capacity of the external world is relied upon. So even in the most “traditional” cognitive science memory is distributed between head and world. The distribution is often marginalized [465], perhaps, but it is there nonetheless. A number of relevant works have continued in this traditional vein and have still managed to take into account some of the impact of artifacts on cognition. Examples of this sort of work include the foundational work by Larkin *et al.* on display-based problem solving [374] and diagrammatic efficiency [115, 375], the work by Card *et al.* [94] on HCI modeling, and the work by Brooks [71–73, 76] on program writing and comprehension.

In most of this sort of work, the external world is explicitly (but simply) treated as an external memory whose currently perceivable contents can appear in working memory. The treatment of the mechanisms involved is frequently simplistic: items stored externally will appear “automatically” in internal memory (as if by magic) when the relevant portions of artifacts are in the field of view and being attended to (c.f. Altmann [10]). This type of treatment of the external world was adopted explicitly by Brooks [71] in his pioneering work on modeling programmers:

The third major [internal] knowledge structure ... is actually information about how to access an external memory, the code that the programmer has already written. It is quite likely that very little of the actual code remains accessible in [internal memory] once it has been written out on paper; when the subject in this study wanted to rewrite or reuse pieces of code, longer than a line or so, that he had already written, he was almost never able to recall them directly from memory. Any use, modification or correction to

code which has been written must therefore retrieve the code from the paper external memory; and the [internal memory] must contain the information necessary to perform the retrieval. [71, pg. 880]

Brooks' models made explicit use of external artifacts in the explanation of thinking. His basic ideas were rediscovered and extended a decade later by Green *et al.* [267]. Brooks' model also suggested how the emerging solution (Section 3.1.5) can structure thought (Section 3.1.3) by implying that a programmer's sub-goal generation is driven at least partially by the external representation rather than "via [an internal] goal stack" [71]. Following chapters expand on these ideas, but the point is that work that can easily be classified as traditional cognitive science has significantly anticipated some of the later work on DC and "situated action" (below). It should also be noted that cognitive science has evolved over time, as all research traditions do. More recently, aspects of task and artifact use have become more central issues in the descendents of traditional cognitive science (see e.g., the review by Gray *et al.* [253]).

Socially Distributed Cognition (SDC)

The second cluster of cognitive science that concerns itself with external artifacts is DC. It is not possible or appropriate to review DC adequately in this section, especially since the main ideas will be covered in depth later. Thus only the sort of DC work that is not well covered later is highlighted here. One type is called "socially distributed cognition" (SDC) to emphasize the social and group aspects of cognition (e.g., Salomon's collection [558]). SDC research as a whole is loosely connected by an underlying conviction that human intelligence and human behaviour is always embedded in a context of collaboration or partnership with others, and hence it has an underlying social and cultural content. For instance, it is a common conviction within SDC that agreements upon meanings are socially arrived at ("socially constructed") through interaction and communication. One of SDC's particular interests in artifacts is that they are viewed as conveyances of knowledge (e.g., Pea [495]), and that they mediate social and organizational activity (e.g., Hutchins [320], Perry [502]). A small glimpse of these ideas has been brought into the world of software development by several researchers [223, 224, 690].

Closely related to socially distributed cognition is research on what is sometimes called "shared intelligence", "collective intelligence", or "group cognition". It has long been recognized that people in groups behave and perform differently than they do individually (e.g., Norman [465]). In fact, Newell's influential "blackboard" model [442] of individual problem solving was inspired by how groups of experts can come together and solve problems in a way that the individuals could not on their own (see Section 5.3). The choice of the term "shared intelligence" exposes an interest in how intelligent behaviour is not purely a function of an individual's cognition, and in the ways in which tasks and problems are shared [65]; the terms "collective intelligence" and "group cognition" imply an emphasis on how individual intelligences combine in a group, and on the structure and dynamics of such group thought processes. Many of the authors from Salomon's collection [558] have come from these sorts of research backgrounds, so the link between that prior research and SDC is quite well established. In the context of building supportive artifacts authors such as Boy [65] have used these ideas to understand the application of intelligent assistants. Others, like Smith [603], have considered how to support collaborative work using groupware.

Situated/Embodied Action

The third cluster revolves around so-called “non-symbolic” or “anti-representational” explanations of “situated action” or “embodied” behaviour. “Situated” refers primarily to the fact that behaviour is contingent on the particulars of the situation in which an actor finds herself [657]. Important claims in this cluster are that (1) behaviour is *ad hoc* (as opposed to “purely rational”) in fundamental ways [623], and that (2) action that appears globally coherent can result from local interactions rather than being a result of a “global plan” [135,136,623]. At a less technical level another important contention is that the actual competencies and behaviours of humans are intimately connected to the “socioculturally constituted contexts in which they are embedded” [377, pg. 6]. In other words, the particular “situated” condition of interest is an agent’s social and cultural setting.

The situatedness of cognition is itself simply a phenomenon that any psychology must address [270, 657], and hence many different schools try do so. In fact, sometimes situated action viewpoints can bear remarkable similarities to other viewpoints such as DC [417]. Nonetheless there are circles within the situated action (and other) schools that takes situatedness as a challenge to certain modeling traditions (see e.g., Thagard [639, ch. 10]). The details of this controversy are not that important here (see e.g., Clancey [133] or the special issue on Cognitive Science [471] for some lively discussion), but some situated action theorists have pursued a modeling tradition which avoids a core reliance on directly representative symbol systems (e.g., of plans for action) and centralized processing.¹³ Frequently their models instead emphasize how global properties *emerge* from local interactions, and they often highlight intimate interaction between the external and internal world. Prominent figures in this loosely connected school include Suchman [623], Greeno [277], Clancey [134], and Clark [136]. Little work from the situated action camp has been applied in software development, although some attempts have been made (see e.g., Law [377], Shukla *et al.* [586]).

Ecological Psychology

The fourth cluster of related research is gathered around what is termed “ecological psychology”, a tradition pioneered by J. J. Gibson [236]. Many of the ecological concerns that captured Gibson’s attention are familiar also to many other strains of psychological study, but certain Gibsonian ideas might be said to be endemic to the ecological psychology cluster. Gibson’s work appears to require several concepts to fit together in order to gel properly [221]. One is the idea of “direct perception” where information is “picked up” directly rather than by involving information-processing steps (computation) [726]. Another needed concept of great concern here is the notion of *affordance*. To Gibson, an affordance is a (realist) property of objects that relates to the perceived or actual capabilities of those artifacts to have actions performed with them. For instance, buttons afford pushing; chairs afford sitting. Affordances of artifacts are human- and task-relative—a single artifact can afford different actions to different people. A screwdriver affords driving screws, but it also (poorly) affords driving nails. Human action is consequently related to artifacts

¹³It is worth noting that these are all still computational models of symbol processing but they are significantly different in character (see also Section 4.1.3).

by the artifacts' affordances and how they relate to the tasks being performed. Because of its way of relating artifact, task, and behaviour, Gibson's work is influential in certain circles, particularly the field of "cognitive engineering" (see Section 3.3.2).

3.3.2 By Problem Domain

Research within any problem domain may be directed towards cognitive support whenever it is realized that artifacts are important parts of the mental lives of people in that domain. Any of the different research traditions mentioned in the previous section might be subsequently employed within that domain.

Design

Many educational and research institutions maintain a separate faculty for *design*, which often includes graphic design, architectural design, and engineering design (e.g., Hubka *et al.* [316]). Sometimes this list of design professionals is extended to include software designers. Some of the ideas in design theory have trickled down into software design either somewhat directly (see e.g., reviews by Terrins-Rudge *et al.* [638] and McPhee [406]) or indirectly (e.g., the influence of Christopher Alexander on design patterns [148,232]). The design community as a whole is philosophically connected by a shared premise that there is an underlying common activity to many different forms of design. The research movement began in earnest in 1967 with the founding of the Design Research Society (<http://www.drs.org.uk/>) after a watershed conference on the possibility of creating a science of design. The first edition of Simon's influential "Sciences of the Artificial" [594] came out soon afterwards, and it helped speed progress in the field. Within this problem domain, the main concern is how people design things, either routinely or creatively.

There are several potential crossover points between design studies and research on cognitive support in software development. First, there are topic similarities. The design field shares with software development an interest in processes and methods for designing things, and in the artifacts and computational support for these processes. For instance they are interested in how things like sketches and models are used during the different phases of design (envisionment, creative design, etc.) [235, 517, 628, 764]. Second, there are methodological similarities. Many studies in the design field are closely reminiscent of studies of software design, programming, and maintenance. For instance, protocol analysis studies in this field bear remarkable similarities to many from software comprehension and reverse engineering (e.g., Suwa *et al.* [627], Purcell *et al.* [518]). In addition many of the ways of researching tool support are similar. For example, some authors [38] have used protocol analysis to try to discover what information designers attend to during design, in order to determine requirements for information management tools. This tactic is remarkably similar to the "information needs" analysis of software comprehension made by von Mayrhauser *et al.* [666,669]. Third, there are definitely opportunities for sharing theoretical content. Some of the ideas from design-related authors such as Christopher Alexander, Herbert A. Simon, and Donald Schön have made their way into computing science theorizing (see e.g., Robbins *et al.* [544], Guindon *et al.* [285], Schön *et al.* [566]). A more direct mixing occurs when authors who publish in design literature (Visser, Hoc, Guindon, Davies, etc.) publish also in journals common to both communities (e.g., IJMMS [661]), or when they publish also in software psychology related conferences and books (e.g., the

Empirical Studies of Programmers Workshops [745]). In addition, design research may utilize the same science base as computing. For example, design may begin to use DC theories (e.g., Gedenryd [235], Perry [504]).

Cognitive Engineering

Cognitive engineering is something of a mixture between human factors, HCI, and industrial engineering [713]. It can be viewed as an offshoot from industrial human factors and industrial psychology—that is concerned with high-level cognitive issues in industrial contexts. These contexts typically constitute complicated socio-technical control systems such as aircraft control, and industrial or nuclear plant control. Researchers in this area are concerned with the design of human-machine systems, and are therefore self-consciously interested in the notion of joint system performance (e.g., Hollnagle *et al.* [312]). For example, typical assumptions in this domain are that the total work is distributed between humans and machines, and that an important design goal is to *allocate functions* to machinery in ways that reduce cognitive overheads of the operators [66]. There is a great deal of relatively mature research work in this field concerning how to analyze and design cognitively-intensive socio-technical systems. For example, see the volumes by Vicente [657], Flach *et al.* [219], Hancock [287], and Rasmussen *et al.* [531].

Many of the ideas floating around in the community seem applicable to designing software development tools (with some modifications, perhaps), yet they have never been applied to this field. For instance, in ecological interface design [531] there is a well articulated theory of how to provide support for cognitive work via the provision of a so-called abstraction hierarchy. This is a testable hypothesis about the usefulness of certain representational forms and these forms seem applicable in software visualization. Some of the theoretical resources from this field are actually integrated into the HASTI framework in Chapter 5 (the SRK taxonomy, see Section 5.4).

Management and Decision Making

Management and decision making is concerned with the products of, and processes involved in, making informed decisions (about purchases, manufacturing decisions, etc.). Artifacts like tables and decision charts have been important resources for such decision making. Thus this area has contributed to the long stream of research on textual and graphical representation effects [332, 656]. Some of this research has already influenced Rasmussen's SRK taxonomy in cognitive engineering, so there may also be potential for points of contact with software development too.

Education

Education research is interested in improving both teaching and learning. One way of viewing it, therefore, is as a *design science* concerned with improving and facilitating cognitive performance and problem solving. Thus it shares some common ground with applied sciences of cognitive support, and with engineering disciplines built around them. Even more so than cognitive engineering, it has a long history of taking theories and applying them in practice as in instructional design research (e.g., see Tennyson *et al.* [637]). For instance, many of the theoretical schools surveyed here are also applied to technologies

for education (see e.g., the special issues of *Instructional Science* on multimedia [8] and metacognition [291], and various conference [80, 684, 760, 761]). Perhaps it should also be noted that cognitive psychology and educational psychology are rather distinct disciplines with their own journals and conferences. This might have something to do with the design-oriented focus of educational psychologists—is there a nascent analogue for software engineering psychology?

Potential crossover points include: (1) studies on what makes for good representations and media for problem solving and learning, (2) studies on tutoring systems and interactive learning environments, (3) studies on training of methods for metacognition or reflective thinking in order to instill skills in problem solving, and (4) possibilities of theory transfer or cross-pollination. On the first point, education researchers have one of the longest histories of research into the benefits of diagrammatic representations and visualizations (see e.g., Paige *et al.* [482]). Although educational researchers have in the past recognized the importance of external media in education, research on the use of external media seems currently vigorous (e.g., Mathewson [398]). Some of this research has had some time to mature, and there may be some hope to transfer some relevant research and cognitive models from studies of reading to software comprehension and software visualization. Likely examples include studies on diagram reading strategies [708], and studies on memory for spatial location of passages [131]. The second point argues that many psychologically-motivated design ideas for computational support have been investigated by educational researchers (e.g., Anderson *et al.* [14], Vosniadou [683], Salomon [559]), and that many of these ideas for learner support (or scaffolding) might conceivably be transformed into analogous concepts of SE support. The third point indicates that some research in education is quite directly relevant to cognitive support in SE: workers in SE frequently need formal training or need to learn on the job in order to enable effective use of novel technologies (e.g., see Lemut *et al.* [759]). In fact, training and technology deployment must frequently go hand-in-hand (see Rasmussen *et al.* [531] for a cogent argument). The fourth point suggests that there may be a great deal of theoretical groundwork in common. Certainly there are many authors working within the computer-based learning field that have related theoretical roots (e.g., Salomon [558], Dillenbourg [180, 181]).

Reading and Writing

Reading and writing research exists semi-independently from related areas like education, computer documentation, and hypertext. Reading researchers are frequently concerned with how people read and understand texts. Writing researchers are concerned with the problem of how to make writing of all sorts (stories, reports, etc.) easier and better. Because of the well-known importance of intermediate representations (outlines, sketches, notes, annotations, etc.) in writing, this aspect of performance is not neglected in writing research. Much like design science, reading and writing research often involves observational studies and verbal protocol work that are similar to those performed in software development tool research. Several recent volumes on writing environments highlight some of the building maturation in the field [313, 383, 577, 652].

There are several possible points of relation between reading and writing research, and cognitive support research in software development. Perhaps the most obvious is the fact that text comprehension research has been influential in software comprehension research (e.g., Pennington [498] and Soloway

et al. [609]). A similar cross-fertilization concerning cognitive models of planning might also be feasible. For instance many models of writing planning (e.g., Hayes *et al.* [293]) strongly resemble programming planning models, which are in turn consistent with the modeling framework presented in Chapter 5. There are two other points of contact that are more specifically related to cognitive support: (1) there has been significant research into the roles of diagrams in text comprehension, and (2) there is some maturing research that has attempted to organize explanations of how artifacts aid writing processes. Some of the best examples of these are the bodies of research from Sharples, Pemberton, and colleagues [496, 573–576], and from Smith *et al.* [603–605].

Hypertext, Information Retrieval, and Library Sciences

Books like those of Marchionini [396], Allen [9], Dillon [183], and collected works like those of Rouet *et al.* [552] are testaments to the increasing understanding within these fields that some of the main problems they face involve cognitive support issues, especially relating to how users perform iterative and media-manipulating interaction with tools. In the past, these fields have had a strong interest in the underlying technology (non-linear representation of documents, document query and retrieval, collections management). But for over 30 years [561] there has been an awareness that on “a fundamental level, information retrieval is inherently interactive.” [561, pg. 1067]. After a watershed workshop on the problems of interfaces in IR in 1971 [561], interest began to grow regarding the importance of overall task concerns (navigation problems, sensemaking and iterative querying, etc.), and regarding the roles of artifacts and their manipulation. There is an increasing awareness of the need to understand user cognition and problem solving, and to subsequently relate them to task environments in order to develop corresponding requirements for tool features. A good example is the Stanford Digital Libraries project, which gave birth to reflective media like the “scatter/gather” interface, and iterative problem solving models such as sensemaking models [301]. Another excellent example is the work by Tweedie *et al.* on “Interactive Visualization Artifacts” [646] and *The Attribute Explorer* [614].

There is certainly a direct point of contact to software development: software developers often use hypertext (e.g., [213, 234, 248, 729]) and information retrieval (e.g., Consens *et al.* [146], Clarke *et al.* [138]) tools. Clearly, one would expect that models of program browsing will need to be highly compatible with models of hypertext browsing if hypertext-like browsers are used to browse programs. Indeed, it may be relatively easy to establish a case for this. For instance, the ASK (Anomalous States of Knowledge) model of information seeking within hypertext (see Marchionini [396]) shares much intellectual ground with Letovsky’s model of knowledge seeking within programs [382]. In addition, many of Storey’s cognitive design issues for developing software comprehension tools were derived from a similar framework in hypermedia [618]. Moreover, Dillon’s TIMS framework [184] (Task, Information, Manipulation, Standard Reading) is an attempt to build theoretical resources for hypertext design, and is in many ways similar in spirit to the present work. Thus hypertext and IR research should be seen as having significant overlap with program comprehension research.

Extraordinary HCI

People are sometimes confronted with lesser or diminished mental capacity. Diminished capacity can result from congenital disability, but also from accident, aging, or more temporary conditions such as fatigue or extreme stress and duress [451]. Whatever the reason for the diminished capacity, there exists the motivation to furnish *prosthetic* mental capabilities: augmented memory, inferencing, vigilance, and so on. The design field concerned with this support is called “extraordinary HCI”. Of course, all humans have limited mental capacity. Thus cognitive support for the mentally challenged is not *fundamentally* different than cognitive support for the most capable, but of course there are the special conditions that make it a genuinely distinct strand of HCI.

The contributions to SE research that might be made by extraordinary HCI could be to a great extent symbolic, but it also could be substantial. The symbolic contributions are perhaps the most readily appreciated. In my personal experience, I have found that for some reason it is frequently difficult for the uninitiated to fully appreciate the cognitive support provided by artifacts, especially simple ones. Support is often overlooked. Perhaps there is a tendency to assume people can do amazing things like build million-line programs simply through mental capacity and discipline alone. The fact that many of these feats would be frankly impossible without external artifacts seems to be frequently glossed over. Cognitive support for the cognitively challenged helps put this faith in the power of human brilliance into perspective. There is something convincing about how simple computer tools can help people with profound memory and cognitive disabilities perform tasks like write cheques and organize daily living [141]. These are activities that just *could not* be done without the prosthesis. Thus, perhaps extraordinary HCI can be of symbolic use in helping to appreciate the importance of cognitive support. The more substantial contribution that might be made by extraordinary HCI concerns the unique disabilities of some of the subjects studied. In experiments with able subjects, it is frequently difficult to determine the support offered by tools since subjects can often make up for lack of support by substituting mental effort or special mental tricks. In extraordinary HCI, however, one can sometimes study people with well established limitations (e.g., zero capacity to remember newly presented information after 30 minutes have elapsed [141]). These well-known limitations may be useful to experimenters much in the same way that studying subjects with various head injuries has helped research on brain function and neurophysiology.

3.4 Summary and Conclusions

Sometimes obvious things have to be repeated over and over before they are realized.

– Stanislaw Ulam, “Adventures of a Mathematician” [649].

This chapter started out with relatively modest goals—to (1) provide a global overview of the types of phenomena that can be considered cognitive support, and the clusters of research work that have contributed to an understanding of them, and (2) to help relate these concepts to SE tools research in order to raise awareness of these issues within the community, and to indicate the potential applicability and

relevance to our research. The survey covered seven distinctive clusters of support concepts, four descriptive theoretical schools for understanding the support concepts, four explanatory theoretical schools that might be brought to bear on understanding cognitive support, and seven quite independent research communities that have a stake in understanding cognitive support, or have histories of trying to do so. The survey reveals that cognitive support is multi-faceted, and that research on it is widely scattered.

To conclude the chapter, it is worthwhile standing back and reflecting on what the juxtaposition of so many different strands of research reveals. One thing that is hopefully quite obvious is that without good and integrative reviews of such materials it is difficult for any SE researcher to have a good global understanding of cognitive support. With examples and discussions of cognitive support so widely scattered, it is unrealistic to suggest that tool researchers—or psychologists or sociologists for that matter—be aware of all the possibilities and flavours of cognitive support, and of their applications in practice. The consequences are predictable. We are destined to rediscover the same facts over and over. And useful techniques and ideas will remain unused through their inaccessibility. I find it simultaneously amazing and incredibly frustrating that research in so many disciplines and problem domains can contain so many unbeknownst similarities regarding cognitive support. Although I have tried to pepper this review with adequate citations, I have left out as many as I have included. And to this day I am astonished at how frequently I discover more papers that essentially replicate similar insights from publications past.

In closing this chapter allow me to point out just one example. Two decades ago, McKim, himself an engineering design professor, wrote a non-academic book about “Visual Thinking” [404]. It was intended as a sort of cookbook for thinking better and was, in fact, subtitled as “a Strategy Manual for Problem Solving”. In it, he noted that people tend to manipulate the world in order to “externalize thinking”, that is, artifacts are used as a *medium for thought*. He then noted some design implications for such media:

Externalized thinking is best accomplished with materials that are easy to form and reform. The sculptor’s Styrofoam, the chemist’s snap-together elements, the designer’s Foamcore and tape all have the virtue of being easily manipulated spatially, much as symbols and images are moved and modified internally in mental space. Materials used to communicate a visual idea that is already formed need not be as flexible. [404, pg. 44].

Similar sorts of observations are sure to have been made in many places. The two essential implications for design noted above are that (1) humans like to think with external media, and (2) if a medium for external thought is being designed, it must be easily manipulable—ideally it should be as fluid as internal human thought. These are relatively simple observations, but they link aspects of cognitive processes and problem solving to desirable properties of supportive artifacts. Observations so straightforward should, one might argue, be incorporated into some type of standard design theory or wisdom. And indeed they have been. Green’s cognitive dimension of *viscosity* [258], for example, nicely captures the need for easy manipulation in this context. Green actually includes an excellent analysis of some of the psychology underlying this cognitive dimension; thus he has incorporated viscosity into something of a broad-brush psychological design theory. A sequence of several papers and presentations have served to expand on the framework, and to extoll its virtues [54, 257, 258, 266, 269, 270, 272].

But from one point of view it is surprising that cognitive dimensions such as viscosity should even need to be argued in so many papers. Indeed, it is surprising that a well-respected cognitive psychologist should have to be the one to introduce this little theory to the software development community, since viscosity obviously has many precedents. Should we not have noted these things (and many others) before? It is, moreover, vexing that many of the psychological observations needed are ones that are either rather “obvious”, or have been known for centuries [454]. Take, for instance, Norman’s popular book “The Psychology of Everyday Things” [469]. It is very well received, contains scores of psychological facts that are extremely relevant to design, and is a standard reading for many undergraduate HCI courses. Yet Pylyshyn argued that:

After all, it did not require knowledge of any psychological theory, or even training in psychological research, to make the sorts of insightful psychological observations of practical foibles that are contained in Don Norman’s *Psychology of Everyday Things!* [522, pg. 43]

At the conclusion of this chapter I find it surprising and perhaps a little puzzling that the basic concepts of cognitive support were not long ago pieced together and brought into loose but functional theories for designing computer tools. Instead, focused articulation of one type of support or another has been favoured over reconciliation and synthesis with other forms. More recently, of course, they are beginning to come together (see Section 6.6), but the progress is so slow. This chapter has highlighted a number of cognitive support concepts, and I think that just shoving these together onto the page can be enormously beneficial. Certainly, a good and more thorough review of cognitive support concepts should help tease apart the issues, organize the concepts, and resurrect yet others from their undeserved exiles in publication purgatory.

Chapter 4

Strengthening the Foundations of Cognitive Support with RODS

Because the framework is not too informal, it can appeal to the clean-living software engineer; and because it does not suppress too much mess, it does not appal the cognitive psychologist.

– T.R.G. Green on his Cognitive Dimensions framework, in “Why Software Engineers Don’t Listen to What Psychologists Don’t Tell Them Anyway” [263], pg. 332.

Sometimes, it is the general, high-level, qualitative theories that make the biggest impact on a field. When this happens it is often because they capture fundamental principles, and thus provide deep and general insights which can be as important as fine details. This trend was noted by Newell and Simon [450] in their 1975 ACM Turing Award Lecture. They observed that the essential characteristics of a discipline can often be stated in short, general sentences. Although any field will have its share of theories explaining specific phenomena, the details are often overshadowed by the “big picture”. Newell and Simon highlighted, in particular, the importance of the cell doctrine in biology, the theory of plate techtonics in geology, and the germ theory of disease. These are all gross qualitative theories which are critical for understanding a domain. They tie together, relate, and organize multitudes of facts.

Consider the theory of plate techtonics, for instance. The theory argues, essentially, that the Earth’s crust is made up of plates that are moving, growing, and pressing up against each other. Stated thusly, it is a simple theory, but it helps explain a whole host of facts, from fossil records to earthquakes. Stated so simply, however, the theory can make few specific predictions or explanations; the details eventually count. But the non-predictive and qualitative nature of such theories does not diminish their importance. General, qualitative theories can be enormously valuable in understanding a broad range of phenomena from up high.

One important function of such high-level theories is that they often provide the first firm foothold for non-specialists. The theory collects and neatly packages together the key insights, which the non-specialist can effectively cling to. Beyond such a theory, the abyss of detail awaits. But not only is the theory an initial foothold, it is really a stepping stone to deeper and more specific theories. One can look to the experiences of typical science students in many countries for evidence to support this view. By the time they are in High School (around age 15), they are usually exposed to all of the qualitative theories listed by Newell and Simon. This does not qualify them as *bona fide* biologists or geologists, but they have the basic conceptual tools needed to be able to understand key aspects of geology, biology, and so on. If they need deeper conceptual tools, these can be learned, with the stepping stone of the qualitative theory easing the transition. However the main point is that the high-level qualitative theories are very widely accepted as being vital and general knowledge. They are general, but not too general to be useful. We also normally think that it is valuable for our children to learn many of these theories, and to be able to apply them to make sense of the world; they are considered part of a well-rounded education.

For similar reasons, a clearly articulated, general, high-level, qualitative theory of cognitive support is likely to be invaluable to developers of software for cognitive work domains. In SE, for instance, a wide variety of tools are proposed in order to reduce the mental challenges of software development. One would expect, therefore, that the developers of these tools would be greatly helped if they understood and could apply the basic psychological principles of supporting cognition. The specific details about cognitive support will be, of course, important at some point, but arguably not as important knowing the core principles. Moreover, the core principles will undoubtedly be the foundation onto which more detailed theories are built. Should the details be needed by the developer—and this is by no means guaranteed—then specialized theories and principles can be assimilated and applied. In any case, it seems prudent to expect that the general theories should be included in the list of things a well-educated tool developer should know.

Unfortunately, such a simple, comprehensive, qualitative theory of cognitive support is not yet clearly perceivable from the literature. The aim of this chapter is to take steps to rectify this. The *form* of the resulting theory is important. The goal is to produce a theory that satisfies the desiderata outlined in Section 2.3.1. This means making sure that the theory is based on mechanistic explanations, explains work equivalence, and so on (see Table 2.4, page 62). Judging from the qualitative theories enumerated by Newell and Simon, we should be able to recognize a suitable initial theory by its simplicity (i.e., storable in a few sentences), and by its ability to help make sense of a host of related facts and theories. The aim of providing a broad survey in Chapter 3 was, in fact, to set our aim widely enough. Thus we will know when we succeed if our theory is simple and yet helps explain all of (or at least much of) the supportive phenomena identified in Chapter 3.

The existing science base makes the extraction of such a theory of cognitive support a non-trivial exercise. The phenomena of cognitive support is generally inadequately understood, and we have to date studied it using a menagerie of limited and incompatible theoretical approaches. In facing this theoretical jumble, perhaps the most sensible course of action is to limit confusion by picking an approach that appears to have good practical promise. If better theoretical frameworks appear later, then we can update what we have already built, or put our frameworks aside and build new ones. This approach is much like

adopting an industry standard when developing software. Nobody is entirely satisfied with the standard, but most agree upon the importance of adopting one. This pragmatic strategy helps side step some of the controversies that rage on in cognitive science and psychology [270]. Once the standard is picked, key principles can be extracted and highlighted as a general, qualitative theory of cognitive support. This can then act as a framework onto which other prior results and theories can be fit.

Distributed cognition (DC) is the overall theoretical approach chosen in this dissertation. DC is a relatively new and developing area of cognitive science, but it seems to hold high practical promise. Several researchers have recently adopted DC as an umbrella approach to HCI research and design (e.g., Wright *et al.* [719], Dillenbourg *et al.* [181], Rogers *et al.* [547], Holland *et al.* [311]). DC appears eminently suitable as a starting point because its scope of enquiry is taken to be cognitive systems composed of humans in combination with the artifacts they use. Artifacts are thus not seen as something peripheral to the cognitive machinery—they are *part of it*. Thus the contributions of artifacts are woven directly into models and theories of cognition.

Using DC as a starting point still leaves us with issues to resolve, however. The fact is, DC is a theory of *cognition*, not a theory of *cognitive support*. A cognitive theory or model—distributed or not—does not by itself guarantee that the means of supporting that cognition are clearly spelled out. Any theory of cognitive support must provide an explanation of the benefits that artifacts provide. This analysis of benefit is necessarily comparative: the benefits of a tool may be understood only in comparison to what is implied by its absence, substitution, or modification. Cognitive support theories therefore compare various cognitive systems to determine their relative benefits. It is the *benefit explanation* that distinguishes theories of *cognitive support* from theories of *cognition*. Theories of cognition explain cognitive phenomena like mental constraints, forms of internal representation, learning, and performance. Given a class of cognitive systems, a theory of cognition can be used to explain or predict those sorts of aspects of that class of systems. Loosely speaking, the “output” of a cognitive theory is thus a prediction or explanation of cognitive phenomena or behaviour, whereas the “output” of a cognitive support theory is an explanation of cognitive benefits provided by some class of artifacts (as compared to some other class). Existing DC theory embeds a kind of cognitive support theory within its description, but it is (unintentionally) concealed and must be flushed out.

Fortunately, it is easy to convert the basics of DC theory into a simple and high-level, qualitative theory of cognitive support. The key idea for doing so was described on page 5 in Chapter 1:

The cognitive support provided by a tool is the computational advantages that the tool provides. Cognitive support can therefore be understood entirely in computational terms: support is the provision of computational advantage.

Notice that this is clearly not a theory of *cognition*, but one trying to explain *support*. A theory of cognition seems obviously necessary to make such a statement, but it is a logically distinct type of theory. Note also that the phrase “providing computational advantage” in this statement is comparative: it means that the computations involved when using one artifact are better in some way than when another is used. Stated in this way, the above blurb qualifies as a high-level cognitive support theory—too high level, unfortunately. If one can imagine a dial that sets the level of detail in a theory, then such a theory

threatens to set the dial too high to be useful [265]. Some details must be added to the statement to let analysis proceed. We wish to lower the detail dial, but not too much.

The dial can be lowered by addressing two questions which quickly arise after some reflection. First, what are the “computational advantages” that are mentioned? Once their existence is mentioned, it seems natural to ask for the complete list. Second, how do we go about determining these advantages? If tools differ according to the support they provide, it is natural to ask for a method for analyzing these differences. This involves knowing how the support is actually implemented by the artifacts (i.e., how the cognitive support principles apply in HCI), and how these supportive mechanisms compare. At first it might not be obvious how to do this. Tools are frequently compared only according to their surface features (widgets used, user-interface metaphors used, etc.) and their over functionality (store, edit, graph, analyze, etc.). In contrast, what we need to be able to do is compare artifacts based on *what they do for the cognition of their users*. Thus although a simple theory can be offered up front, what is really needed is a slightly more complicated one that can follow up with useful (but still relatively simple) answers to these two questions.

RODS is a theory and analysis framework proposed to fit these needs. The core part of RODS is a list of four fundamental principles of computational advantage. These are actually ordinary notions of computational efficiency familiar to computing scientists. These principles are called “task reduction”, “algorithmic optimization”, “distribution”, and “specialization”. The four principles are collectively referred to using the acronym RODS. Because these principles are so central, the entire cognitive support framework defined in this chapter is also called “RODS”.¹

RODS is intended to strengthen the foundations for understanding cognitive support in SE. Adopting this goal has meant that the form and qualities of RODS are engineered specifically according to several criteria. First, it was designed to try to fulfill the desiderata of Section 2.3.1, as was mentioned above. Second, it was specifically constructed with the needs of traditional software designers in mind. These needs are discussed in Section 7.2. Finally, RODS was designed to try to define the cognitive support principles using orthogonal computational ideas. This is a point that is important to other sections of this dissertation, but the discussion is best delayed until Section 6.5. For the time being, these design issues are put aside, and the issue of what constitutes RODS is considered instead.

The chapter thus proceeds as follows. First, the main DC tenets are collected and described in Section 4.1. This review is necessary, in part, because the theory is unfamiliar in SE circles, and it will help to relate the theory to the domain. It is also needed because there exists considerable variability in how DC is treated, so the various convictions used herein must be stated. Second, the four cognitive support principles of RODS are introduced in Section 4.2. Third, ways of analyzing tools for support are described in Section 4.3. Fourth, RODS is evaluated in terms of how it matches the theory desiderata which were outlined in Section 2.3.1. Finally, Section 4.5 summarizes RODS and makes conclusions about its construction.

¹This naming scheme is similar that of Card *et al.* [94], who called their entire analytic framework “GOMS” based on the key cognitive elements in the framework (Goals, Operators, Methods, and Selection Rules). Properly speaking, though, both GOMS and RODS refer to focal parts of the analytic frameworks in which they are embedded.

4.1 DC Principles and Tenets

It is impossible to say what constitutes the DC viewpoint since there is so much variability in the field. Not only in DC *per se*, but in cognitive science more generally. Several sorts of research with differing underlying methods and philosophies fly the flag of DC [417]. It is therefore important to state what basic assumptions are being made [417]. The goal of this section is to present a simplified version of the basic DC convictions. The aim is to make the characterization suitable for SE research, and to relate important DC principles to SE issues. These tenets are necessary for understanding the RODS framework as a whole, and are required to appreciate how RODS can be applied. A small, core set of commonly assumed convictions are singled out. These are chosen such that they enable the theory design in subsequent sections. Each of these is discussed in separate sections below.

For each tenet, an attempt is made to supply representative links to the supporting literature. Also, links to SE works are given where possible. In addition, implications for RODS and its design are noted where appropriate. The presentation attempts to distinguish more-or-less independent or “atomic” tenets. These are divided into two groups consisting of (1) “cognitivist” modeling principles, and (2) their corollaries in DC terms. They are labelled for easy reference using a simple naming scheme. A table of these tenets appears in Table 4.1. The table uses arrows (‘→’) between columns to illustrate the fact that the “standard” cognitivist convictions all have their DC analogues.

A note about these DC tenets needs to be made in advance. Since all of these tenets are fundamental and reasonably widely held, the following may seem at times to be a review of rather well-known concepts and ideas. It may therefore at times seem that they are not worth repeating. Nevertheless, none of these tenets can be said to be universally held—indeed, some are currently hotly contested. As a result, this review has the seemingly dubious dual properties of being both review material and (by certain accounts) incorrect. But it is important to be explicit about the fundamental convictions underlying the overall framework. The explicitness is bound to be beneficial to researchers new to DC. Moreover clarifying the basic assumptions helps other researchers from similar fields understand the relationships to their own worlds [493]. It is quite worthwhile, therefore, to risk the relatively benign hazard of tedium in listing these convictions.

#	COGNITIVIST		#	DISTRIBUTED
C0	human mind is a cognitive unit	→	D0	distributed functional unit
C1	cognition = computation	→	D1	cognition = distributed computation
C2	cognitive interpretation	→	D2	external cognitive interpretation

Table 4.1: Key tenets of DC

4.1.1 C0: Human Mind is a Cognitive Unit

The traditional assumption in cognitive science is that the thing to be studied and modeled is the cognitive functioning of the human mind.

4.1.2 C1: Cognition = Computation

One of the most fundamental principles of cognitive science is that cognition is adequately modeled as a type of *computation*, where computation consists of operations over symbols (cf., Pylyshyn [520], Newell [446]). Explaining cognition as computation has the advantage that a mechanical explanation of the cause of behaviour is advanced. It is difficult to overstate the importance of this advantage [520, 523]. RODS takes this lesson to heart: it attempts to explain cognitive support in purely computational terms. If there is a cardinal rule in RODS, it is that support, if it is to be properly explained, will be explained using computational models, and by appealing to the fundamental principles of computation.

Software Comprehension and SE

Computational models have been a *de facto* cognitive modeling technique in SE domains since the late 1970s and early 1980s [156, 481]. Cognitive models have slowly begun to offer some explanations as to the causes of programmer behaviour. These explanations have offered advantages over “black-box” techniques, which were unable to properly uncover these causes [581].

4.1.3 C2: Cognitive Interpretation

If cognition is modeled as computation, then the simple corollary is that certain computational mechanisms can be interpreted in cognitive terms [521]. In particular, it is possible to talk about how *data* corresponds to knowledge or mental state, and how *computational processing* correspond to perception, attention, thinking, deliberation, reasoning, and so on.

It is important to note that viewing data as knowledge and processing as thinking is an interpretive act. In this work, computational models are assumed to be data-processing systems—systems of purely syntactic manipulation of symbols having no inherent meaning.² So if a model posits that the concepts of “cat” and “house” are somehow encoded, it means that some syntactic entities are being interpreted by the analyst in terms of these semantically meaningful concepts. *At some level*, therefore, it is fruitful to talk about cognition being the manipulation of meaningful symbols. This does *not* mean, however, that one can pry open a subject’s braincase and see tiny inscriptions of the words “cat” and “house”. Nevertheless, an adequate treatment of cognitive support will require it to be possible (in principle) to map cognitive concepts like *goals* and *plans* to their implementation. There is no guarantee that this mapping will be obvious in any way. Cognitive-level concepts like *goal* or *plan* may be implemented ephemerally much like *centre of gravity* or *temperature*: one cannot directly point to either of these yet it may be perfectly reasonable to build models of reality based on them. The importance of these points shall become more apparent in conviction D2, and in Chapter 5.

²This terminology accords well with Turing’s original understanding of the term “symbol”, however in cognitive science, a “symbol” is normally understood as a syntactic element within a *symbol system*, so that its semantic content can be interpreted [117, 446]. This other definition reasonably leads to such notions like “sub-symbolic” computation [97, 606] which, using the purely syntactic notion of “symbol”, is nonsensical. In Section 4.1.6 the reason for presenting cognitive models as an interpretation of data processing will be made more clear.

4.1.4 D0: Distributed Functional Unit is a Cognitive Unit

Cognitive models are built to explain and predict behaviour. Models will be able to do this insofar as they model the mechanisms contributing to it. In *well bounded* systems, these mechanisms have only weak connections to external entities. Consequently, well bounded systems form distinguishable units that can be analyzed relatively independently. Such units often can be modeled as independent computational subsystems with well understood ways of interacting with their embedding contexts. We could say that these independent bundles of causal influences are identifiable as subsystems or “modules”; we may then say that they have *low external coupling* [613]. *Strict* modularity is not found very often in real systems [317], but many systems are still loosely connected enough to be termed “modular” and studied as separate entities [446, 592].

One modular subsystem is the human mind. It is typically assumed to be generated by the brain and bounded physically by the skull. The mind forms the most traditional unit of analysis in cognitive science (conviction C0). However, recently the sufficiency of this traditional unit of analysis has been challenged; DC represents one approach to expand the unit of analysis. An example of where the traditional “unaided mind” unit breaks down is in group interaction. People working in groups can form larger combined systems such that the behaviour of both the individuals and the group as a whole are dependent upon the relationships and interactions between individuals (e.g., Norman [465]). That is, the subsystems combine into larger systems and it is these that become the expanded unit of analysis. Duke *et al.* [192] called such coupled units “*syndetic*” units—a reference to the fact that the subsystems within the unit are rather tightly bound (the term “*syndetic*” comes from the Greek word *syndetikos*, meaning to bind together). In DC, several examples of syndetic units have been studied. Examples include cockpits of airplanes [321], navigation teams on ships [320], and pairs of maintainers sharing a single computer [224]. Humans also form syndetic units with artifacts such as PDAs and, significantly, software development tools and environments.

The key criterion for identifying syndetic units is the closeness of the interaction between subsystems [137], that is, by the strength of the coupling between entities. This coupling is not based on physical or temporal³ proximity, but by being causally or *functionally related* [311]. Consequently, Hutchins uses the term “functional unit” [320] to refer to syndetic units, but a variety of other terms have also been used. For instance, Activity Theory has a similar concept of “functional organ” (see e.g., Kaptelinin [339]), and the term “ecosocial system” has been applied in the context of socially situated cognition (see e.g., Wortham [715]). It is not imperative to settle on a single term, but in this work they will generally be referred to as “joint cognitive systems” when speaking of (single user) human–computer syndesis, or simply “cognitive systems” when it is clear that the systems in question are distributed ones.

4.1.5 D1: Cognition = Distributed Computation

The conviction that cognition be modeled computationally (C1), combined with the conviction that the unit of analysis is a joint system (D0) implies that the joint cognitive systems be modeled as distributed

³For instance one may not return to a marked passage in a book for a long while [454], yet the marked passage can still qualify as an external memory.

computational systems. Hutchins' work on ship navigation is perhaps the best exemplar of this view [320]. In that body of work extended groups of actors and artifacts jointly participate to perform such navigational computations as fixing the position of the ship [319,320]. Cognitive models in DC are simply distributed computation models; all existing computation modeling techniques used in cognitive science can (potentially, at least) therefore be imported and applied. Sometimes different terms are applied to emphasize cognitive implications. For instance, in traditional cognitive science the standard analogue for computation is "symbol processing" to emphasize the semantic level of the processing. Also, in the DC literature, one finds references to the manipulation and "propagation of representational state" [320].

4.1.6 D2: External Cognitive Interpretation

The conviction that computational systems can be interpreted in cognitive terms (C2), combined with the conviction that artifacts can be part of a joint system (D0) implies that it should be possible to interpret artifacts in cognitive terms. Thus we should be able to speak about artifacts in terms of how they function as memories and how they store knowledge. We should also be able to think of artifacts in terms of their participation in reasoning, planning, and problem solving. That is, it is possible to view artifacts through three different sets of lenses: (1) the *cognitivist* lens, which reveals the cognitive functions of the artifacts, (2) the computational lens that reveals the "implementation" of cognitive system in a computational system [320], and (3) the "physical" lens, which shows how computational functions are actually manifested in real-world objects. For instance a printed checklist can be viewed as a *plan* (cognitive), a list (computational), and a body of text on a printed page.

Notice that each mapping between levels generates an abstraction boundary that allows substitutions to occur without affecting higher levels. For instance, the same cognitive function could occur if the printed checklist is implemented in a PDA; the same is true if the plan is represented as (say) a set of actions *A* and a set of ordering constraints between elements of *A*. This point will become important when considering how to compare cognitive systems in Section 4.3.

As with the non-distributed case, the way that cognitive content is encoded may be arcane. For example, pending goals may be represented (in part) using the position of a hamburger [348], or a piece of string [325]. Just as it is not possible to peer inside the head to see inscriptions of "cat" and "house", the cognitive roles of artifacts may not be labelled for our convenience. But, as Hutchins pointed out [320], at least artifacts are much more readily available for inspection than are the inner workings of the brain.

Several different streams of research take care to explicitly interpret artifacts in cognitive terms. The best known approach is the AI approach that begins with the argument that computers may implement (artificial) cognitive systems. That is, the data and processing of an AI system are referred to in cognitivist terms such as "mental states", "knowledge", "inferencing", "reasoning", and so on. In this approach, the interpretation at the cognitive level is *overt*, that is, explicit, openly acknowledged, and reasonably easy to follow. But one can consider less explicitly intelligent artifacts and yet still employ cognitive interpretations to understand how they participate in human thinking and problem solving. In these latter approaches, the cognitive interpretation is often implicit, or *tacit*. Examples of both the overt and tacit approaches are given below.

Overtly Intelligent: AI, DAI, Agents, and DC/AI

The so-called “good, old-fashioned” [320] AI approach to cognitive support generally tries to build relatively autonomous and intelligent programs to assist humans. AI research is thus concerned with how to represent knowledge, how to efficiently implement inferencing, and how to realize various problem solving methods. In many AI efforts, the AI developers use explicitly cognitive-level notations and tools, like formal ontologies [697], knowledge representation languages [179], inferencing algorithms, and so on. These manifestly cognitive-level formalisms help ensure that the analyst’s cognitive-level understanding of knowledge, inferencing, etc. are translated faithfully into computational implementations. A prime example is the Programmer’s Apprentice (PA) project [539]. The PA project aimed to build intelligent assistants that could be smart enough to take over some of the relatively menial coding work that programmers have to do. The PA project utilized a formalism for representing knowledge in the form of *plans* [538]. Thus the mapping from the analysts’ understanding (plans) to implementation (strings of bits) is both explicit and rather straightforward.

“Dumber” Artifacts?

A rather different strain of research studies artifacts that are less explicitly or intentionally intelligent, but still ascribes cognitive-level interpretations to them. In AI-oriented work, the clear intent is to have complicated computer programs behave intelligently and autonomously. Thus when “dumber” artifacts like paper and pen are involved, the cognitivist interpretation of such material artifacts tends to be significantly muted. Nonetheless one may still speak of the roles of these “dumb” artifacts in joint cognitive terms. Norman, in fact, calls them “cognitive artifacts” [470, 472, 493]. Even “ordinary” artifacts that people interact with can be thought of *as part of* the cognitive machinery of the individual. This view of artifacts is often substantially different, qualitatively, from more overtly cognitive approaches as exemplified by AI research. After all, there is nothing magical about implementing AI computations in a digital computer. The exact same computations could be “implemented” (at least in principle) “by hand” with paper and pen. Nevertheless, it can take some mental gymnastics on the part of the analyst to understand a shopping list or a string tied around the finger in terms of a cognitive state. But the world of the seemingly ordinary frequently finds close company with the cognitive.

An example of this sort of interpretation in software development comes from Flor and Hutchins [224]. They described code scavenging behaviour as knowledge reuse. They interpreted external artifacts as embodying knowledge, and the manipulation of these artifacts as a form of knowledge manipulation. Specifically, they proposed the following mapping:

code base	↔	(episodic) knowledge base
grabbing old code	↔	knowledge retrieval
hacking out unneeded parts	↔	schematic abstraction
substituting in particulars	↔	schema instantiation

As it is typically done in cognitive science, one can roughly separate the cognitive interpretations of artifacts into mental representations and cognitive processes. The mental representation brand of this research focuses on how humans extend themselves with external representations of knowledge and

mental states. For instance one brand of research examines the ways in which knowledge exists “in the world” [469,471]. A prime example from SE research are reified or externalized goals (e.g., Singley *et al.* [599], Green *et al.* [267]). Essentially, all of the varieties of mental representations from traditional cognitive science (goals, plans, concepts, etc.) may have their analogues in external representations.

Cognitive interpretations can also be made of external processes, even if they are not overtly considered cognitive processes. These might be “ordinary” computational processes that take place in computers, or they could be any other sort of process. Even non-electronic processes can be interpreted computationally and cognitively, a point made clear by the work on navigational computations by Hutchins [320]. The issue is not whether the processes are done by electronic computers, but that they occur as part of a single joint cognitive process. There are at least two interesting ways in which this can occur: artifacts can process mental states externalized by humans, and humans can manipulate the externalized mental states themselves.

Manipulation by computer. Computer manipulation of externalized mental state is perhaps the more familiar of the two cases of external cognition. This form of cognitive processing ranges from simple calculation to complicated symbolic processing. An example of the former is a spreadsheet program calculating expected profits for a financial analyst exploring various forecasts. An example of the latter is an automated theorem prover being employed by a software developer to check whether a program matches its specification. Often there is a bias towards interpreting only the “symbolic” processing cognitively (as if math symbols are not symbols). However any processing that would have to otherwise occur in the head of the analyst may count as part of a distributed cognitive process.

Manipulation by a user. The other brand of cognitive processing associated with cognitive artifacts involves more direct manipulation of artifacts by users. In this type of external processing, the artifacts function as a *medium* [224] for cognitive states. Then, much as clay is a medium for manipulating sculpture, artifacts can serve as a medium for mental states which are manipulated by the user. This is a distributed version of internal (mental) manipulation—the manipulated medium is merely external, rather than internal. Many classic examples come from mathematics (abacus, slide rule, long division on paper, etc.). Another variant is the work on “epistemic actions” by Kirsh *et al.* [353] and Kirlik [349]. To put it roughly, an epistemic action is an action on the world that functions much as a cognitive process would (e.g., an inference). There are many other variants of such cognitive processing by manipulating external media. Scaife *et al.* [562] called these types of external processing “external cognition”, and they supplied an overview of many different instances. The main point being made here is that DC provides an important twist on understanding external cognition: it models external cognition as ordinary DC processing. From the DC point of view, the “external” artifacts being manipulated are not external at all! Instead, they are to be viewed merely as parts of a joint cognitive system.

Notes on Interpreting Artifacts in Cognitive Terms

Before continuing, some preliminary remarks must be made regarding the forms and realities of representations and other external manifestations of data. The topics of data, information, knowledge, and

so on are very tricky and *exceptionally* thorny. There is *no* hope of satisfactorily resolving all of the controversies before continuing. But even though the controversies cannot be resolved, their impact can be blunted somewhat by clarifying in advance what is being assumed in the following (see e.g., Hayes [294]). The goal is to provide definitions and disclaimers in advance so that when controversial topics are raised the discussion is not paralyzed by a blizzard of counterpoints. Toward this goal, five issues concerning cognitive interpretations of the external world need to be briefly touched on.

First, some notes must be made regarding the implementation or encoding of semantic content. The terms “data”, “processing”, “mental state” and “knowledge” will be used in a loose manner. There may ultimately be great value in pinning down what these terms mean, but using the commonly assumed definitions are precise enough for this work. Also, the terms “information”, “signal” and “sign” are generally avoided. The reason is simple: “data” and “processing” are comfortable terms to computing scientists, and they suffice for the current purposes. This decision goes against some long-standing traditions such as the use of the term “information processing” in cognitive science. But it is worthwhile targeting computing scientists (see Section 7.2). Also, the more neutral computing terminology may help avoid some of the thorny issues that surround other terms like “information”, “signal”, and “sign”. For similar reasons the term “knowledge” will often be treated as a cognitivist analogue of “data”. These decisions may turn out to be inadequate in the long term, but my experience so far have been that distinctions not captured by RODS has not hindered further analysis. Moreover, in my interactions with CS people, I quickly found it to be beneficial to speak in terms of the “lowest common denominator” of data processing.

Second, the notion of equivalence between implementations must be somehow addressed. This work will essentially adopt the assumptions introduced by Zhang and Norman [727]. They introduced a general scheme for understanding distributed manifestations of data within DC systems. Their scheme is essentially an expansion of Simon’s concept of *information equivalence* in representations [375,593]. Their view establishes a way of understanding functionally or *informationally equivalent* manifestations of data. For instance, in their view a problem constraint may be equivalently manifested as: (1) an explicit rule (in the head or on paper) that needs deliberate interpretation, (2) a logical constraint that is easily checked by visual processes, or (3) an implicit physical constraint that is observed by necessity.⁴ This particular trio of implementation types is very reminiscent of the scheme Rasmussen proposed for distinguishing between “signal”, “sign”, and “symbol” encodings [526]. However delving into these distinctions is not productive here. In this work, it shall be enough to remember that there can be many equivalent implementations of data *and* computing process; no implementation is considered privileged.

Third, representation. “Representation” is a slippery concept. One popular way of defining “representation” is in terms of a *representing* relationship—as in a representation *represents* something else (e.g., Norman [555]). However sometimes it seems that some authors are happy to call practically any stimulus a “representation” without special concern with what is being represented (e.g., see Zhang’s [726] definition). Such a definition is closer in spirit to the terms “information” or “data”. If there is nothing to distinguish between “representation” and “data” then perhaps my preference for the more generic term “data” is further justified. For instance consider the case of a program that can be used to track a to-do list.

⁴The complications seem to pile endlessly on top of each other: one may wish to distinguish between “real” physical constraints and simulated ones [474].

If the to-do list is out of date, then can it really be said to represent the list of items to do? Either answer creates its own set of problems. Perhaps more serious is the confusion created when the logical and physical aspects of representations are intermixed. That is, sometimes the term “representation” is used to refer to both the objects manifesting the representing relationship, and the properties of the objects themselves. This confusion is, unfortunately, all too possible since it is quite easy to confound a perceivable depiction of a representation and the representation itself. When discussing the use and design of artifacts, such confusions can be troubling. For instance, consider the to do list program again. Is the output of the list-displaying program the “representation” of the to do list? The program might be able to display the list in a variety of ways—formatted, coloured, filtered, elided, etc. Is it not therefore better to call the array of bytes held in the computer’s memory “the representation” rather than some transient display of it? Or what if just the screen layout changes, or a magnifying glass is used? Is the representation actually the glowing phosphors on the CRT, or the image formed on the viewer’s retina? Some limiting definitions need to be made for progress to occur. In this work, I will gloss over many of the various nuances and consider just two issues: the manifestation of the data (somehow) in the computer or other artifact, and the presentation or way of making it known to the user. The former is henceforth called “representation”, and the latter, “presentation”. In this way “representation” is more-or-less identified with “data” or “data structure”, and “presentation” is equated with ways of accessing the data. This stance parallels the tactic taken by the comparable framework defined by Wright *et al.* [719].

Fourth, the fact that data or data structures may be *distributed* can complicate matters. It is possible for any non-atomic data structure (i.e., most of them?) to be distributed amongst different processing elements. The most obviously relevant case is when a data structure is distributed between human and computer. For instance while a programmer is writing out code, part of it will be written out already, and part of it will still be stuck inside the head. Zhang *et al.* [727] and Green *et al.* [268] give other excellent examples in which data structures are so distributed. In the present work, the choice of examples will tend towards those cases where the data structures are more completely external. This is just because such simplified cases tend to make good examples. Bear in mind, however, that in most practical instances some distribution of data between human and artifact may be involved.

Fifth, duplication. Just because an artifact is interpreted as being an external memory for some data, it does not mean that the same data may not also be held internally. Copies of an external memory may be cached internally. Conversely, just because a user maintains an internal copy, it does not mean that an external memory holding a duplicate copy is not a memory. Users can forget things.

4.1.7 Summary of DC Tenets

DC theory provides a viewpoint for conceptualizing cognition in terms that permit artifacts to be full-fledged members of cognitive systems. Cognition is treated as a type of computation, and this computation is spread out between humans and artifacts. This world view adds an important layer of understanding to artifacts: the cognitive interpretation layer. Thus artifacts can be viewed in terms of how they function as memories and processors, and how they can store and process knowledge or mental state. Because of this conceptualization, DC theory can provide explanations of complicated cognitive systems

such as airline cockpits [321] and paired software maintainers [224]. This world view makes it possible to discuss the contributions of artifacts in cognitive terms. What is left to do, however, is to pick out and enunciate the principles of how beneficial artifacts support cognition. This is done in the following section.

4.2 RODS: Computational Principles of Cognitive Support

Artifacts can reduce the cognitive challenges during problem solving, and so they are said to offer support. In order to speak about this support in a principled way, some sort of “language” is required for arguing what the support consists of. Since it is being assumed that cognition is modeled as computation, it is only appropriate that the language for cognitive support should be based on computing concepts. A simple example helps introduce the idea.

Hutchins [320] enlisted the notion of “precomputation” to understand the benefits and drawbacks of checklists. This was fundamentally a computational explanation of the cognitive benefits of checklists. The standard computing science understanding of the term “precomputation” connotes much of what is needed to comprehend the nature of checklists. For instance, computing the list of items in advance effectively spreads out the computational work over time, potentially lessening the peak demands during use (e.g., the amount of stack memory used). Such rearrangements of computation are standard tricks for enabling real-time performance. In addition, it is well understood that the value of precomputation is increased when the results can be reused many times, and there is bound to be a tradeoff in terms of memory requirements and lookup costs. Moreover, precomputation can spread the computation work over different computing elements: one processor can precompute some values, and another can use it. All of these basic effects are well known in standard computing. Hutchins noted that the same effects occur in cognitive systems, but that the improvements to computing result in improvements to cognition. The cognitive analogues to computer-based advantages of precomputation can be formulated; for instance, cognition can be distributed in time and individuals. The cognitive advantages of artifacts stem directly from their computational advantage to the overall computational system.

This simple example should make it clear that the key benefits attributed to the checklist can be analyzed purely by comparing the implications of differing computational methods. This type of comparative computational analysis is the core part of the entire RODS framework. The ultimate success of this DC explanation framework rests on whether appropriate computational accounts can be made for all of the types of cognitive support that one needs to understand and design.

It may be initially worrying, therefore, that the example of precomputation used above is, a certain sense, quite limited. The basic notion of precomputation provides a computational account of just a few facets of cognitive support. Chapter 3 surveyed many varieties of cognitive support, and even so it was an incomplete and selective survey. It therefore may seem necessary to be able to generate a challenging number of ways of explaining cognitive support.

Yet it would be surprising to find that completely different explanations were needed for each cognitive artifact! A more plausible scenario is that some relatively small set of principles are *in combination* sufficient to account for the many varieties of cognitive support. If so, the situation for cognitive support

would be similar to that of chemistry. In chemistry a small number of different atom types (around 100) combine in endless variety to generate the enormous universe of different chemical compounds. Alternatively, it might be reasonable to compare cognitive support to mechanical support. In mechanical support a small “vocabulary” of simple machines (inclined plane, pulley, etc.) are sufficient to account for the wide variety of mechanical aids [510]. So the veritable zoo of cognitive artifacts might turn out to be eminently manageable if some small collection of principles of computational advantage could be derived. Each computational principle would identify a *equivalence class* of artifacts that provide the same sort of cognitive support. The trick, of course, is settling on a suitable set of computational principles. Perhaps ideally, these would identify *orthogonal* computational principles such that they can be considered independently.

RODS proposes a selection of four computational concepts and principles that appear to account, in combination, for many varieties of cognitive support. It is currently infeasible to know with certainty what combination of principles would be completely sufficient. Fortunately, however, it appears that only this small “vocabulary” of computing science concepts goes a long way: they appear to explain a great number of different types of cognitive support identified in Chapter 3. These basic underlying principles fall into four categories: “task reduction”, “algorithmic optimization”, “distribution”, and “specialization”. The taxonomy is therefore referred to using the acronym “RODS”.

Each of the principles of RODS identifies a distinct *substitution principle*. The substitution says how an *equivalent computation* is better in some way. Of course, the notion of *equivalence* must be defined in some adequate way, but it is the notion of *substitution* that is really the particularly important part. Intuitively, the fact that each cognitive support principle identifies a substitution principle may make some sense. After all, cognitive support is necessarily comparative, and the comparisons must be fair. It would be awkward to argue that an artifact *supports* cognition if, by using it, some entirely unrelated computation is performed using it. An analogous rule in mechanical support is that the total work is conserved—a lever reduces the force needed, but the work done is the same. The issue of equivalence and substitution is revisited in more detail in Sections 4.2.1, 4.2.2, 4.3, and in Chapter 6. For now, it is only briefly mentioned in order to help readers appreciate that RODS is not an arbitrary collection of principles, and to help them to relate the principles to one another.

The following four subsections describe the computing principles underlying each of the cognitive support categories identified RODS. A summary appears in Figure 4.1. The table lists concise statements about each of the computing principles. This is exactly what we should expect for a suitable, high-level qualitative theory. It also provides a short statement about how the substitution principle can be used in design (in parentheses). These statements are included here primarily to foreshadow the design theories discussed in Chapter 7. Because the terminology within the literature varies, and because the intended audience of this framework is computing science-oriented, the presentation will be made using computing science terminology where possible. Within each category a purely computational description of the principle will be offered followed by an analysis of how the idea has been applied in HCI or related fields. Each principle may have wide applicability. Several applications are listed in later sections (primarily Chapters 6 and 8, and Section 9.2.2). Enumerating all of the possibilities for applying these principles is, however, obviously out of the scope of this chapter. Nonetheless, one or two examples will be provided of how each principle can be applied to explain a type of cognitive support. These examples are intended

\mathcal{R}	task Reduction
	<hr/> Cmpt Principle: some functions are easier to compute Substitution Type: substitute simpler tasks for more complicated ones Example (cmpt): removing redundant or unused computations Example (HCI): eliminating unnecessary steps (Design Principle: remove unnecessary work; relax task demands)
\mathcal{O}	algorithmic Optimization
	<hr/> Cmpt Principle: the computational efficiency of functionally identical algorithms differ Substitution Type: substitute equivalent methods, ADTs, or encodings Example (cmpt): changing to doubly-linked list; switching to faster sorting algorithm Example (HCI): switching to Arabic numerals (Design Principle: optimize cognitive processes for task & infrastructure)
\mathcal{D}	Distribution
	<hr/> Cmpt Principle: distribution adds memory or computing resources Substitution Type: substitute external resources for internal ones Example (cmpt): caching memory to a hard drive; client-server architecture Example (HCI): putting a shopping list on paper; automating constraint checking (Design Principle: distribute (i.e., <i>redistribute</i> or <i>offload</i>) data or processing)
\mathcal{S}	Specialization
	<hr/> Cmpt Principle: specialized routines or processors can be more efficient Substitution Type: substitute specialized processors for more general ones Example (cmpt): use a FPU or accelerated graphics card Example (HCI): enable visual search to substitute for “manual” search (Design Principle: change representation to make use of specialized hardware)

Figure 4.1: Summary of RODS computational advantage principles

to anchor the abstract discussion of computational advantage in somewhat more concrete specifics.

4.2.1 Task Reduction

It is sometimes possible to eliminate work that is unnecessary. For example, a pathologically designed development environment might insist on having the developer re-read every line of code in a program before each and every edit (e.g., by forcing a line-by-line scroll through the program). In most (all?) circumstances this is a waste of time and effort since usually very little new knowledge can be expected from re-reading the entire code base again. Computationally speaking, the problem is that there are unnecessary computations being performed; from an HCI point of view, the task can be reduced by eliminating unnecessary steps. Removing unproductive work will influence the performance. This form of performance “enhancement” is quite obvious and will not be discussed further. It is included in the taxonomy primarily for two reasons.

The main reason for including task reduction in RODS is so that one cannot confuse any of the other support types with a simple reduction in the work done. Thus every other principle in RODS will insist on maintaining some strong notion of equivalence in work. Being able to eliminate differences in the amount of work being done is crucial for rationally comparing the support provided by different tools. It would be nonsensical, for instance, to claim that a puppy “supports” the task of doing math homework by chewing it up and thus rendering it unnecessary to do. The “same” task—in some sense—needs to be done *with* the tool as without it (or with a modified version of it).

The above consideration leads to the second reason for including task reduction in RODS: it is sometimes important to identify cases where a designer can simplify task demands by requiring only a “good enough” results. Such cases are not, strictly speaking, instances of artifacts supporting cognition, however. Since RODS is proposed as a tool for performing design (Chapter 7), including task reduction as a category should help the designer consider design options.

Note that for many real-world activities it is problematic to tell whether parts of a task (or even which tasks!) are necessary or unnecessary. The problem is that most—if not all—of a user’s efforts will have side effects. The hypothetical environment that forces developers to repeatedly re-read every line of code may cause bugs to be serendipitously discovered. Or the developer may gain confidence in the correctness of the code, and this confidence could affect later decisions about legally signing off on the security of the system. In both of these circumstances at least some of the re-reading work is not completely unnecessary. The good news, if there is any, is that all of the other types of support principles can be applied to improve even unnecessary work.

4.2.2 Algorithmic Optimization

Algorithmic optimization refers to the modification of data structure, algorithm, procedure, or method in order to improve aspects of performance without changing the essential outcome. In HCI terms this is typically manifested in changes in representations and task solution methods.

Computing Science Principles

In RODS, the term “algorithmic optimization” is used to denote a change in computing methods or data encodings such that the underlying hardware and computational infrastructure remains unchanged. It is fair to say that the intent of algorithmic optimization is generally to get the most out of whatever computational resources or mechanisms are available. This interpretation aligns well with standard computing science sensibilities. For instance Bacon *et al.* [18] say that the goals of “optimizing transformations” in compilation technology are to:

1. maximize the usage of computational resources (processors, functional units, vector units, etc.),
2. minimize the number of operations performed,
3. minimize use of memory bandwidth (register, cache, network, etc.),
4. minimize the size of total memory required,

5. maximize data access locality.⁵

Generally speaking, not all of the above optimization goals can be simultaneously achieved: optimization involves making tradeoffs. In any case, the goals and description of optimization in computing science match very well the goals of optimizing DC systems more generally. The only question is: what are the fundamental computational principles of optimization, and which should be included in this category of RODS?

In RODS, the term “algorithmic optimization” refers specifically to the principles of computational optimization concerning what may be called *algorithmic substitution*. This refers to the fact that changing a data structure, algorithm, procedure, or method can result in performance changes. A key contribution of computing theory in this regard is that it defines a way of fairly comparing equivalent algorithms and data encodings. To compare algorithms one looks for differences in procedure (e.g., loop nesting depth), and, in abstract data type (ADT) and their implementation:

1. **ADTs.** ADTs are usually defined as an encapsulation of a collection of values along with a set of operations on those values [7]. For instance, a *list* ADT could be defined as a set of sequences of values along with *delete*, *insert* and data-accessing operations. An ADT implementation decides on the encoding for the values, and on the methods for computing the operations (e.g., a doubly-linked list). A change in the data structures implementing an ADT can make certain operations in one implementation faster than in another. For example, for a list ADT, a doubly linked list implementation will (normally) make insertion faster than an array-based implementation [7]. Depending upon the algorithm used, different ADT implementations can make important performance differences.
2. **Algorithms.** Algorithms can be roughly defined as terminating, side-effect free procedures operating over ADTs, and computing a single function [7]. Different algorithms can perform identical functions with different performance characteristics. The sorting algorithm *quicksort*, for example, requires fewer comparisons (in the average case) than the algorithm *bubblesort*. Assuming comparisons are of fixed cost, *quicksort* will run faster (on average) than *bubblesort*. Besides the running speed of the algorithm, many other performance differences are possible, such as the maximum amount of memory utilized, number of memory updates, and so on.

ADT implementation and algorithm are *both* implicated in performance differences. That is, two ways of altering the computational performance on a given problem is to change the data structures or procedures operating over them (or both). Because of the intimate relationship between procedure and data structure (e.g., see Rumelhart *et al.* [555]) they can both be considered to be variants of algorithmic optimization.

A Note on Functional Equivalence. It can be tricky to apply ideas of algorithmic optimization in HCI. First, in computing it is often the case that some notion of *functional* equivalence is required when comparing computations [7]. But exact functional equivalence is a stringent requirement that is frequently inappropriate for analyzing real-world systems. Sometimes the notion of equivalence must be expanded

⁵This point assumes that memory access costs are not independent, which is the case for most computing technology (due to caches, lookahead, virtual memory, etc.).

beyond this demanding definition. Strict functional equivalence as a fairness criterion breaks down for non-algorithmic processes, and in cases in which the exact function cannot be defined. For instance, one may wish to compare non-halting computations (e.g., phone switching systems) for the size of their state space. One may also wish to compare running times of heuristic, statistical, or approximate procedures where the equivalence criteria are relaxed. Furthermore, one may be interested in ill-defined problems in which there are no known or set criteria for solutions. Typical examples of such problems include designing aesthetically pleasing buildings, and laying out graphs in an understandable manner. In these circumstances, the function to compute is not known or is not specifiable. Thus, fair comparison of programs relies to some degree on an analyst's evaluation of the program's output. But even in these relaxed equivalence situations it is normally possible to apply the same basic principles of algorithmic optimization: different procedures computing comparable results can perform differently depending upon the data structures and procedures used.

Applications to HCI

The advantages of diagrams, in our view, are computational. That is diagrams can be better representations not because they contain more information, but because the indexing of this information can support extremely useful and efficient computational processes.

– Jill Larkin and Herbert A. Simon,

“Why a Diagram is (Sometimes) Worth Ten Thousand Words” [375], pg. 99

In psychology, cognitive science, and HCI, the application of algorithmic optimization is well known. For example, in cognitive science it is widely agreed that shifts in internal representation or procedure can lead to improved performance (see e.g., Rumelhart *et al.* [555]). The question here is: which applications of these optimization principles can reasonably be called *cognitive support*? For instance, it is possible to improve performance on tasks by teaching users to use better task strategies [47, 657]. It is not unreasonable to say that these newly tutored users may then perform different algorithms to achieve the same function—their use of the tools are “optimized” in the sense discussed above. However is it not stretching the intent of the term to call this type of optimization a form of “cognitive support”? Note that in the above example the issue is not whether the particular *tutoring* is a form of support: a tutoring system may quite easily be considered a cognitive support. But the tutoring system is not the application of optimization in question: it is the use of a more efficient task strategy by the user. In this case it is probably best to say that the support offered by the tutor *lead to* optimizations, but that these optimizations did not constitute an example of cognitive support. So merely *using* a better task strategy is not cognitive support.

So the question remains as to which applications of optimization principles can be justly classified as cognitive support? The criteria adopted here are: (1) the causes of computational rearrangement must stem from the existence or properties of artifacts, (2) the benefits must not be due to the other support principles, (3) the benefits must be a result of changes in ADT implementation or procedure in operation in the DC system, and (4) the benefits must specifically ease the user's cognitive challenges or difficulties.

The first criterion rules out counting changes to internal cognition as cognitive support. So although it may take some learning to make effective use of an artifact, that learning will be considered to be just another independent task (which may, of course, also be supported). The second and third criteria merely limit consideration to cases where the benefits come from optimization and not from, say, distribution. The last criterion is kind of a “catch all” intended to maintain the spirit of the term “cognitive support”. There can be cases where optimization occurs to reduce the user’s overall work, but these may not necessarily reduce the cognitive challenges. For instance, maybe just the physical effort involved in operating an interface is reduced. The optimization in this vein may indeed be considered a form of support, but it seems prudent to reserve the term “cognitive” support for those instances where cognition is made easier, faster, or better.

Even given the above guidelines, it is still necessary to define a method for mapping the principles onto HCI. There are two aspects to this mapping: interpreting DC systems in such a way as to be able to understand how optimization principles can be applied, and understanding how these optimizations lead to cognitive improvements. First, let us consider the second issue, that is, how optimization can lead to cognitive improvements. Let us consider some varieties of optimization in computing science and then map them onto HCI issues.

Optimization in computing. Many different performance measures may be optimized for any given program. When software developers think of the term “optimization”, they often think in terms of speeding up computations. In addition, it is often assumed that the main contributor to speedups is a reduction in the number of instructions executed. It is frequently the case, though, that speedups in optimization are a result of more efficient utilization of limited resources. For instance, speedups are often more closely associated with more efficient use of limited register or cache memory rather than with executing fewer instructions. Alternatively, the optimization may involve executing more operations but reducing the number of expensive operations. In sum, in computing the question of “what” to optimize has three common answers: reduce the number of instructions, reduce the use of non-processing resources such as memory, and reduce the number of costly operations by using cheaper operations (even if more of them are needed).

Analogues in HCI. In terms of cognitive support, all three of the above optimization types are relevant. Collectively they can be used to reduce the “cognitive load” placed on the user. Some optimizations may reduce the amount of short term memory that is required (e.g., by reducing the depth of a goal stack that needs to be maintained). Other optimizations include reducing the total number of operations, and reducing the number of “hard mental operations” [257] that need to be performed.

Mapping to HCI. Given these optimization issues, the essential idea for mapping them to HCI is to interpret human–artifact systems as computational systems such that the optimizations apply. The primary way to do this is to see the user as running methods (tasks) which operate over externally-stored ADTs (interactive systems). This basic idea has been advanced in many different guises. For instance, Green [257] argued that computer *systems* consist of both a notation and the ways of interacting with the

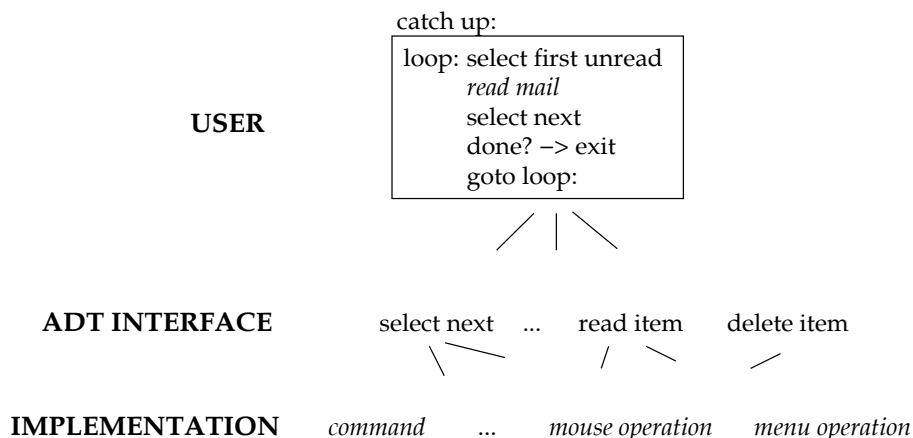


Figure 4.2: ADT view of interfaces, and mappings to implementations

notation. A system can therefore be viewed as an external information store [261,268]. The ways of accessing and altering this external information provide the operations of an ADT. The particular notations and interface manipulations provide an *implementation* of this ADT. In this view, the ADT corresponds to the artifact's *abstract* interface (*read value*, *write value*, *sort*, etc.); the ADT's implementation corresponds to the *concrete* interface (select menu option with mouse, etc.). A simple illustration of the idea appears in Figure 4.2. The figure schematically depicts the situation for the task of catching up on one's new email given some particular mailer interface.

For example, Green [257] gave the example of a speech interface to an otherwise standard (text-based) Pascal editing environment. The interface required sequential dictation of Pascal. The coding environment could therefore be understood in the following manner: the abstract ADT consists of a Pascal store of statements with only one abstract operation (append statement); the implementation of the dictation interface can be seen in terms of Pascal statements (encoding) and speech-based input method (operations). As Green noted, this restricted ADT forced an unnatural coding order. At a yet lower level, this implementation might be viewed in terms of the specific interaction technology and communication mode, i.e., in terms of speech acts (operations) and verbal utterances (encoding) (see Bass *et al.* [36], Moran [418]).

Mental vs. physical optimization. Note that in the above view, the procedures being executed using the ADT can be internal cognitive functions (e.g., inferences), external manipulations (e.g., edit tasks), or a mixture of both. Green's interpretation mentioned above seems to mix aspects of both. Other related interpretations lean one way or another towards internal cognition or towards action. The thinking/action differences closely correspond to the classic perception/action dichotomy in HCI [467]. In Figure 4.2, the "*read mail*" (in italics) may be modeled as a mental operation, whereas the "select mail" may be considered a physical operation.⁶

The interpretation made by Larkin and Simon [375] is an example of optimizing mental operations.

⁶It could be argued that even physical action has a mental component since it still would have to be thought and planned of in order to be performed (e.g., see a GOMS analysis [94]). This fact is actually needed below. But this complication is muted here because it hinders making the helpful distinction between mental and physical action.

They looked specifically at viewers “passively” reading diagrams (i.e., not physically manipulating them; reading a diagram is an active process [708], but not physically so). For instance, they compare the inferences made with diagrammatic and sentential representations of a physics problem. The particular problem involved a configuration of pulleys; the diagram showed the configuration of pulleys and ropes, and the sentential representation effectively encoded the same thing but, by describing it in ritualized English (e.g., “If a weight w_1 hangs from two ropes r_1 and r_2 and ...”). The mental operations required to make inferences needed to answer questions differed depending upon the representation.

Bass *et al.* [36] focuses primarily on designing artifacts to implement efficient action procedures. For instance, they considered an example of an email system. Given some particular ADT for accessing the email, they argued that, to be efficient, the ADT operations should eventually be implemented at the physical interaction level by single manipulations rather than extended manipulation sequences. Although the issue of optimizing physical operations is arguably not an issue of supporting cognition, it is still the case that cognitive resources are used during interaction—even skilled interaction, as GOMS modeling shows [94].

Applications in HCI. Given the above analysis, optimization principles can involve changing the processing involved in performing a task (given some interface). It can also involve changes in representation (encoding) or abstract interface (ADT).

Processing. Given a particular set of interfaces (e.g., a note pad), it is possible to rearrange cognitive processing by making new representations available. For instance Hutchins’ notion of *precomputation* [320], discussed above, is effectively computational optimization enabled by the provision of precomputed data. For example, if a user writes a checklist onto a piece of paper and then uses it later to reduce mental load, the cognitive processing is changed by distributing it in time.⁷ Thus precomputations can provide cognitive support by reducing the instantaneous processing demands during performance. The same argument holds for other external structures used in a similar manner.

Representation. In cognitive science circles, differences in *representation*⁸ are often pointed to for making problems mentally easier to solve (see e.g., Larkin *et al.* [374,375], Norman [472], Peterson [505]). The basic understanding is that the representation is stored on some medium that gives access methods

⁷Note that an external resource is needed in this particular example, making it an example of a *composition* of different support principles (see Section 6.5). However the same effect occurs if the user merely memorized the checklist using a mnemonic of some sort (such as how I have heard the process of crossing one’s self recited as “spectacles, testicles, wallet, and watch”). In such a case, the precomputation rearrangement of computation is still in effect. Memory is needed in either case, but the offloading of memory onto an external medium must be counted separately (see below).

⁸In the normal use of the term, “representation” refers to the encoding of semantic content rather than the structuring and encoding mechanisms. That is, a representation consists of data encoded in some data structure, and carries an additional requirement that the data bear a particular relationship to something else—that is, the data *represents* something (see e.g., Norman [472]). Usually, the structure and encoding format of the representation are enough to discuss efficiency—it matters little whether the structures represent some other entity. Thus including the representing relationship in this discussion actually adds little, if anything, to the analysis of algorithmic explanations of efficiency differences. Consequently it is not emphasized here.

to it (reading, manipulating, etc.). An example of such a representation substitution is the switch from Roman to Arabic number systems (see Section 3.1.6). That substitution can make mental operations like multiplication easier.

Abstract interface. Tools and media can provide various interfaces to external structures and processors. Differences in these interfaces (the ADTs) affect the costs of manipulation ADT [257]. For example if an interface is *viscous* [258], then the depth of the goal stack may need to be increased in order to successfully make updates.

4.2.3 Distribution

Distribution refers to the act of spreading computing costs amongst multiple elements. In cognitive terms, this typically means letting computers or other artifacts hold or process mental states or knowledge.

Computing Science Principles

In the ordinary use of the term, computational systems are called “distributed” when they have multiple distinguishable processing elements and these elements are loosely connected. The concept of distribution is therefore based on hardware-level considerations stemming from interconnection (communication) properties. Particularly relevant to distributed computation are communication costs. Distributed systems are also usually *parallel* processing systems, but the property of parallelism does not imply distribution⁹—it is the high communication costs between elements that is the key criterion. A dual symmetric processing PC with an accelerated video card and SCSI hard drive is not normally considered a distributed system even though the two CPUs, video card, and SCSI drive controller circuitry run computations in parallel. On the other hand, a Beowulf¹⁰ cluster of PCs connected through relatively slow Ethernet connections *is* considered distributed. The difference is the communication costs between the processing elements. The processing elements in the single PC can access common memory locations “directly”, so that communication costs are considered low. Of course, it is possible to consider a single system as being distributed on several levels. The “non-distributed” dual-processor PC might indeed be considered distributed at the finer granularity associated with the internal hardware. At the level of the individual CPU, the PC is seen as a network of processing elements communicating through a slower system bus.

There are several ways of realizing performance differences in distributed systems. A single computational process may be distributed amongst several elements. Using multiple processing elements splits the work, and so the work done by each element is reduced. This can speed up the execution because frequently the work can be done in parallel. Distribution can also mean that computations with resource requirements exceeding the capabilities of one limited processor might still be performed if the excess load can be taken up by other elements. In addition, distribution can lead to many other processing advantages such as scalability and fault tolerance. If distribution is desirable in a computer system, it is because of these sorts of performance advantages.

⁹A distributed system can run serially.

¹⁰See <http://www.beowulf.org>.

Another important aspect of distributed systems is the nature in which the distributed elements combine to form a conceptually single computational system. Several logically independent computations can be performed simultaneously on a distributed system without them being part of what might be called a single distributed process. For example an intranet can connect many office computers, but these might run quite unrelated processes (e.g., different window managers). Criteria must be met for distributed work to be considered as part of a single distributed computation. In particular they must *share* (equivalently, they must *distribute*) conceptually unitary data (state) and processing. This can involve such activity as data replication, state change propagation, rollback, update conflict management, synchronization, coordination, and scheduling. Each of these can have several forms; coordination, for instance can be performed centrally, hierarchically, or even involve self-organizing coordination based on local rules.

It is important to note that distribution frequently introduces a great deal of complexity into the specification and description of computational systems. Much of this complexity can be considered a form of *overhead* (see Section 5.5). The complexities arise from distribution, and from the coordination of parallel processes. Indeed, merely converting a serial program to a parallel one can be considered a *code obfuscation* technique [144]. Distributing an even simple computations can significantly complicate them. The distribution can especially destroy local coherence in the computation's specification or program. Simple, locally-specifiable control structures may give way to distributed coordination mechanisms. For instance, a program with a simple tree-structured control graph can be considerably complicated just by breaking it into a client-server system. The distribution can add cycles, polling, wait states, data translation, access negotiation, transaction and rollbacks. Thus there is conceptual cost associated with distribution that is in a way unrelated to the computation being performed. It can easily hide underlying simplicity. The loss of local coherence is also problematic because it can be extremely difficult to understand the overall computation by looking at only a part of it. What can be understood about a company's accounting programs by examining only the action of the company's database server?

Application to HCI

Humans in conjunction with artifacts and other humans form DC systems, that is, distributed computational systems (Conviction D1, Section 4.1.5). All of the benefits and problems of distributed systems consequently have analogies in DC and HCI. For example the "many hands make light work" principle is an analogue of computational work sharing, but—as in distributed computing—if the work cannot be decomposed into independently processable chunks, then resource and communication contention reduces the speed-ups achievable. For the most part, the traditional computing science intuitions about distribution carry forward into the sphere of human cognition (with the provision, of course, that only certain ways of engineering human cognitive systems are possible). For example:

- joint human–computer systems can solve problems too difficult or large for the human to do alone,
- coordination and synchronization are important problems in HCI,
- coordination overheads can greatly complicate even simple tasks,
- human–computer interactions can spoil the simplicity of an otherwise simple cognitive task.

4.2.4 Specialization

Specialization refers to the construction of specialized computing facilities that function in the same manner as slower, more general facilities. Being “specialized” here means that they have more restricted applicability (i.e., are less general in capability). In computing terms, it means they can compute fewer functions or operate over a restricted input domain. In cognitive terms, this typically refers to efficient but specialized mental capabilities.

Computing Science Principles

Over the years, computing science has created a number of ways of abstracting and generalizing computing methods (procedural abstractions, object hierarchies, etc.). The general rule that has been found is that generality and abstraction usually have a cost associated with them: the more specialized and specific computing mechanisms tend to be faster even if they are less general. This is roughly the computing science analogue of the difference between a Swiss Army knife (generalist) and a grapefruit knife (specialist). The same basic phenomenon is also well known in relation to *simulation/implementation* hierarchies. Computational systems are normally hierarchically decomposed such that operations viewed as atomic at higher levels are “simulated” by compound operations at lower levels. There are several traditional boundaries between hierarchy levels [446]: hardware–microcode, microcode–machine language, machine language–high-level language (although modern architectures can blur these traditional boundaries). Because the levels are related by simulation it is often faster to use operations implemented at lower levels, if possible, than to simulate these same operations at higher levels.¹¹ One way to improve performance, therefore, is to directly use specialized hardware in place of simulating the operations it performs.

Familiar examples of this efficiency-by-specialization principle include: floating-point co-processors, accelerated video cards, and vector processors in supercomputers. All of these examples dedicate specialized hardware to compute restricted sets of functions efficiently. The advantage is that these units are fast and efficient on the operations they are able to perform. Because of the specializations, optimizations in their construction can be applied. For instance it may be possible to take advantage of parallelism opportunities or to use special-purpose algorithms instead of using more general algorithms. But, because of the specializations, certain other operations cannot be performed. Besides hardware specialization, software can also be specialized, typically to perform better on particular problems or data. For example a technique called “partial evaluation” (see e.g., Hatcliff *et al.* [292], Section 6.6 of Bacon *et al.* [18]) can specialize general purpose methods to work more efficiently on partially specified data. Some optimizing compilers can also specialize code that is locally general but, due to global properties (e.g., invariant expressions), can be specialized. Another example is found in the AI literature on problem solving methods: the so-called “weak” problem solving methods work less efficiently than the so-called “strong” methods [443]. The difference between strong and weak methods is essentially that strong methods take the specifics of the problem into account—they are specialists.

This same principle of specialization is commonly found in cognitive science and HCI. In cognitive science one can find models of specialized perceptual hardware (e.g., edge detection or motion detection)

¹¹Not always, of course, but it is the principle that counts.

as distinguished computational components of an overall cognitive system. These low-level cognitive operations are fast, automatic, and are unconsciously performed. Specialization is also a mechanism used to explain how skills are learned (see especially Rasmussen [526,529]). These skills include activities like reading English, typing, and riding a bike. In such learning models the performance of the skill is at first a conscious act and therefore makes use of general-purpose reasoning capabilities. After repeated exposure to the task (the repetition is important) the computations are “compiled” down into more rudimentary and inflexible mechanisms. After training, the performer is no longer conscious of all the activity taking place, even to the point of not being able to say precisely what they are doing. Although the distinction between hardware and software specialization may seem clear enough, it is difficult to say with confidence what cognitive phenomena should be classified as hardware-level specialization [686]. Highly practiced skills (e.g., adding single digit numbers) are also quick and to a great degree automatic so they are phenomenologically similar to “built-in” hardware capabilities (even if their underlying mechanisms differ). Thus it is sensible to group software and hardware specialization together in this taxonomy.

Applications to HCI

In computing science, specialized software or hardware is used to improve performance by *substituting* for the more general mechanisms.¹² In the context of HCI, the specialization principle is applied by recognizing instances where specialized processing capabilities are available. Differences in artifacts can lead to situations which enable these special processing capabilities. These can then substitute for more cognitively taxing processes. There are a variety of examples of this general principle. Perhaps the most well known is what has been called *perceptual operator* substitution [116].

Perceptual operator substitution occurs when efficient, low-level perceptual mechanisms can substitute for deliberate inferencing or reasoning (see Section 6.3 for more). This effect can justly called the “bread and butter” of visualization research [687]. The essential understanding is that certain types of stimuli can exercise highly specialized perceptual mechanisms. The classic perceptual mechanisms involved are the visual mechanisms like motion detection and visual search. Often these create what are called “pop out” effects [687]. The point is that excellent computational accounts of some of these substitutions are available. The essential quality of these accounts is that the specialized mechanisms perform computations that substitute for more complicated inferences (see Larkin *et al.* [375], Casner *et al.* [115,116], Rasmussen [526]). Perceptual operators are just one instance of specialized cognitive abilities. However the blueprint for other specialized abilities is the same as for perceptual operators: artifact differences enable specialized processing to be substituted for more cognitively demanding processing.

¹²As an aside, this notion of substitution is reasonable only if the computational system is composed of distinct levels or processing elements, that is, if it is heterogeneous in some way and therefore distributed [300]. The situation is analogous to the example of the computer with an accelerated video card. At some level, it too can be considered distributed. In the past, I have called substitutions of specialized processing “internal distribution” [685]. While the interpretation is reasonable and unifies two categories in the taxonomy, it is perhaps better to leave substitution as a separate category since it seems qualitatively different, and also explanatory.

4.2.5 Summary of Principles

This section presented a taxonomy of four different support principles: task reduction, algorithmic optimization, distribution, and specialization. These support principles consist of computational principles and a way of using them to explain causes of cognitive support. The computational principles state why different computations or DC configurations generate computational advantages and hence generate cognitive benefits. The taxa were chosen with the intention of providing four quite orthogonal dimensions for analyzing computational advantage. The basic “gist” of each of these support principles is relatively easy to state. Task reduction is essentially the elimination of unnecessary computation. Algorithmic optimization is the use of more efficient algorithms or ADTs, or the use of more efficient ADT implementations. Distribution is the use of additional computing resources. Specialization is the use of less general mechanisms that can be made efficient because they are customized, that is, adapted to the specific task or task environment. Thus RODS meets at least one expected criterion for a suitable high-level qualitative theory: simplicity.

It is important to reflect upon the significant aspects of the support principles, and to relate how they are used in the remainder of this work. One aspect that seems significant is that all of the fundamental principles are familiar principles of computational advantage. In certain respects it should be considered astonishing that understanding the basic principles of *cognitive* support requires so little psychological knowledge. Indeed, *the fundamental principles explaining cognitive support should be quite familiar to most computing scientists*. Naturally, there is still an enormous difference between understanding the basics and applying them well in real situations, but the lesson is clear: the essential character of the DC vision of cognitive support is not inextricably psychological. This might be reasonably taken to be an encouraging sign. For it suggests that reasoning about (certain aspects of) cognitive support may be tractable for ordinarily trained computing scientists. This possibility is further explored in Chapter 7. A second reason for optimism is that the same fundamental principles appear to underly important aspects of both computing science and cognitive support. Some consider this sort of theoretical unification significant (e.g., see Goguen [245] and Simon [594]).

4.3 Analyzing Cognitive Support

DC is a relatively new school within cognitive science. As a result, the ways of actually employing it to analyze and compare various artifacts are still being explored. There are still open questions as to how to best analyze and model DC systems. There are, in particular, three important issues that will be of long-term interest in applying RODS to analyzing cognitive support:

1. How can cognitive systems be compared to understand the cognitive support provided by artifacts? Being able to answer this question is obviously critical for tool evaluation and comparison. But it is also a critical question for analysis and design since cognitive support is understood in comparative terms.
2. How can one generalize DC models across multiple joint cognitive systems? If this question cannot be adequately answered, then we may end up being so restricted that we can only make statements

about *particular* joint cognitive systems, not *classes* of cognitive systems. This would be a serious blow for applications in HCI, for if it were impossible to reuse previous DC models, one would have to begin from scratch for each new system. The cost of analysis would skyrocket. Knowledge about good design could not be accumulated in models.

3. How can one usefully abstract the computational structures of DC systems? Human interactions with artifacts can be extremely rich at the level of physical interaction. Often we wish to understand it at a higher level so as to avoid “death by detail” [264]. For instance, at some point we may only care *that* a user can access a fact from external memory, not whether this access involved particular search mechanisms, scrolling, etc. What are suitable methods for abstracting these details away?

It is not necessary to fully answer these questions here (nor are we in a position to do so). However it can be helpful to make a few initial comments about how these questions might be answered. Even if the full answers are not forthcoming, it is important in a chapter like this to help set the tone and reveal the “ground rules” for future theorizing. Consequently, three proposals will be made as to how these questions can be answered, and how they will be generally addressed in the remainder of this work. These proposals are all based on arguments in the existing literature, but they need to be presented in a more explicit form because of their importance for applying cognitive support theories such as RODS.

The first proposal is the suggestion that joint cognitive systems can be compared by locating them within a logical space of “functionally-equivalent” joint cognitive systems. Cognitive systems within this space are related by substitution transformations defined by RODS. To compare the cognitive support in two cognitive systems, one needs to determine the substitution transformations that relate the two systems. Since these systems are related by variations in artifacts, the logical space is, in fact, a projection of an associated design space. This proposal is outlined in Section 4.3.1.

The second proposal is that DC modeling in HCI and SE should be oriented towards defining and refining standard *distributed cognitive architectures*. Architectures are general (and thus reusable) computational models. Often DC studies are focused on specific cognitive *systems*. To make observations about such systems generalizable, the resulting models and theories must be defined on DC *architectures*. This proposal is outlined in Section 4.3.2.

The third suggestion is that it will frequently be useful to define *virtual architectures* when modeling DC systems. The purpose of a virtual architecture in this context is to encapsulate and abstract meaningful interactions between system components, especially between a user and a computer. It would be surprising if no regular or systematic interactions occurred between users and computers. A virtual architecture is a modeling technique for encapsulating such systematic interactions. This proposal is outlined in Section 4.3.3.

4.3.1 The Design Space Induced by RODS

The DC literature contains several comparisons of cognitive systems. For example, Hutchins’ seminal book on “Cognition in the Wild” [320] analyzes the differences between Western and Micronesian navigation. The analysis shows, in vivid detail, the two navigational systems to be different computational implementations of navigation. In a related way, Zhang *et al.* [727] show several games to be *isomorphs*

if one analyzes them as different implementations of the same problem. Both of these accounts reveal instances of support. But the way of systematically analyzing such support is never adequately spelled out. For instance, can the space of all possible ways of supporting cognition be mapped out? Here it is argued that such a space can indeed be conceived as an aid for comparing tools and reasoning about tool designs.

To suggest that an artifact supports cognition is to indirectly imply that there is a way for cognition to be *un*-supported in the way suggested. For instance, if a slide rule is thought to support cognition when doing calculations, then life without the slide rule can be pondered. In general, the support provided by an artifact may be understood only in comparison to what is implied by its absence, substitution, or modification. One way of thinking about this comparison is to suggest that there is a continuum of different levels of support which ranges from the completely unsupported (entirely mental) to the completely automated (no human thinking involved). In between the extremes are cases where cognition is spread between humans and artifacts. In practice, both extreme ends of the spectrum will be unattainable for interesting tasks. Still, it is instructive to imagine in principle what the unachievable extremes of the spectrum would entail.

For the *entirely* mental end of the continuum to hold up under close inspection, all of the user's problem, evolving solution, and mental state information would need be held internally; all of the processing of such information would also need to be done internally. This might be akin, perhaps, to floating in a sensory deprivation chamber and solving problems mentally. In reality, of course, people always are embedded in an environment and will make strategic use of that. Programmers, for instance, will normally not code anything non-trivial without interacting with external media [260]. No matter, we can still imagine the case where some inhumanly capable programmer keeps a copy of the entire world in the head. The completely unsupported programmer is a logical possibility even if it is not an actual possibility.

Distribution begins to change that picture. As distribution of cognition is increased, the locus of cognition expands away from the individual mind and is dispersed. Our imaginary unsupported programmer may begin to use a notepad, for example, to offload a specific item from memory. Thus we have considered two cognitive systems: the mythical lone programmer, and the programmer-plus-notepad system. The logical difference between the two cognitive systems is a *substitution transformation*: an external memory is substituted for an internal one. In other words, the artifact *reengineers* the computation by making a distribution substitution. It is critical that the substitution transformation is in some sense function-preserving, that is, the cognitive systems belong to the same equivalence class. If they did not, it would be difficult to argue their differences as being due to cognitive support.

As various parts of the computation are distributed, further variations of these systems can be posed by considering specialization- and algorithmic optimization-based substitutions. That is, once data of any sort is distributed, processing can be distributed, specialization substitutions can be enabled, and algorithmic optimizations can be performed. It may be the case that several of these transformations occur in unison. For instance, Sharples noted that writing things down unlocks other types of support:

One way to overcome the difficulties of performing ... complex knowledge manipulation in the head is to capture ideas on paper (or some other external medium such as a computer screen) in the form of external representations that stand for mental structures. So long as ideas, plans, and drafts are locked

inside a writer's head, then modifying and developing them will overload the writer's short-term memory. By putting them down on paper (or some other suitable medium) the writer is able to explore different ways of structuring the content and to apply systematic transformations, such as prioritizing items, reversing the order, or clustering together related items. [575, p. 135]

The above observations suggest one conceivable method for generating the space of all possible cognitive supports for some task (conceptually, at least). The generation is effectively by *recursive application of RODS transformations*: start with an unsupported human, pick a possible redistribution of data and, then one-by-one, apply each possible substitution transformation to generate new compositions. For instance in Sharples' writing example, one could begin by enumerating the possibilities of (at least partially) distributing ideas, plans, drafts, goals, and so on. Then one might consider various ways of processing these externally, of changing the representation to enable specialized processing to substitute for deliberate processing, and of making changes to encodings and methods. Each of these changes represents a possible design option. Thus the logical space of cognitive systems also identifies a design space.

Now it may be the case that the full design space cannot actually be generated. Nonetheless the logically-defined space provides ways of trying to rationally compare the support given by two artifacts or environments. If two tools are similar to each other—one is a successor of an earlier prototype, for instance—then it might reasonably be argued that one transforms cognition according to some combination of RODS transformations. These transformations would identify the cognitive support differences. This is a type of "relative" comparison since the support in one tool is being compared to a neighbour rather than to some "absolute". It is also possible to envision an absolute analysis. One would need to start with an analysis of how an unaided mind might perform the task, and then describe the substitution transformations required to achieve the given system. The reverse transformation can also be considered. Given any user-tool dyad, one would consider the transformations needed to make the relevant activity occur entirely inside the user's head.

4.3.2 Distributed Cognitive Architectures

At what level of generality should artifact analyses be conducted? I believe that rather specific analyses will prove necessary to understand the functionality of artifacts, but that some relatively general analytic tools can be developed for some issues of user interface design, because such issues are often general across a wide range of artifacts.

– Stephen J. Payne,
"Interface Problems and Interface Resources" [492], pg. 133.

Cognitive science uses an ages old trick from computing science to build cognitive models that generalize across multiple individuals and situations. The trick is to use an incomplete model. But it must be incomplete in just the right way. The missing parts must correspond to the variations one wishes to generalize over. Conversely, the parts that are actually specified must correspond to invariants. *Cognitive architectures* in cognitive science are just such abstract models. They are intended to capture invariants

in cognition [315,520,651] across essentially all “normal” humans. A cognitive architecture is a software architecture [580]. It specifies a model of the “cognitive machinery” of the mind. These cognitive architectures are not specific to tasks, social context, or the particulars of the environment such as the tools and artifacts available. This makes the cognitive architectures general. The statements one can make about such architectures (efficiency, behaviour, communication costs, etc.) generalize. But the generality of the architecture also means that work must be done when applying it in specific situations. To be applied, the appropriate details of the analyzed situation must be “plugged in”. For instance, a cognitive architecture like SOAR [446] contains essentially no domain or task knowledge. This knowledge must be encoded before it can be run to simulate users. That is, the user’s knowledge base must be programmed in [721]. Actually, much more than user knowledge is normally required: one must also program in, simulate, or otherwise account for many other aspects such as tools, tasks, and social settings [651].

The generality of a cognitive architecture has obvious and crucial advantages when applying it during analysis. The generality itself is critical, for an architecture will be useful to an analyst only if it can be applied to the situations of interest. Just as important is reusability. The “contents” of a cognitive architecture are those aspects of cognition that it models, that is, those aspects that are invariant. In other words, the architecture captures and encodes certain context-independent facts and knowledge. This content is *reused* by the analyst whenever it is applied. This can save much effort since the content does not have to be recreated for each situation it is applied in.

Currently there are no DC cognitive architectures that one can obviously point to for being candidates for reuse. One problem may be that many types of studies in this field take the point of view that each situation must be studied and modeled individually.¹³ On the face of it, this approach may not seem that unreasonable. The trouble with modeling joint cognitive systems is that (1) they come in so many different flavours and configurations, and (2) “the system” to model is frequently a shifting, ephemeral entity. In contrast, pursuing (individual) cognitive architectures makes a certain degree of sense: most of us have similar brains, and we modify our brains (barring injury) in rather limited ways. We cannot currently pop new RAM chips and co-processors into our brains. This universality and stability makes it *prima facie* reasonable to talk of a model of “the cognitive system” of an unaided individual. Joint cognitive systems exhibit neither this universality nor this stability. One must talk about human–computer *systems* rather than “the” human–computer system. Unlike the brain, the external parts of joint cognitive systems *can* easily be modified. Indeed, from the DC viewpoint the whole purpose of design is to retool the cognitive system (see Chapter 7). Moreover, human–computer systems naturally and fluidly change from one moment to the next. A developer can slide his chair across the room to a coworker’s computer and begin working with an entirely different set of tools. Tool manufacturers continually update and change their products. In fact, developers are well known to customize, program, and tweak their environments themselves. For all intents and purposes, these simple facts rule out the hope of building a single model

¹³On a side note, some DC researchers adopt philosophical backgrounds that tend to reinforce this practice. Certain philosophies from anthropology and communication value a fundamentally “bottom-up” procedure for understanding cultures or systems; they view with suspicion pre-existing theories or preconceptualizations of the phenomena under study. As a result, there may be a tendency to try to model each system by starting from scratch. Although basic modeling principles can still be reused, this sort of philosophy generally rules out assuming a particular cognitive architecture.

that can be applied across all relevant design situations.

Are generalizable DC models therefore impossible? Must context variability universally preclude abstraction and generality? Can DC models be generalized across various tools and tasks? If so, how? To the best of my knowledge, these questions have not been answered in DC literature. So far, most publications from DC have tended to focus on arguing the viability and necessity of the DC view, or have tried to show how particular cognitive systems can be modeled and explained. Generalizability of analysis is not currently a research emphasis. As a result, prescriptions for studying DC (e.g., Hollan *et al.* [311]) emphasize the need to capture situation-specific details rather than, say, the need to understand how the particulars of one system relate to those of similar systems. Consequently, there are no “standard models” to begin with—one builds DC models from scratch. There are some related suggestions on potentially useful modeling *techniques* (e.g., ERMIA [268], Flor’s *media constellations* [223]), but these do not address reuse of the models themselves. Since cognitive model building involves computational model building, it is fair to say that DC modeling is not currently founded on software reuse.

Surely this is not *necessarily* so. There must be similarities in DC systems that can be recognized and utilized effectively. If we could identify useful invariants where they exist then we could capture these in “standard” DC architectures. The trick would be to (1) identify unimportant variations and abstract these away, (2) recognize important variation and then either provide abstractions of these or omit computational details for them so that the situation-specific details can be “plugged in” later, and (3) encode the important invariants into an abstract *distributed cognitive architecture*.

To computing scientists, this prescription for generality will come as no surprise. Parnas is normally credited for identifying the steps that are needed in order to define families of similar programmatic structures [245, 486]. His main prescription was to recognize similarity, anticipate variation, and then structure the system to capture the similarities and allow easy modification for the variations [485, 486]. It is possible, therefore, to define generalizable computational structures, but only for families of similar computational systems. Thus although there may be no *universally* useful DC architecture, we should be able to define some reasonable collection of abstract and incomplete DC models. These would capture important DC invariants so that they may be reused in a wide variety of similar situations. Since humans can be expected to learn and participate in a restricted set of artifacts, an adequate collection of general architectures might be eminently manageable in size. The same basic vision has driven software architectures research in SE [578, 580].

What these standard DC models might be is beyond the scope of this work. The main point of bringing up DC architectures in this chapter is to ensure that the importance of the DC architectural level of analysis is realized. This will become important in later sections since they will be proposing DC-based cognitive support theories. And to be useful, they will have to address the issue of generalizability. To use Parnas’ terminology, the theories need to make statements about a *family* of artifacts. Furthermore, the significance of DC architectures should be emphasized to the wider context of DC research. If the importance for building a generalizable science base is not realized, there is a risk that future DC systems will continue to be unnecessarily difficult to analyze and compare because there are no DC architectures to reuse. This threatens the viability of efficiently using DC-based cognitive support analysis in many instances of real-world design.

4.3.3 Virtual Architectures

If people could plug RAM chips, hard drives, network interface cards, coprocessors, and device controllers into their heads, there might be relatively little need for many of our cognitive artifacts. Why bother with an appointment book if you can slip a flash RAM card into your head and store your appointments there? Your brain could use the extra RAM to store your important appointments on instead of your poor, fallible “wetware” memory. Being able to plug in these components may seem like an “ideal” way of augmenting human intellectual capabilities: one usually imagines that there is little “overhead” involved. When using cognitive artifacts, there is *always* some overhead involved. The overhead of interacting with cognitive artifacts shows up as cognitive effort and interface activity—pointing, typing, speaking, reading, etc. The overhead work mixes with the “real” work of the user. Users must learn how to use the artifacts; they must command, manipulate, and communicate with them; they must also react to, perceive, and monitor them too.

But although cognitive artifacts are not “real” hardware extensions, it may still be possible to go some ways towards restoring a similar level of simplicity when modeling DC systems. The key is to try to encapsulate and abstract some of this “overhead” within the DC models. One way is to employ a model of *virtual hardware*, that is, a *virtual architecture*. A virtual architecture is a simplified model of cognitive hardware. The idea is that the virtual hardware does not really exist—it is in fact *simulated* by operations performed by the user or computer. That is, the *virtual* hardware is simulated by a (more complicated) *implementing* hardware.

For instance, consider a user doing some work using a paper and pencil. At some level it makes sense to consider the paper and pencil as an extension to the user’s memory. But at the “real” DC hardware level all of the memory operations like storing, recalling, and even contents addressing must actually be simulated by the user. She must perform the writing and reading steps manually. Each of these can involve considerable cognitive machinery to perform. Furthermore the user is in charge of devising and following an addressing scheme for the contents on the paper. For example if the data being stored is two-dimensional ordinal data, then tabular or graph based indexing methods might be used (see e.g., Norman [472, ch. 3]). If more than one piece of paper is used as an external memory then not all of the memory may be viewable at once, and the burden of “paging” in the right portions of external memory is passed onto the user [303]. Thus although we may be interested in viewing the paper as a memory extension, when we look closely at how paper is used, there is no simple memory interface to be found.

To create a degree of abstraction in viewing the extended memory, a virtual memory architecture can be postulated. The point of such an architecture is to provide a direct, simplified read/write abstraction to the overall distributed memory system. A rough illustration of such an abstraction is given in Figure 4.3. The figure appears more complicated than it really is: the diagram shows that cognitive overheads are involved in accessing the external memory, addressing it, and paging the right parts into view. Postulating a virtual architecture in this way clarifies the overall work process by encapsulating the overheads involved with interacting with the particular implementation of the virtual memory.

Note that the simulation overheads are not forgotten. They are merely collected together and encapsulated in the mapping to the implementing architecture. The memory and processing overheads involved

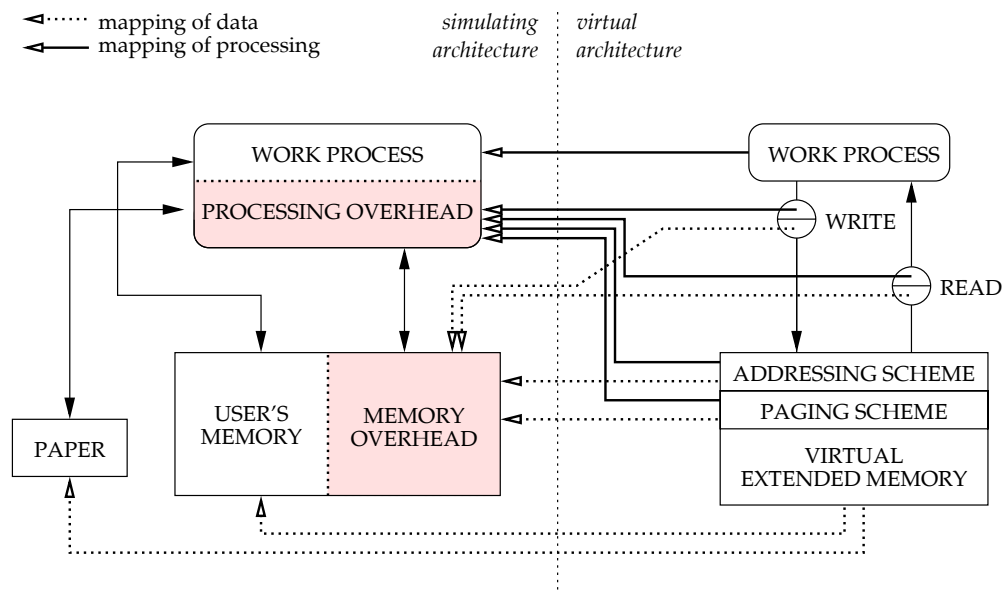


Figure 4.3: Virtual shared memory as an abstraction over complicated interaction

in implementing the reading, writing, addressing, and paging are merely handled at another level. This virtual architecture modeling method is precisely the approach taken in distributed shared memory systems (see e.g., Skillicorn *et al.* [602]). These systems provide an interface to the programmer as if a single physically shared memory were available. But, in reality, behind the scenes it is being simulated. This means that virtual architectures are a way of making the transition from one “level” to another explicit. That is, they bridge the lower level of detailed HCI to a more abstracted view of the system in which those details recede. In a sense, then, they propose a “non-greedy” [177], explicit *reduction* of a high level description to its implementation.

To see the relevance of this possibility, let us reconsider how HCI is frequently dealt with when performing cognitive modeling. For instance, consider how von Mayrhauser *et al.* [675] modeled comprehender behaviour using a high level model. They used protocol analysis to build an interpretation of the mental actions that comprehenders take. But in this analysis they effectively ignored many of the human-computer interactions involved. Sometimes this is natural since many interactions are done with skill and so they will not leave verbal traces to analyze. Other times, however interactions with tools are actively filtered out (see Gray *et al.* [255] for a well considered example). So cognitive analysis often filters out the interface activity, either explicitly or otherwise. This is suitable if one’s aim is to build a simplified model of human cognition that is unfettered by the complications of interactions with tools. This is reasonable to do if one considers that interacting with artifacts constitutes a separate sub-task that the human thinker must simply cope with.

But from the DC point of view, if one throws away the tool interaction, there is a good chance that the full contributions of tools will be missed. If a tool is acting as an external memory, for instance, then some of the system’s memory would be left unmodeled. For example, in the model by von Mayrhauser *et al.* (see above), they coded their protocols for a “Chunk&Store” action without concern for the (internal) activities

that implement the action. But what if external actions were actually implementing a “Chunk&Store” action onto an external memory? Would they be correctly coded as a memory action? If the interactions with tools are ignored, the answer would likely be “no”. But if the interactions are simply thrown back in, then the analysis threatens to become bogged down with the thick interaction details. Here is where virtual architectures may be significant in the future. They allow one to systematically *control* the complexity of the interactions without simply throwing them away. This ability will be used in Chapter 8 to abstract away details about interacting with an external medium.

There is a second possible significance to virtual architectures. In the previous section it was suggested that families of human–computer systems with common structures might fruitfully be described using DC architectures. This could lead to the definition and study of a collection of common DC architectures. If this should come to pass, virtual architectures can make a useful addition: generalized task analyses. A virtual architecture maps between abstraction layers by defining tasks that are performed in the simulation of the virtual machine (e.g., addressing, paging, etc.). What is important about this account is that: (1) the tasks are defined independent of any particular interface or problem domain, and (2) the tasks are assigned cognitively meaningful interpretations. This combination may be hard to achieve with other means. There exists some prior work in describing generalized task structures in ways that are reminiscent (see e.g., the “generalised task models” (GTM) work [336]). But a virtual architecture additionally casts the abstract tasks in cognitive terms; it is an abstraction mechanism suited to DC modeling.

In sum, although external artifacts are not “directly” connected to the user it is possible to step back and view DC systems as implementing virtual architectures. The most essential virtue of a virtual architecture is the logical modularization and simplification it provides. One may encapsulate complicated interaction between heterogeneous components with modular—but virtual—architectural components. The virtual and implementing architectures are related through simulation. The tasks performed to simulate the virtual architecture define a meaningful but abstract set of tasks to perform.

4.3.4 Summary of Analysis Proposals

DC theorists have argued that joint human–computer systems can be modeled using distributed computation models. It is hoped that systematic ways of studying cognitive support can be fitted onto this existing DC research. But three fundamental problems appear not to be addressed well in prior modeling work: (1) how to compare systems in a way that carefully describes the support and the possibilities of support, (2) how to generalize analyses across different types of tools and task situations and then collect such generalized analyses, and (3) how to effectively abstract and encapsulate low-level complexities involving human–computer interaction. Three proposals advanced in response to these issues were overviewed.

First, the concept of a space of functionally-equivalent cognitive systems was defined. Motion in the space corresponds to applications one or more of the three primary substitution principles of RODS. Defining such a logical space clarifies our understanding of cognitive support and how tools may be compared for their support.

Second, the notion of a DC architecture was promoted as a method for providing generalizable DC models. The notion of a DC architecture is a relatively minor extension of common cognitive architecture

modeling techniques. A DC architecture can capture computational structures that are invariant across a family of contexts. The suggestion to investigate DC architectures is significant because it implies a potentially useful shift from studying *individual* systems to studying *families* of systems with common computational structures.

Third, the idea of a virtual hardware architecture was proposed as a technique that could be used to abstract and encapsulate certain human–artifact interactions. These abstractions may be important for analyzing HCI. It may make it much easier to understand interaction at an abstract cognitive level; it may also make it possible to define meaningful abstract tasks.

These three proposals may turn out to be critical parts of RODS-based design and analysis frameworks. They are not well developed yet, but because of their potential significance, it was important to describe them in this framework definition chapter.

4.4 Requirements Check

Certain theories are proposed as an improvement to previous theories, usually by explaining or predicting something better (more accurately, in more contexts, etc.). Others are proposed primarily to try to meet specific desiderata or requirements; less concern is expressed for improving theory accuracy. RODS is an example of the latter. Most of these requirements were generated using the mature field of physical support as an intuition pump (Section 2.3.1). Key requirements are recalled here and RODS is examined to clarify why it is anticipated that these requirements are met.

High-level, Qualitative Theory of Cognitive Advantage

The exposition of RODS given in this chapter hides its core statements within its extended definitions. But a relatively simple definition exists. At a high level, RODS can be stated as:

Artifacts reengineer cognition by substituting one computation for another. The cognitive support these substitutions provide is the computational advantage generated by the reengineering. There are four distinct forms of cognitive support generated by four distinct types of computational advantage. The types of computational advantage are: task reduction (eliminating unnecessary work), algorithmic optimization (switching algorithms, ADTs or their implementation), distribution, and specialization. All cognitive artifacts can be decomposed into applications of these four computational principles.

It seems reasonable to say that this statement of RODS qualifies as a high-level, qualitative theory of cognitive advantage.

Small Vocabulary of Primitive Types

RODS identifies and requires only four primitive types of cognitive support. These rely on distinct computing principles for substitution, and these seem to be orthogonal.

Compositional Language

The primitive types of cognitive support combine because the substitution principles they are defined by identify independent computational substitutions.

Mnemonic, Evocative Names

RODS is couched in terms familiar to most computer scientists. This does not guarantee that the terms will be more understandable, memorable, or more evocative of important implications. However it seems to improve the odds that the terms are at least familiar and meaningful the average computer science student. And computing scientists frequently understand the principles of optimizing computing. This may allow non-specialists to reason by analogy about various forms of cognitive support.

Abstract, Generalizable Description Level

RODS names four categories of cognitive support and defines their underlying principles in cognitivist terms (knowledge, inferencing, etc.) and computational terms (memory, processing, etc.). Both of these are description levels are abstract and independent of their implementations. In particular, cognitive support categories are defined without reference to a task or even a particular cognitive model. For instance, distribution is defined without referring to the specifics about what are being distributed. Plans, constraints, goals, and processing history might be distributed (e.g., see Wright *et al.* [719]). This means RODS can be used on whatever issues of cognition the analyst considers important (goals, social roles, etc.). The significance of this ability is underscored in Chapter 5. In addition, the computational principles referenced are abstracted away from their implementation. For instance, when considering distribution of cognition, any number of artifacts may play the role of an external memory (white boards, computers, even other people [569]). Thus RODS is independent of both task and technology: how they are used depend on the analyst's application context.

Analysis Framework

RODS contains a framework for analyzing cognitive support in terms of computational reengineerings. The core of this framework is a way of comparing cognitive systems. A cognitive system is identified, in part, by the computational architecture created by humans using their particular artifacts. Given two "equivalent" cognitive systems, one can be seen as a computational reengineering of the other based on the four types of advantageous substitution identified in RODS.

4.5 Summary and Conclusions

This chapter has presented an overall framework for generating and using theories of cognitive support. The framework consisted of three main parts:

DC Tenets

DC assumes a particular way of understanding cognition and the way artifacts contribute to it. Six main tenets of DC were reviewed and their importance to RODS and to the goals of this dissertation were considered. These set the overall analytic framework of RODS.

A Simplified General Theory of Cognitive Support

A taxonomy of four cognitive support principles was defined. These principles were: task reduction, algorithmic optimization, distribution, and specialization. These principles are intended to identify

orthogonal computational explanations for cognitive benefits. Collectively they form a theory of cognitive support.

Analysis Techniques

Effectively applying the overall RODS and DC framework to real design work requires an understanding of the principles used to identify and compare the cognitive support in tools. It also requires consideration of how to generalize and reuse DC-based analyses, and how to abstract away unneeded low-level interaction details without completely discarding them. Section 4.3 discussed the beginnings of ways of dealing with these issues. A basis was described for defining a logical design space generated by substitution-based cognitive restructurings. The importance of searching for reusable cognitive architectures was highlighted. In addition, the potential for using virtual architectures to encapsulate low-level interaction details was recounted.

These three parts of the overall RODS framework establish a coherent foundation for studying and exploring cognitive support in tools. Up until this point, SE has not had any similar framework to work with. Bits and pieces of prior works were sometimes used in an *ad hoc* fashion, however what was missing was a clear articulation of a high-level, qualitative theory of cognitive support (and ways of applying it). In fact, for the most part, it has not been realized that a principled theoretical foundation might be possible for constructing generalizable explanations of cognitive support. Most related prior work in SE has been either primarily atheoretical, or has assumed a framework for researching theories of *cognition*, not theories of cognitive *support*.

Realizing that such encompassing frameworks are possible is strategically important. Cognitive support has to this point been dealt with theoretically in a piecemeal fashion. Considering the diversity of support research described in Chapter 3, this might not be the least bit astonishing. A coherent, capable framework puts this diversity in a new light. RODS may ultimately be limited in what types of support it can talk about, but it argues that (1) cognitive support might be understood using just a handful of basic principles, and (2) these principles can be stated within a single coherent, generalizable theoretical framework. The basic principles for explaining cognitive support have already been noted elsewhere, but they have never before been united in this way.

Chapter 5

HASTI: A DC Modeling Framework

There are many theories and models that are potentially relevant to systems design, but each is applicable only to narrow psychological phenomena. ... On the positive side, we should feel fortunate that there is so much potentially relevant knowledge to draw on. On the negative side, it is not at all obvious how to identify the knowledge relevant to any one design problem and then package this knowledge into a form that makes the practical implications for systems design more obvious.

– Kim Vicente, “Cognitive Work Analysis” [657].

It stands to reason that anybody wanting to analyze or design cognitive support would need to have a reasonable command of some salient aspects of applied cognitive psychology. Although informal and qualitative descriptions have their place, so too do more formalized descriptions like computational models of cognition. Consequently, pragmatic-minded cognitive support researchers—especially the non-psychologists by trade—would undoubtedly appreciate simple, standard, unified models of cognition. Perhaps ideally these standard models could simply be plucked from the theoretical shelf and used without modification. Certainly, some well-considered cognitive models would be of great use in applying the support principles from RODS. The reason is simple: the support principles from RODS are practically psychology-free. So although they “explain” cognitive support as different sorts of computational advantages, some details of real-world cognition are required in order to understand what those advantages could possibly consist of. Models help answer the question “what aspects of thinking are supported and how?” In other words, the support principles are the necessary intellectual tool for understanding or developing cognitive support, but one still needs some material on which to work this tool. The aim of this chapter is to try to provide some of this needed material.

At first blush it would seem that modern psychology, especially cognitive science, should be very helpful in this matter since they generally propose to crystallize knowledge about psychology in the form of computational models. Since the models are computational, it might seem at first that they should be

easily employed to analyze various computational rearrangements. Unfortunately, this is not the case. Cognitive modeling in psychology is not “standardized” in the sense that a common consensual base model is pervasively assumed. If a modest cognitive model in combination with some empirical evidence for it is a “good thing”, then psychology clearly has far too much of a good thing. Narrow, little cognitive models abound. To the average non-psychologist, cognitive psychology is a dizzying jumble of independent models, and a squabbling cacophony of irreconcilable theories and modeling methods. Seemingly every paper presents a small isolated model, and this makes integration very difficult, if not impossible. Integration of psychological results is probably the primary impediment to anyone trying to analyze or develop cognitive support.

The integration problem is well known to cognitive scientists. Card described it as follows:

Isolated human performance models cannot necessarily be integrated to together to give a larger model. Trying to match up the miscellaneous hodge-podge of inputs and outputs into an integrated whole is one of the chief problems in trying to build a [practice of] computational ergonomics out of isolated models from the literature. [93, pg. 507]

There do exist some efforts on integration like Newell’s high profile work on “unified theories of cognition” [446]. In the domain of programming specifically, unification of past theorizing has been an occasional interest (e.g., Boehm-Davis [61], Hale *et al.* [286], von Mayrhauser *et al.* [675], Pirolli [511]). Even so, these efforts do not integrate enough aspects of human cognition to be especially useful in analyzing cognitive support. To explain cognitive support using RODS, one needs an adequate characterization of the ways of performing the substitution transformations identified by RODS. Clearly, many facets of psychology and environmental interaction need to be modeled for this to be possible at all. This includes being able to model high level thinking and problem solving, modeling the impact of the environment, and integrating details about different cognitive mechanisms or capabilities. Most existing models fail to deliver on more than one of these aspects.

A second challenge is the narrowness typical of the models. They often simply do not address enough issues or phenomena. For the practical modeler, psychology appears to apply Occam’s razor far too liberally and energetically. What is left after the slashing is done is usually a bare bones architecture, or an enormously limited model. Even “unified” models succumb to such evisceration. It is in a certain sense quite astonishing that there are serious proposals to model something as complicated as human behaviour using a few simple computational mechanisms.¹ For instance, it is not too far off the mark to say that the well-known SOAR architecture [446] essentially explains cognition as: a particular form of rule following, with a pervasive and automatic rule learning mechanism. That characterization is surely too crude and a little gruff, but the essence of the critique is accurate. SOAR’s take on cognition is akin to arguing that a Linux kernel is structured as a RISC processor fetching operation codes and executing them cyclically. This description might very well be defensible, but only at a level totally inappropriate for a maintainer who is planning to tinker with the software. To properly understand Linux for this purpose, one usually needs abstractions in the form of higher level computational structures. The exactly analogous argument

¹See e.g. Simon [594, pg. 53] for a good statement of the basic philosophy of this approach to limiting psychological enquiry to simple mechanisms.

is true for cognitive models since developing cognitive support amounts to tinkering with DC systems. Of course, architectures like SOAR can be used to create more specific models by adding knowledge to them. For instance, analysts can program the models with rules [721]—often many, and with a most tedious amount of detail. But one way of viewing this situation is that, in order to be able to say anything useful, the applied modeler has to supply too much additional information. This means the models are too frequently too bare. Not wrong, not indefensible, just under-employed for the purpose of carrying psychological knowledge. The “base” models must aim to *encode at the start* much more information about cognition-related issues, and in that way provide the practical analyst with a more suitable launching point.

The current situation of HCI modeling frequently creates a third challenge: existing modeling techniques can be costly to apply, making them unsuitable for broad-brush analyses [257]. In particular, the amount of effort needed to get a model to say something useful can be high, giving them a low benefit-to-cost ratio. In Section 7.1.1, these costs will be considered in relation to the relative benefits of user testing, but for now, the main point to note is that many situations require simplified models that are easy to apply.

A fourth challenge to analysis is that existing models do not properly insulate them from the complexities of the science base. Simplified, broad-brush models effectively *mediate* interactions between the analyst and the science base from which they draw. One important function of applied models, therefore, is to *index* the underlying science base. Currently a small fraction of the useful knowledge from psychology and HCI are being used. Partly, the issue is one of awareness [263]. An important mediational function of simplified models is that they can provide a convenient access point; if the model is suitably integrative, it can provide access to a greater fraction of the usable knowledge. A second important aspect of mediation is the models must be in a sense “semi-transparent” so that the more complicated and detailed models can be accessed if need be. This is important because when the model fails to address some question—and all simplified applied models can be expected to occasionally cause this to happen—the analyst needs to be able to “look behind” the model to access the related literature. This is not so much a function of the model, perhaps, but of the way the model is documented. Regardless, some way of making the model suitably transparent is needed. For researchers, this transparency is helpful if they find the need to wade deeper into the related literature. For practitioners, this transparency is needed so that they have a smooth way of accessing more complicated models in order to iteratively deepen their analysis.

The four hurdles of model disintegration, structural simplicity, analytic cost, and mediational weakness currently face researchers in SE. One obvious solution is to provide some suitable “encapsulation” [587] or “bridging representation” [28] which integrates and simplifies existing knowledge, and thereby reduces the cost of the analysis and improves access to the science base. In the context of cognitive support, in particular, there is a need for a suitably capable model for reasoning about the logical space of viable forms of cognitive assistance. This is not to say that there are no potentially applicable models—there are plenty—but they can be costly to use, and most tend to stand alone and work poorly together, if at all. There is therefore really no choice but to try to cobble together a working integration that recovers enough structure to be useful. The aim is not to try to abrogate the modeling challenges entirely, for that would be hubris of the highest order. Instead, the goal is to provide just enough of a stop-gap solution as to palliate

the worst ills heaped on the poor SE researcher who is trying to reason about cognitive support. Where prior models individually fail, the hope is that their integration will succeed. Success, in this sense, means that the integrated model will embody suitable abstractions of the science base concerning DC systems, and that these can then be used to reason at a high level about how to reengineer them in a cost effective manner.

Towards this end, this chapter proposes a modeling framework called “HASTI”.² HASTI is a framework for building DC models of joint human–computer cognition. The intent of this framework is to make it possible to perform “quick and dirty” analyses of cognitive systems in such a way that the possibilities for reengineering them are exposed. In building such a modeling framework many decisions must be made about what form it should take. The key issues are (1) choosing the right prior works to integrate, and (2) deciding on an overall framework and principles for integrating them. In other words the main issues are the *content* and *structure* of the models. Although obviously important decisions must be made regarding content, it is actually the integration mechanisms and principles for construction that are the crux of the entire project. For this reason, the integration and structuring mechanisms that were chosen give HASTI its name.

The HASTI framework comprises a set of modeling principles and structuring methods. The acronym HASTI is formed from the names of the five decomposition and structuring methods it identifies: (1) Hardware models, (2) Agent models, (3) a Specialization hierarchy, (4) a Task taxonomy, and (5) an Interaction abstraction layer. Each of these are described in a separate subsection below (Sections 5.2–5.6). Before these can be described, the underlying principles for framework construction must be introduced. The decomposition framework and the way of mapping the HASTI elements to one another is an attempt to, as Vicente says, “package [psychological] knowledge into a form that makes the practical implications for systems design more obvious” [657]. This requires an appreciation of principles for building models that serve these designer needs. These principles are introduced in Section 5.1. Then, after the five HASTI structuring methods are described, a summary is provided in Section 5.7, and conclusions are drawn.

5.1 Overview of HASTI

My experiences have led me to believe that the central problem that needs to be overcome to make the products of cognitive science more relevant to design is identifying a more productive set of dimensions along which modeling efforts can be decomposed. To support cognitive engineering, the decomposition must be derived from an overall framework capable of ensuring that the resulting research products can be reassembled into a coherent theory useful for design.

– Alex Kirlik, “Requirements for Psychological Models to Support Design” [348], pg. 69.

There is not now, nor will there ever be, a “perfect” way of modeling and understanding cognitive support—at least not with respect to the needs of tool developers. Instead of singular perfection, we

²“HASTI is pronounced to rhyme with “tasty”.

are left to aspire to develop a collection of approximate modeling techniques each capable of addressing some particular set of phenomena—each to their own degree of approximation. Pragmatic-minded tool researchers hope to have modeling techniques with exact sort of abstractions and approximations that enable them to analyze and evaluate tools, and to design good ones. In this context, an important concern is modeling just enough detail of a DC system to enable tools researchers to do their jobs. There are very many models and modeling traditions that can potentially be drawn upon in this project. There are actually too many. The primary challenge remains how to gather and integrate appropriate models in ways that are useful.

This chapter is predicated on the conviction that the needs of SE researchers can be met by steadfastly focusing on decomposing cognitive phenomena in a useful way. Thus a key requirement for success is to determine what aspects of cognition are important to consider, and then determine ways of decomposing these. Most cognitive modeling is driven by the desire to model specific phenomena [446], and to establish the constraints on cognitive theories [348] (e.g., timing constraints [446]). The building of HASTI is, in contrast, driven by the varied needs of general design tasks. Kirlik (see the quotation above) hit the nail squarely on the head when he argued that the way to make cognitive science “more relevant to design is identifying a more productive set of dimensions along which modeling efforts can be decomposed” [348, pg. 69]. His particular proposal for doing this differs from HASTI, but the basic argument is the same: figure out a decomposition that exposes design issues. In this regard, the main modeling consideration in question is the various ways in which DC systems are modified by artifacts. One way to see the importance of this is to appeal to the analogy of a Rubik’s cube.³

A Rubik’s cube effectively consists of interlocking rings with a number of ways of rotating them, that is, with a number of *degrees of freedom*. Motion along any degree of freedom moves a collection of coloured squares from one face to another. The Rubik’s cube thus physically implements a way of generating a number of permutations of the original colouring of the cube. Ignore, for the moment the goal of the Rubik’s cube game (uniform colouring of the sides), and consider just the ability to rotate the cube into different configurations. Rotating a Rubik’s cube is, in a way, similar to reengineering cognition: the cube is like a cognitive system in the sense that it has a number of degrees of freedom for changing it; the cube’s various rotational degrees of freedom correspond to the substitution types of RODS; the various different colours and faces correspond to different aspects of cognition that might be changed by applying substitutions. Using this analogy, the purpose of building HASTI is to determine what the various colours mean in terms of cognition, and to match the rotational motions to valid and interesting rearrangements of cognition. As always, analogies must be used carefully lest one pursues one too far, but it is worth pursuing this one a few steps further so that the implications for the rest of the dissertation are made clear. In particular, we can look to the Rubik’s cube to help explain and compare the roles of HASTI and RODS in analysis.

Consider, then, the following imaginary use of a Rubik’s cube. Imagine that an existing cognitive system is represented by some particular cube configuration. The aspects of cognition that HASTI can make

³A cube puzzle with uniquely coloured sides each split into three rotating rings along both axes (9 total sections on each face, like a tic-tac-toe board) such that the faces can be rotated along two axes; the object of the game is to rotate the rings such that the colours of all sections on every face matches the other ones on that face.

statements about correspond to the colours. These are encoded by the coloured squares and their relative positions. The analyst using HASTI therefore considers some salient aspects of the existing cognitive system, and then maps these aspects to particular squares of the appropriate colour. For instance, if the analyst is considering a shopping task, she might note that knowledge about items are mentally considered. So she might “mark” one particular square as “item knowledge”. Other aspects are similarly marked, so that at the end of analysis, the important aspects of the current cognitive system are represented by markings on the various squares. That is, after analysis, the cube models the cognitive system.

Now imagine rotating one or more rings. Each rotation would correspond to an application of RODS, that is, each rotation corresponds to a substitution transformation which modifies the structure of the computations without changing their essential function. Imagine, for instance, the square previously marked “item knowledge” is rotated to another position. Let us say that this change corresponds to a distribution substitution. Such a transformation might be achieved, for instance, by providing a handheld computer so that memory for the item is externalized onto a shopping list maintained by the computer. The analyst could then make a number of other rotations to see the various ways of reorganizing the cognitive system. Click! Now the progress in shopping is recorded externally instead of having to be kept in the head (e.g., by tick marks on the display). Click! Now the planning of shopping steps is done externally (e.g., by having the computer order the list according the aisles the item is in). Each of these rotations identify ways of rearranging the cognitive system in order to support cognition. The model allows the analyst to explore different transformations of the task at the cognitive level—she is responsible for relating these transformations to ways of implementing them in the design of artifacts.

The above imaginary scenario makes it possible to emphasize the purpose of HASTI. The five dimensions of HASTI enumerate important cognitive issues to consider when reorganizing cognitive systems. These are akin to the different colours in the cube. Furthermore, HASTI establishes relationships between the cognitive issues. This is akin to the mappings of colours onto faces. This is important because it hints of dependencies between cognitive aspects. Note that in the Rubik’s cube game it is impossible to move one square without affecting others. Although in that game there is only one winning solution, the general lesson to remember is that certain configurations are considered “good”, and rotating squares to fix up one face of the cube often fouls up another face. A similar effect occurs in cognitive rearrangements in that changes to one cognitive aspect imply changes to another. The cognitive support analyst therefore reasons about design by “playing” with the model to determine winning combinations (see Chapter 7). The tool evaluator effectively uses such a model to determine what moves are made by some particular design. RODS correspond to the ways of rotating the cube, but it is critical to know what is being rotated and what happens when rotations are made. The role of HASTI is thus analogous to the cube itself: it must provide a way of deriving cognitive models that can be analyzed for ways of transforming them. RODS identifies HASTI’s degrees of freedom.

The remainder of this section describes the principles and methods for making such a cognitive modeling framework. First, the basic principles for building the framework are described in Section 5.1.1. Next, Section 5.1.2 describes the way in which HASTI is structured. There it shall be argued that deciding upon the right structural decomposition of HASTI is the key to developing models useful for analyzing cognitive support.

5.1.1 Framework Principles and Strategies

Basic, bottom-up investigations have been the mainstay of experimental psychology and the literature is filled with tens of thousands of experiments demonstrating thousands of regularities in human behavior. Conducting one more such experiment adds, at best, one small piece of information to the existing pool. Thus, I would argue, basic bottom-up experimentation is not the most efficient use of an HCI researcher's time. Instead, researchers should use the existing psychology literature as the foundation for building computational models.

– Bonnie E. John,

Panel statement on “The Role of Laboratory Experiments in HCI” [711], pg. 267.

Five main principles have guided the generation of HASTI. These are summarized in Figure 5.1, and described below. In building an applied theory, decisions are made as to what to include, and as to what form the result should take. HASTI can be described without enunciating the principles guiding its construction, but having a statement of what they are will help evaluate the wisdom of the decisions underlying HASTI's design.

- | |
|--|
| <ol style="list-style-type: none"> 1. model for use 2. favour inclusion, integration, and approximation 3. embody psychological knowledge in low-detail computational structures 4. decompose phenomena according to model application 5. structure models according to phenomena decomposition |
|--|

Figure 5.1: Summary of principles and strategies for modeling in HASTI

Principle 1: Model for Use

One view of the relationship between science and application is that science is like a fruit tree. The fruit on the various branches eventually will ripen. If a human performance model is not very useful yet, just water the tree, give it sunshine and be patient; eventually the fruit will fall into one's hands. Unfortunately, a look at what is available in models against what is needed to do engineering suggest this view is not accurate...

– Stuart K. Card, “Theory-Driven Design Research” [93], pg. 507.

The primary assumption in all applied modeling is that the models should be well suited to the uses they are put. This may or may not correspond to what the basic sciences are interested in using them for,

or to what they are presently good at modeling [587]. This is perhaps an obvious point but it is still worth stating it since it is the most important principle. The main use being assumed here is that the models will be used to reason about ways in which DC systems are beneficially modified by artifacts.

Principle 2: Favour Inclusion, Integration, and Approximation

... the use of a simple theory of cognition, that is, a theoretical model that adheres to the premises of simplicity, provides a viable alternative that ... acknowledges the impossibility to contain in the model the whole richness of human decision making and performance.

– Hollnagle, Cacciabue, and Hoc,

“Work with Technology: Some Fundamental Issues” [312], pg. 12.

HASTI is being designed for broad-brush analysis. As a result, inclusion, integration, and approximation should be favoured. Many efforts in cognitive psychology and cognitive science take an opposing view by favouring accuracy and minimality. The strategy often pursued is to focus on a specific phenomenon (reasoning, learning, perception, language, etc.), and then introduce a limited “microtheory” [446] to explain it. Although this approach has proven to be effective for explaining many phenomena, rarely are the resulting microtheories stitched together. Rather, once a microtheory is proposed, the goal often becomes to try to make it account for as many phenomena as possible. As Green says, “The normal approach to theorising in cognitive psychology is to propose a theory and then to see how much can be predicted from it, squeezing as much juice as possible from whatever fruit it bears” [259, pg. 28].

Freed *et al.* [229] express well the different focus of the applied modeler:

For models meant to be evaluated on the degree to which their performance fits empirical data, a reluctance to incorporate capable but speculative model elements is easily understood. Our goal of predicting performance in complex domains prescribes the opposite bias: If human operators exhibit some capability in carrying out a task, our model must also have that capability... [229]

The general implication is that applied modelers are willing to put up with having inaccurate models if the model usefully addresses the many relevant facets of cognition. It is wise, therefore, to follow the advice [229] of Freed *et al.*: make the initial model too powerful rather than too weak, and extend or refine the model only as required. Include whatever seems necessary, integrate the models, and approximate or idealize when required.⁴ Of course, doing so introduces an important liability: the approach means we may have no realistic hope of “validating” these models to the standards of rigour normally associated with microtheories. This liability must not be completely ignored, but it can be put into perspective. This will be done in Chapter 8. In the meantime, the pragmatic issues still remain: it must be useful.

⁴My choice of the term “integrate” over the term “unify” is intentional. The term “unified” has come to be associated with efforts to explain many aspects of cognition *in detail*. The hope of “unified theories” is that by making models that account for many different aspects of cognition, it will be possible to “pin down” the theories with enough constraints from observed phenomena [147, 446]. In contrast, the aim here is to determine how to abstract any such unified theories and integrate them with other theories.

In applied settings it is more fruitful to admit the approximation, and then merely enquire about when the models are useful. Indeed, it is normally fruitful to *willfully* approximate model features even when more detail is known [335]. In this work, the aim is to be able to discuss many different forms of cognitive support with broad strokes. Broad-brush analysis techniques are “thinking tools” that are frequently needed, surprisingly useful, and in short supply [93, 262, 268, 719]. Broad-brush techniques permit the generation of highly abstract cognitive support arguments (see Chapter 6). This makes them suitable for use in the early stages of design when the benefits of codified knowledge is most effective (Chapter 7). Lightweight approximations must therefore be favoured, at least initially. They can efficiently kick-start analysis. When more specific analysis is needed, the models can be refined *as needed*. This “iterative deepening” approach may be very important for efficiently investigating cognitive support: often times, the analyst may not know at the beginning what sorts of details are needed. The broad-brush models can thus act as “heuristic tools” for narrowing down the focus; for “raising and addressing theoretical and empirical questions” [495, pg. 47]; for allowing investigators to “isolate variables for designing empirical studies with human subjects” [181, pg. 352].

The above considerations establish a heuristic strategy for deciding what to do with modeling contents once they are decided upon: abstract, idealize, and approximate until what is left is highly general and broadly applicable. Card [93] provides one of the most cogent analyses of this design-driven reduction in detail:

The use of idealizations means simplifying a phenomena so that inference about it is tractable. The simplification is achieved by dropping out details that will have little effect on the outcome. But which details will have an effect may depend upon whether one is interested in broad coverage or in subtle mechanisms. The theory-driven design research paradigm is a heuristic for keeping in the idealizations of the theory those details that will matter for some class of design. [93, pg. 507]

This work concentrates on broad coverage; subtle mechanisms are assumed, by default, to be of interest only when they are demonstrably needed. Until such time, the broad-brush models act to help determine these needs.

It may strike the reader that the goals of being inclusive and deliberately approximate are in some sense contradictory. Not so. Inclusiveness in this context means that knowledge about a diverse range of cognitive and HCI phenomena should be collected. But each of these sources of knowledge may have rich details, only some of which are interesting to cognitive support developers on a regular basis. Thus it is necessary to pick and choose from among the detailed riches, and to turn some complicated aspects of these riches into cruder generalizations.

Principle 3: Embody Knowledge in Low-Detail Structures

Knowledge about psychology or HCI can be formalized in a number of ways, from loose English descriptions, to abstract computational architectures, to programs written in LISP. What sort of models are of interest here? The general rule being pursued in HASTI is to encode the relevant knowledge in the features of computational structures with very little detail to them. These are much like structural or architectural diagrams for computing systems. They describe abstract computational structures but without

many of the implementational details.⁵

The reason for adopting this preference for how to encode psychological knowledge is that it will match well the way in which the model will be used (especially Chapters 6 and 7). The primary use for the models is to provide possibilities to argue about how artifacts rearrange DC systems. Emphasis is therefore rightly placed on making plain and explicit all of the features of DC systems that are essential to these arguments. Any relevant feature from psychology or HCI should therefore have a visible and separable presence in the model. Other models, such as SOAR [446], propose a simplified base architecture and then embed much of the relevant psychology in diffuse and intangible rule sets. In this work, the aim is instead to embody the relevant knowledge in relatively simple computational structure models.

Principle 4: Decompose Phenomena Strategically

Often the classificatory exercise is an integral and inextricable step in the development of theory. The resultant classificatory system provides a consistent conceptual framework, the elements of which eventually are to be utilized in the interpretation or prediction of behavioral phenomena.

– Fleishman and Quaintance,

“Taxonomies of Human Performance: The Description of Human Tasks” [222], pg. 47.

The purpose of HASTI is to provide material with which to reason about cognitive support in DC systems. Thus it is important to have an inventory of the distinct and changeable aspects of DC systems and how they relate—to know a system’s logical elements, their “degrees of freedom”, and constraints on modification. In fact, it is arguably *more* important to simply have an inventory of these aspects than it is to have predictive accuracy. Rasmussen keenly appreciated this fact. He argued that in order to build models appropriate for design, it is important to decompose the relevant phenomena appropriately [526] (also, see Green [259]). This observation led him to develop two related taxonomies, one of which is incorporated into this work. Natural phenomena must be decomposed in some way in order to understand it, but the point is the decomposition must be strategic: it must carve up the phenomena in ways that are design-relevant. The way that this is actually done in HASTI is described in Section 5.1.2.

Principle 5: Match Model Structure to Phenomenon Decomposition

Rasmussen argued that if one decomposes phenomena well enough, then independent models can be used to model each separately [526]. So, for instance, he proposed to classify human behaviour into three types: skill-, rule-, and knowledge-based behaviour (SRK). He proposed distinct general models to account for each type of behaviour. He also proposed *an integration model* that relates the SRK categories

⁵Make no mistake: these *are* models even if they are abstract. Some authors, such as Vicente [657] and Dillenbourg *et al.* [181], exercise considerable evasiveness about whether their models are indeed models. Rasmussen’s processing model is emphatically declared not a model but a “framework” [657], even though it is clearly similar to other high-level layered cognitive architectural models (see, e.g. Brooks [69]). In fact, it very much resembles the multi-stage processing model that Norman [466, 467] holds up as the type of approximate models that need to be developed for HCI.

to one another, and explains the general rules for how they coordinate in real-world cognition. Thus, in reality, he proposed a model with two different modeling levels: the “base” models that modeled specific categories of behaviour using independent mechanisms, and an integrating model that captured aspects of the interplay between these mechanisms. In other words, the base models modeled distinguishable categories of phenomena, and the structure of the integrating model matched the structure of the decomposing taxonomy. *This is the basic recipe for success when integrating heterogenous aspects of cognition into computational models.* Although this modeling scheme is not emphasized in Rasmussen’s exposition, it is an absolutely critical strategy. Generally speaking, the principle is that a model’s structure will in some way mirror the decomposition of the phenomena.

5.1.2 Structure Overview

We knew of many phenomena scattered in the literature of psychology ... that would be helpful for system design. ... To someone who is not a specialist, such as a designer, this literature appears disorganized and contradictory. Psychologists love to split hairs and find small contradictions in published models. The robust but approximate generalizations that might be made to work for engineering tend to get trampled in the debates.

– Stuart K. Card and Thomas P. Moran,

“User Technology: From Pointing to Pondering” [92], pg. 186.

In any integrative effort, *contents* must be decided upon. According to the strategies described in the last subsection, this involves generating a decomposition of these contents suitable for design-relevant analysis. After content and decomposition scheme are assumed, decisions must be made as to how to reflect these in the modeling framework and its structure. This subsection provides an overview of these decisions. The result is a list of five key structures of HASTI (hardware model, agent model, specialization hierarchy, task taxonomy, and interaction decomposition hierarchy).

These five structures are discussed in more detail in separate sections following this overview. The main purpose of the overview is to provide a *guide* to the rationales for why these five structures take the form they do. These rationales are given in later sections, but they are camouflaged by the detailed expositions. As a countermeasure, this subsection provides an simplified account of the content being integrated, the decompositions of this content, and the model mapping techniques being used. In other words, these explain the way that principles 2–5 from Figure 5.1 are observed. In addition, each section below describes the relevance that each structure has in analyzing cognitive support, i.e., how principle 1 from Figure 5.1 is observed. The decompositions, matching model structures, and rationales are summarized in Table 5.1. The table highlights the key issues HASTI addresses: the phenomena or issues to model, the way of decomposing these, the way of modeling them, and the relevance that they have to analyzing cognitive support.

PHENOMENON / ISSUE	DECOMPOSITION	MODEL FEATURE	SUPPORT RELEVANCE
H invariant capacities/constraints	memory, processors	Hardware model	distribution, performance
A behaviour	agents, panel types	Agent model	distribution, alg. opt.
S specialization (adaptation)	SRKM	layering	specialization
T task/problem	D2C2 taxonomy	partitioning	overheads
I interaction	virtual architectures	layering	generalized task analysis

Table 5.1: Decompositions of phenomena and matching structures

Capabilities and Constraints: the Hardware Model

The Hardware model captures task- and knowledge-invariant capabilities and constraints within DC systems. Thus the Hardware model is used to encode basic physical and psychological facts about a joint system. This is a common use for hardware-level models. For example, cognitive constraints such as short term memory limits are frequently modeled as resource bounds in a cognitive architecture. Since the functional or behavioural aspects of the system are orthogonal to this hardware aspect, there is usually some sort of *mapping* between these other decompositions and the hardware level. In computing science circles, such mappings are routine. A prime example is the so-called “4+1 viewpoints” framework [364]. The Hardware model decomposes basic capabilities and constraints in terms of memories and processors. These are important to know for reengineering cognition because they state what computational resources are available for distributing computations onto. The encoding of psychological constraints are also important for reasoning about performance issues in the cognitive system.

Behaviour: the Agent Model

The Agent model captures abstractions of behaviours in the DC system. Humans do not switch randomly between actions, but rather alternate between goal-focused clusters of activities. Thus cognition may be decomposed into task-, or goal-relevant behaviour. The structure of this decomposition does not directly mirror the physical or logical structure of a DC system, so a mapping must be assumed. An Agent model is a way of matching the system structure to behaviour structure. Agent models encapsulate logically cohesive mechanisms that generate coherent action. A generic template for an Agent model is proposed with the aim of capturing common aspects of behaviour. This template includes a taxonomy of six different types of data that the Agents can process: ends, operations, constraints, goals, plans, current state, and past state (history). The Agent model is mapped down onto the Hardware model in order to reason about constraints and behavioural regularities. The Agent model plays a key role in discussing cognitive redistributions. Specifically, it is the primary source in HASTI for identifying generalizable (i.e., task-independent) aspects of cognition that can be rearranged.

Specialization: the SRKM Strata

The SRKM strata impose a partitioning of behaviour according to levels of adaptation, that is, according to levels of *specialization* in the mechanisms responsible for the behaviour. These closely follow the SRK taxonomy of Rasmussen [526, 528, 529]. Rasmussen argued that there exist genuinely different categories of behaviour that people exhibit. He called these “skill-based”, “rule-based” and “knowledge-based” (hence the name “SRK”). These behavioural categories are related to how a system is *adapted* to tasks. Notably, these categories are not related by *function*, or *goal* (as the Agent behavioural decomposition is), but by class of adaptive response to task demands.⁶ Generally speaking, several of these categories of behaviour may be involved in performing a single coherent goal-related activity. Each category is associated with different classes of cognitive behaviour. For instance, skill-based behaviour is associated with rapid perception–action loops that are not cognitively penetrable by the performer. The three SRK categories are ordered according to their relative degree of adaptation (i.e., degree of specialization), with $S < R < K$ (where $A < B$ means A is *more* specialized). A fourth is added, called “meta-cognitive” behaviour, and so the taxonomy is expanded into a “SRKM” taxonomy. These behaviour categories are integrated into HASTI by imposing modeling layers or *strata* onto the Agent model. Agents belong to one particular stratum. Each SRK category is mapped to different Hardware mechanisms, which is used to explain their differing performance characteristics. The importance of SRK stratification is that it provides a way of labeling the degrees of computational specialization within the model. These are needed when discussing the types of specialization within DC systems.

Task type: the D2C2 partitioning

The D2C2 partitioning impose a partitioning on behaviour based on a classification of task types. It is possible to model DC activities as searches within problem spaces [727]. It can be helpful to decompose this problem space according to some meaningful problem taxonomy. Doing so not only partitions the problem space into identifiable components, it can inject a useful vocabulary for the analysis of cognitive support. HASTI proposes a four-fold classification system for sub-problems, i.e., for “*tasks*”. These classifications are: **Domain**, **Device**, **Coping**, and **Cooperation**. The taxonomy is thus referred to using the acronym D2C2. The taxonomy imposes a partitioning on the Agent model. Thus it is used to classify what type of activity each agent is responsible for. This vocabulary and method for tagging task types in this manner will be important for reasoning about cognitive support; in particular they are needed to trace how cognitive overheads within a DC system may be reduced.

Interaction: the Virtual Architecture

Virtual architectures generate a layer of abstraction to encapsulate regularities in interaction. It is easy to get bogged down in interaction details if one has to reason about behaviour at a low level. It is important to be able abstract some of the interactive behaviours so as to be able to consider them at higher levels of

⁶The use of the term “behaviour” by Rasmussen is perhaps a little confusing, but it is not unreasonable. The Agent model is proposed to decompose behaviour according to common goals; the SRK taxonomy proposes to decompose behaviour in an orthogonal way—by level of adaptation in the behaviour-generating mechanism.

granularity. As was discussed in Section 4.3.3, the way to do this in computational models is to postulate a virtual architecture that the DC system simulates. Virtual architectures identify classes of common interfaces, such as direct-manipulation interfaces to external memories. Virtual architectures map down onto lower-level simulating architectures (there may be, in principle, a deep hierarchy of these). The lowest of these is always the Hardware model. Depending upon the granularity of analysis, the Agent model can map onto a Virtual architecture. So far in this work, a simple virtual architecture has been mentioned—a Virtual Memory architecture for abstracting the interactions involved in managing an external memory. In Chapter 8, a “virtual blackboard” architecture is also used. Building a catalogue of common and reusable architectures might be very helpful to designers, however such a catalogue is outside the scope of HASTI. At present, HASTI merely explains how such architectures fit into HASTI-based analyses.

5.2 Hardware Model: Cognitive Capacity Decomposition

In debugging a program, in writing a paper, in doing financial analysis of a firm, in attempting to reason about a machine, limitations on the number of mental things that can be kept track of lay a strong constraint on human cognitive capabilities.

– Card *et al.*, “Window-based Computer Dialogues” [95], pg. 241.

The hardware model describes aspects of a DC system’s structural decomposition, as well its stable constraints and capabilities. Colloquially speaking, the hardware model factors out the basic computational hardware so that other aspects of psychology and interaction (e.g., knowledge) are considered “software”. In a joint DC system, this description includes both the computer hardware and human cognitive architecture. So the background assumption is that what is being modeled is a multi-node distributed computing system. HASTI treats the computer simply as an external memory and processing system that can be interacted with. This simple treatment is depicted in Figure 5.2. For example, a direct-manipulation interface to a program editor can be abstracted in such a manner. Non-computing artifacts may be modeled similarly by taking out the processor. In the remainder, the main focus will be on just one computing node: the human. The aim is to describe an abstract cognitive architecture for users within the DC system.

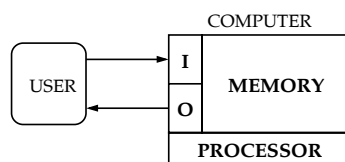


Figure 5.2: Simplified joint system model

There are a great many psychological aspects that might be considered for a hardware model. Some representative sources of this sort of knowledge from HCI include the “Model Human Processor” [94], Mayhew’s overview [401], Barnard’s “Interacting Cognitive Subsystems” model [27,28,30], and the EPIC

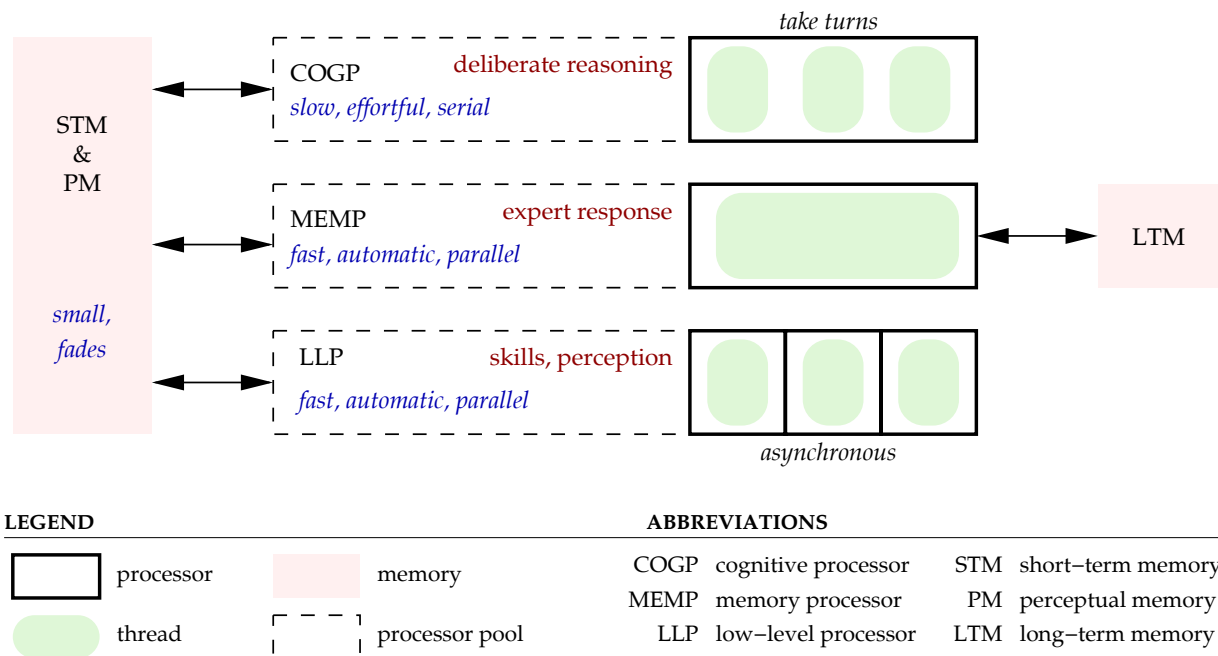


Figure 5.3: Hardware level of description of a user

model [343]. Some representative examples from cognitive science include the cognitive architectures SOAR (e.g., Newell [446]) and ACT-R (e.g., Anderson [13]). Understanding all these (and more) is difficult for the non-specialist. What is more, not all of this knowledge will yield benefits proportional to their cost of acquisition. Following the principle of inclusive approximation, salient generalizations and approximations must be picked such that useful reasoning about cognitive support can be established. In this work, only a few basic items are considered. Primarily, these establish constraints and capabilities needed for other aspects of the models, and for reasoning about the need for cognitive support.

The hardware model is summarized in Figure 5.3, and described below. The intention of the model is to let the diagram encode the salient psychological knowledge. The following aspects are indicated by the figure:

1. **Processing.** Cognitive processing is decomposed into several interacting threads, and these threads are distributed onto three pools of processors. The first pool consists of a single processor called COGP. Threads running on this processor execute serially, although they may switch off as in typical multi-threaded uniprocessing systems. The second pool consists of the single processor MEMP with a single thread running in it. MEMP contains a single thread that accesses LTM. The third pool contains many processors operating in parallel, with a single thread assigned to each. These act much like independent “peripheral” processors [343]. Many similar sorts of multi-processing models are assumed in cognitive and agent models (e.g., Hayes-Roth [297], EPIC [343]).

These processors are named to indicate their purpose, and labelled with their main qualities. These crudely approximate a number of general features of human psychology:

- (a) The COGP models deliberate, introspectible, high-level cognitive processing. Each thread corresponds to some coherent cognitive processing activity. Humans frequently interleave many of these coherent activities, such as when they switch attention between editing a program and writing some email. COGP carries out operations slowly compared to the MEMP and LLP pools. Because multiple threads share the processor, it models limited resources of humans to focus attention.
 - (b) The LLP processing is associated with low-level skills and capabilities such as perception and psychomotor control. These are known to be generally automatic, asynchronous, fast, and opaque to introspection.
 - (c) The MEMP processing is associated with memory-based processing such as recognizing meaningful stimuli and acting upon stored rules.
2. **Memory.** Memory is divided into short-term (STM), perceptual memory or buffers (PM), and long-term (LTM). The PM and LTM are like unstable register pools—fast but limited. LTM is effectively unbounded and permanent but it cannot be directly accessed from by COGP. These are named to indicate their purpose, and labelled with their main qualities.

This crudely approximates several basic features of memory, such as the notorious limitations of short term memory, and the relative independence of different memory buffers (e.g.,). Similar models are frequently found in HCI models (e.g., Mayhew [401], EPIC [343], Barnard *et al.* [27], Card *et al.* [94]).

Within SE and software comprehension, some models of cognition share commonalities with this model (e.g., Shneiderman *et al.* [584]).

This model is bereft of any description of the “software” running in the processors (data types, behaviour, perceptual function, etc.). It selects and models a few key capabilities and limitations of human psychology. The intent is to be maximally useful with a minimal model. If the analysis needs to be iteratively deepened, then the more detailed models cited above could be turned to. As a model for predicting human performance, it may leave much to be desired. However as a model for “quick and dirty” reasoning about performance implications of design decisions, it may suffice. In fact, once the analyst is familiar with Figure 5.3, A simpler “model” can also be rendered as two collections of mental capabilities:

$$\{ \text{COGP, MEMP, LLP} \}, \{ \text{STM, PM, LTM} \}.$$

These terms are indexes into simple psychological knowledge.

5.3 Agent Model: Behavioural Decomposition

The Agent model is a schematic model intended to allow the analyst to model aspects of the behaviour of a DC system. Human cognition does not flit from moment to moment—it is characterized by episodes of coherent, purposive, goal-driven, semi-organized activity. In addition, these episodes tend to be organized schematically and hierarchically (e.g., see Taylor [634], Nielsen [456, 459], Rasmussen [531], Moran [418], Bass *et al.* [36]). Even so, this overall coherence is complicated by the interleaving of various activities, and

by interruptions, breakdowns, and backtracking (e.g., Davies [163]). Many different accounts have been drawn over the years for various aspects of this behaviour. What is needed in this work is a simplified account that can model some mechanisms causing this behaviour. The aim is to provide a simple but broadly-applicable schematic model to describe generalizable aspects of this behaviour. Analysts could adapt this model to suit their particular analyses.

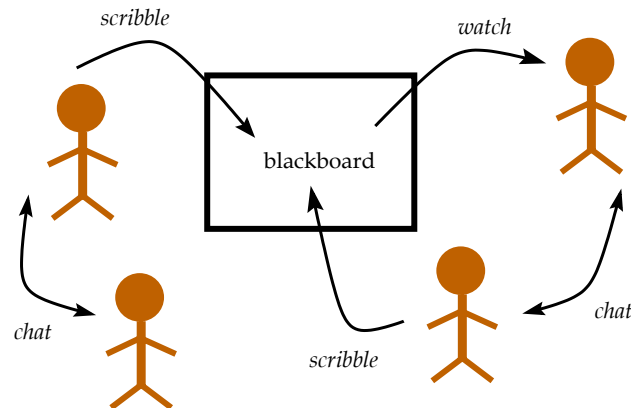


Figure 5.4: Problem solving metaphor of experts working around a blackboard

This need is met by a schematic, *agent*-based model of cognition—specifically a type of *blackboard architecture* (for our purposes blackboard architectures differ from agent architectures in unimportant ways [151]). Blackboard architectures are well-studied in AI, and have a number of descendents and variations. Luckily, HASTI needs to include only the gross architectural features. As a consequence, the needed concepts are described only briefly here and the reader is referred to the existing literature if a more detailed treatment is called for.⁷ Blackboard architectures originated from a problem solving metaphor advanced by Newell [442]. Newell’s original metaphor was of a group of experts collaborating on a shared problem by standing around a blackboard and (more or less) taking turns. An illustration of one variant of the metaphor appears in Figure 5.4. An agent model is proposed as an implementation of this metaphor. It has two key characteristics: it proposes to encapsulate behaviour-causing mechanisms using an *agent* abstraction, and it proposes a classification system (or ontology) of *cognitive resources* that serve to coordinate and organize cognitive processing.

Cognitive resources are function-related collections of data that are cooperatively processed by the agents. A classification of four types are proposed: a *problem* description, an *agenda* of goals, a *control panel* for holding control representations, and a *progress* record. Each of these cognitive resources is in turn composed of a collection of typed data values. To reiterate, *resources* are function-related collections of

⁷A short bibliography includes: Garlan and Shaw [580], Nii [462,463], Craig [151], and Corkill [150]. The reader is directed specifically to the review of blackboard systems by Carver and Lesser [113], the multi-level blackboard architecture of Hayes-Roth [295], and the model of expert decision making by Vinze *et al.* [571, 659, 660]. The first work presents a breakdown of problem solving techniques that aligns well with the breakdown presented here, the second describes a system that mixes autonomous agent processing and a shared thread of control (which will be adapted for use here), and the third gives an example of the application of blackboard models to modeling human thinking that is similar to the one presented here.

data; there are thus two classification systems in the ontology: resources, and data types. This ontology is briefly summarized in Table 5.2. Each of these four resources are described in more detail in the subsequent subsections (5.3.1–5.3.4). These resource types are commonly found in prior works on modeling cognition. The ontology is consistent with other collected accounts of cognitive resources (e.g., Wright *et al.* [719], Howes [314]). The agent model proposes a general scheme for cooperatively processing these cognitive resources. It maps the cognitive resources to data stores. It also models several aspects of how agents behave. This agent model is further described in Section 5.3.5. Afterwards, in Section 5.3.6, a short summary is presented in combination with an example of how this model can be used to interpret prior cognitive modeling work in software comprehension.

RESOURCE	DATA TYPES	PURPOSE AND ASSOCIATED FUNCTION
Problem	ends, operations, constraints	establishes long-term objectives to achieve, and the ways of achieving them
Agenda	goals	used to juggle multiple goals at multiple levels
Control Panel	plans	represents control information to structure activities, and resolve ordering constraints
Progress	curr. state, history	defines current problem solving state and history

Table 5.2: Cognitive resources in the agent model

5.3.1 Problem (ends, operations, constraints)

It is typical to say that users work to solve *problems*. In this sense a problem defines the context in which it is possible to understand the actions and motives of the problem solver (see e.g. Simon [594, ch. 3], Vicente [657, ch. 4]). In cognitive science and AI, defining what these problems are has been treated as the challenge of defining a “problem space” (e.g., see the classic book by Newell *et al.* [449]). A problem space is, essentially, a graph of legal states. There may be several possible ways to represent or define this logical object, but a reasonable one is to define three things: an *end* to achieve, a set of conceivable or possible actions or *operations*, and a set of *constraints* on the actions. Problem solving is portrayed as the process of selecting appropriate operations, subject to the constraints, such that the end is achieved. The terms “end” and “goal” are sometimes considered synonyms, but in HASTI, the term “goal” is used in a specific sense. Because goals and sub-goals may be generated by the user in solving a problem, a term is needed to distinguish the long-term objective defining by the problem, and the shorter-term objectives adopted to try to solve the problem. HASTI thus reserves the term “end” for use by the analyst in modeling longer-term problems, and “goal” for shorter-term objectives. A similar distinction is sometimes made between an “activity” and a “task”, with the idea of an activity being roughly the same as how HASTI treats a problem.

The above way of defining a problem is known to raise a number of difficulties. These can be quieted somewhat by relaxing the above definitions to the point where the difficulties no longer hinder informal reasoning. Since HASTI is being proposed to be useful in analysis of real user problems, it is necessary to

briefly mention these analysis difficulties and to overview how they are resolved.

One potential difficulty is that it may be hard to find a user-independent context from which it is possible to understand the user's problem. In classic problem solving, the problem space is normally defined independent of the problem solver. This fact lead Goel to remark that "It is rather puzzling that the goal should be part of the task environment rather than the agent. In any event, goals are assumed but not explained in information processing theory." [243, pg. 84 (footnote 2)]. This puzzle can raise confusions on how problems might be defined. In particular, the difficulty stems from the fact that what a user does can often be understood only in terms of the user's own conception of their problem. For instance, the problem of writing a PhD proposal is one that is only reasonably defined in reference to the PhD candidate (e.g., their interests). Thus one is lead to the curious conclusion that the context for understanding a user's action is dependent upon the user's own knowledge and belief system. This possibility does not pose an irreconcilable threat to the problem space definition, although it does become recursive and thus difficult to reason about. Note that this problem is *not* the same as the problem of correctly understanding a work domain. In well-established problem contexts (e.g., controlling a nuclear power plant), it is reasonable to establish a firm worker-independent understanding of the problem domain. However doing so may require a distinguished point of view (see Zhang *et al.* [727] and Vicente [657]).

A second potential difficulty is that "the problem" may seem to be poorly defined. In other words, the user's *end* may only be vaguely understood at the start. The classic case the ill-structured problem of design [108]. In design, inadequate knowledge of the problem and constraints is paradigmatic; problem solving therefore involves problem discovery or "problem setting". Whiteside and Wixon elaborated on these problems of goal settings:

In a tightly defined, carefully controlled laboratory situation, where the user has been given a set task, goals are perhaps identifiable. However, in a more representative situation, users are often unable to state unambiguous conditions of satisfaction, and are dependent on the context in which they are operating. To the extent that we can identify the users goals at any time, they are not organized in any strict hierarchy and are radically transformed as events unfold. Further, users are constantly distracted from their goals, make up goals as a rationale after the fact, and state goals in general and often vague terms. In general, we see users acting first, thrown to the situation, and perhaps devising a goal, a posteriori, if asked to. [702, pg. 359]

In a similar way, but to a lesser extent, the constraints and possible actions are also dependent upon the user's conceptions of their problems, and can seem poorly defined at the start. Once again, recursion can be applied to resolve the issue. Thus one might wish to think of problem setting as just another problem, albeit a distinguished one that supervenes on other dependent ones. In this view it just defines a sub-space of the problem space (e.g. Kim *et al.* [345]). Problem solvers then act to refine their solution and problem space concurrently.

A third potential difficulty is that problems a user faces may change just by working on them. Specifically, a recorded history of problem solving activities can change the operations that may be performed. This case is covered below in the section on *progress*.

To use HASTI during analysis, the analyst determines what Problem is being solved, and then tries

to model it in terms of ends, operations, and constraints. For example, an informal analysis of coding in Java might define the end to be the generation of a correct Java program, the operations to be code writing actions, and the constraints to be Java programming language constraints.

5.3.2 Agenda (goals)

Users are goal-directed. They do not simply react on a moment-to-moment basis in response to stimuli reaching their senses. To achieve this, they need to maintain some sort of internal state, which we can call “goals”. They adopt various goals to achieve and work to try to achieve them. But they also interleave the pursuit of multiple, possibly even conflicting goals. As Green says:

...the [users] were willing, within the limit of the tools they were using, to work on whatever goal came to mind, as long as it could be approached without undue difficulty. They did not pursue the strategy of taking one task goal, breaking it into subgoals, solving each subgoal in turn, and then taking the next task goal. Such a strategy is implied by simple theories of performance, but a more realistic view appears to be that users keep an agenda of unsolved subgoals, and can proceed with any one of them at any time. ... the nearest approach to such behaviour in the literature on computational models of planning is the ‘opportunistic’ planning model ... [259, pg. 29]

The agent model includes just this sort of agenda of unsolved subgoals. The agents respond by trying to achieve specific sorts of subgoals. They also manipulate the agenda by posting new goals or refining them. It can be assumed that many of these goals are associated with plans or sub-plans on the Control panel (below).

To use HASTI during analysis, the analyst would determine the potential Goals that might be adopted in order to solve the Problem. For example, in coding it might be reasoned that programmers adopt goals of defining a collection of functions implementing a module, sub-goals of writing a function definition, and so on. The user might then juggle these various goals [267].

5.3.3 Control Panel (plans)

Human action is planful, but not “mechanically” so in the sense that plans are rigorously followed without alteration. Plans are posed in order to achieve a goal. There are two important aspects of planning: *intention* to act in the future (“I plan to fix this up later”), and structuring the action (preparation [293]) to account for *constraints* between acts (as in “No son, first we put on our socks, then we put on the shoes”). In simplistic models of planning the structuring amounts to generating a full control plan. This is now perceived by many as an unrealistic model of human planning (e.g., see Young *et al.* [722], Suchman [623]). Instead, plans are seen as resources that structure but do not entirely dictate action (e.g. Wright *et al.* [719]). Even so, plans are essentially a data encoding of control information. In the agent model, this is modeled by instantiating the plans on a shared panel. This can therefore be called a “control panel” in which partial control information is stored. Any agent, including asynchronous ones (memory processors, skills, or perceptual processors), can act to generate or modify plans, fulfill a plan’s subgoal, act to signal a plan update (e.g., see Hayes-Roth [295], Green *et al.* [274, pg. 32]). It is of dubious value to try to specify in this

broad-brush model how plans are represented internally. However it important to point out that plans are usually presupposed to be hierarchically structured (e.g., see GOMS [94]). Plans, in particular, decompose into sub-goals which are in turn may be achieved by sub-plans.

To use HASTI in analysis, the analyst would determine the operations needed to perform tasks and the orderings and constraints on this action. For instance, in coding tasks, the analyst might note that design documentation (e.g., a UML model) serves as a plan for code generation.

5.3.4 Progress (state, history)

Problem solving can be said to involve the progression through a sequence of states in the problem space. Part of the data involved in defining this progression is the states themselves, and the sequence that is progressed through, that is, the *history*. Each state is defined by (1) the a partial solution being constructed, (2) the internal problem solving state, and (3) recorded problem solving history. For instance if a person's problem is to write a sentence about how they feel about spam email, then a problem solving state could include the partial sentence as it is being constructed, plans for structuring unconstructed phrases, and a memory for the sub-goals that have been fulfilled, and for the phrase ordering possibilities that have been discarded. Storing progress makes it possible to *backtrack* or otherwise use the history to determine moves to perform. Thus maintaining a history provides an evolving session-specific set of *operations* to the problem space definition.

To use HASTI in analysis, the analyst would determine what constitutes the evolving solution and memories of past history of those states (or of a sequence of operations to recover such states). For instance, in coding tasks, the analyst might note that the evolving program, the current statement the coder is working, and test data are all part of the current state. History could include previously attempted solutions. An "ideal" [267] could code unsupported by external memories, and could thus hold all of this data in memory. In realistic situations, of course, the developer is supported by an external memory [260], but if the evolving program is not recognized as part of the cognitive system, then the support of the external memories will not be recognized.

5.3.5 The Agent Model and Its Mapping

The base Agent model is a relatively standard blackboard model with multiple "panels". Panels are a way of logically partitioning the blackboard contents according to their functional roles [660]. Each agent encapsulates a coherent behaviour, activity, or function. This form of abstraction differs from cognitive processing decompositions based on cognitive capability (planner, inferencer, etc.). The agents are fruitfully viewed as experts in performing some particular operation based on the long-term knowledge that they are familiar with. This view accords with Newell's original metaphor; it is the reason that the agents are called "knowledge sources" (KSs) in the lexicon of blackboard systems [463]. It is important to note that the agents are proposed as *abstractions* over the "software" of the mind. A variety of ways of mapping these abstractions onto lower-level descriptions of that "software" are possible (e.g., mapping to related *rule sets* in a rule-based model [65]). This mapping generates performance implications.

The blackboard metaphor argues that each agent will be activated when they have an interesting addition to make towards the problem solution. They will thus read or write to the various panels as needed. This essential insight is used to explain the key property of *opportunism* in human behaviour. For instance, one agent may know how to generate a partial plan for a particular goal on the agenda. It could then post this plan on the plan panel for other agents to resolve. Alternatively another agent may know how to modify the current solution state to achieve a sub-goal. Thus the blackboard architecture treats solution processing (progress) and control processing (plans, goals) uniformly [113]. In addition, it is assumed that there are a collection of agents that cooperatively process perceptual and motor activities. These agents have read access to all of the above four panels, but they also have full access to relevant perceptual memory. The overall architecture is illustrated in Figure 5.5. In the figure, the dashed line is intended to indicate that many different agents could be defined (i.e., it is a modeling framework).

The analyst can use such an abstract model to encode a variety of goal-directed behaviours. This might be done at a high level of granularity within informal design settings. For instance, the analyst might note that when developers are investigating code to make a bug fix, they often switch between activities of searching files, reading documentation, and running test cases. Thus she might draw three different agents labelled “search”, “read dox”, and “run tests”. She might then argue that as the developer performs these activities a plan for making a change is formed. She can then write “bug fix plan” on the Control panel. Such a bug fixing plan might, for example, consist of a sequence of three steps: (1) add new function variant, (2) search for old uses, (3) switch to new function where appropriate. In general, the Agent model is used to model the cognitive activities involved in performing some task. It is also used to expose the types of data being processed and then categorize them into function-related roles.

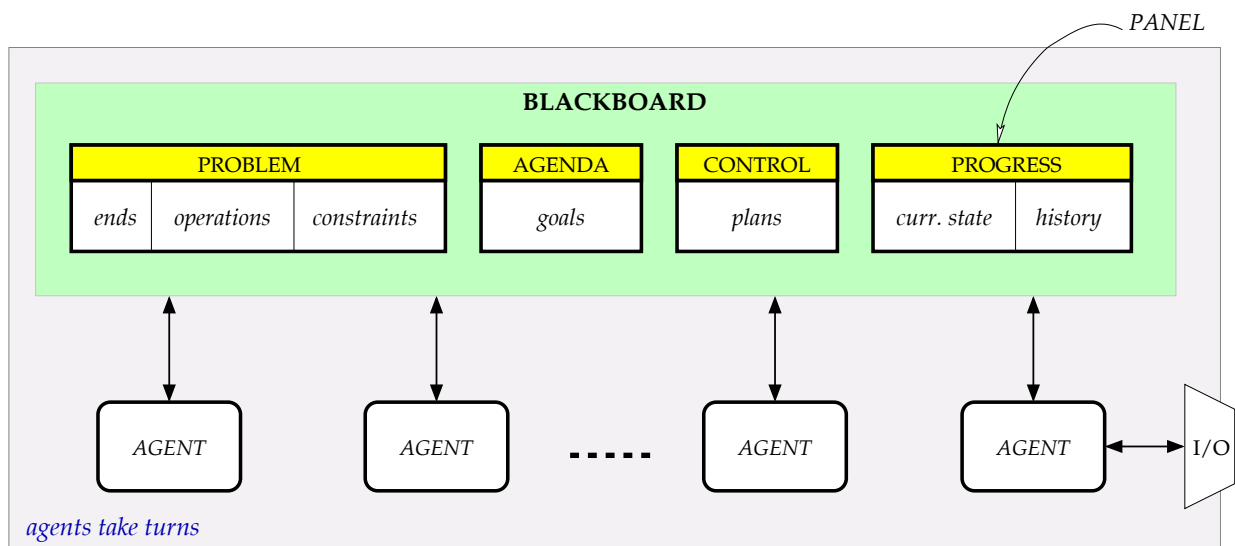


Figure 5.5: Schematic Agent model showing generic agents, panels, and panel data types

Mapping onto Hardware

This agent model is mapped onto the hardware model as shown in Figure 5.6. The blackboard is mapped onto the memories. It is assumed but not explicitly shown that the blackboard contents are held in some combination of LTM and STM. Thus the analyst only is aware that often the contents will be subject to STM limitations, and sometimes will persist. That is, the contents of the blackboard may be forgotten if too much data is needed at once, and some parts of it persist for long terms (e.g., remembering where one left off when returning to a programming problem after lunch).

The I/O-processing agents are mapped onto the LLP processor pools in the hardware model. This allows them to run asynchronously. Flexibility in the mapping is allowed for the remaining agents. Some agents are allowed to be skill-based and mapped to the LLP pools. Other agents are mapped onto distinct threads of the COGP. In most threading, threads are switched according to priority and runnability. Specifically, certain agents are suspended until such time as their potential contribution to problem solving is elevated enough (e.g., see Carver *et al.* [113]). This suspension might be as a result of waiting for a particular event or contents of the blackboard (i.e., blocked on an input or awaiting a signal). This scheme allows both cooperative threading (e.g., turn taking) and competitive threading (e.g., competing goals). Action of the group as a whole will be of a turn-taking nature when they require access to the same blackboard contents. Such turn-taking indicates the contents have a coordinating role [333], that is, they

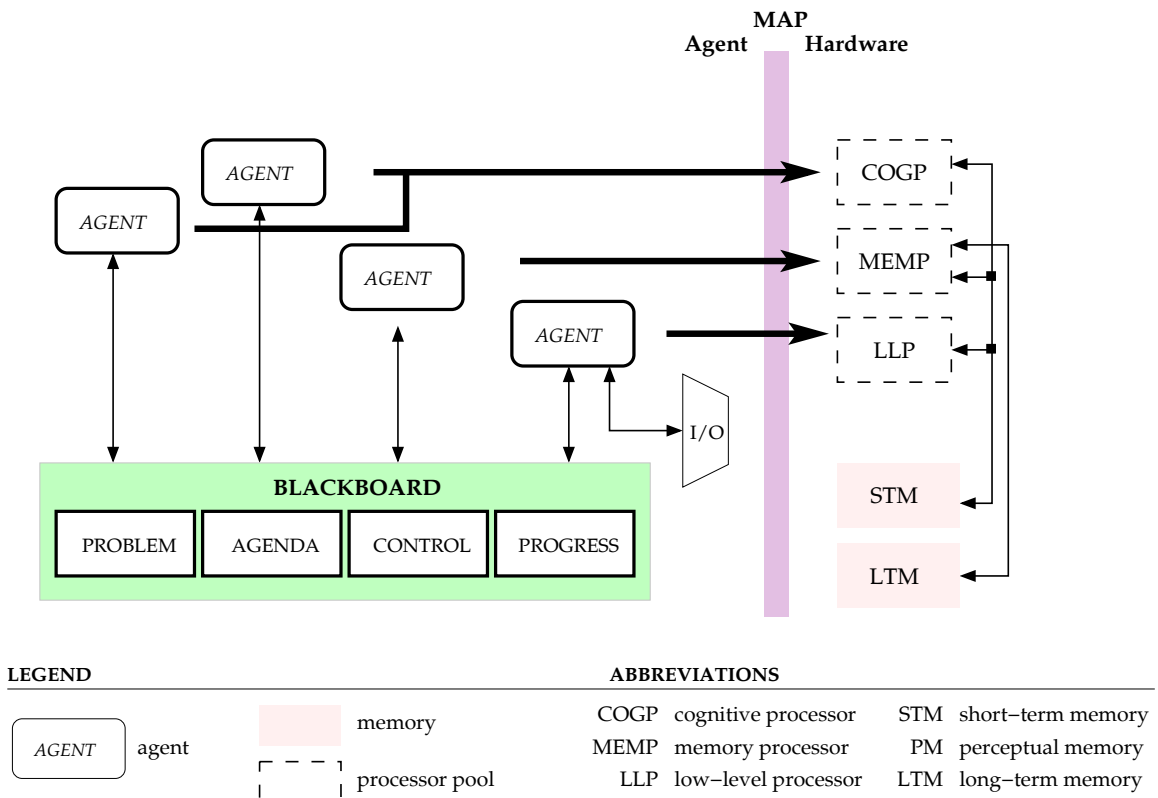


Figure 5.6: Mapping of Agent models to Hardware

affect control [571] of thought and action. Agents doing rather different work may amicably take turns, or an agent may suddenly jump in and interrupt after observing a situation where it thinks it can make important progress. Agents with differing ideas on how to solve a problem may compete for attention.

This is a simplified model that could be expanded upon as needed by consulting other published models from related domains (e.g., Visser [661], Whitefield [699], Vinze *et al.* [660], Zachary *et al.* [723]). This model and mapping can model several important aspects of human psychology:

1. The fact that important parts of the blackboard panels are mapped onto STM means that the limitations of STM affect the capacity to hold complicated contents. That is, human cognition is *resource-bound*. There are limits to the depth and breadth of the agenda, control, problem, and state that people can remember at one time. Furthermore, these things are prone to being forgotten or pushed out of memory, resulting in slips and errors. For instance, the depth of planning is limited, and sometimes plan steps are forgotten (e.g., see Young *et al.* [722]).
2. Mapping agents to separate threads running on a serial COGP models opportunistic switching between different tasks, sub-tasks, or goals based on the recognition of opportunity [113,299,489,661].

5.3.6 Summary

The agent model presents two decompositions mechanisms: (1) cognitive processing is decomposed into a collection of agents that opportunistically switch between coherent goal-related actions, and (2) data contents are stored in separate, function-specific panels on a shared blackboard. Several features of cognitive processing were explained by mapping the agent architecture down onto the resource-limited hardware architecture (goal switching, opportunism, planning limitations and slips, etc). In addition, the cognitive resource types were decomposed into four orthogonal types: problem, agenda, control, and progress. These resource types consist of collections of data with different data types. A problem consists of a collection of global goals, constraints, and possible operations; progress consists current problem solving state and history; control panels store plans; agendas store goals being considered. The data typology was used to explain several properties of human problem solving (i.e., it is goal-directed and planful). The result is an abstract model of human thinking and problem solving that summarizes knowledge from prior models of cognition. Although the discussion focused on prior “internal-only” models of cognition, the DC framework of Chapter 4 implies that in reality all of these resources may be distributed. This is commented on below in Section 5.6.

To illustrate the applicability of this model to software engineering tasks, the “integrated” model of software comprehension by von Mayrhauser *et al.* [675] (the vMV model) can be re-interpreted as a blackboard architecture in a straightforward manner. The vMV model postulates that comprehension consists of a process of incrementally constructing a hierarchical internal representation (knowledge base) of a system. The process is described as consisting of three different types of processes (top-down, bottom-up, and situation) which use different processing techniques, and which build complementary types of knowledge. Processing in the model is described as opportunistically switching between the three processing types. This model is quite naturally re-interpreted using the Agent model. The three processing types are interpreted to be three specialized agents. Thus the model would contain a top-down agent, bottom-up

agent, and situation agent. Each is opportunistically activated when it can make a contribution towards the solution. The internal representation is identified with the progress panel (i.e., the solution state), with the understanding that for long-term processes the incrementally built solution will need to be stored in LTM. Other minor aspects of the vMV model can also be incorporated into this translation. However this simple description should be enough to show that the agent model corresponds closely to existing cognitive modeling in the field. In Section 8.1.2, an example of a different software comprehension model is modeled using the Agent model.

5.4 Specialization Hierarchy: SRKM Strata

Rasmussen was not the first to note that human behaviour falls into several different categories [366, 526]. But he provided an incisive (and reasonably influential) analysis of what this sort of categorization implies to the problem of modeling and designing systems [526, 528, 529]. His proposal was to divide human behaviour according to how well it is specialized or adapted towards particular tasks. Specifically, he proposed that cognition falls into three categories of response: skill-based (S), rule-based (R), and knowledge-based (K). The categorization as a whole is called the SRK taxonomy. A summary is presented in Table 5.3. Briefly, skill-based cognition is quick and easy because of overlearning in controlling external devices. Rule-based cognition is fast because of deep experience in solving classes of problems. And knowledge-based behaviour is slow and difficult because it requires deep thinking and reasoning.

The crucial realization is that properties of cognition match levels of adaptation to the tasks involved. Because of these matches an ordering is imposed on the SRK taxonomy in terms of preferred activity. This ordering results in an ordering on how processing *falls back* to different levels one level of adaptation fails. This ordering is depicted in Figure 5.7. These basic categories of cognition will be very important for understanding cognitive support.

CATEGORY	ADAPTATION	COGNITIVE PHENOMENA	SUGGESTED MECHANISMS
skill	Highly adapted to repetitive activity not involving deep knowledge.	Quick, automatic, effortless, parallel, opaque.	Simple, functional, “non-symbolic” computational models.
rule	Expert response in familiar situations.	Rapid response and pattern matching without rational deliberation.	Rapid memory search for expert knowledge based on abstracted cues.
knowledge	Variations in circumstances mean canned responses fail: need to adapt.	Slow, serial, deliberate. Reasoning, problem-solving, categorization, etc.	Manipulation of internal representations and semantically meaningful symbols.

Table 5.3: Overview of Rasmussen’s SRK categories of human adaptation

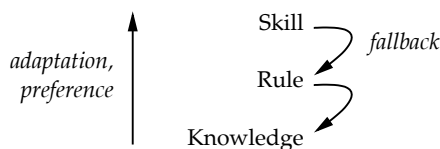


Figure 5.7: Ordering of behaviour categories based on adaptation, preference, and fallback order

In addition to these categories, Rasmussen also established: (1) a taxonomy of interpretation levels on data, (2) a set of hypotheses about mechanisms generating the three SRK behaviour types, and (3) an abstract model for schematically depicting how the behaviour-causing mechanisms interact. In this work, the first aspect is not considered.⁸ The second two are reinterpreted in this framework by mapping his scheme onto the hardware and agent models presented in prior sections. In particular, the mapping is accomplished by establishing a set of layers or *strata* on the agent model.

This stratification works as follows. Agents are located on a stratum. Each stratum defines a typing on the agents located within it. Each agent typing is associated with a specific way of mapping the agent onto the hardware model. It is the properties of the hardware model that explain the different phenomena of the SRK behaviour categories. The hardware model therefore replaces the mechanisms that Rasmussen suggested for each SRK category (i.e., the rightmost column in Table 5.3). In addition, the agent model explains how the different behaviours flexibly interact. Thus the agent model replaces Rasmussen’s model (he calls it a “framework”) of how the different behaviours interact.

The mapping is as follows:

1. **Skill-level.** The skill-level behaviour that Rasmussen describes corresponds to the LLP processing of the hardware model. All agents in the skill-level stratum are mapped to these components. All I/O operations are assumed to be mediated by skill-level agents.
2. **Rule-level.** The rule-level behaviour corresponds to simple memory-based response mechanisms. In the agent model, this is modeled as an agent or agents that match the current contents of the blackboard to knowledge from LTM and then post the resulting solutions or plans back to the blackboard. This type of agent is mapped down onto the MEMP processor.
3. **Knowledge-level.** The knowledge-level behaviour corresponds to the serial, deliberate processing in the hardware model. All agents at the knowledge-level stratum are mapped to threads in the COGP processor of the hardware model.

An illustration of such a stratification appears in Figure 5.8. In the figure, a fourth stratum labelled “M” is depicted. This will be explained below.

The above scheme very much resembles Rasmussen’s “framework”. The main difference is that direct communication between agents is not depicted, and communications through a resource-restricted data store is. Because of the close correlation, the strata terminology will be used when referring to Rasmussen’s framework also. For simplicity agents are forced to reside entirely within one stratum. If they

⁸Rasmussen’s insights are deep and potentially useful, and it may be very fruitful in the future to extend this work by including them.

identify behaviour spanning more than one stratum, they can be simply decomposed into clusters of independent, cooperating agents.

The mappings of the strata onto the Hardware model are also depicted in Figure 5.8. A few notes can be made regarding precedents to this scheme. In agent terminology, Rasmussen's architecture is much somewhat like a *subsumption* architecture [69] which is *vertically layered* (see e.g. Jennings *et al.* [334]). The main differences appear to be related to communication methods of the agents. Note also that many similar schemes have been proposed for multi-layer information flow through cognitive agents. Examples include Norman's multi-stage model [467], and many robot architectures (see e.g., Van de Velde [651] for several examples). Further, it is worthwhile mentioning that Rasmussen is careful to argue that information flows between the environment and the different strata always pass through the skill level. That is, action and perception are always accomplished through agents on the S stratum. This constraint is also adopted; it is depicted in Figure 5.8 by locating the I/O capabilities on the S stratum.

Before finishing, observe that the model includes a fourth stratum, labelled "M". This is intended to indicate a fourth category of cognitive behaviour called "metacognitive" activity. This category is included in HASTI primarily as a placeholder for future work. The intention of this category is to differentiate a special type of knowledge based processing. This knowledge based processing occurs when one thinks about and reasons about one's own problem solving. This corresponds in part to what was termed "reflective thinking" as part of "breakdowns" (see Section 3.2.1; in that work, reflective thought is considered a different "mode" of thought (e.g., Norman [472], Lloyd *et al.* [389])). This is a reasonable extension of the SRK taxonomy because it represents a case where the knowledge-based processing is ill-adapted to smoothly solve a task. In such cases one is forced to reason about how to adapt existing knowledge-based solutions. Because it is related to reflective thought, I thought it would be helpful to indicate future directions for being able to discuss cognitive support in relation to reflective thought.

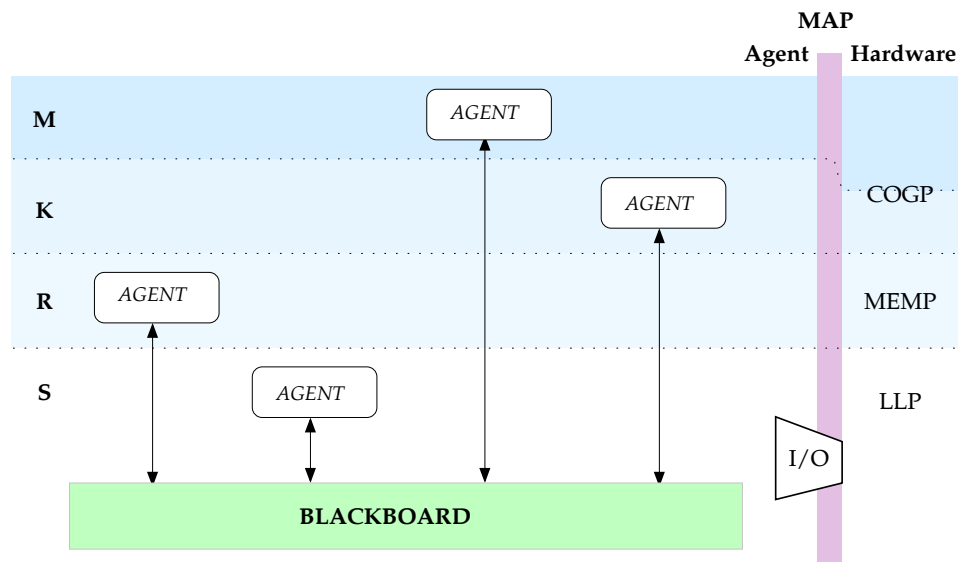


Figure 5.8: SRKM imposes stratification on Agent model

5.5 Task Decomposition: D2C2 Stratification

The notion of a problem space is a powerful one. It can be used to treat many different activities such as moving mice, solving algebra, and negotiating a resolution to a conflict with co-worker. The overall “problem” that a user faces is some complicated composition of all of these sorts of problems, and more. Although this may seem to be a beneficial simplification in logic, the reality is that when discussing cognitive support, a more useful vocabulary is needed. The problem space must be sensibly partitioned.

HASTI propose a four-fold partitioning of the problem space. The choices in this partitioning are related to what sorts of issues are important when analyzing cognitive support. These are called “domain”, “device”, “coping”, and “cooperation” problems or tasks. These are summarized in Table 5.4. They are as follows:

- **Domain.** A domain problem (e.g. modularizing a program) might be solved using any number of different tools or techniques. It is desirable to differentiate between *work* or *domain* task from the specific tasks related to the solution environment. For instance, a commonly noted distinction in HCI is the difference between a *domain* task and a *device* task (see below). There may be some question as to whether it is possible in principle to define domain tasks in a completely device-invariant manner. It is irrelevant for the purposes of this model whether generalizable domain tasks can be identified. Instead, it is sufficient that the analyst be able to confidently categorize part a problem as being part of the domain.
- **Device.** The particular tool being used to solve a problem is usually called the *device*. A device creates its own set of tasks. For instance when writing a letter with a word processor one particular device task might be to save the current file periodically. Such tasks may differ with different tools: no saving is required when using a typewriter. All domain tasks are ultimately mapped by the user onto sequences of device tasks.
- **Coping.** Domain problems present difficulties to the user, but some problems are encountered that may not appear to be part of the domain proper. Perhaps the most significant for this work is the unavoidable difficulty imposed by one’s own limitations, such as a small STM, or a bias for simplified reasoning. People often adopt strategies to cope with these problems (e.g., see Freed *et al.* [229]). Other difficulties might appear in the form of failures (e.g. power outages) or unanticipated interruptions (e.g. a telephone call from a spouse). Regardless of the source, coping problems are problems that are attributable to difficulties not associated with the other categories.
- **Coordination.** It takes effort to cooperate. Cooperating individuals must synchronize, communicate, and maintain joint awareness, understanding, and planning (e.g. Baecker *et al.* [22]). For simplicity we can call these all *coordination* problems. Cooperation between human and computer is no different. One might be tempted to call this category as part of the “device” category. However there are good reasons for considering a separate category for coordination overheads. To see why, notice that so long as cooperation is occurring, a similar collection of coordination problems can exist even if the devices (and thus device tasks) change. For example, if an external memory is being shared

between a user and a computer, coordinating over access to a shared item is a coordination overhead regardless of the particular implementation of the external memory. It would be unfair to the design of the device to lump these costs in with device costs. Thus cooperation tasks are considered independent from the device in the same way that work tasks are.

These four types of task or subproblem are collectively called the D2C2 taxonomy. The actual proposal relates well to previously proposed ways of decomposing task or problems (e.g., Moran [418], Whitefield *et al.* [701]).

There are several reasons for wishing to decompose the overall problem space in the above manner. First, every problem type except domain represents a different type of *overhead*. It is important to be able to distinguish between an overhead and a fundamental problem of the work domain. When discussing cognitive support, the benefits must be placed in relation to the overheads it creates. For example, a designer of an external memory system will need to evaluate the cognitive support provided for work tasks in relation to the device and cooperation overheads it creates. A second reason for providing such distinctions is that without them, it would be difficult to understand the underlying domain-related problem solving activities of the user. The importance of this is emphasized by the experiences of Gray *et al.* [255]. They found it impossible to coherently understand the problem solving performed by their subjects until they recognized the spurious activities caused by device tasks.

The D2C2 taxonomy can be used to identify different types of cognitive processing in the Agent model. The problem types are intended to be pair-wise orthogonal; furthermore the problem type is orthogonal to the SRKM categories (any of the four D2C2 problems could potentially be tackled by any of the SRKM categories of behaviour). These orthogonalities can be depicted by partitioning the multi-layer SRKM mapping from Figure 5.8. This is shown in Figure 5.9.

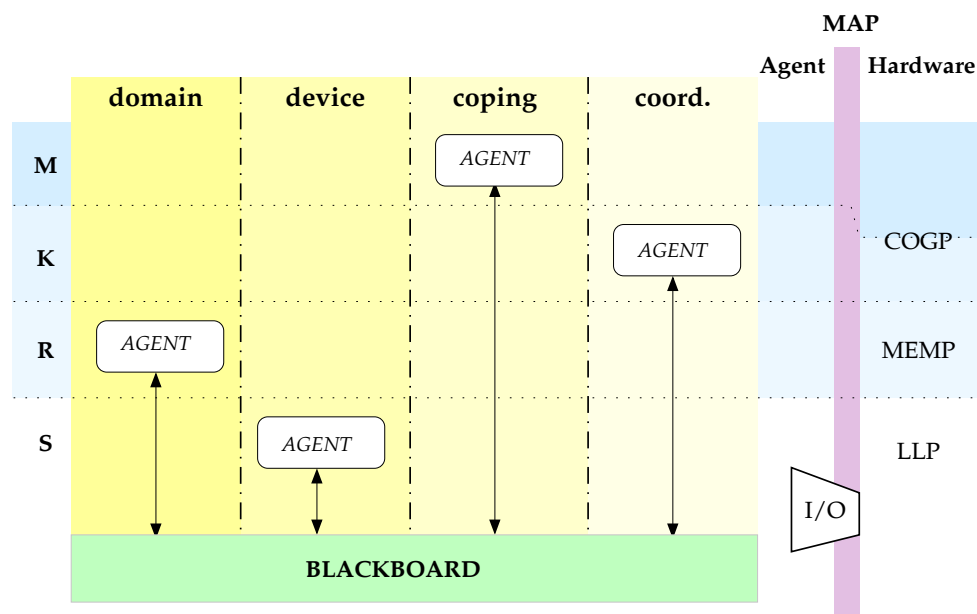


Figure 5.9: D2C2 task taxonomy partitions agents according to their task goals.

TYPE	DESCRIPTION
domain	related to the work context
device	specific tasks for working a tool
coping	work to avoid limitations/contingencies
coordination	overhead tasks when cooperating

Table 5.4: Summary of task taxonomy

5.6 Interaction Decomposition: Virtual Architecture

The importance of distinguishing different virtual architectural abstractions has already been discussed in Chapter 4. That chapter introduced the concept of a mapping between virtual architectures and their implementing or simulating architectures. There is no need to reiterate the basic arguments here. However it is important to discuss how virtual architectures fit in with the rest of HASTI. The main impact it makes is by making it possible to slide a virtual architecture over top of the Hardware architecture. Then, instead of having the Agent architecture map directly to the user's internal cognitive architecture, it can map onto a virtual human-computer architecture.

The Agent architecture models a single user's internal cognition using a multi-agent model. Because it is a multi-agent model to begin with, it is a small step to extend the model to multi-agent systems such as human-computer dyads. Similar concepts have been advanced before, usually in the context of computer-mediated communication between humans (e.g., Demoirers *et al.* [174], Perry *et al.* [503]). Interposing a virtual architecture between the Agent architecture and the Hardware model makes this extension possible. Then, agents can be seen to execute on the human (mental work) or computer (automations). The blackboard can also be seen as being distributed between the human and external memory. For this scheme to work, the interposed virtual architecture must map memory and processing to the implementing Hardware architecture. For a memory system, an architecture such as the Virtual Memory architecture from Figure 4.3 may be used (page 142). In these cases, the work done to simulate the virtual architecture identifies and encapsulates common device and communication overheads. Thus a virtual architecture can be used to elide such overheads from the task view in the Agent model.

Depending upon the analyst's focus, virtual architectures might be considered important or unimportant. For instance, in certain circumstances the designer may wish to ignore the particular virtual architecture. Then she might just assume that the Agent model is a distributed version and delay considering the issue of how it is implemented. In other cases, the analyst might be very concerned how the costs of the simulation overheads compare to the benefits of the device.

5.7 Summary and Conclusions

Several important criteria impinge on any proposed framework beyond the obvious one of utility. First, it must be accurate. This is not to say that it must offer a precise picture of the [cognition and HCI] being supported but what it offers should be correct in the sense that it describes real factors or aspects that influence the [tasks]. Second, it must be relatively non-complex. Invoking psychological descriptors or cognitive structures in a form suitable for non-specialists to use and apply is a difficult but necessary part of a good framework. Third, it must be suitably generic to be of relevance to more than one application.

– Andrew Dillon, “Designing Usable Electronic Text” [183, pg. 123], pg. 123.

This chapter addressed the problem of defining a modeling framework for analyzing cognitive support. In order to analyze different types of cognitive support, it is necessary to have DC models to apply the different support principles of RODS to. In particular, the DC models supply the psychological and HCI materials with which to discuss the various forms and possibilities of cognitive support. The primary challenges that analysts of cognitive support face are the complexity and disintegration of the knowledge base. Furthermore, the knowledge—as it exists in cognitive psychology and HCI currently—tends to be overly detailed relative to the needs of broad-brush analysis. Moreover, the fragmentary nature of existing modeling leads to the result where it is difficult to relate cognitive model features to performance and support issues.

These problems were met by (1) defining a collection of principles and strategies for constructing suitably integrated models, and (2) integrating a number of existing models of DC systems into a generic modeling framework. It is considered a modeling *framework* because it defines only a schematic model which is expected to be expanded and refined *as needed*. The principles of construction emphasized the needs for integration, broad coverage, and approximation and abstraction. An especially important principle for construction concerned how to decompose the modeling efforts. Specifically, it was argued that one of the most important tasks in the entire enterprise is to determine a set of dimension along which it is helpful to decompose DC phenomena. Then, when a decomposition is decided upon, it is important to decompose the models for this phenomena in an analogous way.

Using these principles, a modeling framework called HASTI was proposed. The key idea behind HASTI is that the decomposition dimensions should relate to the ways in which DC systems can be reengineered. This consideration effectively generated the primary feature of HASTI: a collection of five “dimensions” for decomposing aspects of DC systems. These dimensions were called: hardware, agent, specialization, task, and interaction dimensions; each identified cognition-related phenomena and issues, and each had a modeling structure associated with it (see Table 5.1). These dimensions name HASTI itself. Each of these dimensions adds concepts and vocabulary for analyzing cognitive support related issues. The models were designed so that they present an abstracted and simplified sampling of salient facts. Relationships between the various dimensions was established by the use of inter-model mappings, or by imposing orthogonal layerings on models. The emphasis of this presentation was primarily on modeling

users. Nonetheless, with the use of Virtual architectures, it is possible to map the same models onto joint cognitive systems. This capability will be further illustrated in Chapter 6.

It should be reiterated that no “new” facts are being proposed in this chapter. All of these phenomena and their models are discussed reasonably well in the science base. The contribution made here is in selecting the relevant knowledge and integrating it in a coherent set of structured models and inter-model mappings. The integration is very broad, simplified, principled, and unique. In concluding this chapter it is necessary to discuss how HASTI can help cognitive support analysis in ways that the underlying support base cannot.

First, the simplification may be valuable to both the psychology-averse SE researcher, and the specialist alike. For the non-specialist, it is a tidied, selected gathering that is ultimately grounded in rather well-established science base. The advantages to this will be rather obvious to most researchers, but especially to the non-specialists. Current analysis practices make little use of existing cognitive theories. Many existing modeling techniques present an initial hurdle that is too large for many non-specialists to jump over. HASTI is a step towards providing lightweight modeling frameworks which offer smaller hurdles and yet still present stepping stones towards deeper analysis. For the specialist, the integration vividly illustrates how broadly construed cognitive models need to be to make useful design implications. Rarely—if ever—are all of these aspects considered in cognitive science or HCI studies. For instance, the comprehension model by von Mayrhauser *et al.* [674] maps directly in to the Agent model. The suggestion is that it fails to address the other four dimensions of cognition. The integration therefore can help give the specialist perspective in appreciating the limitations of the studies and in understanding how they relate to the broader context [465].

Second, providing the five “dimensions” is important because of the way that many cognition-related issues can be smoothly brought into the analysis as needed. Indeed, one of the main advantages for the analysis is that an inclusive set of concepts and vocabulary is provided. The analyst can refer to memory limitations, discuss goals and plans, compare knowledge-based and skill-based behaviour, distinguish between device and domain tasks, and identify device tasks with simulation overheads. This benefit is discussed further in Section 7.2.1.

Third, the way that the five “dimensions” are woven together is important. In fact, modeling the integration is arguably as important as modeling the individual dimensions. Although it is possible to pay attention to any one dimension during analysis, relating cognitive issues between models is made difficult unless an explicit mapping is available. The mappings effectively create a *structure of implications and associations* that can be followed during analysis. For instance, a simulation operation in the virtual architecture can be linked to cognitive overheads in the form of device tasks, which can be classified as a skill-based behaviour, which leads to consideration of which perceptual skills are being utilized. It is conceivable that an experienced cognitive ergonomist might rapidly appreciate these relationships, but the mappings in HASTI are explicit.

Lastly, HASTI is structured in such a way that RODS can be applied to it to generate a space of cognitive structurings. This last advantage, however, is important enough to have a separate chapter devoted to exploring it: Chapter 6.

Chapter 6

CoSTH: A Hierarchy of Support Theories

... developers and maintainers of software are faced with the ongoing task of assessing the value of new comprehension aids. Currently, this assessment is entirely by trial and error, usually of the most informal and anecdotal kind. ... a useful task for engineering psychology is the development of a set of design guides for the selection of these devices. ... As a starting point for developing such design principles, an analysis of the way in which program comprehension aids operate is necessary.

– Ruven Brooks, “A Theoretical Analysis of the Role of Documentation in the Comprehension of Computer Programs” [75], 1982.

Theories of cognitive support are the intellectual foundation for understanding why tools built for cognitively demanding activities are useful. They postulate how tools are able to actually improve thinking and problem solving. Two ingredients are indispensable for using any such theory: (1) a description of some facets of cognition, and (2) a way of indicating the ways in which tools serve to improve this cognition. These are precisely the two things that HASTI and RODS are intended to help provide. RODS enumerates ways in which DC systems are improved in the form of support principles—specifically, in the form of explanations for why specific computational rearrangements of DC systems can be advantageous. RODS is therefore a high-level support theory that does not explain in particular what sorts of cognition rearrangements are helpful. HASTI proposes a way of modeling joint human–computer systems in DC terms. HASTI can therefore be combined with RODS to build more specific cognitive support theories which make more detailed arguments about what sort of cognition is being rearranged and how. This chapter aims to extract these theories, to illustrate how these relate to cognitive support in SE tools, and to explain how such theories may someday be used to codify knowledge about good tool design. RODS and HASTI set the theoretical backdrop, and this chapter begins the process of applying the theories to improve SE

research.

The structure of the chapter is as follows. First, in Section 6.1, an overview is provided of how HASTI is used to specialize the high-level support theories of RODS. The output of this process is a hierarchy of more specialized theories of cognitive support. This hierarchy is called “CoSTH”. The topmost level of CoSTH consists of three classes of abstract support theories. Each class is based upon the application of one of three types of support principles from RODS. Specifically, each of computational substitution principles from *distribution*, *specialization* and *algorithmic optimization* are used as a basis for generating different classes of support theories. HASTI is used as a basis for refining these topmost classes of theories. These refinements identify equivalence classes of artifacts which are related by implementing a particular type of cognitive support. Thus CoSTH formalizes generalizable *tool ideas*. Each of the three topmost support types are described in separate sections—Sections 6.2, 6.3, and 6.4. Then the way that these branches of the CoSTH hierarchy compose are explored in Section 6.5. Finally, the chapter closes out by comparing this work to prior integrative work on cognitive support (Section 6.6), by discussing some of its limitations (Section 6.7), and by providing a summary of the hierarchy and its implications (Section 6.7).

6.1 Using HASTI and RODS To Formalize Tool Ideas

Whilst there have been numerous empirical studies investigating different aspects of graphical representations there has been little attempt to integrate the findings into an analytic framework. What is needed, therefore, is a more systematic approach for evaluating the merits of different kinds of graphical representations, one that is theoretically-driven and which accounts for the cognitive processing when people interact with them. Without such an approach we have no principled way of either making sense of the vast empirical literature on the benefits of graphical representations or of making predictions about the value of new forms, such as animation and virtual reality.

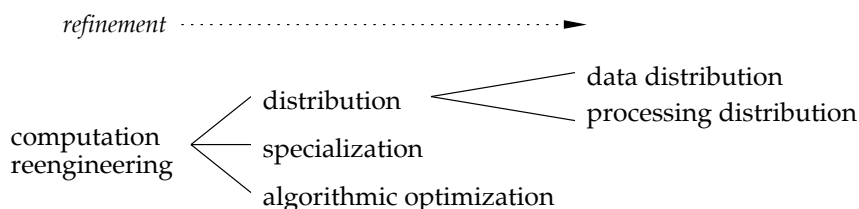
– Scaife and Rogers,

“External Cognition: How Do Graphical Representations Work?” [562], pg. 186.

Tools researchers are commonly interested in *tool ideas* rather than tools themselves. This fact was argued in Chapter 2. Informally speaking, we may say that a tool idea is a statement that a certain class of tools is beneficial in a certain way. The distinction between an *implementation* of a tool idea and the *idea itself* is important. A perfectly good tool idea can be embodied in a horribly bad implementation. A pipe-cleaner, for instance, makes a feeble lever. This fact leads to important implications for testing tools. In order for a test to generalize, a tool *idea* must be tested, else any minor variation in the implementation threatens to render the test results inapplicable to the variants. The existence of such minor variations obviously implies that the same basic tool idea might be implemented in several different forms. This is important to know when, for instance, refereeing papers and grant proposals, or for building re-usable design knowledge.

It is therefore important to be able to formalize tool ideas and be able to classify artifacts into categories of equivalent tool types. RODS works toward this goal. For example, tools that distribute data of some kind all can be said to fall into the broad class of tools that function as an external memory. This categorization involves more than simply clustering together related tools—a tool *idea* is normally associated with an understanding of the usefulness of a tool. For example, we might reasonably claim that the idea behind a fisheye-based source code browser is that the browser helps avoid being “lost in hyperspace” by presenting a detail-in-context view (e.g., see Storey *et al.* [621]). The *explanation* is important. RODS formalizes the *idea* by providing a computational explanation for why that class of artifacts are supportive.

The reader may recall from Section 4.2 that the four cognitive support principles of RODS identify theories of cognitive support. Since the underlying computational principles of RODS are thought to be orthogonal concepts for *substitution*, the four principles of RODS induce four independent support theories. The result is a *hierarchical refinement structure* to the theories as follows:



These theories identify high-level tool ideas such as external memory. They are in a sense independent of the particulars of cognition—independent also of the DC modeling techniques used to model cognition.

This model-independence has its advantages. Conceivably many different cognitive models could be used in conjunction with RODS. These models could be domain-specific, or they could uniquely model certain aspects of cognition like auditory perception. This model independence is just good separation of concerns—good theory engineering. It insulates the definition of RODS, to a certain degree, from changes to how we understand and model human cognition. It also insulates RODS from the vastly complicating details of various domains of application. These advantages are transferred onto the topmost levels of the hierarchy of support theories.

The disadvantage of this independence, however, is that the ideas they formalize are too general. Thus, although the basic RODS principles take a step towards formalizing ideas, it is, *practically* speaking, the tiniest of steps. We should like to be able to formalize more specific ideas about tools. Thus some way of making the tool ideas more specific are needed.

The way this can be done is to define more specifically what sorts of computational reengineering is being done by the artifact. For instance, after it is noted that data is being distributed, a question likely to be asked next is: “what data?” After noting that more-specialized processing can substitute for less-specialized processing, one is tempted to ask: “what substitutions are possible?” This is where HASTI comes in. The main purpose of HASTI is to be able to discuss what sorts of computational reengineerings are possible (see Section 5.1). A key contribution of HASTI in this regard is its five-fold decomposition of cognitive issues into various taxonomies and hierarchies. Since HASTI is derived as a simplification and integration of cognitive modeling literature, it provides a theoretically-motivated decomposition of cognitive aspects that may be reengineered by applying RODS restructurings.

RODS: SUBSTITUTION PRINCIPLE	HASTI: MATCHING STRUCTURE	COSTH: REFINEMENTS
distribution	Agent data typology	distribution subtypes
	Virtual architectures	distribution subtypes
specialization	SRKM	substitution hierarchy

Table 6.1: HASTI structures that align with RODS substitution types

Since RODS is model-independent, other cognitive models can also be used. However, depending upon the cognitive model involved, generating these refined support theories can be easier said than done. For instance, the notion of a specialization substitution would be difficult to apply on models in which no clear distinction is made between general and specialized processing elements (e.g., the comprehension model of von Mayrhauser *et al.* [675]). HASTI has been designed to provide a good starting point. The principle design rationale for HASTI was that the structuring mechanisms should cleave computational models along lines where the computational reengineering can be applied (see Section 5.1). Consequently, the substitution principles of RODS are “matched” to structuring mechanisms of HASTI (see Table 5.1). A summary of these matches is presented in Table 6.1 and described below.

DISTRIBUTION

Distribution substitutions can be applied to the resource and data type decomposition of the Agent model, and the virtual hardware as follows:

1. **Data Type.** The panel types from the Agent model classify the different types of mental states. Any of these are candidates for distribution. For instance, plans could be distributed. The existence of a distributed data type also strongly indicates a distribution of processing of that data. For instance a shared plan can indicate distributed planning.
2. **Virtual Architecture.** Virtual architectures encapsulate interface interactions, thereby abstracting the interface operations. It therefore provides one class of more specific—but still general—details about what data and agents may be distributed. For example, the task of searching an external memory for a match is one that is clearly generalizable across many different external memory devices, and which has external processing possibilities.

SPECIALIZATION

HASTI’s SRKM is a specialization hierarchy that distinguishes within humans four degrees of adaptation to a task and environment. The level of specialization indicates distinct categories of processing which is interpreted by HASTI as involving different (levels of) cognitive mechanisms. The SRKM hierarchy therefore provides a *specialization substitution hierarchy* in which lower level processing can potentially be substituted for higher level processing.

HASTI therefore contributes several ways of refining the CoSTH hierarchy with specific details about different sub-categories of cognitive reengineering. The result is a hierarchically structured family of

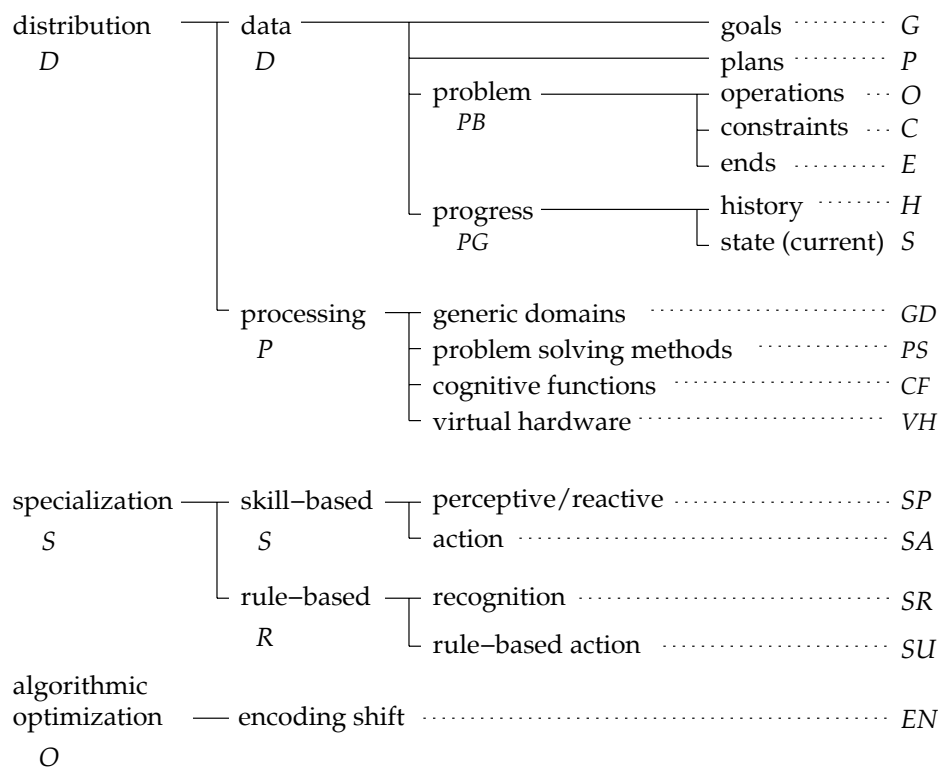


Figure 6.1: Hierarchy of support theories built from RODS+HASTI

cognitive support theories, some more refined than the others. Figure 6.1 illustrates the hierarchy that results from these refinements. This is called the “cognitive support theory hierarchy”, or CoSTH.¹ The leaves of the tree represent support theories specialized using cognitive system information from HASTI in combination with a few additional ways of categorizing support types. It is worthwhile reiterating the fact that these are theories of how certain computational restructurings of DC systems can explain cognitive support—they are not theories for *generating artifacts* that are supportive.

Although HASTI can be used to refine CoSTH, even these more particular theories may be quite “generic”. This is simply because HASTI is an intentionally “broad brush” modeling framework. “Generic” cognitive support theories, in this sense, are essentially broad-brush statements about how large classes of DC systems can be rearranged to support cognitive work. Even though their generality limits their power to make certain explanations or predictions, the generic theories are nonetheless important to have, and is prominently utilized in Chapter 7.

¹The choice to present the body of theoretical work as a hierarchically refined collection of separate *theories* was somewhat arbitrary. I could have chosen to present the whole as a sort of single “unified theory” composed of the leaf nodes, perhaps, or I could have tried to highlight orthogonalities by presenting the theories as inhabiting a topological *space*. I think, however, that the hierarchical refinement presentation reinforces the theory-design perspective (Section 2.4) well by highlighting the multiple decision points that are possible when narrowing the strokes of the broad-brush theories.

Documentation Conventions

The bulk of the remainder of this chapter consists of descriptions of the generic cognitive support theories from CoSTH in Figure 6.1. In the following, UNIX-style file naming conventions² is used to name the nodes in this tree, with hyphens ('-') replacing spaces between words. Also, to save space a set of abbreviation letters are used to refer to various nodes in the tree. These letters are indicated in Figure 6.1 by capital italicized letters (e.g., "*D*") either under the node (non-leaf nodes) or to the right of the nodes (leaf nodes) for which they are abbreviations. So, for example, the strings `distribution/data/goals` and `D/D/goals` both refer to the top-most leaf in Figure 6.1, and both `O/encoding` and `O/EN` name the bottom-most one. Note that many abbreviations are reused (e.g., S, R) in different parts of the tree, but that the leaves are all distinct.³ The duplication does not present any problem if used to name a nodes using the UNIX style. Likewise the leaves can be named without reference to path (when it is unclear whether a leaf node or one of the root nodes are being referred to, a note is made).

In order to structure the exposition of the support theories, a schema is imposed on describing each support type, as follows:

★ RECAP/INTRO

The relevant features of HASTI and other relevant background materials are briefly recapitulated and summarized for convenient reference.

★ INTERPRETATION GUIDELINES

Understanding how to apply the support type to real HCI analysis situations can be challenging. The contribution of artifacts to system cognition can be extremely non-obvious, or otherwise difficult to properly appreciate [311] (see also Section 4.1.6). This guidelines part provides a characterization of what can constitute an instance of the support type being described. It may help the reader to think of these characterizations as things that an experimenter would look for when performing observations of user-tool interactions. It is impossible to be completely thorough in these characterizations, so the emphasis is placed on providing one or more paradigmatic examples.

★ SAMPLE IMPLEMENTATIONS

Examples from the literature are provided here to punctuate the theoretical description with some concrete instantiations. The examples also help demonstrate that the categories they inhabit are nonempty. In some cases the examples are archetypical instances of an unidentifiable sub-category of similar support types. Thus, in addition to the example, sometimes a more general description of a refinement to the support type is also included. It is worthwhile reiterating here the role of the examples in this exposition: they are illustrative *samples* of a support category; they are definitely not exhaustive. Figure 6.2 illustrates how these examples relate to the types of cognitive support and of cognitive support theories. The examples are all categorized by the type of cognitive support

²This naming scheme uses a sequence of node names separated by the '/' character to specify a path from the root node. A mathematician might have chosen a sub-scripted or parameterized notation such as $\mathcal{D}_d[\text{goals}]$, but I think the tree-walking UNIX scheme helpfully highlights the refinement path.

³Notice also that two of these, `D/D/PG/past` and `D/D/PG/current` are a little unusual but is explained later.

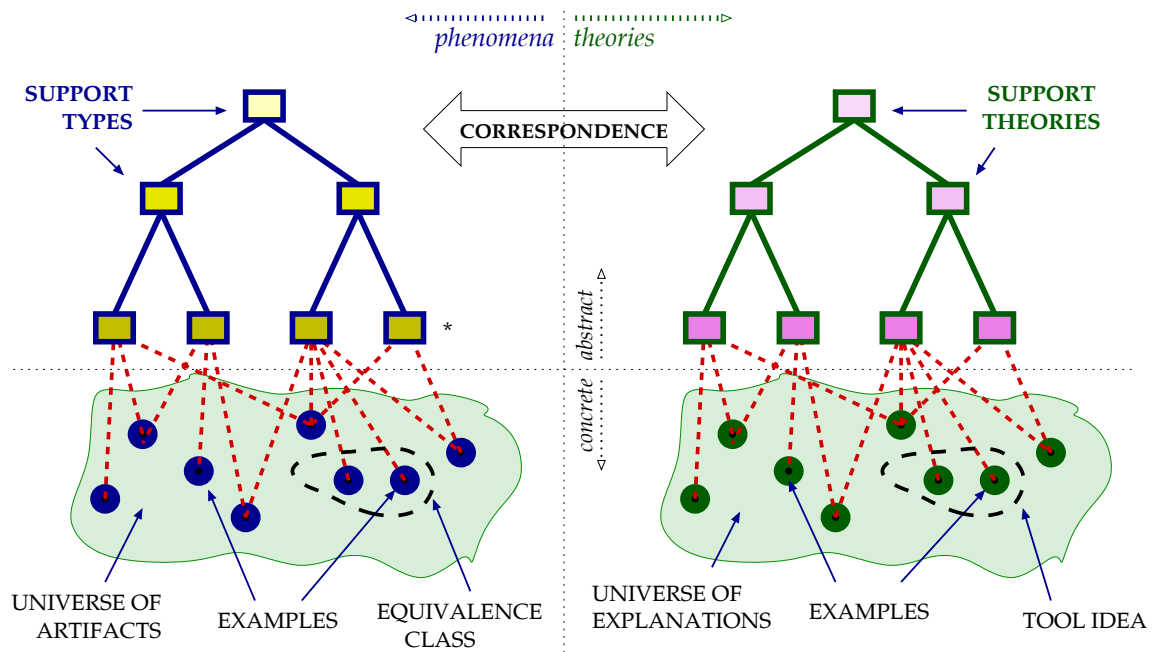


Figure 6.2: Relationships between artifacts, examples, theories, and tool ideas.

they exhibit, which in turn corresponds to the types of theories that explain them. The illustration schematically shows that the examples may be members of more than one category (support types combine). CoSTH inhabits the upper right quadrant (Chapter 3 attempts to elaborate the upper left).

With these organizational issues out of the way, the next four sections can be dedicated to describing the support types and how they combine.

6.2 Distribution

One way of thinking about cognitive support is that there is a continuum of different levels of support that ranges from the completely unsupported (entirely mental) to the completely automated (no human thinking involved). In practice, both ends of the spectrum are generally unattainable for interesting tasks like programming. Still it is reasonable to imagine in principle what the unachievable extremes of the spectrum would entail. In order for the *entirely* mental end of the continuum to hold up under close inspection, all of the problem, solution, and mental state information would need be held internally; all of the processing of such information would also need to be done internally.

Distribution changes this picture. With increasing distribution the problem solving becomes more and more dispersed. With increasing distribution less computational work is done by the user. A common term for this is “offloading” (e.g., Scaife *et al.* [562]) however the terms “automation” (e.g., Boy [65], Parasurman [484]), “allocation” (e.g., Wright *et al.* [719]), and “delegation” (e.g., Bhavnani *et al.* [48]) are also used, and are effectively synonymous. Because “distribution” is both more general (computation can be distributed between artifacts or other agents) and computer science-friendly, it is the preferred term in the

CLASS	TYPE	DESCRIPTION	(SUBTYPE)	EXAMPLES
data	goals	directs/maintains focus	explicit	names in task list, GoalPoster, MediaDoc
			implicit	skeleton declarations during coding
	plans	orders or structures action	explicit	checklist, todo list, reminders, schedule
			implicit	code repair list, query results
			plan-like	scripts, macros, wizard plans remod. plan in StarDiagram, thesis outline
	problem	defines cognition context	operations	menus, buttons, options
			constraints	linguistic/physical/graphical constraints
			combined	requirements documents
	progress	problem solving states	past, trace	visitation history, breadcrumbs, dog-ears
			past, path	undo, revision history
			current, focus	cursor position, object selection, window focus
			current, partial	berrypicking
	process	generic domains	intermediate computations	math/stats
patterns/search				grep, SCRUPLE
abstraction				clustering, concept analysis, graph layout
virtual hardware		abstracted interface tasks	VM	data layout, item sorting, search page management
PSMs		domain independent strategies	iteration	wildcards, tag-&-drag, query-replace
cognitive functions		thinking activity	attn. mgmt.	critiquing
			planning	error list generation
			constr. enf.	type checking
			learning	programming-by-demonstration script library population by peers

Abbreviations: VM = virtual memory attn. mgmt. = attention management
 PSMs = problem solving methods constr. enf. = constraint enforcing

Table 6.2: Summary of redistribution examples

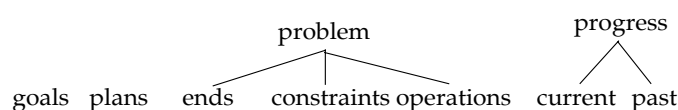
following. Nevertheless “offloading”, “allocation”, and “delegation” are all helpful and evocative terms, so these terms also occasionally find uses.

The remainder of this section is an elaboration of different ways of distributing cognition. For each different type of distribution the principles for categorizing each type are explained and several examples are given. A summary table of these is presented in Table 6.2. The table is organized hierarchically according to distribution class (data or process), type, and what is referred to as a “subtype”. The subtypes used in the table are primarily for organizational and explanation purposes, so in the following they may not be explicitly mentioned.

6.2.1 Data Distribution

There are two characteristic uses of data distribution. The first is the unloading of data from the user's memory for later consumption by the same user. This is typically called "memory offloading". It implies that something functions as an external memory. The second use of distribution is the sharing of that data between user and another agent (e.g., a computer). One might call this "memory sharing". This type of distribution also requires an external memory, but the shared nature of this memory is perhaps more obvious. In the former case one normally can trace a data loop from the user to the external memory and back again. In the latter case the data may be produced and consumed by either participant, so the data loop may be much more convoluted or indirect.

The different types of data distribution are supplied by the Agent model's panel content types (i.e., the cognitive resource types and their data types). These are: goals, plans, problem, and progress. Problem data is further decomposed into ends, constraints, and possible operations. Likewise, progress can be further divided into past and current problem solving states. This data type hierarchy can be summarized as follows:



Goals

★ RECAP/INTRO

An agenda is a resource used to maintain and manage multiple goals. Goals are things to try to achieve.

★ INTERPRETATION GUIDELINES

Any external structure or feature that is employed by users to direct and manage their focus can be considered an external implementation of an agenda. Often the agenda can be recognized by noting cases where goals are explicitly written down, often as presentations of desired problem solving states.

★ SAMPLE IMPLEMENTATIONS

Skeletal declarations. A classic example of goal offloading from programming is given by Green *et al.* [267] in their "Parsing-Gnirap" model of code generation. Programmers planning a complicated addition may externalize pending subgoals in the form of skeletal program declarations to be filled in later. These skeleton declarations serve an external agenda. As Brooks argued [71], such an external agenda can substitute for an internally stored subgoal stack.

Shared goals. Agenda sharing is often an important mechanism for coordinating agents [173]. The Molehill learning environment [598–600] contains a good example of an external shared agenda in the context of human–tool coordination. The computer agent is a tutor called the GoalPoster, and it periodically "posts" the goals which it infers the student is currently

working on. In this case the externally manifested goals are names of meaningful and desired problem states. An analogous function is provided by the “task list” component of the MediaDoc system documentation environment (see Erdem *et al.* [204] for a description). MediaDoc’s task list maintains an agenda of sub-tasks to perform for a task that a user selects. The user’s subtask selection can therefore be partly driven by this external agenda. A similar role for an external agenda is also used in the software design environment Argo (see Robbins *et al.* [544]).

Plans

★ RECAP/INTRO

Planning is about (partially) specifying future intentions or action orders, especially in response to constraints on the ordering of activities. Planning and pre-structuring acts are therefore precomputations of action sequences, or of action ordering constraints.

★ INTERPRETATION GUIDELINES

Since planning is about ordering future action, external plans can be nearly any kind of structure that can indicate or specify constraints, future intentions, or ordering of actions within the system. They do not have to be followed slavishly to be considered a plan in this context. For instance a checklist is a plan; so is a shopping list. Plans can organize user action, computer action, or coordinate the two.

★ SAMPLE IMPLEMENTATIONS

A reminder. A sticky note pasted on the edge of a computer screen; the event database of a popup reminder agent.

An appointment schedule. As in a day planner or online meetings scheduler.

Inspection checklists, test plans. Code inspection checklists (e.g., Fagan [208]) are good examples of precomputed and static plans [319,320]. Checklists are used for performing orderly and systematic software inspections [432]. Another common SE plan is a *test plan* [432].

Error lists. Most compilers will produce an error message list during compilation runs. The message list forms a partial list of fixes which the programmer (normally) has to make.

Search results. One strategic use of query engines is to pose queries that define a structure to traverse in a task. For instance a programmer might call `grep` to query a text base for a list of relevant program locations to modify [596]. The search results can be used to structure the ensuing action and thus can form an external plan of action.

“Todo” lists. Lists of things to do are commonly used ways of accumulating future intentions. Often the lists are used to organize action by imposing ordering schemes. Many development environments (e.g., Argo [544], SourceForge⁴) include todo list tracking features. Such externalized lists are also frequently used for coordinating multiple agents. They often are utilized

⁴<http://sourceforge.net/>, an open-source online development resource.

in conjunction with external agendas (see above). *Argo*, *MediaDoc*, and *Molehill* are all examples of multi-agent systems that use task decompositions as external plans. Many people organize their email mailboxes as a stack of things to do.

Scripts, macros, procedures. Scripts and macros are specifications of procedures (sequences of operations) for the computer to perform. Scripting languages are common across nearly all complex systems like operating systems (Windows scripting host, `tcl`, `sh`) and applications (word processors, reverse engineering systems [324,426]).

Instructions, problem solving methods. Instructions for performing tasks are frequently written in terms of procedures [657]. Sometimes these are instructions for novices which are later “internalized” as they are learned. A classic example is Polya’s popular book [513] of mathematics problem solving. Other external instruction structures include teacher lesson plans, and their computer analogues like structured documentation. These might be generated, for instance, to guide “software immigrants” when starting maintenance on a new system [132].

Tours and lesson plans. Tours are like scripts for humans. They are structures that can be used to guide someone unfamiliar with an application or information space through its contents and organization (e.g., Marshall *et al.* [397]).

Wizard plans. A wizard’s internally stored goal-satisfying procedure [719] is an example of an external plan. Wizard plans are like computer scripts but they are typically incomplete (i.e., they are schematic, see Ormerod [481]) since they do not implement formalized decision procedures at the script decision points (users are queried for these decisions).

Problem

★ RECAP/INTRO

A problem is the context that a person (or group or organization) acts within and gains motivation, constraints, and action possibilities from. A user’s problem changes from moment to moment as their understanding of this context changes. Formalizing a problem context is often difficult. One way to formalize a problem is in terms of a *problem space*, which consists of (1) goals to achieve, (2) constraints on action, and (3) operations that could be performed.

★ INTERPRETATION GUIDELINES

Recognizing what constitutes external manifestations of a problem can be extraordinarily difficult (see Section 5.3.1 for why this is so). Although identifying exactly what constitutes the problem is tough, for our purposes here the externally manifested problem information can still be classified into ends, constraints, and possibilities for action. Of these three, perhaps constraints are the easiest to recognize even though they can take on many different guises. Externally manifested constraints could include physical constraints on action, logical constraints within a language, or

written specifications of logical constraints (see Zhang and Norman [727] for a more extensive comparison⁵). Externally manifested ends are identified just as goals are in an external agenda, however they are associated with setting a context for problem solving rather than determining the moment-to-moment focus for problem solving. Possibilities for action can be any indication of ability to perform an action on an interface, or to make logical moves. Logical moves can be internal operations like inferences, or partly external moves that may be mapped down into a sequence of device-level moves.

★ SAMPLE IMPLEMENTATIONS

Drawing constraints. An architect's T-square can be used to implement a drawing constraint for producing horizontal and parallel lines [48].

Language constraints. A compiler will need to represent, either explicitly or implicitly, the constraints of the language it compiles.

Action restrictions in editors. Some syntax- or semantics- aware program editors enforce linguistic constraints by constraining editing action [630].

Type declarations, assertions. It is widely believed that being able to specify solution constraints is an important part of software design and development. Constraints are thought to help one avoid "shooting one's own foot off". Strong typing is a fine example of externally specifiable constraints [96], but many other examples can be cited, including assertions [549], contracts [405], "aspects" [327], and various types of system structure specification methods (e.g., Moriconi *et al.* [421]).

Menus, buttons, options. Menus, command buttons, and options in dialogue boxes are all visual representations of possible action. Some systems use context-sensitive menus to represent actions possible in the context.

Means-Ends Hierarchy. Externally manifested possibilities for action are called a "means-ends representation" by Rasmussen *et al.* [530]. Rasmussen influentially proposed his "means-ends hierarchy" as a principle for designing interfaces [528]. Many examples are given in derived and related works (e.g., Rasmussen *et al.* [531], Vicente [657]).

Requirements documents. Requirements documents are frequently used to represent the problem that developers face. These are commonly acknowledged to be necessarily incomplete and require iterative negotiation. That is, the programming problem evolves as it is explored and understood further. Thus requirements documentation is an excellent example of an evolving problem conception, but one that is externally manifested.

⁵Since Zhang and Norman give such a nice decomposition of different ways of manifesting constraints, I have considered adding another level of refinement to CoSTH diagrammed in Figure 6.1 in which the D/D/PB/constraints node is further subdivided into several types of constraint manifestations. However this further refinement begins to mix implementation issues with support type issues, and I have decided against mixing them as far as it is possible.

Progress

★ RECAP/INTRO

Progress data includes the current and past computational states of a system and how they evolved. The division of progress into past and current states generate good taxa because they describe a proper dichotomy. A reasonable synonym for past states is “history”, so “H” is used as the abbreviation for past state information. Since “current state” is a mouthful, we can simply drop the “current” and call it simply “state”, abbreviating it “S”.

★ INTERPRETATION GUIDELINES

External manifestations of progress tracking can be generally associated with any way of storing current or past system states. Current state includes the rather obvious objects such as the data being edited (current document, database, program, clipboard, etc.). It also includes, however, states such as problem focus, like the currently active window in a windowing system. Past progress can include snapshots of prior states, an ordered record of the problem solving *path* taken through different system states, or an unordered *trace* of different states that the system has been through.⁶

★ SAMPLE IMPLEMENTATIONS

Evolving solution objects. Many computing environments revolve around (a presentation of) an evolving solution, such as a document in word processing or a system’s source code in software development.

Logical focus. The logical object of the user’s current focus for problem solving may be externally represented. Often this is mapped from the logical position within the abstract problem space and onto the system’s presentation space. Common examples include mapping: logical focus onto a cursor position, object selection onto highlighted presentations, and window focus onto window selections. One special instance of externally representing logical focus is tracing positions with one’s hand or fingers [272, 352]. Note that if a logical object has a distributed external presentation, then the logical focus may map to multiple presentation locations (e.g., Tallis [228, 631], Brown [83], Shneiderman *et al.* [585]).

Visitation history. Many browsers keep a history of visited links (they use this to present link colours differently depending upon its visitation). Other similar examples include edit history and recently-used document lists.

Interaction history. Most command interfaces provide a history of commands or responses. An example of such a history is the scroll of paper that used to cascade down behind the old teletype machines. The modern equivalent of this is the terminal emulator buffer and its scrollbar. History also includes shell command histories [275].

Breadcrumbs, dog-eating, tick marks. A special form of visitation history; people often mark their trail or important places they’ve been to indicate relevance [454]. They also regularly record

⁶The terms “snapshot”, “trace”, and “path” are not further defined here as they are meant to be merely evocative terms as part of the ongoing work of vocabulary building.

progress within a checklist using tick marks. Some environments incidentally record usage informally as wear and tear [214,698].

Change history, revision management. During software development, multiple branches of documents are frequently kept in a revision tree using a revision management system. These effectively store partial solutions from different branches of the search space. Undo mechanisms in applications are similar in character to such revision histories (although undo mechanisms may differ in their state and revision encoding methods). Some systems like SeeSoft [23,198] are able to indicate certain aspects of an object's change history.

Berrypicking. "Berrypicking" [37] refers to the practice of gathering together (references to) interesting things so that the collected works can be returned to later or referred to *en masse*. For instance incrementally adding items "to basket" while browsing online for purchases in an e-commerce site is a simple form of berrypicking. Common non-computerized forms of such behaviour include jotting down names or short notes pertaining to pending tasks on a scrap of paper (see for instance the use of scrap paper reported by Bowdidge *et al.* [64]).

6.2.2 Processing Distribution

Processing distribution implies shared and cooperative processing of data. Distributed processing means that multiple agents split the work; *re*-distributing processing can involve any shifting between these multiple agents, but typically it refers to making computers take on cognitive more load.⁷ Software development is littered with examples where processing is distributed. Probably none is more obvious than program compilation: compilers now automate what used to be a laborious and error prone activity of generating machine-specific code from a more abstract specification of it. Many more examples could be produced, but apart from just listing them, it is unclear at the start how these could be coherently organized. Moreover, there does not seem to be any guarantee that any organization we could produce would carry over for other domains. Thus we arrive quickly at an important question for this section: is there a principled, domain-independent way of saying what sorts of processing can be distributed? For data distribution, the Agent model panel types yield a succinct, well-motivated decomposition of data types. These produce a corresponding taxonomy of data distribution types. The case for processing distribution is not so clear. There are no "obvious" ways of decomposing processing in a generic way.

HASTI includes essentially two structuring mechanisms that make some kind of statement regarding process decomposition. The first is the D2C2 task categorization taxonomy (Section 5.5). It provides categories for labeling certain tasks as corresponding to domain, device, coping strategies, or cooperating overhead. These categories can provide helpful ways of classifying activities once they are known, but they do not make any statements about what these activities are. The second statement HASTI makes regarding process decomposition is the virtual hardware abstractions in the form of virtual architectures. Virtual architectures are proposed as a way of abstracting human-artifact interactions which are common

⁷Distributed processing also necessarily implies that the data to process is available externally, but data distribution is not the focus of this particular subsection.

across many different interfaces (see Section 4.3.3). They certainly are resources for characterizing widely-applicable types of processes that may be distributed, and is utilized in the following. However the virtual architectures still say little about the processes that are implemented “on top of” this virtual hardware. What processes “run” on the virtual hardware? Can those processes be distributed? In sum, HASTI provides some decomposition methods for building a processing taxonomy, but they are insufficient. Other ways of building generalizable taxonomies of processing must therefore be considered here.

There are several possible classification methods for processing that could be considered. These include:

1. **Generic Domains.** A frequently used method for formalizing computation is to map operations from a problem domain into operations within domain-independent computational layers [76] (e.g., algebras, calculi, logics, etc.). This suggests that a fruitful approach for creating a domain-independent taxonomy of computational methods is to enumerate operations defining these sorts of intermediate domains (set functions, matrix calculations, graph traversals, etc.). Distributing computation would then amount to knowing how to map cognitive work onto operations in such intermediate domains. There can be no doubting the value of this approach because it is essentially the way we have been successfully automating previously cognitive work for nearly a century [371,695]. As Landauer [371] noted, however, the limitation is that the problem remains as to how to map domain processes onto these intermediate domains. In some cases this is relatively easy (e.g., as it was for the problem of vote tabulation). But for cognitive work such as software development, design, and maintenance, it is exceedingly difficult. Nevertheless, even if the analyst cannot create the mapping, users may. Thus it is possible to provide users with programmable domain-independent processing systems. The users are then responsible for mapping their particular cognitive problem into a form that can be externally processed. Spreadsheets are an obvious example. Thus, although domain-independent computational layers should not be ignored, they have a rather limited capacity to serve as a generalizable basis for decomposing cognitive processing for the purposes of discussing distribution possibilities.
2. **Domain Task and Process Models.** Task models are ways of decomposing activities and relating them to goals or purpose. An example of a *domain-specific* task is translating a program to machine code. Program translation is clearly a task which is specific to a particular problem domain (programming). For many domains it may be feasible to discern a taxonomy of tasks to perform, or it may be possible to systematically break down large-grained tasks into smaller subtasks. In either case, a collection or taxonomy of tasks is necessarily constructed. Sometimes such a task taxonomy emerges when developing either normative or descriptive models of problem solving within the domain. For our purposes here it does not matter how a task decomposition is created⁸: sometimes

⁸For this work I downplay any distinctions between “cognitive” and (presumably) “non-cognitive” task analyses. From the DC point of view there is little fundamental difference since one can generally substitute for the other (Section 4.2.3). There might be exceptions, of course, so more-or-less equating task analysis to cognitive task analysis may not always work. However, this stance is not so unusual since the DC view tends to blur the distinction between thinking and acting. This view seems especially sensible in highly cognitive domains like software development because it rarely seems worth giving up the simplicity of treating thinking and acting as being interchangeable.

the tasks are elicited during a task analysis [77,657]; sometimes they are a result of process modeling or description (e.g., Tzerpos *et al.* [647], Basili *et al.* [34]).

In SE there are many examples of process and task models. One of the most obvious characteristics of such models is their wide variability in granularity, form, and content. A classic process model is the so-called “waterfall” software development process [613], which consists of a sequence of high-level tasks to perform. In contrast, there are many fine-grained models of program comprehension, both *descriptive*, like Sengler’s pseudocode model of program comprehension [572], and *normative*, like Basili and Mill’s process model [34] for bottom up comprehension of code. Other task models of programming [499], maintenance [664], modularization [64], and reverse engineering [641] testify to the diversity of process and task modeling in the field. Similar types of task decompositions exist in other domains like reading [477,553], writing [92,574,575,604,605], model formulation [660], flight planning [66], decision making [196], and even coffee making [374]. More specific task analyses are performed routinely in standard HCI practice, and increasing emphasis is being placed on the cognitive aspects of tasks [128]. Regardless of the form and content, a task or process model describes processing of some sort and therefore can provide a vocabulary for discussing process distribution. For instance, Boehm-Davis’s [61] model of program comprehension consists of, in part, a cycle of “hypothesis generation”, “verification” and “segmentation” operations. Although these are putatively purely mental activities, one can nevertheless consider how each of these sub-processes may be distributed. For instance, one can contemplate distributed hypothesis generation.

The problem is that it is not at all clear how to make use of such obviously pervasive modeling to produce *general* descriptions of process distribution. The first problem is that many task analyses are better considered “competence” models rather than “performance” models. That is, they suggest what is needed to do a task rather than the ways it is actually performed. The second problem is that domain task decompositions are, by definition, domain-specific—notwithstanding the fact that they may be coarse-grained or written in abstract task languages. Domain specificity is unfortunately the main liability of these models: it is normally difficult to break them free of their domain. When constructing domain task models it is too easy to ignore crucial similarities and patterns of activities across domains. But when seeking a general theory of processing distribution it makes sense to try to abstract away from domain or environment specifics. Brooks [77] chalks up the difficulty as being primarily due to the way that tasks are usually modeled and described, especially in human-factors task models:

... the very low level of the [primitive tasks] and the lack of any formalism for summarization makes it difficult to use human-factors task analysis as the basis for task description in HCI. Suppose that such a task analysis were carried out for controlling a nuclear power plant and for piloting an aircraft. Each analysis might have thousands of instances of the task-analysis primitives. How could the similarities and differences between the tasks be determined? [77, pg. 56]

Regardless of the exact reasons for why domain task models are hard to generalize, they are difficult to use systematically for refining CoSTH. Despite Brooks’ pessimism, there is some hope that useful domain-independent taxonomies can be developed. For instance it is possible to point to the work

by Chandrasekaran *et al.* [119] on the development of a taxonomy of *generic tasks* (classification, data retrieval, state abstraction, etc.) in the domain of AI and knowledge engineering, and the body of taxonomic work spearheaded by Fleishman [222] in the domain of psychology. There is also some hope in generalizing relatively standard task analyses. An example is the “generalized task models” (GTM) of the TKS task modeling method [336]. However, these latter task models are primarily competence models, not performance models. Despite these bright spots, the current state of affairs makes task analyses difficult to use for refining CoSTH. However a similar sort of task analysis is featured more prominently in Chapter 8.

3. **Problem Solving Methods.** Brooks [77] suggested that HCI follow the lead of AI and knowledge engineering (KE) in pursuing abstract descriptions of methods of problem solving (PS). These are called *Problem Solving Methods* (PSMs) in KE, but similar concepts are known variously as *tactics*, *heuristic strategies*, and *task strategies*. In other areas of computing, similar concepts are known as “algorithmic skeletons” [143]. The PSM work is a descendant of work on generic PSMs: the so-called “weak” methods such as “means-ends analysis” [443]. The weak methods are entirely problem independent, and although in principle they should always work, they do not take advantage of particular details of the task or environment in order to be efficient. Generally speaking, the more knowledge one has about the task and task environment, the better one can tailor one’s solution strategy to take advantage of the available invariants, constraints, and resources. PSMs expand on the weak methods by introducing abstraction mechanisms that allow one to add details about the problems in a generic way. They capture *strategic knowledge* of how to solve particular problem types efficiently using the PS environment. Fensel *et al.* [209] succinctly sum up the significant potential advantages of PSMs for KE, but they seem to apply just as well for HCI:

... [PSMs] describe domain-independent reasoning components, which specify patterns of behavior which can be reused across applications. For instance, Propose&Revise ... provides a generic reasoning pattern, characterized by iterative sequences of model ‘extension’ and ‘revision’ ... the study of PSMs can be seen as a way to move beyond the notion of knowledge engineering as an ‘art’ ..., to formulate a task-oriented systematization of the field, which will make it possible to produce rigorous handbooks similar to those available for other engineering fields. [209]

The push to identify, catalogue, and name PSMs in certain ways resembles the design patterns work [232], except that instead of codifying successful and recurring patterns of software solutions, the aim is to codify successful and recurring patterns of reasoning in PS. The PSM approach seems like a promising sort of approach for developing taxonomies of processes that can be used as a basis for reasoning about distributing them. The goals of PSM research therefore match well the needs of tool analysis: PSM research tries to build taxonomies of PSMs that are well matched to task type but independent of domain.

Although I know of no work that explicitly applies AI research on PSMs to HCI, the basic utility of the concept has caught on in several places under various disguises. In the realm of HCI, one instance of PSM-like descriptions of problem solving strategies is within Green’s analysis of cognitive

dimensions [257]. The cognitive dimensions framework started out with an abstract description of a task type (basically a design task) and an abstract description of a PSM matched to that task (basically incremental generate-and-modify). Later work has elaborated this “proto-theory” [272] into a taxonomy of 6 different “generic activities” [266,269]. These are described as “incrementation”, “transcription”, “modification”, “exploratory design”, “searching”, and “exploratory understanding”. These are all related to high-level PSMs in that they classify generalized yet task-related strategies of employing artifacts in the solution of problems. Currently, the problem of describing and classifying such “activities” is a hot topic in research on cognitive dimensions [54]. Another excellent example of the application of PSM-like models is the “resources model” of Wright *et al.* [719]. Part of this modeling framework consists of a short list of what they termed “interaction strategies”. These interaction strategies play a role similar to PSMs. They used these interaction strategies as resources for reasoning about how to redistribute different types of data (or “resources” in their terminology). Unfortunately, Wright *et al.* did not consider how the interaction strategies could be used to categorize types of process distribution. A further instance is the recent work by Bhavnani *et al.* [46,48] on what he calls generalizable *efficient strategies*. In one paper [46], he lists 10 strategies which are grouped into four clusters: “iteration”, “propagation”, “organization”, and “visualization”. These strategies are relatively “low-level” but they share the key idea of capturing strategic knowledge of how to solve problems efficiently using the structure of the task environment. Similar types of taxonomies of PS strategies also appear in many other places; one can usually find them whenever an abstract description of PS strategies is desired (e.g., in hypertext [717] and writing [441]).

The concept of abstract PSMs is also commonly applied in software engineering and program comprehension research. A sizable fraction of the program comprehension work attempts to categorize program comprehension behaviours in terms of abstract PSMs. For example a common categorization is the tripartite split of comprehension strategies into the so-called “top-down”, “bottom-up”, and “opportunistic” strategies (e.g., see Robson *et al.* [545], von Mayrhauser *et al.* [675]). This is an old categorization of generic PSMs adopted from prior AI research. Top-down PS is knowledge-, theory-, or goal-driven PS; bottom-up PS is data driven; and opportunistic PS is an interleaving of the two to take advantage of *opportunities* that arise (e.g., see Carver *et al.* [113]).⁹ PSM-like descriptions of other behaviours have been given elsewhere. For instance Myers [432] defined various debugging strategies like “inductive”, “deductive”, and “backtracking”. Overall, the PSM line of work seems promising. Unfortunately, at this time it does not appear to be mature enough to be used systematically in this chapter.

4. **Cognitive Function Taxonomy.** Intelligent agents of any form must perform basic cognitive functions regardless of the tasks that need to be performed. There is no closed and agreed upon enumeration of these functions, but they include planning, learning, deliberating, perceiving, communicating, being vigilant, and so on. Newell [446, pg. 15] provided a tentative list of such functions,

⁹Sometimes other distinctions are used such as “systematic” versus “as-needed” [388,676], however these distinctions do not yield different problem solving strategies: it is possible to be either systematic or “as-needed” using either top-down or bottom-up comprehension.

however numerous similar lists can easily be constructed just by browsing through the tables of contents for many textbooks or encyclopedias in cognitive science and HCI (for example, see the recent human factors handbook [341]). A rather different sort of list of cognitive functions is put forth by Parasuraman *et al.* (see e.g., Parasuraman [484]). They propose to divide cognition into four stages of information processing: (1) acquisition, (2) analysis, (3) decision/action selection, and (4) action implementation. This sort of decomposition of activity is also popular, and similar decompositions can be found in many different places, such as Mayhew's book [401], and in the multi-stage models of Rasmussen [526] and Norman [465–467].

Whatever the decomposition of cognitive function is chosen, the basic idea for distribution is the same: what is applicable for an *individual* intelligent agent can apply equally well to *extended* or *joint* cognitive systems. Consequently, one may talk about how any of the above cognitive functions may be distributed within a DC system. One may think, therefore, about how to redistribute planning, perceiving, learning, and so on. In Parasuraman *et al.*'s case, they use their taxonomy to discuss different levels of automation for their four types of cognitive function [484]. A list of cognitive functions is thus one possible way of decomposing the different types of cognitive processing distribution. There is definitely promise in this sort of decomposition. Of course there is still the small matter that nobody knows what a list of cognitive functions should include [446], but even so an initial list could be drawn up and ways of distributing these functions could be considered.

Although this is a promising approach at this point only the basics can be covered. HASTI does not explicitly incorporate a cognitive function taxonomy. But in some sense a limited one is already implicit in the Agent model: each of the panel types can naturally be associated with certain cognitive functions (recall that the panels group together function-related types of data). Distribution of those panel data types therefore implies distribution of the associated cognitive function. Plan distribution implies planning distribution, constraint distribution implies distributed constraint proposal and evaluation, and so on.

5. **Virtual Architecture.** As was mentioned above, virtual architectures include generalizations of tasks pertaining to interaction with external devices. The operations within these models are candidates for distribution. For instance, a virtual shared memory architecture might define operations such as searching the memory for a matching item. This is clearly a task that can have some processing distributed onto a tool. In the D2C2 taxonomy the operations of the virtual architectures are *device* tasks.

From the above it should be clear that determining a good decomposition method for describing process distribution types is an unfinished but fundamental challenge. It is not possible at this time to be completely systematic about generating generic process distribution theories from RODS. As a result, at present it may be possible to only go so far as to say that certain instances of support are some type of processing distribution (i.e., D/P), but without being able to classify it further.

Newell [446, pg. 16], when facing the slipperiness of human cognition, took solace in Voltaire's saying "the best is the enemy of the good". Likewise, with a perfect decomposition of processing unattainable, it is time to satisfy [594]. Each of the five possible decomposition methods listed above seem to have

some merit. Consequently the following adopts an attitude of pragmatic theoretical eclecticism, and a few prudent examples are picked from four of the five decomposition methods listed above. These are as follows:

1. **Generic domains.** Three examples are chosen: mathematical and statistical operations, searching and pattern matching, and abstraction. The first is an “obvious” choice and the last two are particularly relevant to software reverse engineering.
2. **Domain task/process models.** Saved for Chapter 8.
3. **PSMs.** The generic PSM of *iteration*—likely the simplest—is chosen.
4. **Cognitive functions.** The following cognitive functions are chosen based on the Agent model data types:

RESOURCE TYPE	ASSOCIATED COGNITIVE FUNCTIONS
agenda	attention management, goal selection
control	planning
problem	constraint observation
progress	progress evaluation
*	learning

Here * means that learning can apply to any type of knowledge. Other cognitive functions could easily be added, but these are a good start.

5. **Virtual hardware.** A virtual memory architecture example is examined. The focus is on the memory management operations it describes. These are broken down into “back-end services”, “layout”, and “page management” types. Page management, in turn, is decomposed into “window management”, “page replacement”, “view management”, “working set management”, and “localization”. These are all described in more detail below.

Note that only the five basic types of process decompositions are shown in the diagram of CoSTH in Figure 6.1. The five further decompositions given here should be thought of as potential, tentative extensions to these five types. In such an extension, these leaves would look as in Figure 6.3. For space saving reasons, several obvious abbreviations are used. Also, as indicated in the figure, several acronyms are used for many nodes as they were in listed in Figure 6.1.

Generic Domains: Statistical and Numerical Methods

* RECAP/INTRO

Automating mathematical and statistical processing is the archetypical use of computation. Because this is by now so mundane, it is good to include it as an example.

* INTERPRETATION GUIDELINES

Basically any computation specified in mathematical language.

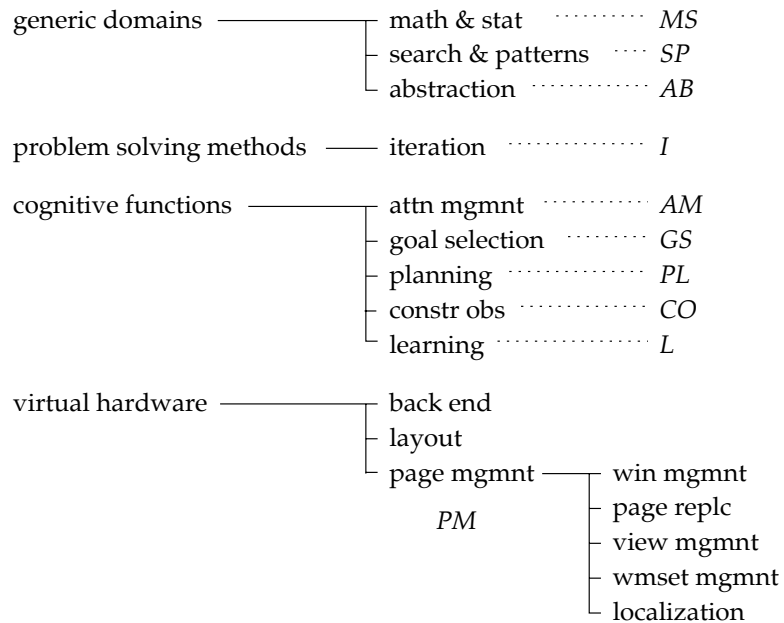


Figure 6.3: Tentative leaves for processing distribution

★ SAMPLE IMPLEMENTATIONS

Notable examples of domain-independent environments include *Mathematica*, *SPSS* and *VisiCalc*.

Generic Domains: Pattern Matching, Search

★ RECAP/INTRO

Pattern matching and search are common functions of models of long-term memory.

★ INTERPRETATION GUIDELINES

Commonly understood.

★ SAMPLE IMPLEMENTATIONS

Grep, for instance [596]. More involved examples include involved pattern matchers like *SCRUPLE* [490], and *QBO* [25].

Generic Domains: Abstraction

★ RECAP/INTRO

Performing “bottom-up” analysis of data involves aggregating patterns and then classifying them using abstractions. This abstraction process is often hierarchical in nature. This type of task is common in reverse engineering (e.g., Müller *et al.* [426]), but also in many other domains such as synthesis tasks in writing (e.g., Neuwirth *et al.* [441]).

★ INTERPRETATION GUIDELINES

Typically involves reinterpreting data at a “higher” semantic level by looking for commonalities and structural modularity.

★ SAMPLE IMPLEMENTATIONS

Statistical semantics. Algorithms can use statistical co-occurrence or other association measures as indicators of “latent” meaning (e.g., object-verb associations [338]). The overall technique can be called “statistical semantics” [372].

Clustering algorithms. Clustering usually means collecting items together according to similarity or connectivity measures, although it also can imply the construction of new abstractions or concepts with which to associate clusters (e.g., Tzerpos *et al.* [648]).

Concept analysis. Concept analysis is a way of generating categories and simultaneously classifying items based on common features (e.g., Snelting [607]).

Graph layout algorithms. Many layout algorithms, like spring layouts, compute “closeness” metrics in order to lay out nodes according to connectivity (e.g., Müller *et al.* [426]). When presented in an appropriate manner they can help the reverse engineering perceive these clusters, or at least perceive good initial candidate clusters [359].

Virtual Architecture: Virtual Memory Management

★ RECAP/INTRO

There are certain overheads associated with any external memory (save, perhaps, if we could implant RAM chips and memory controllers directly into the brain). These overheads include carrying out interface tasks to simulate memory access and management. A Virtual Memory architecture is a trick to raise the abstraction level of the interaction model in order to encapsulate the complexity of memory management. The architecture used in HASTI models a single virtual memory which is in fact implemented by combined user and external memories and managed in part by the user. A task analysis of external memory management will reveal several classes of operations. An important subset of these operations are concerned with the management of the memory resources. This includes creation and management of search indexes, managing caches, and managing memory windows (paging).

★ INTERPRETATION GUIDELINES

Remember it is the user’s view of external memory that is being referenced here, not the application programmers’. It is critical not to confuse the two. For instance to its users, a phone book application is an external memory; to the application’s programmer, the term may additionally refer to a database stored on disk. From the user’s point of view memory management and access operations are restricted to mean the methods provided by the application to store, search, access, and organize this memory.

★ SAMPLE IMPLEMENTATIONS

“Back-end” services. Modern computing platforms have a remarkable number of sophisticated memory systems on which they are built: structured documents, hierarchical file systems, and relational databases for example (e.g., see Jones [337]). Even when the database or filing services are hidden in the “back end” of a program, these sorts of systems already perform a wide array of processing for accessing, indexing, and so on. For instance, a modern relational database will provide index processing—a case-based database will provide even more advanced index processing. These can frequently greatly reduce accessing and indexing costs associated with external memories (c.f. filing cabinets, file folders, card-based indexes, printed subject indexes).

Layout, sorting. Often the main criterion for layout algorithms is that indexing is made efficient. For instance, an array layout, like that of a spreadsheet, makes address calculation simple (see e.g., Larkin *et al.* [375]). The work done by the layout algorithm to rearrange the data into an efficiently accessed display is a form of external memory management processing.¹⁰ For this reason, sorting should frequently be counted as layout processing. For example, sorting file names in a directory listing is layout processing for the purpose of improving access efficiency to the list items.

Page management. In the VM architecture in HASTI, a computer screen is treated as *memory window* onto a larger external memory, that is, it is a small, directly accessible fragment of a larger, indirectly accessible memory. Managing this memory window is a function that can be at least partially performed by the computer. There are several examples from the literature that one may point to:

- *Window placement/management.* Many windowing systems perform work to automatically place new windows so that they do not need to be managed directly. This may also include creating new windows that effectively subdivide the display so that local page replacement schemes can be used within each window.
- *Page replacement.* Wiecha *et al.* [704] propose a “visual cache” as a page replacement strategy that tries to automatically manage a small number of fixed-position windows. Card *et al.* [92] used a related page replacement idea for window management.
- *View management.* On multi-view systems, a single memory system is accessed through different views. Automatically maintaining consistent logical location between views is one example of distributing view management. One prime example is so-called “synchronized scrolling” mechanisms [83, 228, 585, 631], where updates to one portion of the window precipitates corresponding updates in related views.
- *Working set management.* The set of memory locations being accessed in a given segment of time is a process’s *working set*. This management processing can be redistributed. This might involve loading in parts of memory that are about to be, or are likely to be, accessed,

¹⁰This work done by layout is of a rather different sort than the that described above concerning structural abstraction. Here the issue is layout for the purpose of efficient access; there the issue is layout for the purpose of structure perception.

or it might involve scaling the display automatically so that the entire working set can be seen at once.

- *Localization*. When accessing a slower memory through a faster memory window, *memory access locality* [92, 278] becomes enormously important to performance. The reason is that accesses to areas outside the current window will require “paging in” operations (scrolling, loading files, etc.). One way of helping to ensure access locality for a widely scattered working set is by providing a “virtual” or temporary view that presents the scattered locations in close proximity. In other words, one can localize a non-local working set [278]. Methods of doing this include various ways of performing elision [230, 279, 328, 416, 557], filtering, and querying [145]. Processing towards this end redistributes localization processing.

PSM: Iteration

★ RECAP/INTRO

Iteration¹¹ is not really a PSM in the same vein as PSMs in KE. Still, it fulfills the main qualifications: it represents a generalizable computing pattern or structure that is specialized to take advantage of specifics in the task or environment in order to be efficient. Iteration applies operations to elements of a structure when given a way of traversing it. Distribution can mean getting another agent to do some of the iteration.

★ INTERPRETATION GUIDELINES

Usually some method of specifying a structure or traversal is necessary.

★ SAMPLE IMPLEMENTATIONS

Wildcards, tag-and-drag. In command line systems wildcards are often used to specify the structure to iterate a command over. For instance in UNIX the command line “/bin/rm *” applies the *remove* command iteratively to a group of files. In many visual shells, iterated commands are invoked by performing an aggregation operation (“tagging”) followed by a gesture or command (“dragging”) to indicate the operation to perform on each item in the aggregation. Similar distribution methods can be found in many other interfaces, such as in computer-aided design (CAD) drawing packages [48].

Program-driven interaction. Iteration follows a control loop. Sometimes this control loop is delegated onto a computer agent which exercises control over the user’s action. A simple example is a wizard that guides a user through a sequence of steps. Perhaps a more familiar example is a search-and-replace agent that asks the user for a sequence of yes/no responses.

Cognitive Function: Attention Management and Goal Selection

★ RECAP/INTRO

An agenda is maintained in order to enable focused, goal-directed behaviour (instead of being

¹¹I have chosen the imperative form of description rather than a declarative form (*mapcar*, higher-order-functions, etc.) primarily because the state is so prominent when discussing HCI.

purely data- or percept-driven), especially in the presence of multiple tasks or subtasks. The agenda therefore serves a key role in *managing attention* and *selecting action* based on the active goals. The priority or urgency of goals will change from moment to moment [113] as new opportunities are recognized in reference to pending subgoals [489]. Goal selection is also affected by an agent's perceptions of the cost versus benefit of each possible action [113,661]. Both attention management and goal selection may be at least partially redistributed. In these cases, external agents take on some of the load of recognizing opportunities, prioritizing subgoals, and commanding attention.

★ INTERPRETATION GUIDELINES

Attention management and goal selection is usually associated with a shared agenda.

★ SAMPLE IMPLEMENTATIONS

Alerts. Many systems will interrupt the user and, therefore, manage the distributed agenda [212]. An obvious example is an operating system that presents so-called “moded” dialogs when displaying urgent alerts—these dialogues must be responded to first before anything else can be done.

Critiquing. Intelligent critiquing agents will manage a distributed agenda by interrupting when they believe alternate goals should be selected. An example is the critiquing agent of *Argo* [544], which looks for potentially problematic design decisions and interrupts the user at appropriate times.

Cognitive Function: Planning

★ RECAP/INTRO

Planning means either establishing intentions for future actions or else establishing a course of action, i.e., creating plans.

★ INTERPRETATION GUIDELINES

Any processing done to generate structures that are followed by the user can be considered planning.

★ SAMPLE IMPLEMENTATIONS

Error list generation. When compilers generate an error list they are generating a partial repair plan for the program. The list ordering frequently acts as a step ordering since programmers often perform repair activities by sequentially scanning down the list for repairs to make.

Query result ranking. Ranking query result sets is a common technique in information retrieval. Result ranking is external planning when the ranking indicates future exploration order.

Cognitive Function: Constraint Observation

★ RECAP/INTRO

Problem constraints can be obeyed through the vigilance of the user or be enforced by the environment [727].

★ INTERPRETATION GUIDELINES

Generally any type of processing that checks and enforces constraints or reports constraint violation is a form of constraint enforcing.

★ SAMPLE IMPLEMENTATIONS Any externalized constraints are potential candidates for being checked automatically. For instance type checking of compilers is one instance.

Cognitive Function: Learning

★ RECAP/INTRO

Learning can be thought of as a process of accreting and organizing knowledge, *and* as a process of improving the ability to make knowledge available for use. A DC system embodies knowledge in both internal and external forms. This knowledge encompasses both *declarative* like facts, concepts, and maps, and *procedural* knowledge like rules, scripts, strategies, and programs. DC learning therefore means augmenting and refining internal and external knowledge, or adding new capabilities for recalling and mobilizing this knowledge. Distributing learning processes means moving some of the processing required to do this between agents.

★ INTERPRETATION GUIDELINES

Distributing learning can involve instances where work is done to synthesize new knowledge structures, or to construct new rules or programs. It can also involve cases where knowledge is otherwise made ready-to-hand, such as through indexing, adding search or construction programs, and so on. One type of learning distribution is not considered in this chapter: when designers embody knowledge in tools or environments. For instance, populating a plan-matching program's database using knowledge engineering (e.g., Wills [707], Chin *et al.* [127]) certainly redistributes learning, but in these cases the tool developer has a special relationship to the user. This relationship is such that it is awkward, to say the least, to view the developer-user activity as a collaborative learning effort (but see Section 6.7 for more discussion of this topic). In contrast, it is frequently the case that groups of peer software developers cooperatively collect together a code repository, or add scripts or macros to their shared development environment. This would be more easily be considered learning distribution.

★ SAMPLE IMPLEMENTATIONS

Programming by example/demonstration. It is possible for a computer agent to abstract procedural knowledge from examples given, or from demonstrations of how to perform tasks [386]. For instance, a system might abstract procedures for program transformations from given examples [464]. Another example is `Varlet` [330]; in `Varlet`, heuristic rules in a knowledge base are tweaked automatically in response to sequences of decisions that a reverse engineering makes while understanding a legacy system.

Script/macro library construction. People frequently share customizations to programmable environments such as `Rigi` [426].¹² Extensions of these sort are frequently in the form of plugins, macros, or scripts.

6.3 Specialization

... *what makes a presentation interesting are the efficient perceptual procedures that users can perform using the presentation to quickly arrive at a desired result.*

– Stephen M. Casner, “A Task-Analytic Approach to the Automated Design of Graphic Presentations” [116], pg. 112.

The SRKM hierarchy provides several categories of cognitive functioning which are ordered in terms of preference and ease. One category can often substitute for another. For instance, at the beginning of learning a new skill, performance is frequently a knowledge-based activity, but after repetition and practice it becomes increasingly dominated by perceptual, skill and rule-based behaviour. Such practiced activities are easy and relatively effortless, and are frequently described in qualitative terms like “natural” and “fluid”. Cognitive support in this context means provision of artifacts that allow perceptual, skilled, and rule-based behaviour to be used in place of more cognitively demanding cognition. This frequently means that modes of interaction are changed, or it means that the way data is encoded is altered. It is worth reiterating here that CoSTH enumerates advantageous changes to cognitive systems, not the ways of *achieving* them. Therefore this chapter does not enumerate the possible ways of making specializations happen: it only organizes the cognitive effects in terms of the specializations that are enabled.

For the most part, this subsection is an adaptation of the SRK work by Rasmussen [526]. However there are some differences worth making note of. The main difference is the decreased emphasis on the monitor-and-respond type of tasks which the SRK taxonomy was originally derived for, and an increased emphasis on highly goal-directed knowledge-based behaviour. In the monitor-respond paradigm, there is a tendency to focus on tight causal loops (e.g., catching a ball or steering a bicycle). The SBB (skill-based behaviour) and RBB (rule-based behaviour) are, in particular, focused on closely coupled perceptual-action loops or cue-response mechanisms. Here, effort was made to separate the perception and action components of the loops. This seems to make it more clear as to where substitution can occur for more complicated and more goal-directed behaviour. The close coupling of action-perception loops is not in dispute; the intention is only to try to distinguish between specializations occurring primarily for perception or action. As a result, the closely coupled loops highlighted by SBB and RBB are somewhat muted in this presentation.

In addition to this difference in emphasis, the SRK model interpretation is recast specifically in terms of HASTI rather than Rasmussen’s “framework” [657] for relating the three different classes of processing. Recasting it in light of HASTI makes it possible to refine CoSTH based on the types of specialization

¹²Kenny Wong, personal communication, November 24, 2000.

HASTI describes. The recasting of SRK also has the effect of changing the language for specialization: Rasmussen’s specializing substitutions are in terms of behaviour substitution—substitution of behaviour phenomena—and in this work the substitution is in terms of computing mechanisms.¹³

The basic idea of specialization substitution might well seem conceptually simple, but understanding specialization in realistic situations is fraught with complications. As Vicente is careful to note [657], real-world cognition is normally a rich mixture of activity on all of the SRK levels. The apparent distinctness of the SRK levels, and the overall simplicity of the SRK framework, mask the difficulty of converting the idea of specialization into credible explanations of support. The complicated nature of the interactions between SRK levels is counteracted by focusing on examples of “simple” specializations. For my purposes here, simple specialization shall consist of a single, wholesale replacement of a computational method with another. A brief summary table of these specializations is presented in Table 6.3. In the table, arrows are used to indicate action/perception differences (\leftarrow = perception, \rightarrow = action). More complicated rearrangements are discussed in Section 6.5.

TYPE	\leftrightarrow	DESCRIPTION	SUBSTITUTES FOR	EXAMPLES
skill	\leftarrow	low-level hardware replaces recognition-based or generalized cognitive machinery	vigilance	sound at significant events
			logical operation	visual search in graphs
			constraint observation	animation visualizations
			computer animation	envisionment using imagery
	\rightarrow	skilled action	mental rotation	Tetris piece rotation
rule	\leftarrow	rapid recognition and recall of information	structure extraction	cliché plan recognition
			situation assessment	display-based recognition
	\rightarrow	stored precomputed solutions are reused	interface metaphors	reuse of skill

\leftarrow = perception / assessment
 \rightarrow = action

Table 6.3: Summary of specialization substitution types

Skill Level: Perceptual Substitution

★ RECAP/INTRO

Perceptual processes can be rapid, automatic, effortless, and can occur in parallel with other processes. HASTI does not describe what these perceptual processes involve, but it is possible to enlist

¹³It is worth commenting briefly on this point since Rasmussen contended [526,528] that it is often sufficient to consider only categories of phenomena (memory issues, cognitive processing types) rather than models of mechanisms underlying these phenomena. It seems that there is no practical different between describing specializing substitutions in terms of phenomenon or mechanism since, after all, mechanism and phenomenon are supposed to correspond to one another. Furthermore, even phenomena need to be described and formalized some how, and abstract models are perhaps the most parsimonious ways known. Since HASTI is compatible with Rasmussen’s explanation of behaviour, shifting the language in the direction of a mechanism orientation means that my model-oriented presentation can always be mapped back onto Rasmussen’s behaviour vocabulary. See Section 5.4 for more information on the mapping.

the help of other resources in order to fill in some of the details. Casner [116], for instance, lists a catalogue of some 17 perceptual operators that can potentially be substituted for inferences. Certainly there is ample opportunity to expand this list further, but there is rather little guidance as to how to do it. For example, Kosslyn [360] lists four abilities of imagery that might be enlisted. Are these related to Casner's list of perceptual operators, and if so, how? Resolving this level of modeling is beyond the scope of this work, and we shall have to make do with just listing a few representative instances of specializations. Regardless of how one models perceptual capabilities, the number of *logical* operations that can be substituted with perceptual operators is limited only by the ability to encode problems in a representation that enables the specialization.

★ INTERPRETATION GUIDELINES

Perceptual processing is generally difficult to casually observe because it is usually so rapid and un-introspectible. Clearly one of the clues as to whether or not specialization is possible is the form of the artifact: if there are differences in visual form and presentation, these are the changes that are likely to impact perceptual processing.

★ SAMPLE IMPLEMENTATIONS

Observational vigilance. Perceptual cues can replace the need to maintain observational vigilance (essentially situation or condition polling). The most obvious example is probably the use of a sound to signal the occurrence of a significant event like the completion of a lengthy operation.

Simple search. A classic example of perceptual operator replacement is the substitution of tabular data presentations for graphical ones. For instance consider the task of searching for a minimum or maximum value in a list of values. In tabular form it is straightforward but involves number interpretation and memory. Change the presentation into a line graph, and the search becomes a simple visual search for the highest point in the line graph [472].

Constraint violation detection. Kraemer and Stasko [362] give an example where a violation in a program's constraint can be detected by a perceptual judgment when watching an animation of its execution.

Imagery and proxy physical manipulation. Mental visualization skills can operate rapidly on "mental models" in a way that is almost exactly as if perception and physical manipulation were used [360]. Evidence is accumulating that suggests that both mental imagery and perception activate common low-level brain functions [360]. This provides a partial explanation for the efficiency of mental imagery in terms of specialization, i.e., that imagery skills could substitute for other knowledge-based cognition. In this sense the title of "perceptual substitution" at the head of this section is actually a misnomer since no actual perception need be involved to activate the imagery processing capabilities [360]. There is certainly ample evidence to suggest that mental imagery is an important part of a software developer's mental life [507]. The only question is: other than being used in training, how can artifacts be said to support cognition through imagery? Not directly, perhaps, but since imagery can be seen as an internally-generated form

of perception and action, imagery may substitute for more taxing cognitive actions when a visualization or visual is not present or in view. For instance, consider the immediately prior example of visual constraint violation detection of a program animation. After detecting a constraint violation, imagery may come into play. A developer might be able to visualize (i.e., using imagery, not using a visualization program) how the animation will play out if she makes certain changes the program.

Skill: Action Substitution

★ RECAP/INTRO

Skilled manipulation of external artifacts is similar to perceptual processing in that certain operations are easy, rapid, largely automatic, and parallelizable.

★ INTERPRETATION GUIDELINES

External artifacts obviously need to be manipulable in order for action substitution to occur. This manipulation needs to be used instead of perform reasoning or other mental action.¹⁴

★ SAMPLE IMPLEMENTATIONS

Tetris moves. A well-studied example of action substitution is the instances of “epistemic action” noted by Kirlik and Kirsh (see Section 4.1.6). For instance, Kirsh and Maglio’s observations of Tetris¹⁵ game playing [353] showed that players will rapidly rotate pieces—not to move the pieces closer to their final orientation—but to (among other things) effectively structure the problem by eliminating orientation variation on the input.

Media manipulation. Many problem solvers use skilled manipulation of external media to substitute for problem solving that would otherwise require mental gymnastics. Writers manipulate notes and diagrams [575]; designers manipulate models and materials [566].

Rule Based: Recognition-based Substitution

★ RECAP/INTRO

In the purposefully simplistic HASTI model, recognition is a function of an active memory system that constantly searches long-term memory for relevant chunks or facts based on the contents of working memory and perceptual buffers. This retrieval could be based on cues generated by perception, or by other, higher level cognitive processes. The retrieval is automatic, fast, and effortless,

¹⁴Note that skilled action is frequently exhibited in combination with perceptive and reactive capabilities—what Rasmussen terms “skill-based behaviour”. For instance tracing a drawing involves the close coordination of hand movement and perceptual feedback [526]. In spite of this fact it is still reasonable to differentiate between perceptual substitution and action substitution since perceptual substitution can stand well on its own. For example for simple visual graph search the only physical skills that may be needed are the ability to move one’s eyes. The distinction is blurred by “active vision” [349] (see also Ware and Franck [688]) in which perception and action are tightly linked, however this blurring does not negate the utility of making the distinction.

¹⁵A video game in which players must maneuver moving game pieces into winning positions under time constraints.

however it is dependent upon having the right cues available—sometimes the cues required are nearly exactly the same cues used to encode and index the memories. This fact means that recall and expert performance are sometimes situation specific [657]. Many types of memories could be recalled, including past episodes, facts, and images. Memory's basic function is that it obviates the need to reconstruct the recalled information. This is clearly useful when the needed information is transient, as it is for information that has become scrolled off of the screen [10]. But it is also useful when recognizing information that is *latent*, *implicit*, or *implied* in a display. For instance it is usually far better to rapidly recognize a threat from a falling boulder than to have to painstakingly deduce it by calculating its probability of landing upon you with crushing force. Display-based problem solving (see Section 6.5.2), in particular, relies on the rapid perception of problem-solving state.

★ INTERPRETATION GUIDELINES

Recognition-based action is often identified with fluid expert responses that do not leave verbalizable traces [657]. Subjects seem to automatically and rapidly perceive and recognize familiar patterns and cues. Generally speaking, recognition-based replacement requires some expertise in recognizing patterns and cues in the environment and in the problem situation.

★ SAMPLE IMPLEMENTATIONS

Cliché recognition. Experienced programmers can rapidly recognize clichéd patterns (see e.g., McKeithen *et al.* [403]). Much of the research in relation to this ability has been concerned with the recognition of the so-called programming *plans* [51, 241, 609, 707] and programming *idioms* [6, 611]. However the findings probably generalize for all relevant recurrent structures (text structures, control flow structures [266], design patterns, architectural patterns, etc.). As one would expect, these structures would need to be relevant to the programming language, and the programmer would require training or experience to be able to recognize them [241, 375]. Ways of enabling recognition include methods for presenting the patterns in such a way that they can be easily recognized [241], or using names that cue the appropriate clichés.

Situation recognition. As mentioned in the section on data distribution, expert display users often rely on externally represented action cues in order to determine what to do next. For example during coffee making [374], the coffee maker will be able to quickly perceive the state of the coffee making. This recognition will cue a response for what to do next. In this perceive-respond action, the coffee maker does not consciously reason about the state of the coffee making. Presentation of external state (i.e., situation) in a way that enables such recognition is one way of enabling situation recognition substitution.

Rule Based: Stored Rule Substitution

★ RECAP/INTRO

Knowing how to act in a given situation can be difficult and involve a lot of reasoning. Experts have been exposed to many different situations and have learned rules that they can apply to similar situations when they are recognized. Rules are essentially precomputed solutions: the experts'

vast memory with quick cue-based access means the strategy of storing precomputed or “compiled” results will be often faster than recomputing them again.¹⁶ Recalling past solutions to problems is a method of substituting memory-based computing for more effortful reasoning. Expert programmers, for instance, have a vast [72] stored collection of rules on how to write small snippets of code. When presented a situation in which they need to produce a similar snippet, they will not perform laborious reasoning but will instead promptly produce a variation of the canned result. As Sheil said, “If you know how to program, you would neither generate [a familiar fragment] nor synthesize an understanding of it. You would *know* the answer. You would *recognize* the problem, key directly into that knowledge, and pull out a working procedure” [581, pg. 117] (emphasis original).

★ INTERPRETATION GUIDELINES

Rule-based action generally is recognizable by its rapid and unverbalizable nature. For instance an experienced designer may adopt certain problem constraints without comment and seemingly without thought (e.g., Eastman [194]).

★ SAMPLE IMPLEMENTATIONS

Interface metaphors. Experts develop skill at performing routine low-level tasks such as interface interaction. At least part of smooth skilled performance is the mobilization of rules saying how to achieve task goals using the interface (these would correspond to methods or selection rules in GOMS [94]). It has been suggested that metaphors for interaction (e.g., the “desktop” interface) taps into this knowledge base. The user may be a relative expert in the metaphorical world, but a novice in the particular task domain at hand. Using the metaphor can effectively “co-opt” this rule-based problem solving knowledge: the user can solve problems in the analogous world rather than the ones they are novices in.

6.4 Algorithmic Optimization

The term “algorithmic optimization” is intended to refer to cases where computations are restructured by substituting algorithms, ADTs, or encodings. A simple demonstration of cognitive restructuring is Norman’s example of changing number systems from Roman to Arabic [472]. Multiplying numbers in Roman format is much harder (for humans) than in Arabic. The change is best categorized as a change in number encoding formats and the corresponding changes in algorithm. It is noteworthy that the improvement to performance can have little to do with any of the other principles of cognitive support: it involves no reduction in task, no distribution of data or process, and no new specializations in the cognitive machinery (it is essentially a rule-based problem for those skilled in the art). It is only the restructuring of the computations that has a supportive effect.

To the best of my knowledge there are no principled ways of categorizing cognitive restructurings. This means that it is impossible to systematically refine CoSTH any further. There are two main reasons

¹⁶This admittedly simplistic account suggests that learning rules requires someone to have a high-level understanding of the solution first in order for the precomputation to be made beforehand. This is not always the case (see Vicente [657, ch .11]), however the distinction is not important for the present purposes.

for this impasse. First, RODS as it stands does not contain a taxonomy of optimization types. Ideally, a taxonomy could be given (in a generic and principled way) that enumerates the types of computational rearrangements have performance advantages. Even with such a taxonomy (and there are perhaps decent initial candidates [18]), there is still the rather enigmatic problem of classifying representation shifts like the one from Roman to Arabic. Much of the existing knowledge, such as Norman's principles of representation [472], either fails to provide a taxonomy of ways of changing representations, or else makes statements about *other* cognitive support types, especially perceptual substitution. Thus, I know of no principled taxonomies of such representation shifts, although there are some potential resources to consider (see e.g., the review by Blackwell *et al.* [55]). The second cause of the impasse is that HASTI does not propose any features that say what sorts of optimizations are possible. Humans are not reprogrammable like computers are, so human psychology will have much to say about which optimizations are reasonable, and which are frankly impossible. Because of these two problems, there seems to be no resources for further refining theories of cognitive restructurings. Consequently CoSTH is limited to merely noting an example of encoding shifts.

Encoding Shift

★ RECAP/INTRO

Making changes to the data encodings for an ADT will usually alter the way the ADT's operations (algorithms, procedures) on them will be performed. The relevant application for cognitive support is when the number of difficult mental operations are reduced, or when the strain on cognitive resources is lessened.

★ INTERPRETATION GUIDELINES

In HCI, the main application of a shift in encoding is when the encoding of externally manifested data changes with respect to the user's perception and interpretation of it.¹⁷ This type of encoding change will typically involve what is sometimes referred to as a shift in *symbol systems* (e.g., see Rasmussen [526]). Note that the type of change of encoding referred to is not merely a cosmetic change to presentation. For instance neither changing the colours, nor using a new set of letters should be considered an instance of encoding shift for Roman numerals.

★ SAMPLE IMPLEMENTATIONS The canonical example of encoding shift is the change from Roman to Arabic numerals which made (most) arithmetic operations easier to perform [472]. For instance using a piece of paper for doing long division is far easier using Arabic numerals.

6.5 Composite Rearrangement

Three types of cognitive support—distribution, substitution, and cognitive restructuring—have been examined so far. Each category of support was further divided into a small number of important variations. These variations were derived from principled consideration of the ways of applying the three support

¹⁷This is opposed to, say, changing the computer's internal encoding of pictures from TIFF to JPEG.

principles to DC models. The result is a collection of a few dozen distinct support theories arranged in three families.

It is fruitful to view the three support principles as orthogonal bases for decomposing complex forms of support into independent principles. This capability to decompose support into its basic components is enormously useful. As Chapter 3 showed, there are a bewildering number of different forms and variations of cognitive support. A decomposition framework can bring some degree of order to this wild menagerie. It can show the many variations of support to be different manifestations and combinations of just a few handfuls of primary support types. The analogy utilized earlier was that the support principles decomposition plays a role similar to the one played by periodic table of elements in chemistry: the periodic table brought order to the endless variety of chemical compounds (it also, incidentally, was organized into families). The support principles decomposition framework of this chapter may not yet be fully developed and completely satisfactory, but it should be clear that some sort of decomposition framework is both necessary and possible.

Up until this point, it has been quite reasonable to restrict our focus to distinct types of cognitive support. Being able to pick apart distinct types of support can simplify the explanations made for them. However, in reality, the categories of cognitive support is not so easily disentangled. The attentive reader may have already noticed this fact after examining the examples of cognitive support used so far: most of them hint at—or appear to depend upon—other support principles in addition to the one being highlighted by the particular example. To further illustrate the interdependencies between support types, any number of additional examples could be brought forth. Consider the following example, which was described by Sharples in reference to the support provided by writing environments:

One way to overcome the difficulties of performing ... complex knowledge manipulation in the head is to capture ideas on paper (or some other external medium such as a computer screen) in the form of external representations that stand for mental structures. So long as ideas, plans, and drafts are locked inside a writer's head, then modifying and developing them will overload the writer's short-term memory. By putting them down on paper (or some other suitable medium) the writer is able to explore different ways of structuring the content and to apply systematic transformations, such as prioritizing items, reversing the order, or clustering together related items. Writing creates external representations and the external representations condition the writing process. [575, p. 135]

Clearly an account of something even as “simple” as pen and paper will need to be multifaceted. Sharples’ brief synopsis makes oblique references to several forms of support. These are referred to here as *D/D* (ideas, plans, drafts), *D/P* (sorters, clusterers), *S/S/perceptual*, and *S/S/action*. This example demonstrates that even for a simple collection of artifacts, different types of cognitive support may be potentially involved. Consequently, to make sense of realistic instances of cognitive support some way is needed for understanding how the support principles compose. To continue with the analogy to chemistry, something akin to a theory of chemical composition is required. How do atoms compose? What families of compounds are possible? These questions are answered by a *composition theory*. A similar sort of theory is needed for CoSTH.

An encompassing theory of support composition is beyond the scope of this work. However it is possible to gain a sense of what may be involved, and to give at least a few representative examples of cases where support principles composition is at work. Section 6.5.1, reconsiders many of the examples which were used in the preceding sections. These were previously held up as exemplars of the various types of support. These examples are used to show that other support types are usually found in conjunction with them. The aim is not to be exhaustive, but to be illustrative. Then, Section 6.5.2 considers two commonly cited examples of complicated reengineerings of cognition. Decomposing these high-profile compositions of support types using CoSTH hints at the breadth of CoSTH. It also may help to develop intuition as to how the support principles compose.

6.5.1 Revisiting Previous Examples

This subsection revisits some of the previously described examples to give some inkling as to how different support principles combine. A selection of these examples are summarized in Table 6.4. Multiple support types were already mentioned for the first two listed in the table. The last three entries (in italics) are from the next subsection, and can be ignored for the time being. The table is not meant to be searched for patterns, it is only intended to summarize the selected combinations being analyzed. However the table should help make it clear that CoSTH makes it conceivable to rationally compare even vastly different tools and environments based on how they serve to improve cognition.

	Data Distribution						Processing Distribution						Specialization				O			
	PB			PG			GD			VH	PS	CF			Skill			Rule		
	G	P	O	C	G	H	S	MS	PS	AB	PM	I	AM	PL	CO	SP		SA	SR	SU
requirements docs				•	•															
writing media		•						•	•	•							•	•		
skeletal declarations	•																			
...boobytrapped	•			•												•				
MediaDoc	•	•	•											•	•					
wizards		•						•												
Emacs' compile		•	•	•				•			•				•					
Emacs' igrep		•	•					•		•	•				•					
search-replace		•	•					•		•	•	•			•					
context-sens. menu								•							•					
type checker				•												•				
browser links								•									•			
reminder agent		•													•					
<i>DBPS</i>				•				•									•		•	•
<i>precomputation</i>		•						•	•	•	•						•			•
<i>backtalk</i>			•	•				•									•	•		

(abbreviations from Figures 6.1 and 6.3, except DBPS = display-based problem solving)

Table 6.4: Examples which are compositions of multiple support factors

★ D/D/goals

- **Skeletal declarations.** In programming, skeletal declarations can be used to externalize pending subgoals. A trick I have seen used in this respect is to ensure that each skeleton generates compile-time or run-time exceptions.¹⁸ “Booby-trapping” the skeletons in this manner ensures that these pending goals are attended to at some point in case they are forgotten. This is a way of externalizing constraints (D/D/PB/constraints) and externalizing constraint checking (D/P/CF/constr-obs) so that either the compiler or the runtime system performs the checks.
- **Shared goals.** The GoalPoster agent in the Molehill environment can be seen as a collaborative agent involved in plan setting (D/P/CF/planning) and attention management (D/P/CF/attn-mgmt). In the Mediadoc example, the task list serves also as an external plan (D/D/plan).

★ D/D/plan

- **Scheduler.** Schedulers frequently have reminder functions (D/P/CF/attn-mgmt).
- **Scripts, macros, wizards.** These external plans are normally executed (at least partially) by the computer (D/P). Stepping through a wizard requires that progress information be kept by the computer (D/D/PG/curr). Wizards will normally display, at each choice point, a number of options available to the user (D/D/PB/operations). They may also display the name of the task being performed to keep the user reminded of it (D/D/goals).
- **Error lists** Generating the error list is a form of external processing (D/P), specifically planning (D/P/CF/planning). But the presentation of the error list may be made such that the user can apply visual operators to determine the order in which errors should likely be attended to [476] (S/S/perceptual). Many compilation environments have functions specifically designed to allow the programmer to step through the plan. This requires some functionality in common with wizards, namely the presentation of stepping options (D/D/PB/operations) and tracking progress (D/D/PG/curr). A good example is the popular editor Emacs. Emacs has a compile mode that allows the user to step through an error list from a compiler by hitting a “next” key sequence to move to the next list item. When a compilation is invoked, Emacs automatically partitions the screen so that both the error list and an editor page are shown (D/P/VH/PM/win-mgmt), and then pages in files and scrolls to the error location as needed after the “next” key is pressed (D/P/VH/PM/page-replc). Note that even if such functionality is not available, users can frequently simulate it by using multiple windows strategically: one places compilation results in a separate window and then uses the cursor position (or perhaps a finger) to trace one’s progress.
- **Query results.** Query results are frequently used much as error lists are: to traverse a sequence of related locations. Another Emacs mode called igrep works analogously to compile mode,

¹⁸See for instance Wiki Web’s entry (<http://c2.com/cgi/wiki?StubButton>) for “StubButton”

except that instead of stepping through an error resolution plan, it allows one to step through a list of search results created using the tool `grep` (or any other program that creates a suitable list for that matter). Similar types of stepping tools can be found in many other environments such as `tkSee` [596]. A variation of this stepping mechanism are so-called search-and-replace mechanisms, like Emacs' `query-replace` function. Like wizards, such search-and-replace mechanisms step through an update plan calling on the user for decision making concerning replacement (normally "replace" or "skip"). Some interfaces display the choice options (`D/D/PB/operations`). Most search-and-replace involves having the user give up the main loop of control—instead the computer performs a simple iteration over a linear structure defined by the query results (`D/P/PS/iteration`).

★ `D/D/problem`

- **Menus, buttons, options.** Context sensitive menus implement moded dialogues, and therefore need to have a shared problem state (`D/D/PG/state`). They also have to process this shared state in order to determine the possible options in the present context (`D/P/CF/planning`).
- **Language constraints.** External constraints are frequently created in order to have them mechanically checked (`D/P/CF/constr-obs`).

★ `D/D/progress`

- **Visitation history.** Browsers frequently indicate link visitation state using colour. Users can apply visual search routines (`S/S/perceptual`) to locate unvisited nodes.
- **Change history.** Like visitation history mechanisms in browsers, `SeeSoft` colour codes program fragments. It uses colours to show the number of change episodes recorded for each program fragment. Users can apply visual routines to substitute for high-level search operations (`S/S/perceptual`), and the colouring may serve to draw attention to high-priority items (`D/P/CF/attn-mgmt`).

★ `D/P/GD`

- **Graph layout algorithms.** Humans use perceptual capabilities to determine clustering in many graph layouts (`S/S/perceptual`).

6.5.2 Reconstructing and Naming Happy Composites

At the beginning of this section I recounted Sharples' analysis of the values of writing with paper and pen in hand. In that example the cognitive support principles seemed to fall into place one by one, so that the whole was more than a simple combination of the parts. Externalizing mental structures onto paper unburdens memory. Once externalized, it can be manipulated skillfully, processed externally, and be searched visually for pattern and constraint violations. Moreover, being able to manipulate and perceive the externalization is another reason to externalize it in the first place. This fact becomes even clearer if a computer can help process it. Thus, when one analyzes all of these benefits, it seems that the pieces

fit together and lean on each other in a mutually reinforcing way—much like the mutually reinforcing structure of a stone arch. The support principles appear combine gracefully. The design patterns community might say that the artifact *resolves multiple forces* [148]. I do not necessarily wish to discourage the possibly prosperous comparison to design patterns. But for the sake of differentiation, let us not call them “patterns” just yet. Instead, let us agree to call them “happy composites”.

A good theory of cognitive support composition would say how these happy composites arise. This is work for the future. But in the meantime, a bottom-up approach of analyzing good compositions may still yield some fruitful intuitions. With CoSTH in our toolbelt, we are in a position to pick up a few happy composites and decompose them to “see what makes them tick”. In this section a few happy composites are selected from the literature and examined to determine how cognitive support principles have combined. Although the approach is unlikely to yield a theory, it may deliver insight.

This bottom-up approach has two other important and simultaneous benefits:

1. REINFORCING COSTH. If the happy composites are chosen wisely, it may be possible to lend some credence to the *de*-compositional account of cognitive support offered by CoSTH. Specifically, if it can be shown that existing published accounts of the happy composites can be *reconstructed* using CoSTH, it argues for the usefulness and applicability of CoSTH.
2. TOWARD A PATTERN CATALOG? The happy composites describe the pattern of a fruitful solution to a problem. In particular, they describe solutions that appear to “resolve multiple forces”. Thus, even if we do not have a generative theory for building the happy composites, we can helpfully name them and construct a catalogue of them. Such a catalogue would almost certainly be useful for designers (see the theory-based design scenario of Section 2.3.2). If the composites can be described in terms of component support principles, it may prove to be a useful indexing of design knowledge. This topic is revisited in Chapter 7.

Towards these ends, three happy composites are selected from the literature in order to consider their composition. These are called “display-based problem solving” (DBPS), “precomputation”, and “back-talk”. They are reasonably widely known, and they have a rather firmly established terminology. The three happy composites are listed as the last three entries in Table 6.4 (so that they can be easily compared to the composites already mentioned).

It is probably unwise to push the analogy to chemical composition too far, but another interesting similarity can be made. Chemical compounds can be (somewhat) uniquely named by listing their molecular contents. For instance water is HOH (or H₂O to avoid repetition), and table salt is named NaCl. Since the leaves of the CoSTH are also given distinct abbreviations, a similar naming scheme for cognitive support compositions can be entertained. Since lower case letters were not used, the support types need to be separated somehow, and one might as well choose a comma (’, ’). Given this naming scheme, the “chemical name” equivalent for the “common” name of DBPS would be C , S , SP , SR , SU (the order is based on reading Table 6.4 from left to right).

Display-based Problem Solving

Larkin [374] invented the term “display-based problem solving” (DBPS) to describe a form of problem solving that makes extensive use of external displays. It is clear that to Larkin there are three essential qualities of *display-based* problem solving: (1) that all or almost all of the relevant problem solving state can be read from the display, (2) that because of the nature of external displays, perceptual inferences can be used in places where otherwise more taxing logical inferences would need to be made, and (3) that little planning or deliberation is required for any of the steps—the solver employs local control. These requirements appear to be relaxed in other uses of the term “display-based problem solving” (e.g., Davies [165]), however it is not important here to decide what should or should not be considered DBPS solving. Larkin’s notion will do. Using her conception DBPS differs from the “purely mental” equivalent as follows:

- D/D/P/C. The current state of the problem is readily available.
- S/S/perceptual. The solver is able to use perceptual skills to make inferences concerning problem implications, goal selection, or problem constraints.
- S/S/recognition. Because of practice, the user is able to rapidly recognize salient aspects of the problem state.
- S/S/rule-action. The recognized states of the system cue the recall of rules saying what to do in the current state.

As can be seen from this list, DBPS is essentially a skilled way of solving problems requiring little need to store state related to planning or backtracking. In light of newer analyses than Larkin’s, it may be prudent to augment this list of cognitive support types. Specifically, a new analysis of physical implementations of the Towers of Hanoi problems (which was used as an example of DBPS by Larkin) was produced by Zhang *et al.* [727]. In this analysis, the physical implementation provides external constraints that can be perceived instead of determined internally (D/D/PB/constraint).

Precomputation

Hutchins [320, 321, 470] introduced the notion of *precomputations* to refer to artifacts¹⁹ that are constructed in advance to help structure ensuing activity. There are several computationally interesting aspects of precomputations. The most obvious is that precomputation redistributes processing across time. This can reduce the computational load during subsequent performance of tasks. Not all of these interesting aspects of precomputation can be captured fully by CoSTH. However a number can; based on Hutchins’ accounts [320, pg. 164–170] [321], the following support types may be seen to apply to precomputations:

- D/D/P. A common type of precomputation is a checklist.

¹⁹Note that precomputations are *nouns*. They involve *performing* precomputations (transitive verb), but Hutchins uses the term “precomputations” predominantly as a noun. This use of terminology is similar to that of “representation”.

- *D/D/PG*. Many precomputations such as maps and charts are frequently drawn upon. This can record a history of location and computations thereof.
- *D/P*. Precomputations allows other agents to perform part of the work. This work could involve performing and then storing simple calculations; for example, generating a table of precomputed numbers for lookup later [321] (*D/P/GD/MS*), or pre-sorting a stack of charts in preparation for an ordered sequence of actions (*D/P/GD/PS*). These examples suggest that one class of precomputation involves making the subsequent tasks less cognitively demanding by formatting or laying out the artifact to enable skillful manipulation or perceptual inferences (below).
- *S/S/perceptual*. Certain precomputations can be rendered in a form such that perceptual inferences can substitute for complicated logical operations (e.g., distance-rate-time calculations during ship navigation [320]).
- *O/EN*. Hutchins makes it clear that when constructing a precomputation, frequently a change takes place in the data encoding used (e.g., from tables to bar charts). This can make a difference to the mental operations subsequently performed on them. Sometimes this encoding change may allow substitutions by perceptual operations. He gives a clear example in the context of an airline cockpit [321]. In the cockpit he studied, digital representations of the desired aircraft speeds are transformed to analog “speed bug” representations (small indicators on a circular dial). These allow quick perceptual judgments of relative speed (see also Vicente [657]).²⁰

There may be more overlap between Hutchins’ analysis of precomputations and CoSTH. This would not be surprising since Hutchins’ analysis of precomputations is really a mixture of (1) an analysis of the power of artifacts to beneficially alter cognition, and (2) the importance of the pre-construction of such cognitive artifacts. This mixture makes it somewhat challenging to explicitly map his account to CoSTH.

Backtalk

In Schön’s work on design thinking, he noted that designers like to manipulate external materials in order to elicit what he termed “backtalk”. Backtalk is part of the “conversation with materials” such as sketches, models, and simulations [566]. Interpreted within the world of software development, this conversation might be made with prototypes, or possibly with various sketches or diagrams [544]. Schön’s analogy to conversation is apt because real conversation is not a smooth transmission of ideas from one party to another. It is an active process involving breakdowns in communications, and actions to repair these breakdowns. Breakdowns are instances when the conversation cannot be made sense of by one of the parties, so that feedback is elicited. This ability of the parties to provide feedback is enormously important because it frees speakers from making everything clear at the start, and thus extensively planning their speech [235]. Backtalk in conversation with materials means that designers can set forth to experiment

²⁰Hutchins also gives an example of a more encompassing cognitive restructuring in the differences between Micronesian and Western methods of navigation [320]. However the relation of this restructuring to precomputation is much more complex, having more to do with learning at the cultural level.

with materials; they provide feedback that can both help correct invalid reasoning, and provide pause to *reflect* upon assumptions, or upon the chosen course of action.

This rough account of backtalk can be analyzed using CoSTH as follows. In order for backtalk to work in the first place, some aspects of the solution must be externalized (D/D/PG/current). Then backtalk can happen in several ways. One particular way is that the evolving external solution cues the designer's rapid recognitions of unexpected problems, opportunities, or constraint violations (S/S/perceptual, S/S/recognition, D/D/PB/constraint, D/D/PB/operations). Another possible way is that the designer can "play" with the externalized solution rather than consciously perform deductive reasoning. This play may involve skilled manipulations (S/S/SA), which can further lower the cost of external manipulation.

6.6 Comparison to Related Work

The main contents of this chapter are the enumeration of a family of support theories, and an analysis of some of the ways in which they combine. These theories are organized into a refinement hierarchy in which RODS provides a core set of support explanation principles, and HASTI supplies the ways of applying these core principles (although some elaboration of HASTI was needed). This presentation has drawn extensively from many different sources for theories, such as the works by Hutchins [320], Larkin *et al.* [374,375], and Zhang *et al.* [727]. Because this chapter tries to synthesize these, it is not especially illuminating to compare and contrast the integrative account with the many sources it draws from. A more interesting comparison is with other integrative explanations of cognitive support.

Many, many other works have reviewed various forms of cognitive support. It would be cumbersome and rather uninformative to attempt to exhaustively compare them all. But among these, there are a select few that (1) propose comparable theoretically-motivated explanations, or (2) provide some sort of integration of different cognitive support concepts. A focused subset of these exceptions is listed briefly here:

Neuwirth & Kaufer [441] (1989)

These authors begin with an internal/external representation dichotomy that is similar to mine. They utilize a domain task analysis for hypertext environments. They touch on many explanations of cognitive support that are in common with this work, including perceptual substitution and redistribution of process and agenda. However, they pursue a rather different tactic for enumerating advantages of external representations. Like I do here, they begin with a cognitive model and try to determine how external features affect the model. But their model is tightly focused on specific aspects of cognition; it contains only four fundamental cognitive processes (encoding, match and execution, control of cognition) and only implicitly considers how control and data are redistributed. Although their approach is distinct, many of their explanations for support are very compatible with CoSTH.

Tweedie [645] (1995)

Tweedie's account of the role of artifacts in problem solving provides an interesting contrast to the

development presented here. There are certainly much in her approach that is common or consistent with the account of support presented here. For example her account implies (1) an SRK-like decomposition of cognition types into skill/knowledge modes, and (2) some similarities in decomposing types of external representation (like the inclusion of R/PB/operations). The interesting contrast concerns Tweedie's 8-fold decomposition of problem solving according to the three orthogonal dimensions of: situatedness (planned vs. opportunistic), cognition driver (conceptually motivated vs. perceptually motivated), and action mechanism (automatic vs. intentional). This decomposition provides an alternate way of understanding the context of cognitive support in that the supportive nature of artifacts are placed in relation to the mode of problem solving being engaged in. This decomposition could conceivably provide a sixth type of processing redistribution decomposition for Section 6.2.2.

Scaife & Rogers [562] (1996)

These authors try to provide an integrated account of why external representations are helpful. They utilize some of the same references as I do here. The core part of their analysis is the decomposition of cognitive benefits into what are called *computational offloading*, *re-representation* and *graphical constraining*. These three have close analogues in this work in the form of redistribution, cognitive restructuring, and problem constraint redistribution, respectively.

Narayanan & Hubscher [433] (1998)

Analyzes the "cognitive utility" of visual languages; they touch on situated reasoning and visual inferencing, and give a brief account of what conditions are required for inferences on representing words can substitute for reasoning about the represented world.

Petre, Blackwell & Green [508] (1998)

This paper contains an eclectic *mélange* of questions regarding the effectiveness of visualization in programming. Its presentation of support is hardly integrated, but in its winding course it briefly touches on many aspects of cognitive support. Some of these aspects are addressed in this chapter, others are not. They touch on external perceptual effects (S/S/perceptual), display-based planning (R/D/plans) and re-representation (CR/encoding). They also discuss more complicated concepts of support that are hard to categorize in this framework, like the advantages of providing a "secondary notation" in which important relationships are kept implicit.

Several more papers could easily be added (e.g., Ware [687], Cheng [122], Sharples [574], Orhun [480], Perry [502], Kirlik *et al.* [351], Robbins *et al.* [543]), but the ones listed best qualify as focused, theory-backed integrations. There is certainly overlap between these accounts and the theories presented in this chapter. However none of these other works integrate the number of diverse forms of support presented here, and none provide the same type of theoretically-inspired decomposition.

Two other bodies of work bear a special relationship to the support theories of this chapter. One is the "resources model" work by the group of Wright, Fields and Harrison [210, 211, 290, 718, 719]. The other is the cognitive design work by Rasmussen and colleagues [526–528, 530, 657]. The latter was essentially merged into my RODS and HASTI frameworks, so it forms a core part of the analysis. The former bears

a more complicated relationship to this work. The overall support framework presented here had its genesis before I became aware of the resources model of Wright *et al.*. The fact that it bears remarkable similarity to their work, yet it was developed quite independently, seems important enough to mention. In ways, the similarities should not be surprising since there are clearly a number of significant overlaps in the literature that we have both drawn upon. The result is that their work is highly compatible with this one; many of the observations made by Wright *et al.* have their analogues in CoSTH. Probably the most significant difference is that, in this work, RODS provides a more thorough computational account of support, and HASTI provides a more rationally motivated decomposition of what Wright *et al.* term “resources”. Thus the main divergence between the two is that this work further clarifies the theoretical foundations for analyzing distributed resources in cognition.

Other than the unique theoretical framework for integrating cognitive support, there is one other notable difference between this work and that of Rasmussen *et al.* and Wright *et al.*: the way that design rationalization is treated. Here the support theories are completely separated from issues of “good” or “bad” design. Support is cast as an accounting of the computational benefits of artifact features. The support involved in a system may or may not make a beneficial impact on performance. For instance, the advantages of offloading working memory may be completely offset by the costs of the offloading. In comparable work the difference between support and good design is muted. For instance, in the cognitive engineering work of Rasmussen *et al.*, the notion of support is clearly enunciated as a substitution of modes of behaviour, but the various means of substituting behaviour are tied to just a few design concepts like the so-called “abstraction hierarchy” [528].

Another potentially interesting comparison of CoSTH is to other theory-organized collections of design ideas for software development tools. Unfortunately, there are not too many of these to compare with. For the most part other surveys have organized tools by their features or products (see Section 2.2.1). The main points for comparison are the surveys by Storey *et al.* [619] and von Mayrhauser *et al.* [668,669]. These are both surveys motivated by theory-backed analyses of cognitive issues in software comprehension and reverse engineering.

The survey of Storey *et al.* hierarchically clusters tools according to a “hierarchy of cognitive issues” relating to their design. Like this chapter, the survey employs exemplar tools to illustrate each of the leaves of the hierarchy. This hierarchy is probably best described as being complementary to CoSTH. The main distinction between the two is the type of nodes in the hierarchies, especially the leaves. Storey *et al.* essentially provides a decomposition according to *design goals* (see Section 7.1.3). Each design goal might be potentially satisfied using a variety of different classes of cognitive support. Thus the leaves in CoSTH cluster together examples using similar cognitive principles, whereas the hierarchy of Storey *et al.* clusters together examples of tools satisfying broad design issues. Logically, the hierarchies therefore belong to different categories, although because the design goals often relate to the provision of cognitive support, the relationship is complicated.

The surveys of von Mayrhauser *et al.* are quite distinct from CoSTH also. Their work started with an analysis of the tasks involved in program comprehension (e.g., “gain high-level view”, “determine program segment to examine”, etc.). They generated these by starting with a model of program comprehension, and then using these to “reverse engineer” a set of common tasks from observations of the

activities of actual maintainers. From these tasks they then determined what information could be needed to solve them. Given these needs, they made many suggestions as to what sorts of tools or tool features could provide the information. The result is a reasonably fine-grained analysis of different uses for tools (29 total), and different classes of tools (22 total). The classes of tools usually abstractly specify tool *functionalities* (e.g., “smart differencing system”) rather than list specific tool *features* (e.g., display formats, interaction methods, etc.). Each of these functionalities could be implemented using a variety of cognitive supports. Thus their survey is also in a different category from CoSTH. An interesting fact emerges if one interprets their survey with CoSTH in hand. CoSTH seems to indicate that von Mayrhauser *et al.* tacitly used craft knowledge about cognitive support in order to generate their particular list of tools. This fact was mentioned briefly in Section 2.1.1. The way that they deduced the tool categories from the task demands is not explicitly acknowledged. Yet, it is fairly easy to interpret examples of D/D (e.g., “history of browsed locations”), D/P (e.g., “function counts”), and various virtual memory operations (e.g., “keyword search”). Thus their categorization is not based on identifying different examples of cognitive support implementations, but on associating certain tool types with their uses.

6.7 Limitations

CoSTH is a limited theoretical framework in the sense that it is preliminary and approximated. It made use of limited models of cognition in the form of HASTI, and it employed limited ways of analyzing tasks. Although it can offer basic explanations for a wide range of cognitive support types, the explanations derived from it are not quantitative or precise. In addition, even though many things are successfully integrated, if one goes through the phenomena described in Chapter 3, then one is likely to find several instances of support whose spirit is not completely captured by CoSTH. For instance, one could point to Hutchins’ idea of precomputation and note that nothing was specifically mentioned regarding the role of preparation. CoSTH is not fine-grained, not detailed, and not comprehensive.

Let us agree to put aside questions about these sorts of limitations for the time being. One can hardly fault *any* theory for failing to be simultaneously fine-grained, detailed, and comprehensive. Moreover the entire point of a broad-brush applied theory is to approximate usefully, and to provide good coverage of salient aspects rather than to fill in fine details, or to be entirely comprehensive. It is therefore more important to step back a bit to try to place CoSTH within the greater context. Specifically, how does CoSTH address main problems we wish to address—the problems of understanding tool usefulness, and of building useful tools? The most important criterion for an applied theory is if it can be usefully applied, so the limitations of the theory that impact application need to be discussed. These issue might be phrased as the “bottom-line” question: “what will I need to really *use* it?” In this regard there are four important limitations that affect this “bottom line”: (1) the *negative consequences* of tools are ignored, (2) the full implications of cognitive support are not traced from support to eventual performance, (3) human learning and training is ignored, as is the issue of tool adoption, and (4) little help is available for understanding partial, coordinated human–tool collaboration.

No negative consequences

The support theories are not full design rationales: they contain no tradeoff information. All design involves tradeoffs. For instance, the aid offered by an automation may be more trouble than it is worth [347], and an external memory may be too costly to be worth storing knowledge into [165,567]. Thus although CoSTH could be employed for arguing a good or bad design by saying what is or is not supported, it has no grasp whatsoever on the tradeoffs. On the one hand, being able to separate the mechanisms of support from the issue of negative consequences is good. It affords some insulation from the many complications associated with implementing good and useful tools. As I pointed out in Chapter 2, this separation is critical if one is to evaluate or test *design ideas*, which may or may not be implemented in a polished tool. On the other hand, this is bad news for designers. Linking tools to their negative and positive consequences is important to them [110,269,272]. They need it during tradeoff analysis. Although I have not attempted to add tradeoff information, it seems likely that CoSTH could be augmented to provide much tradeoff information, and it is certainly a possible direction for future work.

Full implications are not traced

What difference does it make if a plan is offloaded? The possibilities are vast. Perhaps the memory freed up by the offloading could be used for better planning, resulting in fewer cases where backtracking is required? This in turn could reduce the need to easily undo operations. Or maybe the user's mental workload is simply reduced without otherwise affecting the user's productivity? Maybe bigger problems could be tackled? The implications seem to spiral out unbounded, cutting a swath through psychological, technological, and organizational aspects. The same holds for any of the nodes in the CoSTH. Tracing these implications will require more than RODS, HASTI, and CoSTH frameworks. It would most likely be a daunting task that would appear to need cognitive models with much more detail. A hint at some of the requirements can be detected by examining the work of Stacey *et al.* [615] and Freed *et al.* [229]. The details of these papers are not of concern here—only what these authors needed to do to derive design implications for supportive technologies.

Stacey *et al.* needed an analysis of the ways that *cognitive biases and limitations* showed up in task performances. They identified a problem that product designers (their target user population) had: they usually had a bias for following *habitual paths* based on what they have learned from past experiences. This bias leads them to sometimes use solutions that were less appropriate. This is a variation of the old saw “to a baby with a hammer, everything looks like a nail”. To understand how to counteract undesirable biases, a list of them would need to be generated, and the effects of various forms of support would need to be established. Unfortunately, many of these biases would be based on personal motivation, or on training and organizational or sociocultural setting. This makes them all the more difficult to model. And without a model for biases in software development it would be difficult to trace how the reengineering provided by cognitive support would impact them. Fortunately there may be some promise in simply building a list of cognitive biases in software development (see Stacy *et al.* [691] for a start). After building the list one might be able to employ empirical legwork to determine how cognitive support affects them. In this, at least, CoSTH makes such an empirical endeavor possible by providing a list of experimental

variables (i.e., the support types) to consider.

Freed *et al.* helpfully suggested that performance problems need to be understood in the context of coping strategies that people have for dealing with cognitive limitations (see Sections 3.1.1). As Freed *et al.* say:

Human resource limits are much easier to identify and represent in a model than are the subtle and varied strategies people use to cope with those limits. ... People cope with memory limitations by maintenance rehearsal, writing notes to themselves, setting alarms, and other methods. [229]

But cognitive support counteracts cognitive limitations. As a result, before the implications of cognitive support could be understood, the ways in which coping strategies are relieved by it must be appreciated. Unfortunately our knowledge about coping strategies in software development is rather pitiful [41]. Nonetheless, there are a few bright spots, such as the work by Davies [165] on coders' strategic use of external memory during coding, and that of Bowdidge *et al.* [64] on developers' bookkeeping strategies for coping with high demands on working memory during maintenance.

Learning, training, and adoption

Learning and training play a significant role in determining which strategies people employ, and which skills they develop. CoSTH does not specify what these should be. It only presumes that learning and training have produced the appropriate capability so that the support can be effective. This seems appropriate anyway, since the definition of support should probably be independent of training: designers want to know if an artifact is supportive, even if some particular class of users will not be able to use it effectively. But it is important for design analysis to understand the importance of learning. For example, consider exploratory learning (sometimes known as “learning while doing”). A well-known facet of exploratory learning is that users “asymptote to mediocrity” [109], that is, as they become more competent at an interface the need to learn better methods abates in proportion, and so they never become maximally skilled. In other words, learning how to use tools is an instance of classic negative feedback: learning is squelched by competence. This means that even if one provides a supportive mechanisms, users may never learn to use them effectively [48]. But tool designers will want to know when and how well the support can be used. The upshot is that cognitive support only tells part of the story the tool designer wishes to hear. CoSTH is limited to explaining the support only if it is actually used; it does not specify the conditions upon which the support is actually rendered.

On a related note is the issue of tool uptake or adoption [424]. There is a cost to learning a new tool. Even if there is potential for an eventual improvement in performance, developers are loath to adopt them. This is nowhere more apparent than how developers cling onto their favorite editors with near religious fervour. The implication is that knowledge of cognitive support is not enough for effective design—one may also need to account for learning costs, or other hindrances to tool adoption.

Understanding collaborative human–tool interaction

A core part of CoSTH is the distribution framework: in any DC system *distribution* is the key element. Without it, it is useless to consider the other support principles. In this review, a conscious decision was

made to try to provide examples that highlight each individual form of cognitive support. This emphasis has resulted in a review that did not highlight well the possibilities of *partial* distribution of data and processing. CoSTH describes many different types of data and process distribution, but it does not elaborate well on how to develop partially redistributed data and processing. For example, in reverse engineering it is for the most part acknowledged in that fully automated clustering of real legacy software is never likely produce satisfactory results, at least, not in the immediate future. Yet it is certainly not appropriate to simply wait until perfect automated clustering mechanisms appear. Thus we are necessarily left to consider how to *partially* automate clustering in ways that humans and tools work well together. Currently the ideas being pursued to that end have been more simplistic than they might need to be. Typical suggestions involve simple human–tool coordination schemes such as a “propose–validate” style of dialog, in which humans pick and choose amongst automatically generated candidates [359]. This method places human judgment at the periphery of the clustering process. In other words, the processing does not currently use tight collaboration between human and software agents.

This type of cooperative processing [290,320] of shared, partially distributed [268,727] data is actually a core fixture of the DC viewpoint. Sadly, CoSTH gives too little insight into developing it. The main culprit is the weakness in characterizing types of processing distribution. What is needed is a more complex decomposition that enumerates classes of partial distributions. Consider an analogue: a automatic parallelizing, distributing compiler. The compiler takes a description work process (a serial program) and analyzes it for ways in which to distribute the processing amongst different elements. Often times such compilers look for clichéd patterns that they know how to distribute. In a similar way, a generative cognitive support theory for distribution would be able to take a description of a problem and say how to split it up into joint human and computer processing. This issue is revisited in Section 7.2.2.

6.8 Summary, Commentary, and Implications

Theories cumulate. They are refined and reformulated, corrected and expanded. ... Working with theories is not like skeet shooting—where theories are lofted up and bang, they are shot down with a falsification bullet, and that’s the end of that theory. Theories are more like graduate students—once admitted you try hard to avoid flunking them out, it being much better for them and for the world if they can become long-term contributors to society. Theories are things to be nurtured and changed and built up. One is happy to change them to make them more useful.

– Allen Newell, “Unified Theories of Cognition” [446], pg. 14.

This chapter defined a space of generic cognitive support theories and provided examples of support that inhabit the space. The space was generated in part by applying the support principles of RODS to the model features of HASTI. RODS and HASTI were well-matched partners in producing these support theories: RODS provided computational explanations of the different types of cognitive advantage that artifacts provide, and HASTI provided a breakdown of cognitive features to apply these computational

explanations to. Since RODS is grounded in theoretical computing science and HASTI in prior work in cognitive science, the result is a principled, integrative, and theoretically motivated decomposition of support types. Since many details were suppressed by strategic abstractions, the resulting theories are broadly applicable, that is, they are reusable. Some of these theories have quite a lot of backing from basic sciences, while some of them are quite preliminary. Lack of solid evidence is not a knock on the framework, however, since the maturity of the basic sciences on which it draws are themselves rather shaky and full of holes [508,562].

The aim of the effort was to bring existing theory together so that tacit and folk knowledge can be replaced with whatever science knowledge is currently available. And the CoSTH is certainly broadly integrative in this regard. At the beginning of Chapter 3, I asked the reader to consider a paper and pen, a typechecking compiler, and a search tool like `grep`. Although the cognitive supports exemplified may have seemed to be quite distinct, this chapter has demonstrated that they may be rationally compared within the same theoretical framework. Thus, even though the work is preliminary, one thing should be clear: the RODS framework in combination with the structuring mechanisms of HASTI make a convincing case for a good starting theoretical framework for cognitive support in software development.

This particular exposition of cognitive support theories has advantages other than relative completeness, generality, and theoretical pedigree. First, the decomposition of support types into three orthogonal groups appears to be immensely helpful. There is considerable parsimony in providing a principled decomposition of the many complicated combinations of support. The decomposition simultaneously demonstrates that a few principles common to all of computing carry over, to a great extent, to explaining cognitive support. Applications of these computing concepts to cognitive support have to be attenuated to the issues of human psychology, of course, but the principles do apply: cognitive support is computational advantage. Furthermore, note that there seems to be some utility in gathering together different support examples and implementations in the coherent organizing context that CoSTH provides. In particular, the examples populate a design space mapped out by CoSTH. This suggests that a more thorough survey of how the CoSTH types have been applied to software development would yield a useful cookbook of design ideas. As it is, this chapter presents a rudimentary cookbook. The breadth and variety of this cookbook is an early testament to the thoroughness of the framework, and hints at the potentially broad applicability of the resulting support theories.

In concluding this chapter, it may be appropriate to comment on some of the implications which CoSTH makes regarding the research questions covered in Chapter 2. First let us consider the fact that this chapter has been a close mixture of theory and examples to which it applies. The examples were portrayed as expository devices (see Figure 6.2). In choosing the examples an attempt was made to use ones that are likely to be familiar to SE researchers—although for scholarly purposes, the selections were also chosen to cite examples that boast publications that can buttress the theories with argument or evidence. I certainly hope many of these are examples that strike the reader as being familiar and even a little mundane and well known—as necessary knowledge for anyone skilled in the art of creating SE tools. If so, it is excellent news. It would strongly indicate that the support theories capture some of the important and tacit tool building knowledge from the field. Being able to transform the enormous quantity of craft knowledge into explicit knowledge was singled out as a critical long term goal for the evolution of the

field (see Figure 2.1). Synthesizing a broad, widely applicable theoretical framework for cognitive support is one accomplishment—building one that brings principled explanations to heretofore tacit understanding is an excellent feather in the cap. CoSTH provides a vigorous demonstration that the search for such theoretical foundations is not vacuous. And if the examples *are not* familiar, then all the more reason to appreciate their exposition.

This last point brings up a final issue for comment. Chapter 2 argued that many forms of empiricism that we engage in are limited because of their lack of theoretical basis. In particular we tend to engage in what I called “simplistic comparison” experimentation. We pit tool against tool with little theoretical basis for rational comparison. My sincere wish is that we as a community can put to rest the unreasonable hope that we can continue to pursue this course. While we should not abandon these experiments (they can be extraordinarily useful) I hope that it is clear from this chapter that considerable theoretical apparatus can be brought to bear on the problem of understanding the merits of software development tools. For one thing, CoSTH indicates that principled tool analysis is a viable precursor to experimentation: there can be a principled basis for comparing the merits of different types of tools, a way of categorizing the support-relevant features of tools, and a reasonably fine grained basis for differentiating among tool variants. For another thing, within each example of this chapter lurks a plausible, theoretically motivated, but unverified hypothesis about the supportive nature of artifacts. Any empiricist looking for a problem to examine need only scan down Tables 6.2, 6.3, and 6.4 to find dozens of support issues that are nearly completely unexplored in the realm of SE. Potential PhD topics lurk in every category. CoSTH is not only a door leading to codified tacit knowledge, it is a window onto the uncharted future of a potentially central aspect of SE research.

Chapter 7

Building Theories Fit For Design

Historically and traditionally, it has been the task of the science disciplines to teach about natural things: how they are and how they work. It has been the task of engineering schools to teach about artificial things: how to make artifacts that have desired properties and how to design.

– Herbert A. Simon, “The Sciences of the Artificial” [594], pg. 111.

*D*esign is about constructing things that have properties the designer desires. At the most abstract level, designers take as input an existing world, decide what they dislike about it, and then determine how to create artifacts to fix it. Design and engineering are therefore constructive, or *synthetic* activities. Their basic goals differ sharply from the goals of the sciences. Herbert Simon, in his seminal work “The Sciences of the Artificial” [594] describes the distinction crisply:

We speak of engineering as concerned with “synthesis,” while science is concerned with “analysis.” Synthetic or artificial objects—and more specifically prospective artificial objects having desired properties—are the central objective of engineering activity and skill. The engineer, and more generally the designer, is concerned with how things *ought* to be—how they ought to be in order to *attain goals*, and to *function*. [594, pg. 4] (emphasis original)

Because analysis is an important part of design activities, it is probably better to contrast synthesis with *explanation* or *evaluation* rather than *analysis* (see e.g., Long *et al.* [392] for a similar dichotomy). The differences between “explanation” and “synthesis” activities are further revealed by examining what they consider *fixed* versus *dependent* variables. Kirlik understood the essential differences:

A standard modeling approach in cognitive psychology is to hold a task environment relatively fixed and to create a description of the cognitive activities underlying a person’s behavior in that environment. The designer, on the other hand, is faced with the reverse challenge of creating an environment to elicit a desired behavior ... In problem solving terms, the solution space for the scientist is a set of plausible cognitive theories, whereas the solution space for the designer is a set of technologically feasible environments. [348, pg. 71]

The simple corollary to this argument is that designers do not use *explanation* oriented theories to do design, except during evaluations before or after synthetic steps. Designers want artifacts, not explanations. *How then do designers know what to design?* What resources are used in the synthesis step?

One simple answer is that designers utilize their craft knowledge and experience. If this were the only conceivable answer there would be much less to talk about regarding the development of theories for design. The more interesting answer to the question of synthesis is that designers might be able to use *design theories*, that is, theories that provide to the theory wielder a description of a design solution rather than an explanation of some phenomenon. Note that the explanation–synthesis distinction is *not* to be identified with the commonly cited basic–applied dichotomy [348]. It is conceivable to have both basic and applied design theories. Instead, the difference is in the *product* of the theories: one produces explanations, the other produces (suggestions for) artifacts. This chapter is concerned with (applied) design theories for cognitive support.

The topic of design theories for cognitive support is particularly relevant to SE. As I argued in Chapter 2.1.2, it does the field as a whole a great disservice to keep our knowledge of cognitive support tacit. To do so means to suffer a discipline that is craft on one of its most important dimensions. Mind you, the status of SE theory is certainly not unique to SE: HCI design is predominantly craft-like and atheoretical. Even though there certainly do exist many theories in HCI, these are not particularly suited to design. Most are instead oriented towards evaluation. Long [390] invented the terms “fit-for-design-purpose” to indicate knowledge or theories that are well suited for synthesis, and “fit-for-understanding-purpose” for knowledge or theories that are suitable primarily for understanding or explanation. Let us agree to drop the “-purpose” ending and just call the two categories “fit for design” and “fit for understanding”. It is fairly easy to argue that most work in cognitive science and HCI has been primarily fit for understanding and *not* fit for design. This state of affairs should not be considered surprising. After all, HCI has leaned heavily on cognitive science, psychology, and sociology for theories and the vast majority of the theories from such fields are distinctly explanation-oriented, not design-oriented.

In this chapter, I wish to make a strong case that the theories and models of the previous chapters can be made, with a little work, to be fit for design. Chapter 6 has already made a good initial argument that they are fit for understanding, and Chapters 8 and 9 will reinforce this argument. RODS, HASTI, and CoSTH were actually developed with the needs of design in mind, but even so there is still work to be done to make them especially fit for design. Converting explanation theories into theories fit for design is neither trivial nor automatic, and these facts are seriously under-appreciated. Design theories are tools, and like all tools they need to be carefully crafted with the characteristics and needs of their users in mind. Since there are many different design contexts there is inevitably a need to reshape the same basic theoretical resources into many different forms. In Barnard’s terminology [28], these different design-specific forms would be called “application representations” because they are the forms that basic theory takes when being applied during synthesis. RODS, HASTI and CoSTH are good starting points but they are *only* starting points.

This chapter is structured as follows. First Section 7.1 describes reasons for why the theories of HCI and related disciplines are so notoriously unfit for design. An argument is put forward that too little attention has been paid to what *type* of knowledge the design-oriented theories should represent. To this

end, a taxonomy of design knowledge is proposed. The taxonomy is organized by the type of reasoning it enables. This taxonomy is used to argue that most current HCI theorizing has failed to compile knowledge that enables what may be termed “*forward reasoning*” about “*positive consequences*” of artifacts. Next, Section 7.2 considers several different ways of converting the support theories of previous chapters into forms that are more fit for design. Finally the chapter concludes with a brief summary.

7.1 The Trouble with Theory

The basic assumption of researchers who study programmers is that by understanding how and why programmers do a task, we will be in a better position to make prescriptions that can aid programmers in their tasks.

– Soloway and Iyengar, Preface to
the Proceedings of the First Empirical Studies of Programmers [741], 1986.

There is a belief, quite widely held, that good science invariably leads to usable design knowledge. An argument commonly used to bolster this conviction is that knowing more about the design context (pick any: the user, user psychology, work domain, work practices, etc.) will naturally enable better design. This belief is certainly held by many researchers in SE and HCI. For instance, when motivating the development of a model of software comprehension, Vans argued that a “better grasp of how programmers understand code and what is most efficient and effective should lead to a variety of improvements: better tools, better maintenance guidelines and processes, and documentation that supports the cognitive process.” [654, pg. 4]. The popular expectation is that good theories inevitably find application.

But this is really a myth, or, at least, it seems to be dishearteningly indistinguishable from a myth. For there is a readily noticeable lack of useful theory that is applicable to real HCI situations in general, and even fewer theories exist that can be called “fit for design”. This scarcity persists despite decades of quality research working to erase it. The noticeable lack of usable theory has created something of a running debate within the HCI community. In 1989, the “Kittle House Workshop” was called to rally together HCI theoreticians so that they could reflect upon the notorious shortcomings of theory in HCI [101]. Barnard summed up many of the problems nicely:

The difficulties of generating a science base for HCI that will support effective ... design are undeniably real. Many of the strategic problems theoretical approaches must overcome have now been thoroughly aired. The life cycle of theoretical enquiry and synthesis typically postdates the life cycle of products with which it seeks to deal; the theories are too low level; they are of restricted scope; as abstractions from behavior they fail to deal with the real context of work and they fail to accommodate fine details of implementations and interactions that may crucially influence the use of a system ... Similarly, although theory may predict significant effects and receive empirical support, those effects may be of marginal practical consequence in the context of a broader interaction or less important than effects not specifically addressed... [28, pg. 107]

In the decade that has elapsed since the workshop, agonizingly little has changed which could dispel Barnard's disparaging remarks. Is good high-level, design-oriented, relevant, and applicable cognition-related theory in HCI impossible?

Some, like Landauer [370, 371] and Carroll [102, 104, 106] have argued that the application of explicit theory to design in HCI will always be marginal, limited, or indirect. However, rather than giving up on theory, a number of authors have concluded instead that not enough attention has been paid to what designers really need from theories. Kirlik [348] articulated this stance very well in his own field of cognitive engineering:

Although it may be fashionable within the cognitive engineering community to bemoan how little guidance modern psychology provides the designer, the psychological nature of the design product is inescapable. The correct response to the current and unfortunate lack of applicable psychological research is not to attempt to do psychology-free design (because this is impossible—the design will not be apsychological but instead reflect the designer's folk psychological theory), but rather to ask what kinds of psychological models are needed to support cognitive engineering, and to begin the long range empirical and theoretical work necessary to realize them. [348, pg. 72]

Besides Kirlik, a number of others others, including Green [257, 259, 263, 264, 268, 270, 272], Rasmussen [531] and many former members of the AMODEUS-2 project¹ like Barnard [28, 30], Buckingham Shum *et al.* [43, 57, 84], and Blandford [57, 58] have reached similar conclusions. They concluded that useful and usable theory *is* possible, but what has been missing is appropriate attention to the designer's needs. Plus, perhaps, too little attention to what it takes to transfer theory into practice [57, 263]. The trouble with computers—as Landauer [371] argued at length in a book by that title—is that we have not striven adequately to make them usable and useful. In an analogous way, the trouble with theory, according to the above authors, is that we have not striven to produce usable theoretical products. Their prescription is to begin to better understand the needs of designers, to understand the design context, and to understand what forms the theory should take to be readily taken up by designers. Only then will theory become fit for design.

I am highly sympathetic to this theory-as-designed-artifact point of view. However I do wish to add one small but important point that seems never have been well articulated. The point has to do with what it takes for design-oriented theory to be *useful*. Obviously all the critical usability problems like learnability must be addressed, but little has been written about what designers would find most useful in a theory. I contend that designers are in serious need of design theories that allow *forward reasoning* about *positive consequences* of artifacts. Because it is such a mouthful, that type of reasoning is abbreviated FP-reasoning. By “forward reasoning” I mean using theory to reason from designer goal towards conceivable artifact, rather than from an existing or envisioned artifact back to explanations or predictions. By “positive consequences” I mean the facets of artifacts that make them desirable, as opposed to facets that negatively impact usability. I distinguish *design theory* from other theories that might be fit for design by requiring that design theories enable FP-reasoning; they are *FP-theories*. I would argue that FP-theory is the type of

¹The AMODEUS-2 project was a large three-year European project to investigate methods for modeling and designing in HCI (see e.g., Bellotti *et al.* [43]).

knowledge that is most useful in a direct attack at researching and designing good tools.

The important distinction between forward reasoning and backward reasoning, and between reasoning about positive and negative consequences seems to be poorly understood. Even the theoreticians who try to design usable HCI theory seem to be frequently unaware of the important distinction. This may be one of the reasons for why these sorts of theories are not only poorly developed, but for why so few researchers appear to be aware of the possibility of developing them. It is important therefore to describe the needs of designers and then explain the possible roles of theory in design.

This subsection consequently unfolds as follows. First the problems faced by designers are examined in order to understand the requirements that need to be met. Two key needs are identified in this analysis: the need for survey knowledge of design possibilities, and the need for resources that allows FP-reasoning. The former is argued in Section 7.1.1 by first casting the problem of design in terms of evolutionary algorithms, specifically as a search in what is called a “fitness landscape”. Posing design in this way (very helpfully) casts what we currently do in HCI in a rather unflattering light. This makes it clear that it is *survey knowledge* of the fitness landscape that is most immediately needed by designers. Introducing the fitness landscape also makes it relatively simple to properly highlight the importance of *synthetic* reasoning in design. In particular, the difficulty of synthesis is portrayed as a *gulf of synthesis* in Section 7.1.2. Analyzing what the designer must do to cross this gulf leads to the conclusion that the most pressing needs is for knowledge enabling FP-reasoning. Along the way it will be shown that the most common types of theoretical work in HCI simply do not address these needs. The implications for building theories fit for design is summarized in Section 7.1.4.

7.1.1 Navigating the Fitness Landscape

Here followed a very long and untranslatable digression about the different races and families of the then existing machines. The writer attempted to support his theory by pointing out the similarities existing between many machines of a widely different character, which served to show descent from a common ancestor. He divided machines into their genera, subgenera, species, varieties, subvarieties, and so forth, He proved the existence of connecting links between machines that seemed to have very little in common, and showed that many more such links had existed, but had now perished. He pointed out tendencies to reversion, and the presence of rudimentary organs which existed in many machines feebly developed and perfectly useless, yet serving to mark descent from an ancestor to whom the function was actually useful.

– Samuel Butler, “Erewhon” [88], pg. 192, (1872).

With the application of sufficient persistence, it takes astonishingly little intelligence or knowledge to be successful in building new and useful tools. In fact, good tools can be constructed entirely by accident and without any explicit intent. This lesson was first taught over a century ago by Darwin in the context of creating biological organisms. But the lesson need not apply only to biological entities, it can apply to artifacts such as mechanical or computer tools. As the quotation at the head of this section indicates,

Samuel Butler—a later contemporary and fierce critic of Darwin—was one of the first authors to apply the concepts of Darwinian evolution to artifactual evolution. But he was certainly not alone.² The formula for generating good tools is simple and general.³ Only three things are needed: (1) a way of preserving good existing tools and forgetting the bad, (2) a way of generating new tools based on the pre-existing ones—random changes will do, and (3) a way of evaluating each tool to determine how well it works so the bad ones can be detected and eliminated. In the lexicon of evolutionary algorithms, this algorithm is cast in terms of “reproduction”, “mutation and recombination”, and “selection”. To make good tools, just iteratively run the generate-and-test process, accumulate the good tools, and throw away the rest. This is the basic Darwinian method for building good tools without even knowing how.

It is quite easy to see that the above formula for building tools is too good of an approximation of how we currently build successful tools. The connection was made briefly in Section 2.1.2, and the mapping is straightforward. Much of our knowledge about good tool design is tacitly and silently encoded within existing tools. Successful tools are emulated. When designing new tools one rarely, if ever, starts from scratch, but builds from prior designs [62, 102, 177, 599]. We design by “hillclimbing from predecessor artifacts” [598]. Because we know so little about what makes tools great, our changes to existing tools tend to approach random modifications more frequently than we would like. To deal with this fact, we have our own versions of natural selection. The single most successful idea in HCI is user testing in combination with iterative design [251]. User testing weeds out poor designs. We cull the “unfit”.

In natural selection, “fitness” and “adaptation” are concepts used to explain the success of organisms and species. The precise definition of “fitness” is entirely dependent upon the ecosystem the organism lives in. This concept of fitness also applies to tools, that is, tools need to be ecologically fit. Traditionally one of the main fitness criteria for tools has been the performance that the tool makes possible (user productivity, product quality, etc.). However clearly any performance measure of interest can conceivably be one indicator of fitness (see Section 3.2.4). So, for example, the degree of cognitive support can be an indicator of fitness. There obviously cannot be any *globally* fit tool since fitness depends upon local ecology. Since so many factors affect the overall fitness, it can be expected that variations on any of the factors will affect the overall fitness of an artifact. So, for example, it is widely known that the fitness of a representation is task-dependent (see e.g., Casner [116], Green *et al.* [273] Peterson [505], or Mulholland [422]) and user-dependent (see e.g., Good [250]). Moreover, because of the so-called task-artifact cycle [106], the work ecology will tend to change in response to the introduction of new tools. This tool-ecology change process is somewhat akin to predator-prey co-evolution. But the notion of fitness does not need a globally fixed definition of ecology of use. In general, the analyst investigates a tool’s fitness with regard to some combination of performance measures, characterizations of user population, and situations of use.⁴ The issue of fitness is simply given in relation to some chosen ecology.

²See Basalla’s “The Evolution of Technology” [32] for a good overview of various attempts to apply evolutionary theory to mechanical technology.

³I am presenting the simplest form: there are many good reasons for complicating this most basic formula but for my purposes here these complications are only distracting.

⁴The parameter optimization style of modeling design search uses a different but analogous vocabulary. For instance, a tool’s overall fitness is defined by the analyst in terms of a “utility function” and the manipulable features are called “command variables” (see e.g., Simon [594, pg. 117]).

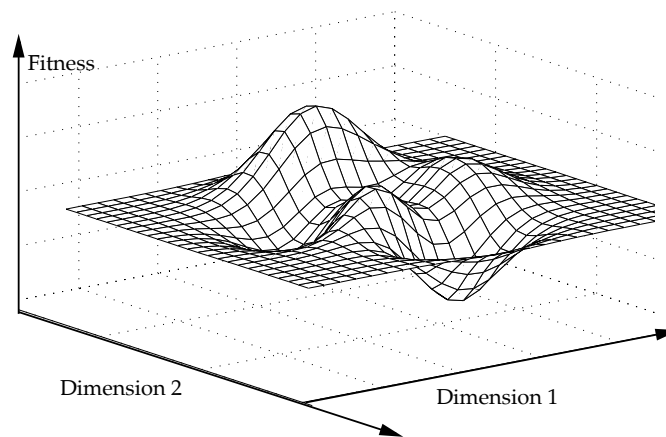


Figure 7.1: A “fitness landscape” visualization

Each variation in an individual tool induces changes in the tool’s adaptation to the ecology in which it is used—each slight change in a tool affects its overall fitness. These changes can run the gamut. They include changes to screen presentation, screen size, colour selections, layout options, input mechanisms, command language, commands provided, and so on virtually endlessly. It is enough to even try to understand the dimensions on which a tool can vary, but what is more, variations along one dimension modulates the fitness of variations on other dimensions. These co-dependencies make fitness very difficult to understand. In order to illustrate this difficulty evolutionary biologists have developed a useful visualization trick called a “fitness landscape” (also called “adaptive landscape” or “adaptive topology”, see e.g., Dennett [177] or Fogel [226, ch. 4]). The idea behind the fitness landscape is that the features which may covary are placed along different axes, and the resulting fitness is plotted as a function of these variables. Although such fitness landscapes will generally have extraordinarily high dimensionality, a two-dimensional slice can be very illustrative. Figure 7.1 is an example visualization. Given two dimensions for design variation, the resulting fitness results form a surface; the height at any point indicates the degree of fitness.

This sort of visualization be quite helpful in thinking about the properties of design methods and the possible roles of knowledge in design. From this viewpoint, design can be seen as the search for appropriate high points in the fitness landscape. Iterative design based on a preexisting tool can be seen as starting at some particular location, and then tracing a trail up one of the slopes. As far as how knowledge affects the design, it can be assessed by how it alters the properties of search in the landscape. Without adequate knowledge, search risks being too dumb.

The Problems with Dumb Moves

The most important benefit to Darwinian construction methods is that very little knowledge (i.e., none) is needed in order to stumble upon good tools. To make the scheme work practically any sort of mindless

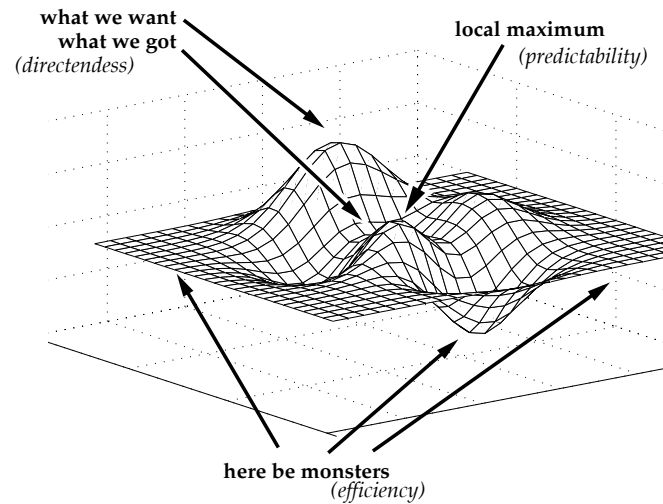


Figure 7.2: Problems of mindless search

random mutations of preexisting artifacts is sufficient. So long as the mutations are modest, the iterative build-and-test process will follow slopes up to the top of hills in the landscape. Even “dumb” moves can be good. This is seemingly fantastic news for tool designers. They can be profoundly ignorant and still manage to produce good tools. Unfortunately, Darwinian dumbness is closer to the current state of the art than we should like. It is unfortunate not (only) because the ignorance is embarrassing, but because of the problems of the mindless slope following. There are at least three problems with Nature’s maximally dumb design method. These are illustrated in Figure 7.2, and are as follows:

- **Inefficiency.** The most obvious problem of mindless search is its stupefying inefficiency. Normal design spaces are *vast* [177] and filled primarily with bad designs. Dawkins, ever the wordsmith, said “the vast majority of theoretical trajectories through animal space give rise to impossible monsters.” [170, pg. 73]. That is, most of tool design space is inhabited by inappropriate and unworkable tools. Consequently mindless search will happen upon the good ones only rarely.
- **Unpredictability.** The second most obvious problem of mindless search is its unpredictability and uncertainty. Many interesting parts of design space may never be ventured into. One particularly pernicious aspect of unpredictability is that it is possible to get stuck in a *local maximum* and never find more globally optimal designs. At a local maximum, all slopes lead down so that slope following (based on small incremental changes and doing iterative design and user testing) will never converge on a better solution. The problem is especially acute if the more global maxima are a result of multiple dependent design factors. Rasmussen *et al.* [531] are fully aware of this problem, and they even use a fitness landscape as an illustrative device. As they put it, missing a design consideration “in only one dimension may cause an otherwise optimal design to fail” [531, pg. 24]. From the point of view of searching in a landscape, the problems of unpredictability are that (1) random walks end up in random places, and (2) random walks make finding fit designs more difficult if they lurk in far flung locales.

- **Undirectedness.** The third most obvious problem of Darwinian design is the undirectedness of it. Without mindful selection of direction, fit designs are essentially the outcomes of a series of accidents. The outcomes might be fit for *some* use, but the result may have no obvious relation to any of the designer's original goals.

All of the above problems stem from the dumbness of the search, and specifically from the dumbness with which new designs are selected or constructed.

Now it is clear that the way we build new SE tools does not *really* rely on a truly *random* search of design space. Nonetheless it is helpful to look at the implications of random search because each of the problems noted for random search exist to some degree in current design practice:

- **Inefficiency.** Inefficient search is well known problem for HCI. User testing is an expensive process so it makes sense to leap up to the tops of hills as fast as one can and to avoid accidentally tumbling downhill. In fact, the process of inching up hills is a limiting method of iterative hillclimbing search: the most efficient searching methods will be able to leap from hilltop to hilltop without needing to sample the valleys in between. This corresponds to radical redesign of the existing tools⁵ rather than small incremental change [531].
- **Unpredictability.** The unpredictability and uncertainty of random search implies that many useful tool ideas are waiting to be uncovered if we only knew where to start looking. The problem of getting trapped into local maxima is also worrisome. It suggests that iterative improvement may allow one to race to the top of hills, but it subsequently traps one there even if there is a taller hill close by. Many times this problem is a result of the inter-dependencies between features of artifacts: sometimes good tool ideas are successful only if the appropriate supporting tool features are also present. If the designer does not realize the dependencies, the good idea will seem bad because user testing will show poor performance. We saw this in Section 2.2.1 for *SuperBook*, where the fisheye view was a good idea, yet careful testing showed that it performed poorly unless other features were added to the interface.
- **Undirectedness.** The undirectedness of dumb search may seem to be less obviously a problem for tool designers. Taken naively, the undirectedness is not a problem at all: normally nobody would start out trying to build a compiler and accidentally end up building a video game! Design, in the normal sense of the word, is all about intention (the term "design" is often used as a synonym for intention or purpose), so designers will generally not to pursue leads taking them far away from their intentions. Instead, designers will try to use whatever knowledge or beliefs available to them to find their way in the landscape. This knowledge may or may not be adequate, and Nature's

⁵There are two separate notions of radical (re)design. Here I am talking about making multiple, significant changes to the users' task environment as often occurs when, for instance, a computer system is brought in where none existed beforehand, or when introducing a large number of design changes to existing products. Another notion of radical redesign is a paradigmatic shift in tool form as described by Newman [452]. The latter type of radical redesign implies a situation in which truly and globally novel ideas are investigated. The former merely implies that the ideas are novel in the workplace the tool is being deployed in, and it does not rule out the possibility of having established engineering knowledge to say how to do so.

random search is the limiting case: random search uses no knowledge of where the high points are, and has no knowledge of which direction is better. The clear implication of undirectedness is that the worse off the designer's knowledge of the landscape is, the more the wayfinding resembles dumb Darwinian search.

The way to counter these problems is, of course, rather obvious: build in some knowledge and intelligence into the "generate" part of the "generate-and-test" cycle.

Building Knowledge into Design

...a practical goal for frameworks and models in HCI is to guide the derivation of suitable initial designs which, by virtue of their accuracy or utility (and these terms are not equivalent), reduce the number of iterations required before an ultimately acceptable design is achieved. Evaluations would subsequently act as confirmation or rejection of the design (or parts thereof)

– Andrew Dillon, "Designing Usable Electronic Text" [183], pg. 122.

If one wishes to improve the search in the fitness landscape with knowledge derived from theories, it makes good sense to resist applying one's pet theory for long enough to first determine where adding knowledge holds the most promise. One place it could be added is in the "generate" part of the "generate-and-test" cycle, that is, in trying to improve the synthesis of new candidate tools. How important is good candidate synthesis? Consider:

- **Inefficiency.** Good candidate synthesis can speed up hillclimbing by making it possible to make larger jumps up the hill, and by avoiding wasted iteration cycles going in the wrong direction. It also can make it possible to jump from hill to hill instead of following a convoluted path using conservative iterative improvement.
- **Unpredictability.** Good candidate synthesis can avoid getting caught in local maxima by reducing the chance of starting out on the wrong slope, and by either jumping long distances, or taking paths through low-fitness designs in order to break out of local maxima.
- **Undirectedness.** Good candidate synthesis can help designers start from the right location, and to set a course in the direction matching their design goals. Knowledge for doing this is called *survey knowledge*.

Knowledge about candidate synthesis is therefore a critical resource for good design. A good case can be made, in fact, that the *most important* role of theory in design is the provision of knowledge to synthesize good initial or succeeding design candidates.

The above presentation might at first seem somewhat unnecessarily belaboured in order to make relatively simple point. My sincere hope was that the presentation makes it very obvious to the reader that one of the most important roles for theory in design is for building good candidate synthesis knowledge.

I hope this fact is made blindingly obvious because the significance of this role is *not* universally acknowledged. In fact, the role does not even appear to be particularly well known or investigated. Instead, a great majority of theory in HCI is concerned with evaluation instead of synthesis, and with fine detail rather than survey knowledge. I shall briefly consider in turn why evaluation and fine detail should be considered secondary targets for theory development.

Synthesis, not Evaluation

... once in hand, the theory makes it possible to zip out products faster than cut-and-try engineering because one can skip a lot of the cuts and the tries. ... far from being a leisurely luxury, theory can be even quicker than "quick and dirty" engineering methods.

– Stuart K. Card, "Theory-driven Design Research" [93], pg. 502.

One of the most popular applications of theory to HCI is to substitute prediction for empirical testing. The clearest example of this principle is given by the authors of GOMS [94,448]. GOMS is, to put it very simply, a theory that produces predictions of performance given a description of an artifact, user, and task. Synthesis works the "other" way: it takes a desired performance and generates a new design intended to achieve that performance [348]. One can say that GOMS works "backwards" in relation to the direction of synthesis.⁶ The difference between synthesis and evaluation is illustrated in Figure 7.3. For synthesis, one needs resources to reason *forwards* from the design context to features of new artifacts such that the design goals are satisfied. Evaluative theories like GOMS, by their very definition, only serve to reason backwards. To use them one needs to have an artifact, prototype, or design in hand.

There can be no question that applying theory to the problem of evaluation can be very valuable [254]. But the preceding discussion on navigating the fitness landscape places the contributions of evaluation into its proper place. Theory's role in evaluation is of a secondary tool that helps out only *after* the important decisions are made. Evaluation's role in search is limited to establishing current position so that it can be compared to other established positions. As such, it can assuage the problems of inefficiency, unpredictability, and undirectedness in only the most roundabout ways. This limited role for theory in design appears to suit Landauer just fine, because he feels that "empirical cut-and-try" is the "foundation of design" [370, pg. 60]. He argued that the basis of design is inspired guesswork followed by iterative slope following, with theory playing a small and limited role in the guesswork portion. Now, it is true that we will surely never have enough theoretical knowledge to make perfect design moves. It may also simply be cheaper to do some guesswork and test the guesses even if the theory did exist. Nonetheless, we must resist being led into thinking that the most promising application for theory is in evaluation. The brightest promise for better design always was, and always will be in improving synthesis. Human are not more efficient designers than Nature because of their evaluation skills, but because of their construction knowledge.

⁶Calling synthesis "forwards" and understanding "backwards" corresponds to industry-wide terminology. E.g., "forward engineering" and "reverse engineering" [126].

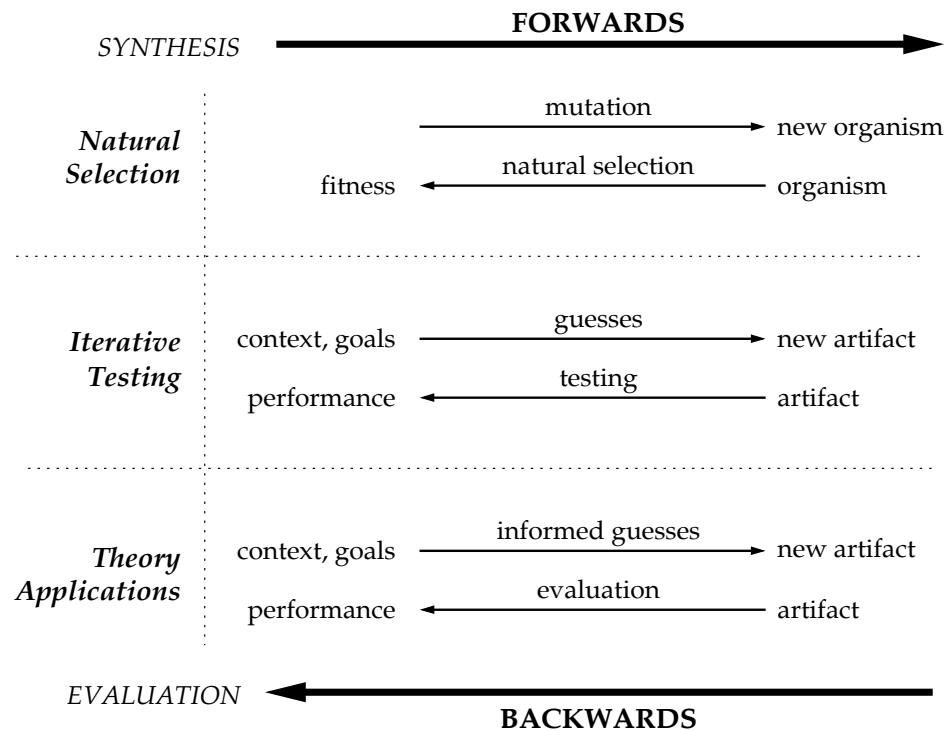


Figure 7.3: Roles of theory for synthesis versus evaluation

Because it is a slippery slope between evaluation and design, some elaboration on this point is warranted. When Card *et al.* [94] outlined their vision of an applied psychology for design, they held up, as a guiding example, a scenario of how it might be used. Specifically, they imagined a hypothetical designer of the future using GOMS analysis in order to compare the performances of users using two competing designs for text editor interaction mechanisms (these were, incidentally, keyboard-based vs. light pen-based solutions). The GOMS analysis showed that one design was easier to learn, but slower, and the other was harder to learn, but quicker. These results were then used in order to make and rationalize design decisions. Is this not the use of GOMS to do synthesis? Answering that question satisfactorily requires a modicum of subtlety, but the answer is definitely “no”.

The first key point to notice is that the space of design options (i.e., keyboard vs. light pen) was already set out somewhere else. Perhaps in the world of text editors the option space for input devices is well known, and needs no “theory” in order to lay out the possibilities. However, elsewhere the space of options is enormous, and the issue of understanding design options quickly becomes a dominant hurdle. Currently we rely on craft knowledge of the design space built from exposure to large portions of it in the form of many different existing artifacts. It is fine to say that designers should be able to learn to build an internal model of the structure of the design space. But there are few things so parsimonious as a theory that generates it from fundamental principles. As can be seen from Section 6.5, even a restricted decomposition of this design space can be important for reigning in the multidimensional space of options.

The second important thing to understand is just what the designer is forced to do with GOMS. GOMS’ role in the scenario is to analyze the design implications of the elements in the design space. The output

from this design space analysis using GOMS is a *decision table*, that is, a situation-specific design theory. Given a designer's preferred mix of learnability and performance, the table is a theory specifying (or predicting, if you prefer) the artifacts that deliver those goals best. It is the space of design options in combination with the decision table that form the knowledge the hypothetical designer used for forward reasoning. GOMS is used to produce this knowledge, but it is not the knowledge itself. It would be far, far better for the designer to have had a description of the design space and its decision table in hand to begin with. This is, in fact, the critical point to the whole question of analysis vs. synthesis.

Consider the cost to the designer of producing the situation-specific decision table: GOMS is needed for essentially every slight artifact variation because there is no equivalent of a generalized decision table—i.e., a generalized design theory. Not even a heuristic or approximated one. The designer is left to sample design space and evaluate each point. She tediously reconstructs the fitness landscape. Put another way, the very fact that an evaluative theory like GOMS is so often required is definitive proof that no appropriate design theory exists that could mitigate the need for it. Instead, the designer must sweep GOMS across the entire design space in order to point out the high ground in the fitness landscape. This last point conveniently brings up the issue of survey knowledge.

Survey Knowledge, Not Details

There are many ways of evaluating the adequacy of a theory [348] but one of the most prevalent in HCI and cognitive science is its predictive accuracy. Since most theories in HCI and cognitive science are for evaluation rather than synthesis, this means that much attention is paid to the accuracy with which a theory predicts a phenomenon such as user behaviour and performance when given an artifact. The reason for this attention is straightforward and understandable: accuracy of prediction is one of the best ways of establishing whether the basic ideas underlying the theory are sound. Unfortunately because predictive accuracy is used as a measuring stick, there is an overpowering bias against theories that try to explain many things [446], and a strong preference towards "limited theories" [259] that pick out some small phenomenon and predict it reasonably well. The price one usually pays for predictive accuracy is a limited scope [192] (see also Section 5.1). Predictive accuracy is still a valid measuring stick for design theories, but the above discussions of the difficulty of navigating the fitness landscape argue against applying it too aggressively lest limited theories are unduly favoured. And there are good reasons for considering accurate but limited theories to be less desirable than more general theories even if these are less accurate or less predictive. A key to realizing this preference lies in the cost difference between tool analysis and empirical testing.

Evaluation in the form of user testing is relatively easy and it can rather cheaply pick out many of the small flaws in design, especially problems in usability. That is, for many "minor" aspects of the design it is easier to test guesses than generate good design decisions in the first place. Of course, this is a generalization that needs some qualification, but even when qualified not everyone is likely to agree with it. For example, Landauer [371, ch.10] advances the argument that developing new ideas is easy but the evaluation is hard. He suggested that good new tool ideas like spreadsheets and direct manipulation are quite easy to create but properly evaluating their merits is much more difficult. This is in ways true and yet in many important circumstances it is generally false. The ultimate proof of the matter is that many

sorts of simple usability evaluations have proven very effective at detecting the obviously bad parts of designs (e.g., Nielsen [460]). In fact, the average *untrained* user is apparently quite good at detecting bad designs. Just ask any novice computer user whether they think operating it is easy; ask pretty much any expert computer user whether they think the system they are using is perfect. They will both generally be able to give a list of problems, and neither will have put in as much effort in listing these problems as the designer had in conceiving a novel design. The ultimate proof is that if users could not detect bad design, the push towards ensuring usability would be seriously muted. There may be many limitations to user testing, but even so testing design guesses will often be cheaper than building good initial designs, especially for the minor usability quirks. Using the metaphor of the fitness landscape again, user testing is the cheap way to inch up a hill once you have located it, but locating the hill is hard. The “test and tweak” method may be very hard to beat so long as only minor tweaking is required.

Given the relative cost of local hill climbing the problem with limited design theories is apparent. Limited theories—no matter their accuracy—by their very construction reflect a knowledge of *local topology*. Metaphorically speaking this is the equivalent of knowing well a small neighbourhood of a large city. Improving predictive accuracy of such a limited theory may do little more than eliminate a few steps on the slope-following curve: once you get to the right street, you can just go door-to-door. Thus the most important role for theory in design may very well be for hill finding, not hill climbing. Hill finding absolutely requires a non-limited theory, that is a theory that is very broad in scope and general, integrates many high-level design concerns, guides one roughly to the right neighbourhoods, and lets one quickly realize when one is in the wrong neighbourhood. A critical role for theory in design is therefore to provide broad survey knowledge rather than knowledge of limited local topologies.

In addition, the cost structure of theory application further disadvantages accurate but limited theories. Simply put, there is generally a cost to predictive accuracy and power in terms of analytic effort. This cost creates what Shum *et al.* [86] call the “cost gulf” for an analytic technique. It is clear that what goes on when human interact with computers is incredibly complicated; so in general, being able to predict what sorts of artifacts will be maximally fit requires accurate and detailed understanding of the users, tasks, environments, and so on [344,625]. One can expect that better initial designs only come as a result of better analysis and more powerful theory. This not only makes the analysis part difficult, it also makes the techniques much harder to learn for the HCI specialist and non-specialist alike. Barnard and May [30] describe the problem as follows:

[Many techniques] require detailed specifications to be generated for each application modeled. Often each application can require the specification of many rules, the construction of which requires a modeling expert. All of this work has to be redone for each problem considered. Therefore application of such techniques requires a large commitment of resources. In many design contexts, this is difficult to justify. [30, pg. 105].

Good and accurate theory application may be very costly and user testing can be cheap. This puts the squeeze on applied theories: in many circumstances the only cost effective role for theory in design may be hill finding. To put it bluntly, there may be rather little practical use for highly accurate but limited design theories.

The upshot is that if one is looking to build design theories, as I am in this chapter, the obvious choice to begin with is “quick and easy” survey knowledge that gets the designer into the right general area of design space. The minimum requirements for these theories are that (1) they be broad in scope, (2) they require only cheap, broad-brush analysis to apply, and (3) they need only yield general, high-level suggestions about the gross forms of tools. This suggests that applied researchers should tend to eschew limited theories in favour of aggressively broad and inclusive ones. I am hardly the first to point out many of these facts (see e.g., Young and Barnard [720], Green *et al.* [270], Dillon [183]), however these voices are difficult to hear against the din made by papers emphasizing the predictive accuracy of limited theories. As a consequence, some authors apparently feel the need to offer apologies for their broad-brush theories (e.g., Wright *et al.* [719]), probably to attempt to appease the “hard-line” theoreticians. No apology should be needed. Because these issues bear strongly on both how I will design and evaluate design theories, the above argumentation for the necessity of survey knowledge is worthwhile.

Summary of Navigating the Landscape

Theory builders should be aware of the overall importance and possibilities for theory in design. Initial design theories should probably favour breadth and generality instead of fine detail and predictive accuracy. The cost structure of theory application and user testing supports this point. Further, it should be realized that one of the most critical roles for theory in design is to allow the designer to reason about how to synthesize good designs, rather than to better analyze existing tools.

Even if these facts are realized, it is still necessary to understand what is required to enable synthetic reasoning. As the field stands now, not much attention has been paid to what it takes to improve synthetic reasoning. So in the next subsection I shall take on this issue by outlining the distinction between design context knowledge and synthesis knowledge, and then identifying the types of theoretical resources that can enable designers to cross what can be termed the “gulf of synthesis”.

7.1.2 Crossing the Gulf of Synthesis

...there is an assumption that understanding the programmer's mental model is an efficient route to designing effective tools. However, it is not at all obvious how to design a tool given a specification of the programmer's mental model. For instance, how does knowing that programmers will sometimes use a top-down strategy to understand code ... inform tool design? It doesn't tell us what kind of tool to build, or how to integrate that tool into the workplace or the programmer's work.

– Singer *et al.*, “An Examination of Software Engineering Work Practices” [597], pg. 210.

It is one thing to understand the world, and another quite different thing to know how to change it. In much of the cognition-oriented research work in software engineering and other fields, a great deal of attention is paid to understanding the contexts for which tools are designed. This is quite understandable.

Knowing the ecosystem that the tool is intended for is critical for knowing what would be “fit” in that ecosystem. In fact, one of the original rallying cries [20] for HCI was to “know the user” [288]. The cry led to wide-spread acceptance of the ideal of user-centredness [475] in the design of human–computer interactions. The basic call for understanding design context has been refined over the years. Now, more and more aspects of the design context are being studied: in addition to individual psychology, we are become more concerned with such things as collaboration and group interaction [44,591], the social and organizational backdrop [103,140,157,160,690], and the essential structure of work domains [530,657,658]. We are also told to study “authentic” users [275,583] “in the wild” [221,311,371] to know what they “*really* do” [60,596,658].⁷ In contemporary HCI much emphasis is placed on understanding the ecological context during design.

Cognition-oriented theories clearly have an opportunity to play an important role in understanding the relevant aspects of the ecological context. Researchers have turned to a variety of theoretical disciplines in order to shed more light on users, their activities, and other like issues in the design context. The typical presumption is that the theories can, in some vaguely understood way, be applied to better “inform” design. Now, it is hard to imagine that it is ever a bad thing to know more about the design context. Even so, it is important to make a distinction between being informed about the design context, and knowing exactly what to design. To paraphrase the quotation from Simon that started this chapter: designers want to know what *ought* to be rather than what *is*.

It would be a profound pity if theory could only be applied to “inform” the designer about what *is*, instead of what *ought* to be and how to achieve it. I am reminded of the old joke where the patient goes to the doctor and says “Doc, it hurts when I do this” and the doctor says “Don’t do that.” The joke is no less applicable if the patient says “Doc, I think I have a broken arm” and the doctor runs an exhaustive battery of sophisticated tests and says “Yup, its a broken arm all right” and sends him home, broken arm untreated. Designers want the equivalent of the doctor’s knowledge of how to actually mend the broken arm—it is not satisfying to just know the current state of affairs. Restricting theory’s role to simply “informing” design by articulating aspects of the design context is a seriously limiting conception of theory’s role. We want to use theory to cross what can be called the “gulf of synthesis”.

The Gulf of Synthesis

As Long [390] pointed out, it is possible to learn something about the design context without necessarily also knowing what to build in order to change it for the better. Say you learn that short term memory is limited. How would a theory of how short term memory works tell you what features to add to your source browser? It certainly tells you something about what *not* to include—features that require lots of short term memory—but nothing about what remedies a lack of it. The problem is that, in general, knowledge about a design context is not easily convertible into ideas for design—it is “inert”. This knowledge can serve to inform designers of the problems the users face. It can also serve to constrain design, but because the design space is so vast such constraints are not generative. Imagine going on a treasure hunt

⁷Blomberg [60] produced an excellent review of how the push towards understanding design context as a precursor to good design has come about.

when the only information you have been given is that the treasure is not in Mobile Alabama. Although one can try to add more and more constraints, theory will always under-constrain design [110], so exploring design constraints is helpful, but not sufficient. Knowing the user's problems and the constraints imposed on design does not actually tell one what to build.

After studying the design context the designer is therefore inexorably lead to the precipice between analysis and synthesis. Norman [322,467,468] used the term "gulf" to describe similar sorts of impasses. For instance he coined the term "gulf of execution" to name the impasse that occurs between a user's plans for action, and the performance of these. In an analogous way, the problem of developing design ideas when given some understanding of the design context can be called the "gulf of synthesis".⁸ Design context knowledge primarily leads one to better appreciate the gulf rather than ferrying one across it.

The gulf of synthesis is actually even more devilish than one might at first suppose. Not only is knowledge of a design context inert, it can also act like a set of blinders for the unwary designer. Pylyshyn [522] provided a clear example in the case of designing for designers themselves. Design studies have shown that professional and expert designers do not follow strictly phased development methods (such as the well known "Waterfall" method), despite suggestions that such phased development is the "right" way of doing software development (see e.g., Carroll *et al.* [108], Parnas *et al.* [488]). Pylyshyn correctly asked how this knowledge of the actual existing practices of developers informs the design of tools for designers:

What are we to make of such findings? Do they suggest that design cannot be automated, or that we should not study designers? The fact that certain ill-structured problems, such as design, are approached in a certain way by experts may not tell us anything about how they *could*, or *should* be approached given certain computational aids. [522, pg. 48] (emphasis original)

Now it may certainly be the case that design can really never be automated, but *claiming* that it cannot be automated based on observations of what designers *currently* do simply begs the question. So even if one studies "authentic" situations, one must be wary of confusing what exists with what is possible, and with what *ought* to be. The designer must have vantage point from which the possibilities for changing the status quo can be understood; she must be in a rather privileged position [657]. Traditionally, the issue of comparing what *is* with what *ought to be* has been posed as a distinction between *descriptive* theory (that describes what is) and *normative* theory (that describes what should be). For instance, a normative theory of design might *prescribe* that design should be performed as a group activity in which the appropriate shareholders cooperate. It is not appropriate to further unpack these various meanings in this chapter. The point is that a variety of models and theories can be used to understand the target ecology (the user's tasks, preferences, needs, social setting, and so on), but even once that is understood there is the next step of knowing the ways disturbing the existing world to meet the perceived needs. The important question to ask is therefore: how do we build theories for crossing the gulf?

⁸Singley and Carroll [598] have described a related—but different—problem called the "analytic-synthetic gap". The analytic-synthetic gap refers to a putative limitation of deduction from theory for the purpose of generating new designs.

One might suppose that application-oriented theoreticians spend a lot of time pondering just this question. Indeed, one would hope they have by now come up with a number of satisfying and detailed arguments as to how theory can be used to bridge the gulf of synthesis. In reality, we have only the barest inklings of how to do so. The claim that a theory or framework provides “design insight” or “design implications” can seem embarrassingly flimsy. This is not to say that existing theoretical works are not useful. In many cases claims for improving design might very well be true. For instance, I am inclined to believe Holland *et al.* [311] when they claim that a DC perspective really does provide “new insights for the design of conceptually meaningful tools and work environments” [311, pg. 180]. However since useful design theories are so hard to come by, the skeptic deserves a convincing argument. Alas, with the exception of a rather sparse set of outliers (e.g., Shum *et al.* [86], Rasmussen *et al.* [531]), there are few arguments that force us to believe the claims for improving design practice are more than hopeful wishes. Perhaps one reason for this state of affairs is that too many theory developers succumb to the fallacy of the “magician’s design method”.

The Magician’s Design Method

The magician’s method for designing good tools is to study the design context until magical insight occurs. This method is diagrammed in Figure 7.4. The “magic” occurs at the boundary between understanding and action, and it seems that too many theoreticians appear content to keep the synthesis step an enigma. I do not doubt that if researchers study a problem domain for long enough, then a few good ideas might pop into their heads. This seems especially likely for exceptionally clever researchers who publish papers on design. Moreover, I am certain that expert designers will in most cases be able to call upon their vast knowledge of past designs in order to come up with some workable ideas. In contrast with either of these methods, I am searching for something that takes as input a partial understanding of a design context and, *by using explicit theory*, generates useful design ideas and concepts.

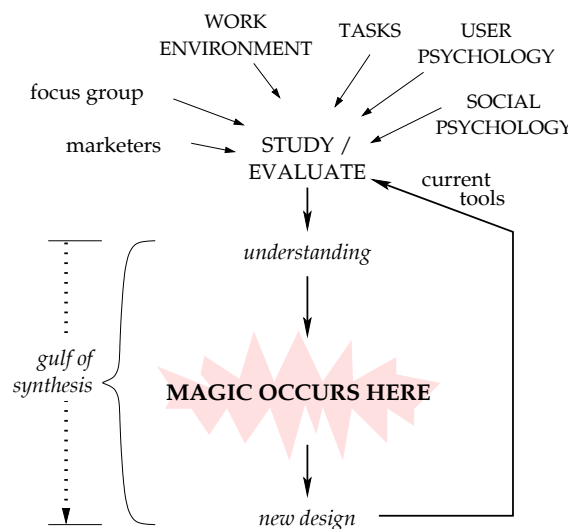


Figure 7.4: The magician’s design method

This is not an unreasonable request. It is precisely what is required for any theory that can be directly applied to cross the gulf of synthesis. The theory does not have to mechanically or infallibly generate new designs as, for instance, an ideal automated designer might do. An automated designer would take descriptions of the design context and design goal. It would use a theory to generate design moves (it can be implicit), and would accordingly construct an appropriate artifact. For generating a restricted variety of information displays, this theory-driven approach has already been tried (e.g., Casner [116], Zhang [725]). Automating the construction of certain suitably restricted aspects of SE tools is certainly conceivable. For example, it might be possible to automatically generate task-relevant program visualization displays when given some description of a maintenance task (or at least partly automatically [195]). Automating SE tool design to any significant degree still seems farfetched, however. But full automation of design synthesis need not be the goal. The theory does not have to authoritatively or infallibly “dictate system design characteristics” [370]. Instead theory can simply *guide* design reasoning.

On this view, the role of a design theory is to provide recommendations and suggestions, or to produce guidelines [28]. Designers—even when they use a design theory—will still make poor design choices, rely on intuition and craft knowledge, and iteratively test out new ideas. Nonetheless, it is entirely worthwhile to take seriously the problem of constructing design theories. To not do so is to meekly accept the mysteries of the magician’s design method, and to thereby miss out on the possibility of reducing the reliance on intuition and iterative testing. But what is first required is some way of making scrutable the processes of idea creation and design decision making. Shedding some light on these is necessary in order to generate a decent argument detailing how theories may be used during the critical step of synthesis. To make this argument one needs some kind of model of the knowledge and resources used during design reasoning. In the following subsection, I will propose one simple model. The purpose this model is to argue the importance of theories allowing designers to reason about what I call “positive consequences” of tools, that is, to facilitate *FP-reasoning*.

7.1.3 Theory for FP-reasoning

Theoretical integrations along the environment dimension ... are hardly ever attempted but are critically needed to support the cross-environmental reasoning inherent in design. It should not come as a surprise that most cognitive psychologists are not overly concerned with this type of theoretical unification, because an acceptable scientific product is a model of behavior in a specified environment, and rarely is reasoning backward from cognitive theory to environment required.

– Alex Kirlik, “Requirements for Psychological Models to Support Design” [348], pg. 74

In order to demystify the magical synthesis step, it is necessary to model something about the design process [62]. Fortunately a little detail will often go a long way. It is enough to consider here some

of the resources that designers rely on to synthesize new designs.⁹ The HASTI framework can actually be employed to model the sorts of knowledge employed by designers.¹⁰ During synthesis the designer explores what HASTI defines as the *problem*, i.e., designer's overall goals, the constraints placed on their design options, and the design moves that are possible. These define the designer's problem space and so design activity can be interpreted as an exploration of this space. That is, design is an exploration of possible design moves according to the design goals while respecting the design constraints. Since this definition of problem space changes as design and development progresses, we can therefore say that these resources comprise the "synthesis state". The point is that if we can understand something about how the synthesis state is constructed we can consider how to impact synthesis without needing any more of the gritty problem solving details. In other words the analysis can proceed at what Newell called the "knowledge level" [445]. The question for this subsection is how theory can be applied to define the problem space as it is constructed and explored by a designer.

To begin answering this question it is helpful to elaborate what things should be considered design goals, constraints, and action possibilities. A short survey easily rendered:

1. **Goals.** Designers adopt design goals based on what they believe tools *ought* to be. From these beliefs, designers adopt high level design goals that serve to globally organize their exploration of design space. Examples of general design goals from the literature include (in no particular order):
 - (a) *Automation.* Designers may adopt a goal to try to automate as much as is possible.
 - (b) *Reduce cognitive overhead.* Navigation can be viewed as an extra burden, so the cognitive load it imparts can be a focus for reduction (e.g., Storey *et al.* [619]).
 - (c) *Information provision and formatting.* The designer may think the purpose of software comprehension tools is to provide the right sort of information in the right format. This expectation may set up goals to pursue certain search and visualization tools (e.g., von Mayrhauser *et al.* [668,670]).
 - (d) *Skilled performance.* One may adopt the position that the goal of design is to provide environments in which manual and perceptual skills can be employed in place of intensive reasoning (e.g., Kirlik [348]).

Note that none of the goals are directly operational, that is, one cannot actually generate designs just by adopting them. Also note that one important aspect of goal setting in practice is that designers are known to follow *habitual paths* in design space (e.g., Stacey *et al.* [615]). For example, Carroll and Rosson, who have both a penchant and an aptitude for applying psychology during design, noted offhand that "it seems that almost everything we design involves example-based learning" [110, pg. 194].

⁹Design is frequently a group activity. Appropriate extensions to a group problem solving model may generalize, but these extensions will not be considered here.

¹⁰This observation was partly inspired by Rasmussen *et al.* [531] and Buckingham Shum *et al.* [86], who both expertly performed a similar trick of recursively applying design theory to designing theory.

2. **Constraints.** Design constraints effectively prune the design space: they identify the parts of design space that should be avoided. That is, they identify which tools are impossible, unsuitable, or simply undesirable. Thus design constraints implicitly identify the *negative consequences* of tools [110]: the undesirable implications of the tools such as usability problems. For instance a tool may force the user to remember too many items, and therefore violate a constraint imposed by the memory limitations of users. By respecting design constraints designers avoid usability problems.
3. **Possibilities.** Design possibilities are the steps that designers can take to achieve their design goals. Since designers typically try to make useful tools the design possibilities correspond to the moves that make useful artifacts. In other words, design possibilities are ways of creating artifact features with *positive consequences*.

How can theory be applied to construct these three aspects of the designer's problem space? In the previous subsection I identified three main classes of theory application: *backwards* from artifact, *forwards* from goals, and "informing" theories that help the designer generally analyze and understand the design context. If one composes these three classes of theory application with the above three resource types, one gets a matrix of 9 different possible applications of theory. I shall consider five of these: the four categories defined by $\{forward, backward\} \times \{negative\ consequences, positive\ consequences\}$, and the case where design goals are inspired by a theory's *design stance*. The first four types are abbreviated *FN*, *FP*, *BN* and *BP* where *B* and *F* stand for "backwards" and "forwards", and *N* and *P* stand for "negative" and "positive" consequences, respectively. Each of these four categories of theory application assists in a different category of design reasoning. These categories are thus called *FN-*, *FP-*, *BN-*, and *BP-*reasoning, respectively. The fifth application (providing a design stance) can be said to enable strategic goal setting. These five applications of theory are depicted in Figure 7.5 (the ones applied here are circled).

Theory Guided Reasoning

Each of the five applications of theory provide a way of guiding design reasoning by allowing the designer to use theoretical resources to establish and consider goals, constraints, and possibilities. The problem space that the designer may reason about is dependent on the forms of such theoretical resources. Table 7.1 summarizes the applications for the four types not concerned with goals. These are:

BN BN theory enables reasoning backwards from an artifact to the negative consequences that it embodies. This is the prototypical form of using theory in HCI. Methods for invoking theories from

	BACKWARD	FORWARD
NEGATIVE	predict / explain problems	constrain design
POSITIVE	rationalize / explain benefits	expose solution space

Table 7.1: Four types of reasoning and roles of theory for supporting them

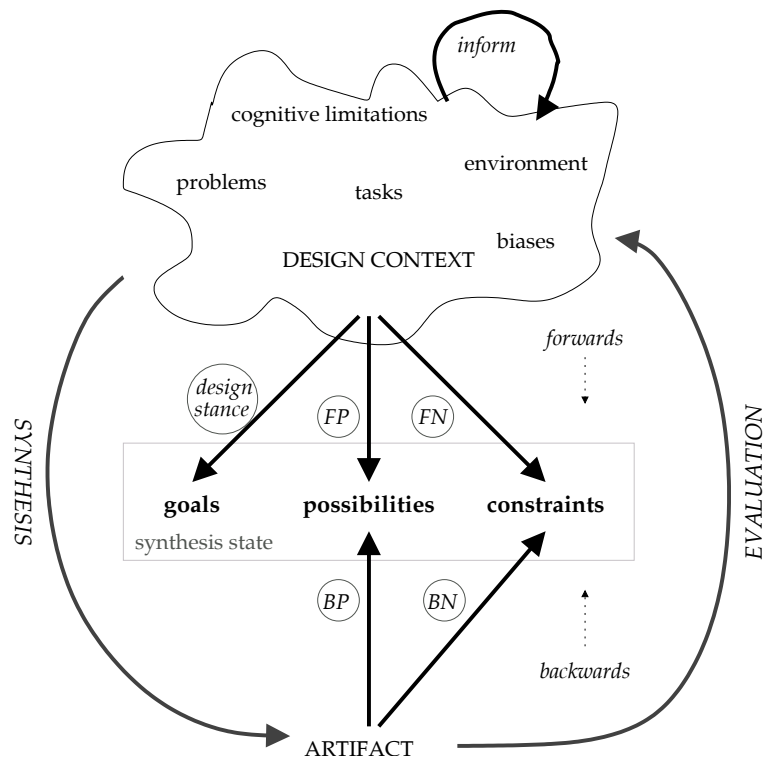


Figure 7.5: Resource flow model of theory application in design

artifacts include the various forms of “cognitive walkthroughs” [110, 272], or the execution of programmable models [86]. The role of theory in these methods is to predict usability problems in an artifact without actually testing it (or else to explain their source).

BP BP theory enables reasoning backwards from an artifact to understand how the artifact can be considered useful or otherwise beneficial. The role of theory in this case is to either rationalize the design decisions or else explain the source of the designed benefits. The CoSTH of Chapter 6 is a collection of BP theories.

FN FN theory helps one reason forward from design goals to anticipate the possible negative consequences that should be avoided. Normally this means helping steer design away from problematic or unusable features and thereby shortcutting the need for BN theory during retrospective evaluation.

FP FP theory allows reasoning forward from design goals to features of artifacts that can be beneficial. These possibilities lay out parts of the design solution space. Although CoSTH is primarily a BP theory, it also has some potential to enable FP reasoning since it identifies possible reengineering that could be accomplished. In particular, redistribution, substitution, and cognitive rearrangement are theories for reengineering cognition to create support.

The fifth type of theory application being considered here is the provision of a design stance in order to help generate goals. Theoretical frameworks often carry with them a way of looking at user problems and

the benefits of artifacts. The theories therefore provide a *design philosophy* or *design stance* for the designer. Colloquially speaking, different design stances let designers try on differently tinted glasses. Each design stance serves to highlight different problems in the design context, and to develop goals for achieving them. For instance Kirlik [348] outlines a design stance that honors skill-based task execution as a way of reducing cognitive burdens. Adopting this design stance can lead the designer to generate goals of providing skill-based methods of completing cognitively challenging tasks. By providing a design stance, the theory serves to bridge between problem comprehension and goal setting, and subsequently acts as a heuristic strategy for searching the design space. Since designers tend to follow habitual paths, having a collection of differently coloured glasses can open up the design space and afford to designers multiple ways of tackling design problems.

7.1.4 Summary and Implications for Designing Design

... Viewed in this way, the user interface is not a gulf, but a resource for action. Unfortunately, current cognitive models of how action is generated do not throw much light on the way such a resource might be exploited.

– Stephen J. Payne, “Looking HCI in the I” [491], pg. 185.

In this section I have gradually constructed a taxonomy of ways of applying theory in HCI. The taxonomy has two main orthogonal application dimensions: (1) the direction for reasoning, and (2) the types of artifact consequences to reason about. The two directions of reasoning were called “backwards” (B) and “forwards” (F). Backward reasoning was characterized as starting with an existing artifact, prototype, or design idea and working towards explaining or predicting properties of the resulting artifact. Forward reasoning was characterized as starting out with a design goal to achieve and coming up with artifact features that can achieve the goal. The artifact consequences to reason about were called “negative” (N) or “positive” (P). Negative consequences are the “bad” implications of artifacts such as usability problems they create, and positive consequences are the “good” implications of artifacts such as the cognitive benefits they provide. Using this taxonomy and a simple model of the design problem space, I considered five applications of theory to guide designers in constructing their design space: BN-, BP-, FN-, and FP-theories, and design stances that help designers set design goals. I then argued that the DC theoretical resources developed in previous chapters are potentially very well suited for BP- and FP-reasoning, and for setting design stances.

In terms of designing cognitive artifacts, some of these five applications of theory have definitely received more attention in the past. Most notably BN theory, but also to a lesser extent FN-theory. For instance, human performance models are primarily useful for backwards reasoning. Returning to the lever example of Section 2.3.1, performance models are analogous to engineering models that can determine the load limits of a given lever. This is an important, but complementary issue. Design theories suggest instead that perhaps a lever might be appropriate solution. In addition, much of the less theoretical work in HCI (like checklists and usability inspection methods) can also be categorized as resources for

FN- or BN-reasoning. Chapters 3 and 6 showed that there are many different studies of cognitive support. Yet the use of cognitive theories in HCI is still dominated by performance theories, and theories useful for predicting or diagnosing usability problems. Why this particular focus? As Payne [491] has implied, HCI researchers tend to treat interfaces as posing problems rather than providing aid. This bias could explain a general tendency towards BN- and FN-theories.

The analysis in the earlier part of this section suggests that current prioritization of HCI research is unwarranted. Specifically, the types of theories most keenly needed are FP-theories and theoretical frameworks that can set a useful design stance. If one were to build a priority list for theories to develop, two types of theories would top it: FP-theory that lays out the solution space, and design stance setting theories that provide strategies for navigating it. In addition, one would have to give a much higher priority to broad-brush theories that provide global survey knowledge. These provide the biggest cost-to-benefit ratio for the designer. The executive summary is most useful now; details can be filled in later. The somewhat belaboured analysis presented in this section would be unnecessary if this prioritization were acknowledged in more than a few isolated instances. Unfortunately this is not the case, and the result is that the profile of theory application in HCI reveals unsightly gaps in the most inopportune places.

How can this insight into theory types and priorities be converted into theories fit for design? I propose that cognitive support theories such as CoSTH can be employed to fill in some of the gaps. Although it clearly has promise for FP-reasoning, CoSTH is a theory that is built primarily for BP-reasoning. Given an artifact, one can use the CoSTH to understand what benefits are embodied by the artifact in terms of various types of cognitive support. The question to settle is whether it is possible to more effectively reverse the direction of reasoning, that is, whether RODS, HASTI, and CoSTH can be massaged to provide a useful and usable design stance, and whether they can be put in a form that makes FP-reasoning simpler. This is the question I tackle in the following section.

7.2 Cognitive Support Knowledge Fit For Design

The tacit assumption in the modeling world has always been that science and modeling is hard, but that if one can get some results the design, while a little scruffy, is relatively easy. In fact, one of the greatest difficulties [we found] has been in figuring out what designers could use models for and what is a reasonable design method that would incorporate models. Theorists have been observed to undergo severe culture shock when required to extract from their model or theory some piece of information necessary to answer some design question.

– Stuart K. Card, “Theory-Driven Design Research” [93], pg. 506.

The main idea behind cognitive support is that artifacts can be a partner in cognitive work. Therefore “support”, as a concept, is squarely aimed at explaining why tools are beneficial rather than merely tolerable (or a hindrance). With the exception of Section 2.3, theories of cognitive support have been treated in this work as a method of *explaining* how the support works: how a tool—or, properly speaking, changes

in the tool environment—alters a DC system in such a way as to improve cognition. In other words, in their present forms RODS, HASTI, and CoSTH are tailored to analyze existing tools and are thus fit for BP-reasoning. BP-reasoning is very important during design since, at the very least, it is necessary to remember past successes [110]. But what can these resources contribute during synthesis without first having a tool in hand to analyze? It seems that CoSTH in particular could be used to enable FP-reasoning and provide a useful set of design stances. Even so, it is entirely likely that these resources are better classified as *basic* design theories rather than *applied* design theories: they are not especially tailored to be used by practitioners. The question is whether they can be changed in form to suit designers better, that is, to make them both useful and usable.

In this section I will argue that they can. I shall put forward three different ideas for molding and repackaging the concepts from RODS, HASTI, and CoSTH so as to be better suited for application. None of the ideas for repackaging theoretical resources are novel, but since RODS, HASTI and CoSTH are new, they can be applied in novel ways. With the exception of a few tentative forays [331,685] into doing so, the results are all distinct from prior work. This avenue of research can be explored only briefly and tentatively in this work. Although it is a poor substitute for thoroughness or rigour, I shall strive to convey the *promise* of these approaches towards building applied design theories. If theories are designed artifacts, then what I am proposing would qualify as early design envisionments or skeletal prototypes—they are not even beta versions. Nonetheless, I think presenting these early prototypes is very worthwhile. Exploring these ideas even briefly should help point out promising directions for future research. Hopefully, this foray should also help convince the reader that the theoretical resources developed in other chapters have a reasonable chance at providing knowledge fit for design.

The three theory-repackaging ideas are: (1) building a working vocabulary, (2) providing useful design perspectives, and (3) reifying the design space by providing checklists and the like. These are outlined in three subsections below. This is an exercise in designing design knowledge, and so the effort will necessarily be tailored towards a particular type of design or designer. During each subsection I will thus begin by introducing the design idea being pursued, and the intended target audience and design setting. Then I will overview the ways that the design idea can conceivably be implemented by using RODS, HASTI, and CoSTH. In Section 7.2.4, I will conclude with a brief summary of the design ideas.

7.2.1 Engineering Concepts and Vocabulary

The contribution of cognitive science to HCI research is to provide the conceptual background against which the engineering principles can be understood.

– Barnard and May, “Cognitive Modelling for User Requirements” [30], pg. 102.

Design Ideas

Within linguistics there is a reasonably well known hypothesis—the so-called “Whorfian Hypothesis” or “Shapir-Whorf Hypothesis”—that the languages a person is able to speak determines what they can or cannot think about. While that hypothesis may not be entirely well founded, it is almost certainly the case that a concise and appropriate vocabulary and its associated conceptual background can effectively influence and aid thought [608]. To properly unpack the idea, one needs to consider the properties and content of both the conceptual background in question, and the actual lexicon being proposed.

In terms of conceptual background, the important concepts for cognitive support are at the cognitivist’s level. If one wishes to design tools that improve the cognitive work of software developers, it may be wisest to analyze and reason about tools in cognitive terms. This was one of the lessons implied by the lever scenario of Section 2.3.1. Reasoning about tools in cognitive terms abstracts away from the details of the implementation (see also Section 4.1.6). Brooks [77] in fact argued that an appropriate abstract conception of artifacts is a critical foundation for any design theory:

How then do engineering theories arise? A necessary kernel is the development of an appropriate abstraction that discards irrelevant details while isolating and emphasizing those properties of artifacts and situations that are most significant for design. Indeed, this property of abstraction may be more important than the extent to which the abstraction gives rise to manipulatable formalisms or prediction; by indicating what properties of an artifact really are significant, a good abstraction may lead both to invention of new artifacts that produce these aspects in novel ways and to novel uses for existing artifacts. [77, pg. 54]¹¹

Brooks makes the two relevant contentions: (1) the *abstractions* provided may be important than modeling methods or prediction, and (2) abstractions are a crucial part of synthesis and invention. If true, the implications are that a programme of concept and vocabulary building may ultimately be one of the most important avenues for providing theory to designers.

If one wishes to try to support these claims with evidence from research, one possible place to look is in studies of expertise. Experts are competent where novices are not, and capabilities of experts may reveal what competent cognitive support designers need. Numerous studies have suggested that experts owe much of their performance advantages to their ability to quickly abstract away from the low-level

¹¹Brooks uses the term “engineering theories” which for the present purpose can just be considered a more specific term than “design” theories.

details of a problem, and to more directly consider the abstract, function- or goal-relevant aspects of the problem (e.g., Soloway *et al.* [6, 608]). It is reasonable to expect, therefore, that expert cognitive support designers would be able to rapidly interpret tools in cognitive terms, especially their functional roles in terms of how they aid performance. A cognition-oriented set of concepts that include cognitive function-oriented ideas relating to design would likely be a necessary prerequisite. This is a direct extension of Soloway's argument [608] about teaching programming abstractions to programmers: one needs to teach cognitive abstractions to cognitive support designers. Knowing about support-related abstractions does not automatically make one an expert on designing cognitive support, but it is hard to imagine an expert cognitive support designer that does not inherently use them.

Next, consider a vocabulary to go along with the concepts. Vocabulary building is an effort to influence the moment-to-moment thinking and conversation of designers [266]. Vocabulary building is a kind of conversation-level analogue of the long standing belief in mathematics that coming up with an appropriate notation is sometimes the most important contribution one can make. The vocabulary in question here relates to the concepts for generating positive consequences. A useful vocabulary for positive consequences ensures that the appropriate concepts can be employed smoothly during analysis such that not only are the ideas of cognitive support made more precise, but the terms themselves bring into the conversation, through connotation and association, important design issues and options. This programme of vocabulary building is familiar to anyone acquainted with design patterns [205, 232] or taxonomic categorization [516]. The idea is that a common and well defined vocabulary simultaneously add control and rigour to existing design practices, especially informal design practices like brainstorming and casual design argumentation. It is a mechanisms for improving communication between designers by providing a common vocabulary with which it is possible to quickly establish a common understanding or common ground. As Green and Petre say:

Explicitly presenting one's ideas as discussion tools is, we believe, a new approach to HCI, yet doing so is doing nothing more than recognising that discussion among choosers and users carries on interminably, in the corridors of institutes and over the Internet. Our hope is to improve the level of discourse and thereby to influence design in a roundabout way. [272, pg. 132]

It is an open question as to whether more than one lexicon should be developed, each tailored for a particular community based on their existing terminologies and conceptual backdrops. For instance, it is easy to envision three separate vocabularies: one for cognitive scientists and HCI theoreticians, one for SE researchers, and one for in-the-trenches practitioners. Green weighed in on the side of a common vocabulary, arguing that a *lingua franca* is important for bridging the various communities [263]. He also suggested that using psychology-laden engineering-oriented terms ensures that interest and acceptance by both communities. Currently the dimensions work of Green *et al.* appears to be leaning towards a single universal (English) vocabulary [54]. I will leave for others to consider the question of whether a single lexicon should be developed. In the meantime, I shall advance one that I think will suit SE researchers.

Target Audience and Setting

While our ability to build more powerful machine cognitive systems has grown and promulgated rapidly, our ability to understand how to use these capabilities has not kept pace. Today we can describe cognitive tools in terms of the tool building technologies (e.g. tiled or overlapping windows). The impediment to systematic provision of effective decision support is the lack of an adequate cognitive language of description...

– David D. Woods, “Commentary: Cognitive Engineering in Complex and Dynamic Worlds” [713], pg. 116.

Most software tool researchers, and nearly every software tool practitioner, has little tolerance or enthusiasm for applying “real” psychology-oriented theory in design. It is only a small exaggeration to say that many software developers actively dislike HCI theory—even HCI specialists tend to eschew HCI theory [86, 183]. A compelling but challenging target audience are those tool researchers that are computing science savvy but largely HCI and cognitive science theory illiterate—i.e., the majority. If a good vocabulary and set of concepts can be developed for these potential clients, then perhaps it is not too farfetched to think that informed practitioners could one day benefit from similar resources. As indicated above, the vocabulary building effort will be aimed initially at informal design settings such as brainstorming sessions and early design meetings.

Resources

Science is likely to pace technological progress when ... [it] provides tools for thought, either conceptually or mechanically. That is, science can be effective when it helps us see a new way for conceptualizing the design space or it allows us to make inventions that themselves aid intuitive design.

– Stuart K. Card, “Theory Driven Design Research” [93], pg. 502.

RODS, HASTI, and CoSTH contain many important concepts, and I have tried to be conscientious when choosing terms for each of them. The general naming rule that I established early on is to prefer terminology familiar to computing scientists even if other terminology carries connotations important to others. Consequently the key design-related concepts are all couched in computing science terms: the support principles, and the support theories. It is not very productive to try to reconstruct all the important concepts and vocabulary built here. Perhaps an appropriate handbook can be produced sometime in the future as Green *et al.* [269] have done for the Cognitive Dimensions framework. In the meantime, I will list some of the key terms and group them into broad categories:

1. *General DC view of cognition.* Important terms include: joint system, distributed processing, shared processing, distributed knowledge, external knowledge, external memory, augmentation, media.

2. *Modeling ideas.* Virtual memory, caching, paging, memory management, agenda, goal, plan, problem, constraint, operation, problem state, history, snapshot, trace, path, perceptual operator, skills, rules, skill-based, rule-based, knowledge-based, domain task, overheads (coordination, device, coping).
3. *Support principles and support types.* Distribution, distribution, specialization, optimization, substitution, reengineering. Offloading, allocation, precomputation, display-based problem solving, mediation, backtalk. Extended working memory.
4. *Support targets.* Cognitive overheads, overload, cognitive limitations, depth-first bias.
5. *Support features.* Memory bandwidth, access costs, update costs, reference locality, localization effect, working set, effective working memory size, page fault rate.

Most of these terms are associated with concepts that I have already tried provide workable definitions for. The third groups of terms is the most critical. The names of the support principles connote many design-related issues of how to engineer positive consequences. The fourth group list cognition-related motivators for finding support. The fifth group gives hints that design tradeoffs may be analyzed in terms of measurable variables of a DC system.

Unfortunately, defining a useful vocabulary provides no assurance whatsoever that it will be taken up and wielded effectively. It may be difficult to get non-psychologists to think in cognition oriented terms—to view bookmarks as an extended memory, to see compilation errors as partial repair plans, and so on. Nevertheless, if they do, then they will have a chance to key in on the relevant design issues in response: an external memory is being used? What are the access costs? How is it indexed?; a plan is constructed? How are plan steps ordered? Should execution state be stored externally? The hope for vocabulary building for cognitive support is that discourse can be raised above the suffocating details of keystrokes and colour schemes, and onto the cognitive level.

7.2.2 DC Design Stances

What is required is a description of level of discourse that ... actually supports valid descriptions of human activities in a form that is most meaningful for system design. This is not an easy task but one is helped by at least knowing where the goalposts are. Within the electronic text domain a suitable analytic framework should provide designers with a means of posing appropriate questions and deriving relevant answers.

– Andrew Dillon, “Designing Usable Electronic Text” [183], pg. 65.

Design Idea

Design perspectives, or *stances*, are used by designers in order to decide what *ought* to be. In this sense they are more of a philosophy than a science of design. Design stances allow designer to pose questions

about what goals to pursue; they predispose designers to consider certain options rather than other ones. Craft disciplines often inherit culturally propagated design stances. For instance, a common theme that keeps being propagated in SE is the belief that SE researchers should seek out and automate the mundane and repetitious [275, 539, 596]. Designers who embrace this perspective will tend to set their sights on just the mundane and automatable. Not that this is *necessarily* bad: no design perspective is right or wrong in any absolute sense. However each stance biases the designer to explore a restricted area of the design space. Since cognitive support is multifaceted and designers exhibit a tendency to follow habitual design paths, a broad collection of available design stances is desirable. DC generally, and CoSTH particularly, provide many different vantage points for thinking about how to design cognitive support. If these can be documented and passed on, then there is hope that researchers would stop to think more critically about broadening their goals when considering joint cognitive performance. They may provide productive counterpoints to their existing design biases.

Target Audience and Setting

My particular concern is SE tools researchers, but researchers from other domains (and even the thinking practitioner) could benefit from defining suitable design perspectives. SE tools researchers are of interest because they are a special type of designer: by default they are trying to bend the status quo, and so they are not generally satisfied with reusing prior tried and true solutions. The alternative design stances may help them in this matter by steering them out of habitual solution paths. Furthermore, in the domain of SE, cognitive costs are a central issue, and so SE researchers are likely to be interested in knowing how to reason about how to lower cognitive costs with technology. This background may make them more receptive to theoretical approaches than the average practitioner. As with vocabulary building, the anticipated setting would be in early design reasoning when the core functionality and behaviour of the tools is being considered.

Resources

The DC point of view takes seriously the ideas of joint and mediated cognition. Here I consider three popular design stances that can be found in SE research, and then show that a DC point of view can offer valuable alternative stances. These can help generate distinct design goals:

1. **Automation.** Many important software development tasks are nowhere near being automated. Yet many researchers doggedly pursue full automation only to later resolve themselves to acknowledge the limitations of their automations, and to duly note their points of failure. Typically, the failures are considered to indicate “directions for future research” rather than being foreboding indications of the limitations of the full-automation approach. Two examples from reverse engineering (where the limitations of full automation are easily reached) are in cluster analysis [648] and program recognition [714]. Neither cluster analysis nor program recognition can currently be automated, so semi-automatic techniques must be eventually considered. This limitation of the full automation design stance is widely acknowledged, yet there are comparatively few examples where alternative design goals are rationally developed.

Counterpoint: Cooperative problem solving. Automation is an unrealistic goal for many interesting tasks, so why not give it up at the start? Instead, one could begin by carefully analyzing the task and look for ways of redistributing the data and processing between computer and human. On this view, the overall design goal is to put the human deeply and intimately “into the loop” of processing. Returning to the example of clustering and program recognition, there does exist work that tries to integrate the human into the problem solving loop of the tool. For example in joint clustering there is the body of Rigi work [359,426], and for joint program recognition there is the DECODE work [127]. Let us just consider Rigi for now. Rigi’s general view of joint clustering is that humans bring in knowledge of how to specialize and run clustering algorithms, which the computer performs. Then the human exercises further manipulation of the results and applies more informed judgments. Rigi is an example where the design stance taken assumes joint performance of tasks that cannot be automated.

Even though joint cognition is sometimes considered, existing design stances can be considerably expanded in terms of what design goals can be developed. For instance Jahnke *et al.* [331] argued that a useful design stance for reverse engineering tools is to assume that a specific, key objective is to redistribute *imperfect knowledge*, i.e., knowledge that is tentative, contradictory, heuristic, or vague. Once it is redistributed it can be manipulated externally and cooperatively processed. Design goals under this stance then shift attention towards discovering the types of imperfect knowledge which can be redistributed onto an external medium, and towards discerning how the external knowledge can be manipulated (or cooperatively processed) to remove the imperfections. Even from this lone example it is clear that the DC design stance can expand on our currently simple forms of design reasoning, like the push towards full automation.

2. **Right Information, Right Time.** There is a rather entrenched perception within the reverse engineering and program comprehension (RE/PC) community that “the purpose” of RE/PC tools is primarily to display information. von Mayrhauser and Vans articulate this stance well:

If we can identify and present the programmer with information that best helps to understand code, we can significantly improve quality and efficiency of program understanding... [680, pg. 316]

Researchers who adopt this stance typically focus their efforts on methods for information provision and presentation. The resulting tools are typically structured in the form that might well be called the “holy trinity” of RE/PC tool structures: the “parse–analyze–display” pipeline (e.g., see Tilley *et al.* [424,641,642]). In visualization circles, this architecture is called the “visualization pipeline”. Posing tools as a visualization pipeline channels attention towards techniques for gathering (parsing, analyzing, etc.) and transforming (clustering, filtering, etc.) data into an understandable visual presentation (graphing, layout, etc.). Since the guiding goal is to present the desired information simply, easily, and at precisely the time it is needed, let us agree to call this the “right information, right time” design philosophy, or RIRT for short. The overall goal of RIRT is to efficiently transmit the information contained externally to the comprehender.

Counterpoint: Hands-on, Mediated Problem Solving. Most of the time the RIRT design stance implicitly

buys into the belief that comprehension and understanding is a rather simple one-way *transference* of information or knowledge from tool to user. However transference, as a notion for comprehension, certainly seems to be limited. Viewing software comprehension as a simple transmission of data across a channel is almost certainly grossly simplistic. Many modern education research groups present a stark contrast in design stance. It is fair to say that most modern educational psychologists disagree in principle with simple transference models of learning (e.g., see Mayer [400]). Students cannot learn complicated material simply by listening to a talking head at the front of a lecture hall: they must work through examples, try hands-on experiments, and ask questions. Learning, in this view, is not simply transmitting knowledge across a channel. Even in the field of information retrieval—which is usually defined in terms of being able to supply information to satisfy a user’s information demands—there is wide recognition that information seeking is an active process involving problem finding and task reconceptualization (e.g., Bates [37], Savage-Knepshield *et al.* [561]).

Taking heed of the limitations of simple transference as a model of comprehension necessarily alters the designer’s stance. Trying to achieve simple knowledge transmission might conceivably be a reasonable stance when the system being studied by the user is already well understood, but it breaks down quickly when less is understood to begin with. In such cases comprehension is necessarily a hands-on problem solving activity. Indeed, it frequently is better categorized as a *learning* or *discovery* activity (e.g., see Corbi [149]), since frequently even experienced maintainers are inexpert in the domain, language, or application type [679], and must work to even understand the nature of the problems they must solve. The design focus in such circumstances can shift towards the active processes of problem solving and problem finding, and thus towards how to support them. This can involve searching for ways to elicit and then subsequently resolve inadequacies in a comprehender’s conceptions. It could therefore involve helping the comprehender to perform experiments to probe and resolve their uncertainties and hypotheses. From the DC perspective, potential design goals to adopt in response includes the provision of a *medium for thought and reflection*. Users can use such a medium to externalize problematic aspects of their understanding so as to reflect upon it as it evolves (see Jahnke *et al.* [331] again).

3. **Comprehension = Internal Knowledge.** The nearly universal aim of software comprehension research is to effectively generate a complete and accurate model of a system within the mind of the comprehender. Success of a comprehension tool is measured by the quantity, veracity, and completeness of the knowledge internalized.

Counterpoint: Distributed comprehension. From the cognitive support point of view, the above design stance is fundamentally misguided. In DC systems, knowledge is always (at least potentially) distributed and jointly processed, albeit to lesser or greater degrees. There *may* be legitimate reasons for why only the user is the appropriate holder and processor of system knowledge, but it must not be accepted *a priori* as an immutable dogma and design goal. Instead, the design stance of redistribution is to seek to *offload* the knowledge from the human and therefore reduce the cognitive burden. The need for full internal representation of the external system therefore represents the *worst possible*

situation—a complete failure to support maintainers with external memories, external knowledge processing, or external media for thought.

If a distributed comprehension design stance is adopted, then attention can then shift towards addressing the balance between external and internal knowledge: determining what needs to be internally held for what tasks, and why; trying to build effective external memory systems; shifting processing burdens onto tools so that comprehension burdens are lessened. Comprehension tools are then evaluated on whether they avoid the need for the comprehender to unnecessarily be aware of or memorize certain facts about the system (unless they are truly needed, in which case they should be, of course, easily ascertained). This goal of reducing knowledge burdens to a need-to-know basis is nearly universally recognized as a principle for language design¹² but it is commonly forgotten when it comes to maintenance tools.

Reclaiming the goal of avoiding comprehension may lead to a more productive exploration of tools. In particular, the DC viewpoint suggests that software development can normally be considered to involve a joint processing of a distributed understanding of a software system. Such a “comprehension redistribution” view therefore attenuates the “comprehension = internal knowledge” stance in important ways. In the distributed comprehension view, knowledge about a subject system is distributed: some is maintained internally, but important aspects are essentially maintained externally, with a certain expectation of overlap between externally and internally maintained knowledge [331]. One rule to apply in design is therefore to minimize the knowledge the user needs subject to the costs of maintaining the distributed knowledge (see Section 3.1.2). This is a very general rule. Butterworth *et al.* [89] utilized a simple version for screen design: “Our heuristic for screen design is that the screen should contain the intersection of what information is necessary and what information is difficult for the user to maintain.” [89, pg. 460].

This is essentially an argument about minimizing human memory use subject to coordination and externalization costs. There are two sides to this coin. On the one side is the issue of knowledge externalization: developers can externalize knowledge in order to forget it, and to allow the tool to maintain it. On the other is the issues of minimizing the developer’s internal knowledge requirements, and of maximizing how much is storage and processed externally. In this view, the goal is to allow the developers to internally maintain a thin veneer of key concepts, and a cadre of indexing knowledge. The developers use this knowledge to crawl over external knowledge sources, much as Simon’s ant crawls over the undulations of the beach. The knowledge maintained internally is also used to negotiate the revelation of needed knowledge from other sources.

This design stance, incidentally, is very much in line with observations of what actual software maintainers do. They rarely, if ever, sit down to fully comprehend a system, they try to avoid deep comprehension if it is unnecessary [609], they work to comprehend parts of programs only

¹²For instance automated memory management in many languages make it possible to be blissfully unaware of most memory allocation and reclaiming issues. The “maximal ignorance” principle also holds for the advantages of ADTs (data layout ignorance), information hiding (internal details), and so on.

as needed [357,595,597], they seem to frequently remember only knowledge to access other knowledge instead of full representations [11], and they tend to want to interleave comprehension with action and only switch back to comprehending when an impasse occurs [64]. In short, in the field one sees a fairly systematic attempt to minimize knowledge of the system and this frequently involves utilizing tools and external representations effectively. But that is only half of the story. Software developers make sure that knowledge is available and accessible externally. They carefully name and organize their systems so tools can exploit this structure [278], they add documentation to code so that they can refer to it when they forget what they wrote the code for, and they write little scripts and programs to look up facts about their programs as needed. They take care of their external memories. Expert comprehenders minimize their personal knowledge of the system while maximizing the joint system's knowledge.

The ideals of automation, RIRT, and internal knowledge construction currently make up much of the received wisdom of the RE/PC research field. Yet each of these design ideals are limited in terms of the way they can set potentially fruitful design goals. These three examples illustrate that the philosophical and theoretical backdrop of DC can provide potentially useful alternative design stances. I wish to mention once again that none of the design stances should necessarily be considered superior. But being able to state the alternatives is important because each stance develops quite distinct goals.

7.2.3 Reifying Design Space

The purpose of the cognitive dimensions framework is to lay out the cognitivist's view of the design space in a coherent manner, and where possible to exhibit some of the cognitive consequences of making a particular bundle of design choices that position the artifact in the space.

– Green and Petre, “Usability Analysis of Visual Programming Environments” [272], pg. 133.

Design Idea

It is possible to reify—i.e., make concrete—a design space by representing design options, constraints, and goals. In terms of CoSTH, this is data redistribution, specifically problem redistribution (D/D/PB). Reifying the design problem in this way creates external structures that can be used as structuring resources (see Section 3.1.3). Thus external structures can act as plans (D/D/P), which can be used to step systematically through the space of design options. All the familiar advantages of such an external structure are possible: it can make the task of following progress easier, it can help ensure that important aspects are not missed, and it can make the exploration more systematic. These three advantages help remedy some of the many deficiencies of craft design, although probably the problem of missing design issues is the most notorious [263,272,370].

Two classic examples of these sorts of external structures are the checklist and the cognitive model. More complicated and inclusive external structures might be contemplated (see e.g., Denley *et al.* [175]),

but for the current purposes checklists and cognitive models are enough to consider. Checklists of design principles or guidelines can lead the designer through a list of questions and issues that would or should be considered during design [401, 525]. One well known drawback of checklists is that they can be overwhelmingly large (lists thousands of items long not all that uncommon). For cheap, “discount” design analysis during early tool envisionment, the lists should ideally be quite short [461]. A cognitive model can effectively complement checklists since models are one of the most compact ways of representing or summarizing many related checklist items. Probably the best known example of a using a cognitive model for just this role is Norman’s multi-stage model of interaction [467]. That model depicts interaction as a cyclic 7-stage activity involving activities such as goal and intention setting, planning, acting, and perceiving. The model is simple and very general, so it can be applied in many situations. It can be used during design [525] or analysis as a compact structure “that can be ‘walked through’ by the HCI specialist with question prompts at each stage that indicate the type of new claim that may be discovered at each stage.” [626, pg. 224]. Although Norman [469, pg. 53] generated his own specific checklist from his model, such models can facilitate less structured and more open ended reasoning by the informed analyst. For instance, Carroll *et al.* [110] explained that they came up with a list of both negative and positive consequences for their tool

... by considering Norman’s stage theory of action For each stage, we imagined the general kinds of psychological consequences an artifact might have, translating those possible consequences into questions that one might ask about the artifact. [110, pg. 194]

In sum, checklists and models are alternative ways of representing structure that can be used during design reasoning. In the terminology of CoSTH, the external structures correspond to *plans* (D/D/P) since the user tends to walk through the structure and consider each in turn.

The challenge here is not to try to supplant any of the existing resources, but to supplement them with the unique capabilities of RODS, HASTI, and CoSTH. To know where supplementation is really needed, it is helpful to first air some of the main problems with current checklists and models. Many problems are already well known. For instance they are frequently too vague or only occasionally applicable. However, there three essential complaints that need to be especially considered here:

1. Many principles and guidelines apply to rather low-level details of interfaces and interaction rather than higher level cognitive issues that are core concerns of complicated tools. For instance Mayhew [401, pg. 496] lists the display design guideline “Avoid using saturated blues for text or other small, thin line symbols.” This is simply too low a level of a view from which to properly address cognitive support issues; ideally guidelines should abstract away the details of the tool and let the designer think at the cognitive system level.
2. Many checklists include principles or guidelines which are not “actionable”, that is, they are non-operational or “inert” in the sense that they provide goals but very rarely indicate ways of achieving them. For example the analysis by Mayhew [401, pg. 44] suggests that one of the design goals implied by the limitations of short term memory (STM) is to “Keep STM storage requirements at any given time to a minimum.” This is a cognitively relevant principle that is hard to argue with in

most situations. But what designers need are hints at the sort of steps that can be taken to achieve it. CoSTH's vocabulary of positive consequences can provide some of these hints. For example redistributing data from STM can reduce the cognitive burden. That is not a full prescription for design, but it fundamentally replaces ends with means.

3. Many, perhaps most, checklists primarily contain items that are concerned with either (1) negative consequences, or (2) minimizing user-tool communication efficiency. Neither of these make it easy to directly state what useful features might be constructed for a tool. My goal here is to concentrate exclusively on reifying the space of possibilities for generating positive consequences.

The first problem implies the need for structures defined at a cognitive level rather than implementational level. The last two problems imply that the contents of the structure should not be *design constraints* (what the designer should not produce), but actually the *action possibilities* available to the designer (steps toward a good solution). Using the terminology of CoSTH, they should redistribute the possible operations of the problem space (D/D/PB/O). They therefore constitute a type of *ends-means* [531] structure: the designer has ends in mind and the structures designate means of achieving them.

One additional note to make is that there are many possible uses of external structures like checklists and models. Although the traditionally cited use is to act as plans to help systematically search the design space, other uses could be entertained. One possibility that is rarely cited is to intelligently drive the research into the design context (see Kaptelinin *et al.* [340]). Prior to any synthesis step there is normally an analysis step in which the design context is understood better: by observing users, analyzing tasks, understand work domain problems, and so on (see e.g., Figure 7.4). In an idealized setting, all of the design context is fully understood before design begins. However the reality is that the design context will only ever be partially understood—usually *very* partially. What parts of this design context does one explore? A list of design possibilities can be used to explore the design context for information concerning how to apply them; with a means one may search for ends to apply them to. For example the designer might initially hypothesize that the users will need support for complicated planning. Knowing that plan redistribution and problem state redistribution are possible means to achieve support, the designer can study users to understand how planning is currently achieved, and to determine what sorts of plans could conceivably be redistributed.

Target Audience and Setting

If you think there are countless ways to organize information, you are not likely to want to try them all during the design process. If you think there are only five, you can imagine trying all of them in several variations.

– Marc Rettig, “Hat Racks for Understanding” [536], pg. 22.

Essentially the same as vocabulary building: trained designers performing early or informal design. The design context is when trying to understand which features to pursue.

Resources

1. **Model-based.** The main resources for building applied design models are HASTI in combination with CoSTH. HASTI is something of a merged version of several modeling methods (see Section 5.1.2). Significantly, the model features are explicitly associated with support principles (see Section 6.1). This means that by “stepping through” HASTI, one can iterate through the associated support principles. In other words, at each stage of considering HASTI features, the CoSTH can be invoked to consider what design options are possible. For example, when considering the Problem panel of the Agent architecture of HASTI, the designer may be prompted to investigate possibilities for redistributing constraints (D/D/PB/C). Overall, HASTI presents a compact catalogue of model features to consider. If a group of designers are brainstorming about how to build a reverse engineering tool they could consider each feature of HASTI in turn to determine how it applies to the reverse engineering situations in mind. They could then turn to CoSTH to consider how to add the appropriate support which relates to these features. An example of stepping through HASTI in this manner is provided in Section 8.1.4, where a model of distributed comprehension is stepped through to examine ways of redistributing the data within the process.
2. **Checklist-based.** It would be a rather straightforward task to generate a checklist of design questions from CoSTH. For example from D/D/PG/H one could simply list items such as “Does the user need to unnecessarily remember past progress states?” or “what task constraints can be externalized?” Such a checklist would just be a linearized form of CoSTH itself, but written in English sentences. Obviously, since CoSTH is hierarchical, the checklist could be hierarchically decomposed. It is not entirely clear that transforming CoSTH into a simple checklist form would be much of an improvement. In fact, this doubt leads one to wonder if checklists are generally poor forms for applied design resources. Since the checklist version of CoSTH is roughly equivalent, and CoSTH is more clearly organized, then any preference for CoSTH over a checklist would suggest that the many existing forms of checklists should ideally be replaced with model-based design theories. That is, either checklists are a good alternative format for theories, or else they might best be junked in preference for approximated design theories.

There may be one reasonable case for preferring a checklist-based representation over the CoSTH: in cases where additional, more specific theoretical apparatus is needed. HASTI contains few psychological details, in part because there is a cost-benefit tradeoff to adding details to the model. A specialized checklist might possibly be a more appropriate place to add details when they are known and relevant. For instance, in certain situations it may be helpful to have a list of perceptual operators (see e.g., Casner [116], Ware [687]) that can be enlisted for reasoning about how to engineer perceptual substitutions. Such a checklist might resemble an expanded version of the “User Profile Checklist” provided by Mayhew [401, pg. 61].

3. **Tabular Worksheets.** Besides the common prose question type of checklist, one may also consider (more) graphical representations like tables. These representations can be created in the form of a worksheet, which would allow the designer to fill in relevant information, or to sketch out design

options. Such a worksheet could collect together related design issues. This collection might be especially well motivated if the design issues involve tradeoffs, or if they address orthogonal aspects of the design. Moreover, certain graphic representations make it simpler to express “vague” or “sketchy” design reasoning: prose form checklists can sometimes be forced into using strict dichotomies (yes/no, true/false) or fixed categories. There are many possibilities for encoding CoSTH options in such a looser, graphical worksheet format. As an example of the possibilities, consider Figure 7.6. The intent of the table is to enable designers to think about and jot down various ideas and concerns. The example table of Figure 7.6 is not meant to be taken too seriously. It is intended to illustrate the general idea of directing designers to important possibilities, and then letting them record thoughts in whatever format seems appropriate. The figure shows an example of how it might be used in designing a debugging system. For instance, the check marks and question marks might indicate prior and future design work. The underlined “bug!” under the “internal” data column might be used as a reminder that the designer is unhappy that the bug the programmer is working on needs to be remembered due to too high of an “up front” cost of externalizing it first. The “remind”, “prior”, and “todo” markings could be related in the sense that the designer thinks a reminder agent for prioritized todo lists would be needed for bug tracking within the “bug base”.

7.2.4 Summary of Design Ideas

Developers of abstract design representations ... should be thinking from the start about the eventual users of their techniques, just as the HCI community urges designers to ‘involve the user’ from initial conception of the software artifact.

– Buckingham Shum and Hammond, “Delivering HCI Modelling to Designers” [86], pg. 312.

This section has explored three possible ways of packaging theoretically based knowledge about cognitive support for use by designers. These were: vocabulary building, design stance construction, and reifying the design space using various external structures. All of these theory repackaging ideas were geared towards improving design reasoning at the earlier stages of design envisionment and analysis. The main purpose of this exercise was to show that CoSTH, HASTI and RODS have potential to create highly applied “bridging representations”. These bridging representations can effectively hide away many of the complications of the theories from which they derive, and yet they can still be used for synthetic reasoning in design. The primary sources of optimism in this project are that (1) they make it possible to reason about support at a high level, that (2) they present design alternatives before artifacts, prototypes, or designs are even constructed, and that (3) they have the potential to improve the scientific foundations of design practice, their approximated nature notwithstanding.

At this point none of these theory packaging ideas are explored in any but the most tentative manner. One reason for this is that there is rather little in the way of prior research to suggest what repackaging methods work well. What little is believed about how to package such theoretical knowledge comes primarily from guesswork by other researchers, although there is some recent activity in analyzing [43,

	DATA		PROCESSING		EXTERN.	PRE-	COSTS	
	← INT.	EXT. →	← INT.	EXT. →	LEARN?	COMPUTED?	EXTRN.	COORD.
GOALS		prior		remind				
PLANS		todo				day's end		
ENDS	<u>bug!</u>						up front	
OPERATIONS		??						
CONSTRAINTS						sets a filter...		
CURR STATE		✓						
HISTORY		✓			bug base			
WINDOW MGMNT								
PAGE REPLCMNT		focus		bug map				
VIEW MGMNT								
WM SET MGMNT	file?							
LOCALIZATION								

Figure 7.6: Hypothetical example of a tabular-form worksheet and its use

272] and testing [86,625,705] theory delivery vehicles. But the fact is that most theories in HCI tend to be used primarily by their authors. This suggests that too little is currently known concerning how to make theories useful and usable to designers at large.

Since it is far beyond the scope of this work to make further inroads on this aspect of the theory delivery problem, there is little that can be done here other than think tentatively about future bridging representations. Even so, I have shown that at least three previously proposed techniques for delivering theory can be reasonably applied to RODS, HASTI, and CoSTH. The prospects for eventually creating useful bridging representations with these seem bright. They show promise that they could provide a valuable toolkit that can be used for FP-reasoning during synthesis. This type of design representation has to this point proven to be very elusive.

7.3 Summary and Implications

From [our] vantage point, three difficulties suggest themselves: (1) we need a better understanding of design, (2) many theories will never be useful for design, and (3) transfer of theoretical knowledge to practice is hard for most engineering disciplines.

– Stuart K. Card, “Theory-Driven Design Research” [93], pg. 506.

In this chapter, I argued that most theory in HCI fails to be especially fit for design. There is a great deal of research that has the *potential* to be relevant to design. However there is a real risk that such research can fail to be fit for design if the nature of design and design reasoning is not taken into account. I argued

that, in the past, the various roles for theory in design have not been properly understood. Because of this, the ways of prioritizing research in the field has been inappropriate. This argument was made by casting design activities as a search in a “fitness landscape”, and then posing the problem of design reasoning as a “gulf of synthesis”. The metaphors were used to argue that some of the most important roles for theory in design is to provide *survey knowledge* of the fitness landscape, and to provide ways of performing what was termed “FP-reasoning”: reasoning *forward* from design goals towards potential design actions that could bring about *positive consequences*. Thus this chapter has essentially argued that a specific *type* of theory is most important: broad-brush FP-theories.

Next I considered how to create broad-brush FP-theories for cognitive support. RODS, HASTI and CoSTH can be used during design, and CoSTH can be used as a type of design theory. But they may not ultimately be very usable in many realistic design contexts. So I outlined three different avenues for trying to transform these theoretical resources into “bridging representations”, that is, representations of the theories in a form that is more fit for design. These three ways of reformulating the theories included: (1) building a vocabulary and matching set of concepts that can help raise the level of designer discourse, (2) articulating alternative design stances based on the principles of DC, and (3) reifying the high level design options using checklists or models. There already exists ongoing work on all of these avenues. The work herein suggested that RODS, HASTI, and CoSTH may be able to provide valuable additions to all of these ongoing streams of study. The work may be considered quite preliminary, but it strongly indicates that design practices need not be so atheoretical as they are now.

In conclusion, it is important to consider the broader implications of this chapter. The problem of creating usable design theories for cognitive support is a veritable juggernaut that simply cannot be conquered in a work such as this. Nevertheless, this chapter was still able to present some initial evidence to suggest that the theoretical resources like CoSTH and HASTI have potential for creating theories or other “bridging representations” that are truly fit for design. Although the steps being presented are small, the conclusions they point to have far reaching consequences for theoreticians. Many excuses have been given for why psychological theory has failed to make inroads in HCI and other design fields. Failure has, more often than not, been attributed to theoretical or methodological weakness, or to the existence of usability problems in the bridging representations. But rarely is it suggested that the problem is that the wrong sort of theories are being pursued. The arguments in this chapter suggest that this may indeed be the case. HCI is driven by a nearly single minded pursuit of finely accurate evaluation theories that primarily serve to help reason about usability problems. It is time to begin working in earnest towards broad-brush survey theories that enable reasoning about positive consequences like the cognitive benefits of tools.

Chapter 8

Application: Where Craft and Science Meet

Cognitive Engineering ... is a type of applied Cognitive Science, trying to apply what is known from science to the design and construction of machines. It is a surprising business. On the one hand, there actually is quite a lot known in Cognitive Science that can be applied. But on the other hand, our lack of knowledge is appalling.

– Donald Norman, “Cognitive Engineering” [467], pg. 31.

Software engineering tools research is at a point where there are a great number of questions floating around regarding tool efficacy, but few scientific answers. Our knowledge about cognitive support, in particular, is grounded almost entirely in craft knowledge and folk psychology. The status quo is not ideal. It would no doubt be better if this craft basis could somehow be replaced with a more solidly scientific foundation. But the scientific basis is currently sparse, contentious, and difficult to apply. If we were to insist on working only with rigorous and widely accepted theories, then we would be exiled onto some tiny islands of knowledge, and be unable to talk about important issues. Shoring up the necessary science base is a long term proposition. Realistically speaking, we shall have to make do with something less than a fully adequate scientific practice for a long time—possibly forever. In the interim, what are we to do? In SE, our tools are rife with psychological implications; simply ignoring them is not a satisfactory option. That tactic merely ensures that explanations of tool usefulness will remain tacit, and be grounded mainly in folk psychology. Then the cognitive support within tools will continue to be hard to compare, difficult to teach, and impossible to test. So *some* measures should be taken to strengthen the scientific basis of tools research and design—some middle ground must be found between staunch, rigorously scientific proof, and folksy, unreliable craft skill. What can these measures be? The answer, it may turn out, might best involve a coordinated attack by tools researchers and applied theoreticians.

One step forward would be to begin making better claims as to the cognitive support offered by our tools. Our current level of claim explicitness does not lead to effective testing and comparison of tool ideas. So it is important to further open up our arguments, make our claims as explicit as possible, and at least try to be diligent at building good arguments for these claims. If we fail to be thoroughly and convincingly scientific, at least we can strive to be *credible*. Credibility demands competent exposure of support claims, and due diligence in justifying them. It is not realistic to insist that *all* cognitive support claims be empirically tested and verified. A single tool may embody dozens of interesting claims, and each claim could take months of work to test; tools research would grind to a virtual standstill. Moreover, from a developer's standpoint, the purpose of theory testing is to ensure that when the theory is applied, the need for validating the results is abated. This is simply good decomposition of research efforts. Tools researchers build tools, make claims about them, and justify them using applied theories. But it is beyond the scope of that research to evaluate or develop these theories. Until such applied theories are available, it seems unlikely that the status quo in SE research will change. Cognitive support claims might still be made, but without a reasonable applied theory, the claims and their justification are apt to be based on folk psychology. Thus *application reaches out to science for applicable theories*.

Changing the practices of tools researchers is only part of the story. The field may in some ways lack a certain amount of scientific respectability, but it is not just to blame only the tools researchers [263]. It is important that applied theoreticians supply SE with the right applied theories, even if they are tentative and weak. Applied theoreticians can use their knowledge of the basic sciences to construct new theories suitable for application. In this way, applied theories are structures that mediate a SE researcher's interaction with the scientific knowledge from other disciplines [257]. In many cases these theories must be abstractions and idealizations of the basic science theories they represent. This is because the applied theories must typically apply in settings that are much broader in scope than the basic science theories from which they draw. To some scientists, this view shall will undoubtedly seem unappealing and unscientific. Nonetheless it is required. Some interim relaxing of scientific hauteur is needed if many of our interesting claims are to be advanced. It is important to provide abstracted theories of broad scope, and it is even more important that they can be used to argue claims of interest to the tools researchers.

These last two points were argued well by Young and Barnard [720], albeit in a slightly different context. They proposed that theoretical works in HCI be "test driven" according to a set of scenarios that act as "sufficiency filters". Given a collection of scenarios, the test driving process

involves taking each theoretical approach and attempting to formulate an account of each behavioral scenario. The accuracy of the account is not at stake. Rather, the purpose of the exercise is to see whether a particular piece of theoretical apparatus is even capable of giving rise to a plausible account. The scenario material is effectively being used as a set of sufficiency filters and it is possible to weed out theories of overly narrow scope. If an approach is capable of formulating a passable account, interest focuses on the properties of the account offered. [28, pg. 115]

Their idea is to weed out theoretical treatments that fail to apply in interesting situations, or that fail to say something of interest to the applicant. The overall aim is to establish a way of achieving a markedly different goal for theoretical works:

...one of the main reasons for using scenarios as sufficiency filters is to try to redress the balance between scope and accuracy, shifting the trade-off point towards a greater emphasis on generality. In fact (and although for saying it we risk being drummed out of the psychological societies we belong to), the empirical accuracy of the scenarios is comparatively unimportant. If what the scenario claims to happen is wrong, then some different statement, but like it, is right, and either way it can serve its purpose of making sure that the theory has sufficient scope to cope with the scenario, i.e. to say something about it. [720, pg. 293]

Their point is that some way of favouring and establishing generality and relevance is needed. Whether the scenario method of Young *et al.* is the best way of doing so is, ultimately, not that important for the present purposes. What is important is that some method is established for placing a high priority on applicability. Right now, the way that applied theories tend to be developed is to take a pet theory and try to extend it to see where it might conceivably be applied [259]. To many tools researchers, this approach seems completely backwards. It might eventually work, but it risks being far too slow and undirected. To them, the right way of building applied theories is to first determine what sorts of theories are most needed and then do the best one can to supply them, tentatively filling in missing knowledge with approximations and generalizations. Theory building, in this light, springs from the important issues in a domain of application. In Young *et al.*'s proposal, the collection of scenarios stand in for the concerns of the HCI practitioners. The scenarios embody the criteria for scope and relevance; they are proxies and advocates for the developers who seek to apply them. Thus *applied theories reach out to application for problems to solve.*

With this point we reach the main issue of this chapter. There is an important relationship between theory application and applied theory building. Craft disciplines depend upon applied theories to effect scientific reform, and applied theory generation is helpfully driven by the problems of domains to which they may apply. It is a mistake to consider these two issues completely independently. Application is where craft and science meet. There may be considerable synergy if applied theories are evaluated for applicability to current research and trends in their domain of application. In particular, it may be helpful to "test drive" theories using scenarios derived from current research tools. A list of points may be drawn up in favour of this approach:

1. Tentative, idealized, and approximated theories are only partly anchored in solid data and experiment. Applying theories derived in controlled settings always brings up the question of whether their application to some uncontrolled situation is warranted, or even valid (e.g., see Wolf *et al.* [711]). In a complementary way, craft disciplines often build a wealth of experience-based knowledge they do not have scientific explanations for. If the theories align well with the received wisdom in the domain, it lends some credence to the theory, and to the validity of its application.
2. In a symmetrical way, if the craft knowledge aligns well with science-backed theories, it adds credibility to the craft knowledge. Reformulating this craft knowledge using a theoretical framework takes an important step towards converting craft knowledge to science knowledge (see Figure 2.1). Doing so does not add new knowledge. Nonetheless, casting it from a different theoretical angle can add another valuable layer of understanding within the field.

3. It is all too easy to under-estimate the difficulty of applying theories in realistic situations. Examples of applying a theory can be pragmatically important for would-be applications. In the present situation, the theories of these past few chapters are likely to be novelties to most SE researchers—that is, they will likely be novices in the application of RODS and HASTI. Examples have been known to be invaluable in other learning situations (e.g., see Fischer *et al.* [215]). Examples bind the abstract theory to accessible, concrete instances. The concreteness may help researchers understand the relevance of the theory. And the theory application can be “reused”—assuming judicious modification—in analyzing tools that are similar to the ones analyzed in the examples. This facility is bound to be easily recognized by any Unix programmer who has ever used man pages to cut and paste example code into their own programs. In a similar way some salient examples of theory applications can serve as analysis templates that an analyst may reuse by “cutting and pasting” from. In addition, such application examples may be inspiring. They can convey a theory’s potential uses to those who did not realize them. It is thus an eminently sensible strategy to promote new applied theories by showcasing their applicability to several of topical examples from a client domain.

For the above reasons, this chapter presents two examples of applying the theoretical resources developed in earlier chapters. They are applied to existing research tools from program comprehension and reverse engineering. The main example is an analysis of a reverse engineering tool called `RMTool` [430]. For the purposes of establishing breadth, a second reverse engineering tool, `Rigi` [425] is considered. Both `RMTool` and `Rigi` are tools which are of current research interest within the SE community, show encouraging promise, and have few explanations of their merits concerning cognitive support. They are thus outstanding sufficiency filters. Analyses of these tools are performed in Sections 8.1 and 8.2, respectively.

These examples of theory application are analogous to—but not identical to—the scenario-driven evaluation method proposed by Young and Barnard. Recall that theories are tools to their wielders (see Section 2.4). In the examples being considered here, scenarios are being envisioned in which researchers (users) are applying theories (tools) to analyze or design cognitive support. The issue is whether the theories can be applied in these scenarios, and what they say. This approach differs from that of Young *et al.* in that the scenarios are not detailed accounts of interactions with the tools; rather they are informal analyses of how or if the theories could be applied in analysis or design situations. Nonetheless, the approach is analogous in that a collection of cases is drawn up with the intention of capturing the salient scope of applicability for the theories.

The result of this exploration into theory application advances the causes of tools researchers in SE. For tools researchers, it recasts experiences within the community in a new theoretical light, adding credibility and a distinguished viewpoint. It also provides salient examples of the sort of analysis that can be performed using HASTI and CoSTH. At the same time, this exploration continues the evaluation of HASTI and CoSTH. It establishes that they can be applied to analyze important tools issues within this research domain. Not only that, but it also shows that they have interesting things to say about them.

8.1 RMTool Example

RMT₀₀₁ [428–430] is a prototype tool that was designed for reverse engineering and software comprehension. Although nothing really precludes it from being applied to other tasks and problem domains (e.g., as *Rigi* has been [643]), the original intent for RMT₀₀₁ was to understand software systems. Consequently that is the application scope adopted in this section.

RMT₀₀₁ is an interesting case to examine for several reasons. Firstly, it is a relatively general tool that can be used to perform several different tasks. This makes it a good “test driving” case. It immediately provides a strong bias against explanations of support that are overly task-specific because the explanations need to generalize across tasks. Furthermore, it is a relatively novel tool within the field. It differs from many other tools in the way it combines visualization and automated software analysis. In terms of tools research, it is important to understand the novelty so that the lessons learned can be easily applied elsewhere [331]. In addition, RMT₀₀₁ has good craft credibility. Its publications contain evidence gathered from authentic and realistic case studies using the tool. And, for a research prototype, the tool is successful and well received; several other researchers have adopted it or similar approaches (e.g., Sidarkeviciute *et al.* [589], Clayton *et al.* [139]). Finally, RMT₀₀₁ is an interesting tool to analyze because the authors of RMT₀₀₁ have a reasonably well-documented design iteration cycle. They iteratively refined the design after they presented prototype tools to engineers in the field. Thus it is possible to travel back in time to earlier design stages to see if design theories could have anticipated the changes the designers made. Being able to doing could suggest that the theories have generative capabilities.

This section unfolds as follows. A brief description of RMT₀₀₁ and its use is presented first (readers are encouraged to refer to its literature for more thorough descriptions). Following that, the main features of the tool are analyzed for cognitive support using HASTI and CoSTH. This theory-based account is compared to the experience reports for RMT₀₀₁. Then, in Section 8.1.4, CoSTH is shown to be able to reconstruct some of the design insights that have been gained through experience.

8.1.1 Tool and Usage Description

RMT₀₀₁ is a “lightweight” tool that is designed largely in the Unix tradition: it can be flexibly programmed, and it combines easily with a variety of other tools. This makes its boundaries and features somewhat difficult to establish. In essence, it consists of five logical entities:

1. *A low-level model (LLM) extractor.* This analyzes source code to interpret its features according to some (possibly weak) semantic understanding of them (functions, variables, function calls, etc.), that is, according to some *ontology* (e.g., see Welty [696]). This extractor produces a low-level model of the code.
2. *A high-level model (HLM) editor.* RMT₀₀₁ does not actually implement this editor: the user is presumed to be able to define the HLM in some manner, either by modeling it in the head, or by using a piece of paper, a whiteboard, etc. The model is specified using a simple text format, so any text editor could be used.

3. *A Map editor.* Maps specify relations between HLMs and LLMs. Maps are represented as text strings, so in reality the editor is any text processor the user finds handy. The key enabling technology is a declarative map specification language. The main advantage of the map specification language is it has a simplicity and flexibility so that it can leverage regularities in the way that source entities are textually represented (e.g., regular naming). The Map and HLM combine to form what may be called an interpretation of the LLM.
4. *A reflexion model calculator.* This is an automatic tool that takes as input the HLM, LLM, and Map, and generates a *reflexion model*. Loosely speaking a reflexion model is a representation of how well the HLM and Map correspond to the actual structure of the system. It is appropriately described as **evidence** for the accuracy of the **interpretation**. This is described in more detail below in the use scenario.
5. *A few tools to investigate reflexion models.* One is a graph visualizer that depicts the reflexion model in a graphical form. This tool allows the engineer to examine the **evidence**. Another tool summarizes the way in which the (declarative) Map is interpreted.

The software actually constituting `RMTOOL` consists of Items 4 and 5.

Context of Use

`RMTOOL` is designed to be used in situations where an experienced systems developer is trying to modify or evaluate a system with which she is unfamiliar. Because of her experience, she has a great deal of knowledge that can be applied when understanding the system. The particular software development tasks being pursued by the user are not that important; it is only required that some understanding is needed of how the system is structured.

Use Scenario

One of the best ways to understand `RMTOOL` is through a scenario of use. A scenario of a software engineer understanding a Unix kernel is used here. This is the same scenario used in the `RMTOOL` literature [428] (although the narrative is modified slightly for presentation purposes), so the reader may refer to that literature for more details. An illustration of the general process of using the technique is shown in Figure 8.1. It goes as follows:

1. **EXTRACT LLM.** The engineer selects and configures tools to extract the low-level source features and relationships of interest. In this example, let us assume that some tool is configured to scan the source code and generate a graph of all functions and their calling relationships.
2. **DEFINE AN INTERPRETATION.** The interpretation summarizes the engineer's current focus and beliefs about the possible structure of the system. Defining the interpretation involves defining two entities:

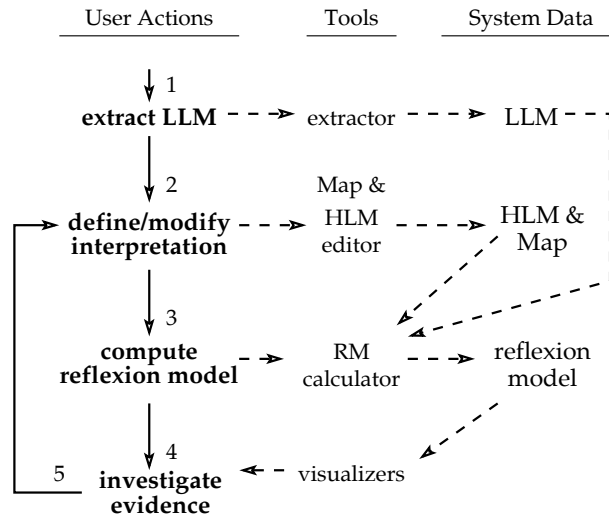
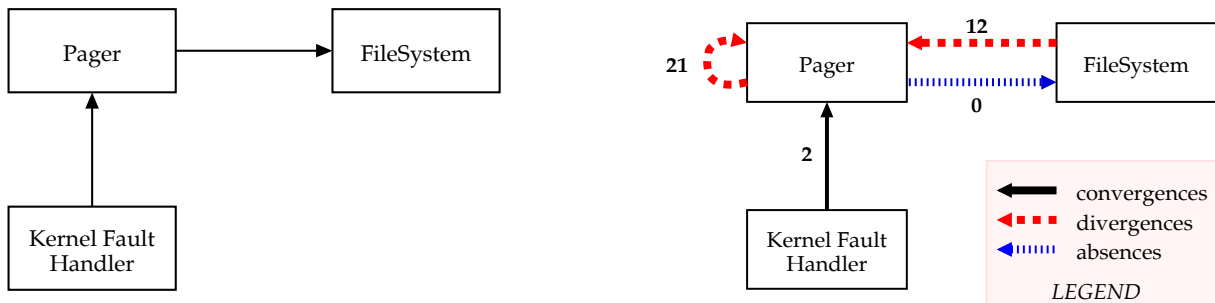


Figure 8.1: Simplified flow of RMT001 sessions

- (a) *Define a HLM.* Using deep domain knowledge, the software engineer generates an model of the *expected* structure of the Unix system being considered. This is an abstract and partial model. This model defines a set of high-level entities and relationships that can be mapped onto a LLM. A simplified illustration appears in Figure 8.2(a). In this diagram, boxes represent logical entities and arrows represent expected relationships between them. In the present example, the relationships are presumed to be calling relationships between modules.
- (b) *Define a Map.* The engineer defines a Map between the HLM and LLM. The intention of the Map is to map HLM features to LLM features, and map relationships between HLM features to relationships amongst LLM features. For this example, assume the Map for the HLM in Figure 8.2(a) maps all functions in the file `pager.c` to the `Pager` node, and all functions declared in the file `filesall.h` to the `FileSystem` node.



(a) HLM

(b) reflexion model

Figure 8.2: Illustrations of RMT001's HLM and reflexion model outputs

3. COMPUTE A “REFLEXION” MODEL. This generates the *evidence*. An illustration of a reflexion model appears in Figure 8.2(b). A reflexion model consists of two main classes of entities:
 - (a) *convergences* — relationships defined between HLM entities are supported by an analogous relationship within the LLM. Figure 8.2(b) shows two convergences between the `Pager` and `Kernel Fault Handler`. In this example, this means that (in the LLM) two function calls occur from functions of `pager.c` to those of `filesall.h`.
 - (b) *divergences, absences* — divergences are relationships between LLM entities that were not anticipated by the interpretation; absences indicate expected relationships were not found in the LLM. Figure 8.2(b) shows two sets of divergences, and one set of absences. The divergences mean that 12 function calls occur which are unaccounted for between functions defined in `filesall.h` and those in `pager.c`, and 21 within `pager.c` itself.
4. INVESTIGATE THE RESULTS. The reflexion model indicates where the interpretation of the LLM is accurate or inaccurate. Since the engineer in this scenario believed the interpretation matched the LLM structure (step #2), this evidence makes implications as to the validity of the engineer’s current beliefs. The engineer is free to investigate the results in any way desired. The main aim of this investigation is to establish the validity of the interpretation, that is, of the HLM and Map.¹ As a rule of thumb, the convergences generally indicate the interpretation is accurate, and the divergences and absences generally indicate problems with the interpretation. More specifically, divergences usually indicate that the engineer did not anticipate some relationships in the LLM, and the absences generally indicate that the engineer expected certain relationships when, in reality, there was none. There are many potential circumstances where this rule of thumb may be broken, so the engineer often has to investigate the evidence to see if it is based on unimportant or unrelated issues. For instance, the engineer may find that the divergences result from relatively minor or unimportant violations in the modularity of a system. She may choose to ignore these exceptions.
5. Go to step 2 and refine the interpretation. The evidence *drives refinement*.

The process is one of (hopefully) convergent evolution. The HLM and Map are iteratively refined until they are found to abstract the actual LLM structure to a satisfactory degree. Investigation of the evidence drives the refinement, particularly investigation of the contradictory evidence (divergences and absences). Since the initial interpretation is a “guess” as to the structure of the system, the length of the iteration cycle is related to how good this initial guess is. In other words, one of the best reasons for using `RMTool` to refine a HLM is that there are inaccuracies in the user’s knowledge, or uncertainty in its accuracy. Thus the tool externally represents *imperfect* knowledge [331]. In addition, the engineer investigates most or all of the contradictory evidence at one point or another. In the end, the engineer gains both an understanding of the system, and an increased level of confidence that her interpretation is valid.

¹A secondary aim is to establish the suitability of the LLM extracted by the tools. Its scope may need to be expanded, for example. Although this activity can be significant, the present analysis focuses primarily on the fixes to HLM and map. Repairing or updating the LLM can be added to the ensuing discussion without changing its character—but that is left as an exercise for the reader.

So far, reported experience with `RMTool` has indicated that this general process is relatively simple, quick, and successful. Knowledgeable engineers do not have trouble defining the HLM. They are able to specify maps after some quick browsing of the source base to determine names and likely candidates. There are no undue difficulties in sorting through the evidence in order to discard superfluous arcs in the reflection model.

8.1.2 Interpreting RMTTool Using HASTI

The `RMTool` literature does a good job of describing many of the advantages of the overall `RMTool` approach. Nevertheless, the literature is thin on advancing clear explanations for the apparent success of `RMTool`. This is not a knock on `RMTool` or its authors, for there is only so much that can be done in a single dissertation and a few papers. Still the problem of adequate explanation remains. To wit, no references are made to psychological principles even though the primary goal is to aid in software comprehension—a task that is obviously laden with psychological implications. Should it not be the case that a tool for aiding software comprehension should be successful, at least in part, as a result of principles from psychological sciences? This subsection provides one possible interpretation of the cognitive support principles underlying `RMTool`.

To perform this explanation, HASTI has to be augmented with a domain-specific model of comprehension activities. Recall that HASTI is a framework for modeling, and it is assumed that this framework will be augmented and added to as required for the area of application. In the case of analyzing `RMTool`, some features of the general comprehension task and comprehension processes need to be added. For the present purposes, this can be done without proposing many modeling novelties. Instead, it is possible to rather directly refine HASTI with a prior model of software comprehension: Brooks' model [76].

Although such a preexisting comprehension model does not lead directly to design ideas, it can be employed as a starting point [685] for further analysis. Specifically, once HASTI is so augmented, it is relatively straightforward to use CoSTH to examine how `RMTool` distributes cognition (Section 8.1.3). The distribution analysis provides a complementary understanding of `RMTool`'s features from the viewpoint of cognitive distribution. In addition, a realistic design scenario is evaluated. The motivating context for the design scenario is provided by published reports of a design iteration. The analysis proceeds to show how aspects of the design improvements can be reconstructed using CoSTH. This is explored in Section 8.1.4.

Brooks' "Top-Down" Comprehension Model

Many years ago, Brooks [76] proposed a model of expert comprehension of software. Although Brooks' original works studied modestly sized programs, recent evidence suggests that some of the basic points generalize to large-scale system comprehension [674]. The central argument behind Brooks' model is that in some circumstances expert software developers will use their extensive knowledge to drive their comprehension processes. Such a knowledge-based process is precisely the context expected for the effective use of `RMTool`. This suggests that if one wishes to understand the benefits of `RMTool`, it may be fruitful to attempt to use Brooks' model as a starting point in analysis. Specifically, the match between Brooks'

model and the context for using `RMTOOL` suggests that Brooks' model might be used to refine HASTI to the point where interesting arguments can be made about how `RMTOOL` supports comprehension. This can be done as follows.

Brooks' model contains three key features: (1) a cognitive task analysis, (2) a suggestion as to the mental representations being used during comprehension, and (3) an analysis of comprehension processes. Each of these can be inserted as augmentations of the Agent model in HASTI.

- 1. Cognitive Task Analysis.** Brooks argues that comprehending a program amounts to generating (or *reconstructing*) a hierarchical mapping of models. He called this "domain bridging" [73,76]. The models start at the domain level and proceed through various intermediate levels such as mathematical methods, or system structure models. Each model consists, in part, of a set of objects and relations; the mapping between models consists of bindings between higher-level objects (or relations) to lower-level objects (or relations). There is nothing particularly unusual in this hierarchical way of modeling software systems, as it resembles many other hierarchical models of software systems (e.g., Byrne [91], Müller *et al.* [425]). Comprehension of a system is posed as a problem of generating an internal representation of this hierarchical mapping, that is, a *mental model*. Brooks explicitly argues that for specific tasks, the required model will be partial, consisting of a partial mapping of relevant aspects. Cognitive tasks involved in generating such a model include: *retrieving* relevant structures from expert memory, *verifying* a binding, *searching* for evidence of a binding, *recognizing* conditions that contradict the current assumptions, and *backtracking* by refining the model. These tasks are described in more detail below when discussing the processing model.
- 2. Mental Representation Model.** At any point in the comprehension process, it is assumed that the mental model of the system is a tentative collection of *hypotheses*. The models at any level are considered high-level hypotheses about the system (e.g., "this is a standard Unix virtual memory system"). Bindings to lower level models are considered sub-hypotheses (e.g., "The file system must be implemented in these functions here..."). As evidence is accumulated about hypotheses, the comprehender's certainty about the hypotheses are assumed to be recorded somehow. In that way, if contradictory evidence is found later, then the comprehender can determine how to refine the model.
- 3. Processing Model.** Brooks' model is called "top-down". Partly this is because it portrays comprehension as a *knowledge-driven* activity; it is also called "top-down" because the hierarchical mapping is built by starting at the high-level domain models and working "downwards" to low-level code models. This aligns with so-called "top-down" software development methods, which propose that programs are to be hierarchically refined in an analogous manner. Brooks argues that somehow comprehenders will develop high level hypotheses about the meaning and structure of the system being studied (e.g., because of the program's name). These set up the gross hypotheses which are hierarchically refined until bindings are considered verified. Verification of a hypothesis is performed by searching for confirmations or disconfirmations. Sometimes this search fails, or encounters contradictory evidence. This leads to a failure to verify a hypothesized binding. This causes backtracking

to occur, resulting in refinements to higher level hypotheses. Processing occurs until the full (or partial) hierarchical model is constructed and confirmed to the degree required.

There are other significant aspects of Brooks model, but they are not used in the following.

Refining HASTI with Brooks' Model

Most aspects of Brooks' model can quite easily be integrated into the Agent model of HASTI. Let us call this integration "TD-HASTI". It has the following features:

1. The mental model refines the `Process` panel, becoming its primary contents. History within this panel includes prior states and revisions.
2. The `Control` and `Agenda` panels contain pointers into the mental model. In the case of the `Control` panel, the pointers indicate that high-level hypotheses (model guesses) organize (plan) the exploration of sub-hypotheses. In the case of `Agenda`, they indicate that attention is divided between various related hypotheses, often on different model levels.
3. The main cognitive tasks of Brooks' model are assigned to separate agents. For the time being, relatively few considerations will be made as to SRKM stratification or D2C2 strata. For now, let us merely assume that the memory and recognition processing occur at the rule level or below (since these do not seem to show up in verbal protocols [76]), and the remaining can be assumed to occur at the knowledge level or above.

It may be helpful to visualize the resulting model; a sample is given in Figure 8.3.

Although this is a simple extension of the Agent model, using other aspects of TD-HASTI can lead us to consider some properties of this comprehension process. Using the Agent-to-Hardware map of HASTI, it is possible to anticipate both the hypothesis (`Progress`) and sub-hypothesis exploration state (`Agenda`) will be subjected to restricted STM capacities. This suggests that understanding complicated software or complicated hypotheses will be difficult because they will overload the limited STM resources. It suggests that pending hypotheses will be occasionally forgotten and systematic hypothesis investigation will suffer planning failures. The SRKM taxonomy suggests that backtracking, searching, and hypothesis verification processes will seem effortful since they will not normally be done through skilled actions. In addition, the D2C2 analysis generally suggests that coping strategies may be employed to reduce the number of hypotheses explored at once.

8.1.3 Tool Analysis Scenario

TD-HASTI is essentially a disembodied, unassisted model of comprehension. It is tacitly assumed that comprehension occurs within a context of simple tools (e.g., a simple editor or set of code printouts). Even so, Brooks' model is compatible with the projected applications of `RMT001`. The trick to analyzing cognitive support in `RMT001` is to re-conceive of `RMT001` use as a distribution of TD-HASTI. This exposes a number of cognitive supports. This re-interpretation of `RMT001` is done by viewing its features

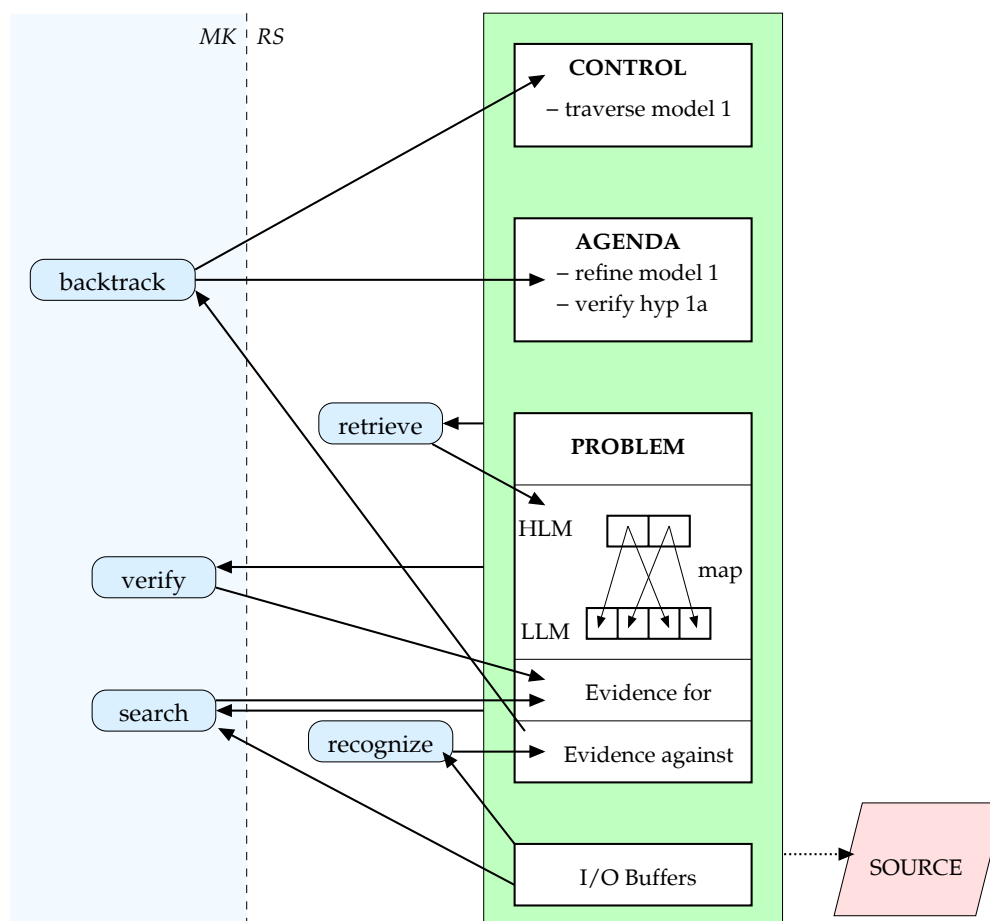


Figure 8.3: Refining the Agent model with Brooks' comprehension model

as mechanisms for distributing the data and processing modeled in TD-HASTI (hypotheses, verification, etc.). This can be done as follows:

1. *Data distribution: hypotheses.* A HLM can be identified as a high-level hypothesis in TD-HASTI. The Map can be considered the collection of sub-hypotheses (bindings) associated with the model hypothesis. Thus the construction of an *interpretation* is RMT₀₀₁'s version of domain bridging.² The key step in RMT₀₀₁ is thus the externalization of the hypothesis to be refined (D/D/PG/state). As Brooks notes, initially this is extracted from expert knowledge, but the comprehender is uncertain as to its veracity.
2. *Processing distribution: hypothesis verification.* Hypothesis verification is partially distributed by RMT₀₀₁. This involves externalizing the *search* and *verify* cognitive tasks from TD-HASTI (see Figure 8.3). Computing the reflexion model uses the *interpretation* to effectively *search* for possible

²A small aside is required here. Brooks' domain bridging is more complicated than the example interpretations in the RMT₀₀₁ literature, which are high-level structural interpretations. However these are still consistent with Brooks' theory; besides nothing seems to preclude RMT₀₀₁ from being applied more generally to what Brooks had in mind in terms of domain bridging.

evidence that may confirm or disconfirm the hypothesis. The arcs in the resulting reflexion model constitute potential sources of evidence for the accuracy of the sub-hypotheses (bindings). Both confirmation evidence and disconfirmation evidence is generated. This is only a partial distribution of *verification* because many of these arcs must be investigated by the engineer in order to determine what implications they make (the reflexion processing is not capable of determining evidence relevance for the user). However even this saves a great deal of work because otherwise these would have to be individually navigated to and examined. Because hypothesis exploration in TD-HASTI depends upon the diligence and capability to remember pending goals, the wholesale processing of the hypotheses also suggests that evaluation of hypotheses will be more systematic and thorough.

In this view, RMT001 distributes the computation of TD-HASTI without significantly altering its essential qualities. To visualize this, consider the illustration in Figure 8.4. It shows a rough illustration of what RMT001 use might look like if it were drawn using a *virtual blackboard* architecture (uninteresting features are omitted, as in a partial HLM). The idea behind this virtual blackboard architecture is to encapsulate the details of how the user and computer share memory and coordinate processing. From the viewpoint of the virtual abstraction, blackboard memory is directly shared between user and computer, and processing on it can be done by either human or computer. From this abstract view, RMT001 merely shuffles around the location of data and processing.

Using the above analysis, the main cognitive support that RMT001 provides is the distribution of the hypotheses and their processing. Externalizing the hypotheses can reduce cognitive burdens and increase the complexity of the hypotheses explored. The external processing means that cognitive processing loads are reduced. New task burdens are introduced, of course: externalizing the interpretation, invoking tools, etc. These are *overheads* in the form of device and cooperation overheads. These burdens can be tolerated because of the cognitive support they provide. This overall evaluation of cognitive support in RMT001 aligns nicely with the analysis of Murphy *et al.* The main difference between the two accounts is that here the considerations stem entirely from (1) an application of a pre-existing task analysis, and (2) a theory of cognitive support applied to this analysis. The two evaluations are completely independent but eminently compatible.

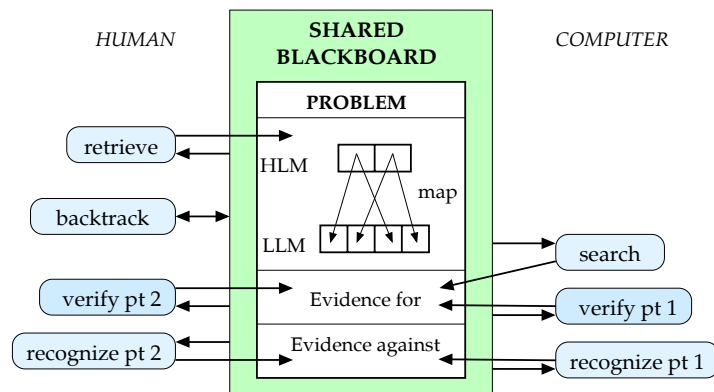


Figure 8.4: "Virtual blackboard" illustration of RMT001 processing

Other arguments about the cognitive support in `RMT001` can be made using HASTI and CoSTH, although they are less significant than the above, and are not supported by `RMT001` documentation. For example, consider the visual presentations provided by `RMT001` for accessing the *evidence*. The main facility provided for this is a source summary that is displayed when an arc in the reflexion model visualization is clicked. The summary displays the LLM features that the *interpretation* is bound to. This presentation serves to collect together (i.e., localize) references to these LLM features. If the source code is viewed as external knowledge held in external memory, this localization can be viewed as a way of collecting together these memory references so that paging (scrolling, file loading, etc.) need not be performed. That can be considered an example of automating memory management operations (D/P/VH).

The above investigation is an informative first “test drive” of HASTI and CoSTH. This tool evaluation “scenario” gives a strong indication that HASTI and CoSTH are broad enough in scope, and say interesting things to the analyst. `RMT001` is a real tool of current research interest. HASTI is quite easily refined so that it applies to `RMT001`. CoSTH is able to recast many of the arguments about `RMT001` in terms of being instances of cognitive support.

8.1.4 Design Envisionment Scenario

Early prototypes of `RMT001` lacked some of the features that were added to later versions in response to user feedback [428, 431]. This type of iterative development is not in any way unusual within HCI or SE. For instance, software development tools are frequently refined to better match the work practices of individual organizations, users, or tasks (e.g., Singer *et al.* [596], Gillies [237]). The question is, can design theories be used to help anticipate some of the requirements for tools so that the necessary features do not have to be discovered after the tools are delivered to the users? It is impossible to fully answer this question with a retrospective analysis of prior design histories: hindsight, as they say, is 20:20. But the results can be suggestive. `RMT001`'s published design history provides a good case in point.

Given the analysis in the prior subsection, other types of support may be considered. It is sensible to suggest that analyzing this distribution could turn up several points the `RMT001` authors originally missed. One particular aspect of the above analysis may quickly draw an analyst's attention: the partiality of the processing distribution. Specifically, if one adopts a *design stance* (see Section 7.2.2) which places a high priority on distributing problem solving data, then this stance can lead to a search of the design space (see Section 7.2.3) for additional ways of doing so. The following presents a summary of how this sort of design reasoning might reasonably have progressed using CoSTH.

Thinking About Distributing TD-HASTI

Although much of the evidence analysis is automated by `RMT001`, it is only partially checked: the user needs to sort through the *evidence* provided, and to refine the *interpretation* appropriately. This is an example of where human and computer must coordinate their efforts. Specifically, they must coordinate over the resolution of the evidence. The computer does not fully evaluate the evidence and relies on the human to be able to distinguish irrelevant and important cases and to account for them. The user makes a series of decisions regarding the salience and meaning of the various bits of evidence. If the

computer is to take this into account in further processing, the user must work towards incrementally externalizing the relevant decisions and interpretations. At the same time, it may be fine to say that the user should keep track of their decisions about evidence, but the `Hardware` model (state is mapped onto limited memory) suggests that this is apt to be forgotten. Thus it should often be helpful to offload this knowledge. Moreover, if the computer can display the externalized state of evaluation, this can in turn help regulate and coordinate the user's actions. So in addition to being an external memory, it can act like a shared memory used for coordination (the computer records and displays action cues for the user). Finally, note that the *retrieval* and *backtrack* steps are unsupported. It may pay dividends to consider how to distribute these sub-process.

The preceding analysis leads to a few suggestions of cognitive support:

1. `D/D/PG/state`. Memory for the state of evidence evaluation could be offloaded. This includes decisions to ignore particular LLM features, and to remove from consideration those features that have already been understood as being important and accounted for in the *interpretation*. Another term for this data might be "evaluation tracking state".
2. `D/P/VH/VM`. Evidence processing (reflexion model calculation) can take externalized evaluation tracking state into account. Reflexion model processing can use the decisions to ignore evidence to improve the display of evidence. A prime candidate is to filter the ignored evidence and thus pare down the *effective working set size* of the problem state. This would reduce the contention for working memory (i.e., reduce display clutter).
3. `DBPS`. The display could be made to indicate the system's progress in evaluating evidence. Specifically, the evidence visualizing programs can use the externalized exploration state to display the visited/unvisited status in some form. In combination with the elision of unimportant links, this implies that the unexplored plan states (i.e., hypotheses yet to be evaluated fully) are externalized (`D/D/PG/state`). This externalized plan-execution state can structure ensuing sub-hypothesis resolution. In many cases, these unexplored plan steps will form an agenda of pending sub-goals (`D/D/agenda`). The user can then employ a strategy of using the externalized agenda to determine the next sub-hypothesis to examine. This type of externally-cued activity is a form of display-based problem solving (DBPS).
4. `S/S/SP`. DBPS is assisted if perceptual cues are available to determine what action to perform next. For instance, it might be helpful to visually highlight the arcs in the reflexion model visualization which contain unevaluated evidence. The user might be able to use visual search to determine which evidence to examine next.
5. `O, C, S, SP, SA` (backtalk): Generating an initial HLM may not be a simple matter of "recall" since the user may not be able to simply "transcribe" their internal understanding of the system. Such knowledge externalization difficulties are well-known in the knowledge engineering community. The implication is that the model externalization process itself might be a problem-solving step in many circumstances. The general notion of backtalk (refer to Section 6.5.2) can be applied to the

HLM editing tool as a way of helping users reflect upon their own knowledge by encouraging backtalk. `RMTool`'s current implementation currently allows only a roundabout form of feedback. More immediate feedback gained while externalizing the model may be very helpful. In particular, the modeler might be able to use their skills at visually judging the correctness of a model's diagram as it is being externalized. In this view, the externalization process can become a minimally-planned skill-based loop, with feedback serving to quickly correct anomalies. Little is mentioned in the `RMTool` literature regarding initial HLM generation, other than that it was found that users easily externalized a working initial hypothesis. For instance, it is not mentioned whether the users worked the HLM out beforehand on some suitable medium such as a whiteboard or scrap paper. These would provide a reasonable medium for backtalk.

Regardless of the specifics of the examined cases, the above analysis makes suggestions as to other possible usage contexts. In cases where the modeler is uncertain as to what structure might be employed in a system, a modeling tool may be important for externalizing a good working hypothesis. In such cases, the modeling tool may closely resemble knowledge modeling tools used for brainstorming and knowledge acquisition (e.g., see Kremer [363]). In certain other cases, extensive modeling may be considered to be a prerequisite step before trying to map these models onto code (e.g., Clayton *et al.* [139]). In such cases, the support provided by an externalization medium may especially come to the fore since the complexity of the modeling process may require such an external memory.

6. `D/D/PG/state/history`. Backtracking can be expected when hypotheses fail, or when unanticipated interactions in the LLM are found. Backtracking may involve unrolling (1) decisions about the features of the HLM or Map, (2) commitments to ignore certain aspects, or (3) judgments and interpretations about supportive evidence. Since all of these problem solving states are partly externalized, mechanisms for recording their past states or derivations may be helpful. Specifically, it would offload some of the memory requirements for the user. In addition, if rollback methods are made available (undo, version management systems, etc.) then some rollback processing (`D/P`) is distributed.

The updated `RMTool` contains some features that implement the above cognitive distributions [428, 431]. In particular, the second version added *tagging* and *annotation* features. Tagging features allow the user to "tag" specific arcs in the reflexion model, indicating that they are to be considered temporarily irrelevant. The visualization engine uses these tags to elide the **evidence** arcs. If the interpretation is changed such that the relevance of that **evidence** might change, these tags are undone. The annotation mechanism allows the user to externalize whether and how an arc is resolved. The visualization engine subsequently indicates this resolution status visually (by displaying the fraction of **evidence** resolved for any given arc on the diagram). Although this might not enable visual search for the next goal to examine, it still enables a form of display-based processing. Thus the tagging and annotation facilities effectively implement the first three cognitive distributions suggested above. Experience has shown that these features are significant aspects of the overall `RMTool` approach.

The fourth, fifth, and sixth considerations (visual search, backtalk, backtracking) are currently poorly

explored by `RMT001`'s implementation. The point of including them is to illustrate the relatively systematic way in which design considerations can be brought up. What was needed was a task analysis and a collection of cognitive restructurings provided by CoSTH. The result reconstructed three supports offered in the design iteration, but it also suggested three additional design choices. This suggests that CoSTH is not limited to purely retrospective analysis.

8.1.5 Summary and Implications of RMTool Analyses

An analysis scenario and a design scenario were described for applying HASTI and CoSTH to the problem of rationalizing `RMT001`'s design. The analyses generated for these scenarios demonstrate that HASTI and CoSTH have powers to reconstruct central rationales of `RMT001`'s design. The analysis involved understanding how `RMT001` can reorganize cognition in comparison to a relatively unsupported cognitive task analysis. Support analysis consisted of using CoSTH theories to reconstruct arguments for specific design features. Design envisionment analysis consisted of “walking” through CoSTH to see how different ways of distributing or otherwise supporting the cognitive tasks could be performed.

It seems important to note how HASTI and CoSTH interact with existing cognitive models. HASTI was shown to be able to be sufficiently compatible with an existing cognitive model to let HASTI-guided support analysis to proceed. This is an example of being able to use prior scientific work to jump-start tool analysis [685]. At the same time, the application of CoSTH highlights the limitations of prior cognitive modeling methods. For instance, consider the problems of applying a cognitive model of software comprehension such as that of von Mayrhauser *et al.* [675]. The model by itself cannot capture the important roles of the externalized hypotheses or the automated reflexion analysis. The main reason is obvious: the model by itself does not explain how `RMT001` changes cognitive processing for the better. It is assumed that the model holds regardless of the tools employed. Thus the comprehender would still be modeled as posing hypotheses and trying to answer them, but the contributions of `RMT001` would need to be accounted for in some other—probably *ad hoc*—fashion. Past experience with these sorts of models have not led to similarly methodological design argument reconstruction. For instance, in the past the model of von Mayrhauser *et al.* has primarily been used to critique toolsets that did not provide adequate information access methods [665], or which over-constrained program investigation methods [619]. As a sufficiency filter for analytic theories, `RMT001` clearly demonstrates that HASTI and CoSTH encompass a pragmatically important scope of application, and can say interesting things to tool analysts.

The design scenario is also enlightening because it props up the drive towards theory-based design. The main purpose of a design theory is to suggest design options based on an understanding of a problem context. CoSTH clearly could do this for `RMT001`. Recall that I quoted Singer *et al.* [597], who asked “...how does knowing that programmers will sometimes use a top-down strategy to understand code ... inform tool design? It doesn't tell us what kind of tool to build... .” I concur wholeheartedly. What is required *in addition* is a design theory similar in form to CoSTH. This small design scenario demonstrates that a top-down model of code comprehension (TD-HASTI) is helpful indeed—but only if considered in combination with design theories. Design theories can not only provide interesting theory-backed explanations of the usefulness of tools, but they appear to hold promise for being able to suggest useful

tool features.

8.2 Rigi Example

The prior section on `RMTool` has established the general procedure for applying HASTI and CoSTH to analysis and design scenarios. This section considers the reverse engineering tool `Rigi` [425]. The purpose of looking at `Rigi` is to reinforce this view using another example, and to expand on the breadth of scope to show that the theoretical frameworks are not limited to analyzing `RMTool`-like tools. This example further emphasizes the independence of the theory-based derivation by presenting an analysis of program comprehension in advance of the tool descriptions.

8.2.1 Bottom-up Comprehension: BU-HASTI

Brooks' model of top-down comprehension can be "reversed", that is, the hierarchy of mappings can be constructed from the bottom up. This means that the processing will be primarily *data-driven*, that is, it will be based on inductively interpreting lower-level inputs. The implied context is that the comprehender does not have suitable knowledge to generate good hypotheses about the code. She must therefore begin interpretation by looking at the code and inducing useful abstractions. Many blackboard models assume that processing will be a mixture of top-down and bottom-up interpretation (see e.g., Carver *et al.* [113]). Consequently a bottom-up model, BU-HASTI, can be defined as a relatively simple extension to TD-HASTI. In addition, other studies and models of bottom-up comprehension can be employed to fill in the details to TD-HASTI.

The following analysis enlists the studies by Pennington [497, 498] in order to augment HASTI. As was the case for Brooks' model, these prior works have primarily considered intra-modular comprehension, yet there are strong hints that they also generalize for inter-modular comprehension (e.g., see von Mayrhauser *et al.* [674]). The research of Pennington will be used to supply a cognitive task analysis. Only certain aspects of these studies are used here. Pennington's work contains some important arguments which are mostly irrelevant for the current analysis. For example, she makes a strong and important distinction made between external structures (stimuli) and the internal representations that are induced. That issue is muted in the following.

TD-HASTI can be "reversed" with the help of Pennington's models as follows:

1. **Cognitive Task Analysis.** The process of comprehension is modeled as the incremental construction of a (partial) hierarchical mapping as in TD-HASTI. This involves a number of cognitive tasks, including: *recognizing* familiar structuring idioms (e.g., by the idiom of assigning functions to files by their logical module), *cross-referencing* related items, and *inducing* higher-level abstractions based on the cross-referencing. The abstraction-generation task makes use of the properties of the source and cross-referencing relationships to make judgments about the best interpretation of the source. Specifically, the cross-references help determine the functional relationships between entities that work in close concert [498, pg. 326-327]. These are then interpreted, or "chunked", as forming cohesive, functionally-related units. To remain consistent with prior terms used, these abstractions can

all be referred to as HLMs.

2. **Mental Representation Model.** This is analogous to TD-HASTI, however the binding evidence is of a different sort: cross-references made between entities in the LLM combine to either support or contradict any particular abstracted interpretation of them. Specifically, functionally-related and closely cross-referenced entities tend to be interpreted as forming entities within HLMs. In a similar manner loose bindings provide evidence to separate entities within the HLM. Thus inter-entity cross-referencing densities in combination with functional cohesiveness are the primary evidence used to decide on mappings from LLMs to HLMs. For instance, in SE terms, program modules are normally decided upon by collecting together functionally-related items such that the “coupling” between modules are minimized. All HLM abstractions are considered tentative collections of *hypotheses* that need to be refined based on evaluating the cross-referencing evidence.
3. **Processing Model.** Bottom-up processing means that the external cross-referencing structures to a great degree drive comprehension processes. When evidence about clichéd structuring is found, standard structuring schemas (TS schemas in Pennington’s model [498]) can be recalled to organize the ensuing interpretation. For instance, if function location conventions are used in the source base (e.g., the `pager.c` in the `RMTool` example), regularities in the source base can be used to generate an initial abstraction hypothesis (i.e., the engineer might suppose initially that the functions in `pager.c` together form a page management module). Iterative cross-referencing is presumed to incrementally and hierarchically build up and refine the resulting HLMs.

8.2.2 CoSTH Analysis of BU-HASTI

An analysis of the possibilities for restructuring cognition in BU-HASTI can proceed in a similar manner as for TD-HASTI. The following points can be made:

1. *D/D/state*. STM limitations in the `Hardware` model suggest that only for small HLMs is it easy to hold the hierarchical abstractions in internal memory. Some sort of external representation can be postulated to offload this data. Significantly, this representation will need to be able to represent HLMs, LLMs, and the evidence for their structure (cross-references).
2. *D/P (structuring)*. When looking at a system, structuring clichés known to the engineer may be recalled. These can be used as a basis for constructing an initial HLM hypothesis. The construction of the initial HLM might therefore be externalized. This would involve either some mechanism for defining the structure (a declarative map), or for representing how to construct it (procedural methods).
3. *D/P (evidence)*. Evidence evaluation can be partially externalized by formalizing a decision procedure or metric space for evaluating cross-referenced entities. In the case of a formalized decision procedure, this corresponds to externalizing clustering mechanisms (*D/P/GD/abstraction*).
4. *S/S/SP/cluster (evidence evaluation)*. Evaluating cross-referencing evidence requires the engineer to somehow integrate all the various pieces of cross-referencing data and then to mentally

create and compare alternative groupings. A critical problem is to recognize which functions might belong together. According to SRKM, it would help to be able to substitute some of this work with specialized perceptual operators. A visual cluster perception operator is a candidate if visual cluster perception can be made to substitute for the analysis of related functions. This would require mapping cross-referencing evidence onto a suitable visual presentation (i.e., classic cross reference tables would likely not be enough).

5. *D/P/VH/VM* (layout). Processing may be done by the computer to lay out the visual presentation in order to invoke cluster perception.
6. *SP, SA* (backtalk): If the visual presentation can be manipulated manually, the user can use cluster perception as visual feedback in order to try to arrange the presentation into better configurations. That is, the visual interface becomes a medium for generating feedback to those skilled in its manipulation.

8.2.3 Matching Features in Rigi

Rigi can be interpreted to implement each of the six distributions suggested in the prior subsection:

1. *D/D/state*. *Rigi*'s primary external state is the "resource-flow graph", or "RFG" [426]. This logical entity is represented using a formalism of the same name. It can represent hierarchically decomposed abstractions and interconnections between lower-level entities. The RFG's contents are made available through a variety of visualizations.
2. *D/P* (structuring). Knowledgeable engineers are able to script *Rigi* to make use of recognized existing structures (such as file naming conventions and file hierarchy information) [712] (clusters).
3. *D/P* (evidence), *D/P/GD/abstraction*. *Rigi* defines and allows various metric spaces for computing initial abstractions. These include business rules, and cohesion/coupling metrics [426]. As in the case of evaluating *RMTTool*'s evidence, it is still up to the user to verify the interpretation by examining the evidence closely, if necessary.
4. *S/S/SP/cluster* (evidence evaluation). Cluster perception is made possible in *Rigi* by representing the entities and cross-referencing relationships using nested box-and-line graph presentation format [427].
5. *D/P/VH/VM* (layout). *Rigi* provides several layout methods for organizing nodes in the RFG visualization [426]. Some layout algorithms are chosen to map logical clustering to visual-spatial clustering.
6. *SP, SA* (backtalk): *Rigi* provides many different methods for manipulating the graph, including spatially arranging nodes [426,427]. This allows users to move entities around in order to get visual feedback.

8.2.4 Summary of Rigi Analysis

Rigi is a complicated research product. It contains features other than the ones listed in the previous section, and there are many rationales for its design other than the ones covered here. This does not detract in any way from the fact that rationales for many of its features can be derived solely from a theory-based account of cognitive support. Several of the published rationalizations of Rigi's features are consonant with the foregoing analysis (perhaps most particularly Müller *et al.* [427]). Experience reports appear to bear witness to the advantages expected (predicted) by CoSTH. As a result, Rigi provides another example of where applied theory and field experience are mutually reinforcing.

This analysis of Rigi helps to outline a lesson about the possibilities of design theories. The thing to note is that the design analysis may precede tool feature elaboration. Whiteside and Wixon once asked HCI theoreticians "Where does the design come from initially?" [702, pg. 365]. This scenario answers they may spring from quite systematic application of design theories. It is therefore tempting to suggest that Rigi's basic form could have been originally developed by using relatively unremarkable theoretical resources. What was required was a reasonably straightforward task analysis based on existing theories, and a design theory similar to CoSTH. As far as I am aware, Rigi was not constructed by such a systematic application of design theory, but by basic ingenuity in combination with long and incremental exposure to the problems that software developers face. This is not to suggest that many of the basic design insights did not in some way underscore Rigi's genesis. On the contrary, Rigi's authors seem to have been quite aware of most or all of the essential insights all along. That is, they appeared to use a craft design theory of sorts. The contrast made in this section is between explicit and implicit design knowledge and methods. The extra lesson here is thus not about Rigi *per se*, but about the prospects for using design-based theories. It suggests that the design of cognitively supportive tools might not have to be a result of heroic insight, but of relatively methodical analysis and application of design theories.

8.3 Summary

This chapter paints a picture that may be interpreted on at least two levels. The first level deals with the benefits shared mutually between applied theory evaluation and theory application. From the applied theory evaluation side, it demonstrates that HASTI and CoSTH are broad enough in scope to deal with high-level tool design issues from a relevant tools research discipline. RMTool and Rigi are research tools that are of current interest. Not only are the theories shown to be broad in scope, they also "say" things of interest to researchers in the discipline. They provide a second interpretation on the experiences within the field. Particularly interesting is that they can reconstruct many of the perceived advantages of the tools.

From the theory application side, this is a welcome sight. The applied theories provide mediated access to underlying science theories. Tools research has plowed along without many of these sorts of science-backed accounts. Certainly, significant attempts have been made in the past to apply various theories to explain development tools (e.g., Choi *et al.* [129], Redmiles [532]). However these efforts have been infrequent and have been, so far, less than satisfying. In addition, the intuitions about these tools

have been building along similar lines within the field, but they have to this point lacked a suitable and powerful enough framework with which to articulate these ideas. This chapter demonstrates that latent psychological claims existing in research tools can be articulated. This means that they can be explicitly tested for, both for evaluating existing ideas, and when designing new tools and variations of them. Right now, little else is known about how to systematically evaluate such tools. Currently we pit tools against one another and pick through the secondary evidence of performance data. Being able to articulate psychological claims about tools makes it possible to test the claims more directly. For instance, do perceptual judgments in *Rigi* substitute for knowledge-based reasoning? This is (at least in principle) a directly testable hypothesis. The deeper implication is that tool building expertise within the field can be made more credibly founded by exposing such claims and encoding them in independently validated theories. Thus, if domain wisdom is the raw materials for tools knowledge, then applied theories appear to be the most likely catalyst for scientific reform; credible claims about tools are the precipitate. Unfortunately there is a veritable sea of tools to analyze: one chapter's worth of claims analysis will make only the smallest of advances. Nonetheless, sometimes the symbolic first move is the most difficult to make.

The second layer of interpretation for this chapter concerns the relationship between applied theorizing and tools research. Specifically, this chapter provides a strong indication that they are best performed hand in hand. The tools research disciplines set a motivating context for theory breadth and content, and the applied research discipline provides a parallel and longitudinal accumulation of experience suggesting how well the theories generalize to practice. So far as I can tell, this close relationship has not received its fair share of attention. Building applied theories has for the most part been seen as a problem of taking existing theories and trying to adapt them so as to be able to apply them as widely as possible (e.g., see Barnard [28]). The alternative presented here is that a research discipline comes up with a list of interesting questions it wants to answer, and appropriate applied theories are pieced together from whatever is most suitable.

From the other side of the fence, transforming a craft discipline into a scientifically-backed discipline has been treated more or less as a process of accreting tiny, point-form results. For instance, one particular theory from cognitive science may be imported to explain some small aspect of some particular tool. A small experiment is run to "validate" this explanation. This is like painting a large picture by dropping small points of paint here and there. The alternative presented here is of using broader-stroke, integrated applied theories to mediate access to the science base. This is akin to drawing a painting by starting out with a fuzzy—but panoramic—outline of the result, and then incrementally filling in details. Starting out with integrated theories might help avoid the problem of getting the individual drops of paint to come together as a coherent picture. This has been a problem for cognitive science in the past [446]. More importantly though, the intermediate results are much more likely to be useful.

Before concluding this chapter, I find it impossible to resist including a small historical analysis comparing theory development and tool development across fields. A review of the history of some of the various works referred to in HASTI, CoSTH, and this chapter reveals some curious facts. A selected 20-year timeline appears in Figure 8.5. Some of these publication dates cluster around the same year, so in some cases they are merely placed in rough proximity. A number of relevant building blocks for HASTI

and CoSTH are shown on the right hand side. The rightmost column indicates some of their main contributions to those theories. On the left hand side are listed the early genesis dates for the tools of this section. Also listed on that side are a few reference points and background works. For reference, the citations for both the left hand and right hand sides of the timeline are listed in Table 8.1, in order from top to bottom.

As can be seen from the timeline, many of the critical parts of HASTI and CoSTH were clicking into place in the fertile years of 1983–1989. Analytic frameworks similar in many ways to HASTI and CoSTH could conceivably have been constructed soon afterwards—all before (or at about the same time) as the first version of *Rigi*, and a decade before *RMTool*. It is grossly unfair—but tantalizing—to speculate that it would have been possible to create and refine *Rigi* and *RMTool* using theory-based design and analysis methods. Likewise, the rationale for *RMTool*, or a tool very much like it, could have been conceived through theory-based analysis at about the same time; certainly more than 15 years ago. This is certainly not brought up to suggest that these tools could or would have been constructed earlier. Rather, the point is that, as a field, we could have been applying similar theory-based methods to design and analyze tools for quite some time. This chapter has been a *long* time in coming. I mentioned in Chapter 1 that integrated theoretical accounts of cognitive support are overdue, that it is important to begin collecting functional theory together in order to analyze and design tools. This timeline, I suggest, strongly argues the case.

SIDE	REFERENCES OR SECONDARY REFERENCES
left	[71], [73], [299], [465], [418], [76], [94], [526], [466], [296], [92], [567], [498], [375], [267], [374], [257], [61], [470], [116], [667], [472], [727], [320]
right	[594], [432], [582], [564], [423], [741], [623], [469], [101], [307], [767], [766], [430]

Table 8.1: Summary of references in Figure 8.5



Figure 8.5: Timeline of main ideas and tools

Chapter 9

A Field Study of Cognitive Support

RODS, HASTI and CoSTH are proposed as an intellectual toolkit for tools researchers and developers. They are candidate theories and frameworks built specifically for the purpose of raising and answering questions about cognitive support. To fully realize this potential, researchers must have access to effective methods for using the toolkit, either analytically or empirically. Chapters 6, 7, and 8 have developed ways of applying RODS, HASTI, and CoSTH analytically. In this chapter, the aim is to begin answering the question of how to apply them empirically. The theories encode knowledge; the question is how to wield it empirically to yield answers regarding tool design questions. By this, I mean using the toolkit in the context of a laboratory or field study for the purposes of exploring cognitive support issues not directly answered by the theories. Theories are not all-powerful, and they often lead to important questions rather than answering them outright. Data gathered from careful observation can be a powerful way of augmenting the toolkit. If suitable techniques are made available, then theory-driven empiricism may have the potential to be another weapon in the arsenal of methods for systematically designing and engineering cognitive support.

Applying cognition-related theories empirically in such a way is a relatively unusual proposition. In cognitive science and HCI, the main reasons for going to the lab or field are to validate or verify theories, to gain some new understanding suitable for deriving new theories, or to otherwise refine one's knowledge about the context for design. For instance, consider the research programme advanced by von Mayrhauser *et al.* [678]. They proposed to apply a cognitive model of software comprehension in field studies. Their purpose for proposing this was to further validate their cognitive model, and to use it to understand new things about comprehender behaviour. This type of empirical study programme is not at all unusual. Even proposals to study tools typically place a priority on discovery and theory building rather than importing and applying theories as is (e.g., see Redmiles [532]). Theories and models are generally proposed to be *developed* empirically, and *used* analytically (e.g., GOMS [94]). Being *used* empirically to answer directed questions about tools is a relative rarity for these sorts of theories.

Nevertheless, there are sometimes valid reasons for combining theory and empiricism. What if you do not question the validity of a cognitive support theory, but it does not say enough to you analytically? Then one may be forced to observe tools being used—in a “user study”, for instance. In a user study,

good theoretical frameworks are important assets to have (see Section 2.2.2). A theory can guide the interpretation of activity, and it can focus both observation and analysis. Thus different types of theories have been proposed as a way of guiding user studies. For example, the cognitive dimensions framework can be helpful in directing attention to important issues in HCI [54]. Furthermore, theories and models are also useful for interpreting HCI so that tool implications can be understood. For instance, von Mayrhauser *et al.* [665,669] used a cognitive model of software comprehension to interpret a comprehender's needs for information. These needs could be compared to what information a tool easily provides, thereby rendering judgment on the tool's efficacy. Thus theoretical frameworks can be useful even if they do not give answers by computation and prediction alone. Theories can guide and assist in either focused or relatively open-ended exploration. A key question is therefore to understand the potential of various models and theories: to know how they can be used and what sort of information they can yield in observational studies. Theories and models are tools for empiricism, but they need a user manual—an operator's handbook.

The main aim of this chapter is to explore possibilities for using HASTI and CoSTH empirically. Essentially then, this chapter is primarily an exploration of *techniques* rather than *theories*. The goal is to provide techniques which are useful for exploring cognitive support questions whenever they pop up during tool design and evaluation. The particular focus is on understanding possibilities for leveraging the theories to yield directed insight from observations of tool use. The overall objective is to make it possible to perform routine, efficient, and reliable theory-based investigations of cognitive support in realistic situations. In other words, this chapter is engineering-oriented rather than discovery-oriented. Consonant with this goal is the choice to focus on efficient and "lightweight" empirical techniques.

The search for lightweight empirical techniques is an important challenge to meet. In many research and development contexts, timetables are measured in months, weeks, or even days. This puts tight constraints on the sort of empirical techniques that can be used. Many existing theory application methods break these constraints. Many are known to be incredibly tedious and expensive (see e.g., von Mayrhauser *et al.* [678]). Sometimes, the needed experiments or fieldwork are lengthy, costly, and difficult; other times, the analysis methods become arduous. As a result, many empirical techniques are at odds with the schedules and predilections of tools developers. Recently, suggestions for reducing the costs have been advanced, especially for making it simpler to collect and process data from fieldwork. Examples of such lightweight techniques include variations of "discount ethnography", "rapid ethnography" [408], and motivated rapid observation techniques [596]. However little work has been done to apply reasonably fine-grained cognitive theories such as CoSTH and HASTI in a suitably lightweight manner. To a great extent, the possibilities of theories like CoSTH have remained untapped.

Towards these goals, an exploratory field study of software developers was performed. Although it may sometimes be useful to study cognitive support in laboratory conditions, field studies can be effective also. Furthermore, a field study provides a "litmus test" for lightweight methods: chances are, if an observation technique can be applied in the field, it could be applied in the lab too, where conditions can be controlled more effectively. Thus the field study was performed in order to serve as a basis for exploring different ways of applying the theories; it was intended to be a "testbed" of sorts. The study provided a corpus of data on which to set the theories loose. In addition, the field study provided an

opportunity to experience firsthand some of the challenges of collecting and interpreting data while using HASTI, RODS, and CoSTH as guiding frameworks. As an added benefit, the study also provides a way of establishing ballpark evaluations of the validity of HASTI and RODS. Although this possibility is not ignored (see Section 9.5), it is not the central focus. The central focus remains exploring the potential for engineering-oriented observation and analysis.

The choice to collect observations from practicing developers in the field is strategically and philosophically significant. Hutchins [320] has made a strong argument that we must study cognitive processes “in the wild” if we are to study genuine ones. We get unnatural behaviour if we study it in the lab. This contention in more detail in Section 9.1. For now, suffice it to say that “the wild” is one place where the natural use of cognitive support is likely to be observed. Hutchins’ call to study cognition the wild is appropriate for discovery-based inquiry: for discovering situation-specific facts about a particular cognitive ecosystem, or for building novel theories. The present chapter has a different aim. It is an attempt at employing existing theory to shed light on uncertainties regarding the support provided by tools. Thus, although one may choose to work in the wild, it may be with the specific intentions of bringing an existing theory into it, rather than returning home with a new one. This focus attenuates the call to study cognition in the wild.

This chapter unfolds in five main sections. First, in Section 9.1, the field study design and objectives are described, and my initial experiences are outlined. This section begins to examine how to make observation techniques lightweight. Specifically, ways of using HASTI and CoSTH to perform in-situ “shadowing” observation methods are considered. Second, a particular observation session is chosen from the corpus of collected data. This session will be used as a test case for applying HASTI and CoSTH. The test case is described in Section 9.2, and an analysis is performed to enumerate some of the cognitive support involved. The specific cognitive support issue in question will be support for repairing errors in programs. Third, Section 9.3 explores techniques for coding and analyzing the test case. This analysis is used to drive an exploratory evaluation of the cognitive support found. The effect of these first three sections is that they provide a context for evaluating some research scenarios. Since the purpose of the study was to explore methods that others might use, the study is documented such that data collection and analysis methods are highlighted.

Section 9.4 begins to consider scenarios in which these techniques might be applied. Two tool development scenarios are considered, both of which are adaptations of scenarios first presented in Chapter 2. The purpose of these scenarios is to argue that the analyses presented in Sections 9.2 and 9.3 may be useful for answering certain support related questions. The first scenario is of a tool analyst who has built a cognitive support claim and then wishes to find some evidence for its validity. Theory in this case allows the analyst to make claims, and empiricism acts as a check on the theory application (or as an inexpensive alternative to applying more powerful theories). The second scenario is a scaled-down adaptation of the design scenario from Chapter 2. The scenario is of a cognitive support designer who wishes to understand the specific support types that need to be investigated. Theory in this case acts as a heuristic for examining specific design options, and empiricism acts to fill in parameters and values missing from the theory. In this manner it fuels considerations of tradeoffs. The section illustrates that methods from the first three sections have application in both of these scenarios.

Section 9.5 provides a brief evaluation of how the field observations comment on the validity and usefulness of HASTI and CoSTH. Although this study was not conducted as a validation experiment, it is still reasonable to present an analysis of what evidence has been uncovered in support of the theories. At the very least, this is merely responsible reporting. Reporting even highly preliminary indications of validity may be helpful for future researchers who wish to expand the validation studies. The chapter is rounded out with a brief conclusion section.

9.1 Field Study Description

As a means for developing a testbed for exploring theory application methods, a field study of software developers was performed. This section incrementally introduces the study. Only the basic mechanics and outcomes are described here; later sections contain more detailed theory-based analyses of the resulting data. The motivations for the study are considered, the study design is outlined, the observational techniques that were used are reported, and an initial summary of the study is described.

9.1.1 Motivation and Background

The science of mind is, I assert, not the science of what the mind can do in a laboratory situation artificially rigged to make it relevant to one of our theories, but what it does in a situation naturally or artificially rigged by itself and its culture... Given this view, the only way to find out what the mind is like, and to stumble across questions that need to be understood more fully, is to study it in its natural habitat. Since the mind does not have any set natural habitat, we need to study it in the habitats in which it frequently finds or wishes to find itself. At the moment it seems to wish to find itself in constant interaction with computer systems, so this is where we must track it down.

– Thomas K. Landauer, “Relations Between Cognitive Psychology and Computer Science” [369]
(1987), pg. 19–20.

The motivations for this study are generated initially from considerations of the plight of tools developers, not of theory developers. When studying or designing software development tools, theories and frameworks are important for building and analyzing support claims. But all analytic techniques have their limits, and when these limits are reached it usually is necessary to augment them with empirical methods. When theories cannot help answer a question about a tool, observing somebody using the tool can be extraordinary helpful. However going to the lab or field to observe users raises many challenges (see Chapter 2). Cognitive support is a slippery thing to observe and to understand. It easily goes unnoticed because it removes problems instead of creating them. One possibility is that cognitive support theories can assist in making the most of observation effort in the field or the lab (Section 2.2.2). A field study was performed in an attempt to assess this possibility. Why was a field study chosen as a testing ground? Why not a more controlled laboratory study? Three answers are offered here.

The first reason for a field study is a concession to the needs of this particular dissertation. This work proposes theoretical resources derived from many different cognitive science and HCI studies. Many of these are generated and validated using laboratory studies. When these lab-generated theories are incorporated into any applied theory, there always remains the issue of whether the resulting theory works in real-world (non-laboratory) settings. Do they generalize? Furthermore, there is always the question of whether the models apply to significant behaviour actually occurring in the field. Are the theories and models relevant? It may not be possible to answer these questions completely here, but what is abundantly clear is that another controlled laboratory study will do little to increase confidence in the generalizability or applicability of the theories in real-world development contexts. Going to the field is a preemptive defense against the charge that the theory will not be usable in realistic tool deployment situations (e.g., in commercial settings). Later, in Section 9.5, the theories are evaluated for their capability to work well in the field. This is a limited study. Limited evidence for validity could be collected. The impact of this evidence would have been significantly blunted if the task context were manipulated in the laboratory to generate the expected behaviour. In this study, industrial strength tools being used by semi-professional developers were examined in the field. It is hard to imagine controlling the observation conditions any less. If the theories strike paydirt here, it makes it all that more believable that they will also apply in many real-world design settings.

The second reason for attempting a field study has more to do with the problems of tools developers than with theory evaluation. It is reasonable to expect that CoSTH and HASTI could be used in both controlled conditions and in the field. Performing field studies generally presents the greater technical challenge for finding the relevant behaviour and collecting suitable observations of it. For instance, in a field study it will be normally impossible (and unnatural) to get subjects to wear eye tracking head-gear (e.g., Crosby *et al.* [152]), or to set up a horde of video cameras to capture behaviour on a forced task in a restrictive task environment (e.g., Pohthong *et al.* [512]). Unfortunately, instruments like an eye tracker might be one of the few good ways of determining whether or not certain cognitive supports are being used (e.g., if perceptual substitution is effective). Therefore a field study quickly brings up many limitations for “lightweight” theory application. Perhaps more importantly, it guards against attempting studies requiring unnatural task conditions or unrealistic data gathering techniques. The observational constraints inherent in field studies better reflects the realities of tools researchers. It keeps the exploration of realistic observation techniques honest.

The final reason for attempting a field study is that it is likely that much of the behaviour that tools developers wish to study is only available in the field or in highly realistic simulated situations. Although realistic work scenarios can often be set up to elicit ecologically valid behaviour [348, 402], it is often difficult or expensive to do so. For this reason, many developers may need to observe tool use in the wild. This third reason for choosing a field study is the most significant from a theory user’s point of view, and is therefore worth discussing in more detail.

Advantages of Field Research for Studying Cognitive Support

For a period of months, the programmers generic skills in programming design, comprehension and debugging are made ineffective. The programmers attempts are thwarted by their lack of facility with the concepts and details of their new language, the software system, and the myriad of ancillary programs.

– Lucy M. Berlin, “Beyond Program Understanding: A Look at Programming Expertise in Industry” [44], pg. 22.

Up until recently, models of cognition in software development have been primarily disembodied, history-free models. These tend to portray comprehension or development processes as essentially universal and purely mental activities. In this way they ignore the details and capabilities of the tools, and fail to acknowledge the importance of many of the interactions with these tools. They also remove from consideration differences in individual problem solving knowledge and skills. Although it is true that different models for novices and experts are often proposed, developers within any category are painted with the same brush. It is assumed that they can be sampled at random with little concern about their expertise in tool use. Laboratory studies or constrained studies with packets of printouts are often considered sufficient for exploring such cognitive models. To be sufficient for this purpose, it must be assumed that the environment will not impact the aspects of behaviour being modeled.

Although tool use is downplayed in most cognitive modeling work in the field, tool use is precisely the focus for tools researchers. Any model that cannot make statements about how tools affect behaviour will be of limited use in studying them. In this sense the universality and tool-independence of the models are of great concern. In fact, the questions of universality and tool-independence go hand in hand: universality of the models is broken by tool and environmental dependencies. Specifically, experts are adapted differently to their specific environments: they have individual “ecological expertise”. So when one studies expertise, it is not simply a matter of studying a generalized expertise that is universal across all capable software developers. Expertise is adapted; it has an environmental component. Several different aspects of adaptations are important when studying tools.

One aspect of adaptation is the development of fluid skills which enable higher-level problem solving to flow naturally. When a user is experienced with using a tool, skills can often be employed, and this frees cognitive resources to be used to pursue hard problems. For instance, consider the experience of constructing a tricky program within an unfamiliar editor. Figuring out how to work the editor becomes a task requiring conscious problem solving, and this is likely to intrude on the programming problem solving. For (primarily) this reason, classifications of expertise in HCI are usually not made according to a single dimension of skill (novice vs. expert). For example, one popular distinction is made between domain and tool expertise [218], with both types having an affect on overall levels of expertise.

Another aspect of adaptation is that expert use of many tools requires strategic knowledge. Often this strategic knowledge develops only after a great deal of experience in applying the tools to various tasks. This has shown to be true even for simple forms of support. For example Bhavnani *et al.* [47,

49] demonstrated that even relatively simple tool features are ignored by users unless special training is provided. Their result holds for simple forms of cognitive support; we can expect that the importance of ecological expertise will only be magnified for less routine problem solving. Then, deep strategic skill can be involved. The type of skill in question was described in Section 3.1.1. An example of such skill is the flexible and strategic uses of `grep` by certain experts [380,696]. Although `grep` is a simple tool to describe, it requires a great deal of strategic knowledge to use it for cross referencing and impact analysis. Another example is the clever use of type checking to perform impact analysis as described by Cardelli [96]. In Cardelli's example, a developer would need to call on a skill for recognizing that the compiler could be mobilized for solving the problem of finding program dependencies. If this skill were not developed, other methods would doubtless be pursued.

A third aspect of adaptation is the number of local ecosystems of expertise involved in software development (what Hutchins called "cognitive ecosystems" [320]). In some cases the ecosystem is relatively uniform across developers, and therefore tool expertise can also be relatively uniform. For example, sometimes an organization will foster an enculturation process of using tools similarly. This can be done through formalized mechanisms (e.g., tools training programs [282]), or through more informal means (such as apprenticing junior developers with senior ones [44]). Thus corporate culture and training regimens can lead to some similarity in tool expertise. However, frequently the ecological expertise of software developers is created through long years of self-directed exploration. And often the tool sets used in this process differ greatly. Certainly this accords with my own observations and experience. In other jobs and professions (e.g., airline pilots [321,323], navy crews [320], radiologists, air traffic controllers, etc.), learning how to solve problems with the available tools is significantly systematic and explicit, or based strongly on apprenticeship. In contrast, software developers—like many other tools users—learn substantially by doing [98]. That is, they learn how to use the tools available as the needs arise, and by performing work during the learning process. Learning thusly will be haphazard. As a result, individual offices may develop vastly different tool use cultures, and these will affect the tool use strategies of its workers [49]. Furthermore, real (professional) development environments are rich and complex. Undirected learning in this flexible space leads to idiosyncrasies [190,275, ch. 3]. The specific history of problems tackled may lead to a different repertoire of tool uses [275,550]. Program editors are a simple but prime example: programmers become greatly attached to specific editors, and this leads to efficiency and strategic use of the individual command sets. Their proficiency on complicated editing tasks drop quickly with unfamiliar editors. Some developers may make heavy use of a programmable text editor, while others may become sophisticated users of programmable tools like `sed` or `awk`.

A fourth aspect of adaptation comes from the specializations that developers make to their own environment. Developers are known to adapt their environment by writing scripts and macros, customizing their windowing environments, and organizing their information space (file systems, documents, etc.) strategically [550]. Berlin [44] found this to be true in her field study of programmer expertise. Effectively, the developers construct their own local ecosystem in which they work best. Often times this leads to a familiar condition in which one developer will find it significantly awkward to use another developer's environment and setup. Such cases of co-adaptation of user and environment help to blur the distinction

ADAPTATION	PROBLEMS FOR LABORATORY STUDIES
low level skills	expert users typically are needed
strategic knowledge	expert users typically are needed
local cognitive ecosystems	population sampling is problematic
environmental customization	poor transfer of expertise to lab situations

Table 9.1: Potential problems caused by dependency of cognitive support use on adaptations

between humans as a topic of study and human–tool systems as a topic of study. Although expert developers may be quick to adapt to different environments, that is not the point: in unfamiliar terrain, they will not be able to easily use their tools and problem solving strategies that they are familiar with.

Implications for Empirical Studies of Cognitive Support

All of these aspects of ecological dependence pose potential problems for studying cognitive support. A table summarizing these problems is presented in Table 9.1. To actually observe fluid use of cognitive supports, the subjects may need to possess (1) deep familiarity with the tools, (2) skills for using them efficiently, and (3) problem solving knowledge for using them strategically. The first requirement is challenged by idiosyncrasies in tool expertise and by the number of local cognitive ecosystems that developers work in. Furthermore, for this knowledge and skill to be elicited, the tool environment and task demands may need to closely match those of the field. In some contexts, some of these problems are less prominent, or they may be compensated for in some way. In many other cases, the easiest route for satisfying these conditions is to observe tools as they are used in actual work settings. This need plays out differently for practitioners and researchers.

Practitioners are interested in developing cognitive supports that work in real client settings. They often work with quite well-polished and comprehensive tools; sometimes there is an established client base. For practitioners who develop commercial tools, the main implication of adaptation dependency is that they may often need to visit client worksites in order to understand how the tool is used in practice. Only there can the many contextual factors which impact tool use be observed. Many times the importance of apparently mundane artifact features can only be realized by watching how these features are actually used [311].

Tools researchers are normally interested in developing techniques that are generalizable. They frequently deal with research prototypes on which perhaps nobody can truly be called an expert. For many researchers, the main implication of adaptation dependency is that they will usually be unable to find subjects with enough experience on the tools or the tasks. Simple training sessions will not normally be enough for the subjects to use the tools effectively and strategically [64, 359, 618]. Even for relatively simple programming tasks, a week of learning will likely not even be enough to elicit informative uses of a development tool [407]. Even if experts can be found, it is often very difficult to get much of their time to do a study. Watching them do their own work is much less demanding on subjects.

In sum, field studies are frequently desirable for studying cognitive support, and the challenges they present make them good sufficiency filters for any techniques being proposed. For all of these reasons a

field study was thought to be the most worthwhile method for exploring cognitive support theory applicability.

9.1.2 Study Design

This subsection describes the design of the study, including its stated objectives, format, participant recruitment procedures, study session procedures, and interview session procedures.

Objectives

The study was designed as an exploratory investigation. In a more directed study, there is usually a clearly articulated study objective or a hypothesis to test. In this case, the objective was not to discover something novel from the observations, but to show that cognitive support could be effectively analyzed using HASTI and CoSTH. In addition, it was assumed that much would be learned about the possibilities and difficulties of fieldwork just by performing such a study. During the time of the study design, CoSTH and even HASTI were in an early prototype form. Although I had some ideas as to how they would eventually be applied, it was not entirely clear at that time exactly how this would be done. For all of these reasons, there was considerable uncertainty throughout the study design phase.

This uncertainty impacted the study design by creating its own objectives and problems to overcome. Because it was unclear at the time which data needed to be collected, I was concerned with recording a rich collection of data that could be later analyzed at leisure. I wished to have a corpus of realistic observational data that could provide a suitable testbed for later analysis work. Thus the main objectives of the fieldwork were to establish what was possible under limited time constraints to gather rich and authentic observations of developers using their tools. An important secondary objective was to try the “shadowing” method of in-situ coding to determine its advantages and pitfalls. This is described in more detail in Section 9.1.3.

Study Format

The study took place over a three week span. Participants were instructed to contact me at suitable times when they were prepared for me to observe them. The observation sessions could be scheduled for specific times in the future, and I made it clear that I could normally be ready to collect the observations with short notice (during the full time of the study, I was located at the work site within moderate walking distance of all participants). The participants were encouraged to initiate at least two sessions, but were informed that any number of sessions from zero to a maximum of five were allowed. Only one session per day was permitted. An optional interview session was also encouraged. Because of this format, each participant was engaged in two or more sessions: a recruitment session, one or more study sessions, and (optionally) an interview session.

The participant-initiated observation technique was an important part of the study strategy for two reasons. Firstly, it helped limit observations to relevant development activities. All of the participants

were expected to perform a variety of activities during any given day or week. By initiating the observation times, the participants themselves could ensure that the sessions being observed would be “productive”, that is, be filled primarily with tool-using development activity. Since I wished to collect data from authentic tasks rather than assigned tasks, the alternative would have been to follow them around for a great deal of time. Doing so would have made it much more difficult to gather the detailed data that I was able to capture. Secondly, the participant-initiated protocol was expected to be beneficial for the participants. They would have control over the sorts of activities that they permitted me to observe. This allowed them to select activities they were comfortable with me observing (or permitted to let me observe). It also allowed them to schedule the observation sessions when it was convenient for them. In my informal discussions with the participants, my impression was that they were satisfied that neither of our time would be wasted.

It should be noted that, in other circumstances, this self-selection setup could be criticized as suffering from a participant selection bias. For example, it might be charged that the participants select only “showcase” tasks which they felt comfortable and competent in. In an exploratory study such as this, this threat to accurate behaviour sampling was considered inconsequential. Moreover, from the observations I have collected, I can confidently say that none of the participants selected “showcase” activities: they all appeared to be solving ordinarily challenging activities in which they displayed a considerable degree of difficult problem solving.

Participants

Six participants were recruited from a large research institution. All of the participants were involved in ongoing research and development work. Participation in the study was in all cases with the expressed knowledge and approval of the participants’ supervisors. Participants were unpaid volunteers. All were pre-screened to ensure that they had at least two years of programming experience, were working on a programming-related project that was at least 1500 lines of code, and had at least four months of experience in the general programming environment they were working in.

Recruitment Protocol

With the exception of recruiting one subject directly through a personal contact, recruitment was performed in a two stage process. First, a supervising manager of a team was approached, and the general purpose of the study was explained. Permission to carry out studies on employees was then sought. It was considered important to first receive permission to recruit participants for inclusion in the study. In addition, the supervising manager was asked to recommend potential participants based on the aims of the study. Specifically, the supervisor was asked to recommend active software developers.

At the initial recruitment meetings with participants, the general aims of the study were explained, and the methods for collecting observations were described. All potential participants were advised that their participation was entirely voluntary and that they could withdraw at absolutely any time. They were informed that they could request at any time to have any or all collected data destroyed. It was emphasized that the study was concerned with their tools and how they were being used, rather than

on their performance or their particular work. They were told that the main aim of the study was to help future researchers build better tools. They were told that each session would last at most 40 minutes. They were also informed that they could schedule an informal interview session after finishing the observation sessions. At the recruitment session they were also provided a short questionnaire that was intended to establish simple facts about their backgrounds. The text of the recruitment invitation, research description card, survey questionnaire, and participant instruction card are presented Appendixes A, B, C, and D, respectively. Each participant received a copy of these four items during the recruitment session.

A key part of the recruitment meeting was the provision of instructions for establishing a suitable time for an observation session. The participants were informed that the goal of these sessions was to observe tool use during program maintenance and program understanding. They were told that I would like to observe them for between 30 and 40 minutes, but certainly no longer than 40 minutes. They were instructed to contact me via email to schedule subsequent sessions; they were informed that I would respond to their email as soon as possible. They were also allowed to schedule an initial session right then. Four of the six participants chose to do so.

Observation Session Description and Protocol

The main part of the study consisted of a sequence of one or more observation or “study” sessions. These took place in the participants’ normal workplaces. Before starting the first session, recording software was installed on their workstations (see below). In addition, survey responses were collected, if they were completed. At the beginning of the first session, instructions were given as to how to generate a verbal report. The instructions closely matched those given in Appendix E. These are slightly modified instructions taken from a standard experimenter’s handbook [256]. On subsequent study sessions I asked if they wished to review the instructions for verbal report generation. None did.

At the beginning of each session I configured the recording equipment (see below), and I ensured that the workplace was private by shutting doors as necessary. All participants worked alone in their normal office space; none of them shared this office space. Working alone in this manner appeared to be a routine activity for all participants. During the observations, I would interrupt the participant only if they fell silent for an extended period of time, upon which time I would gently remind them to think aloud. This happened twice.

Interview Sessions

The study protocol included an optional post-study interview session. Only one of the participants chose to make use of this post-study interview. Instead of such a formal interview setting, all participants except one chose to chat with me informally before and after study sessions. I made notes of these conversations afterwards. These less formal notes have so far proven to be valuable, however the interviews may have been better if they were pursued more vigorously, and if they could have been conducted after reviewing the collected data first.

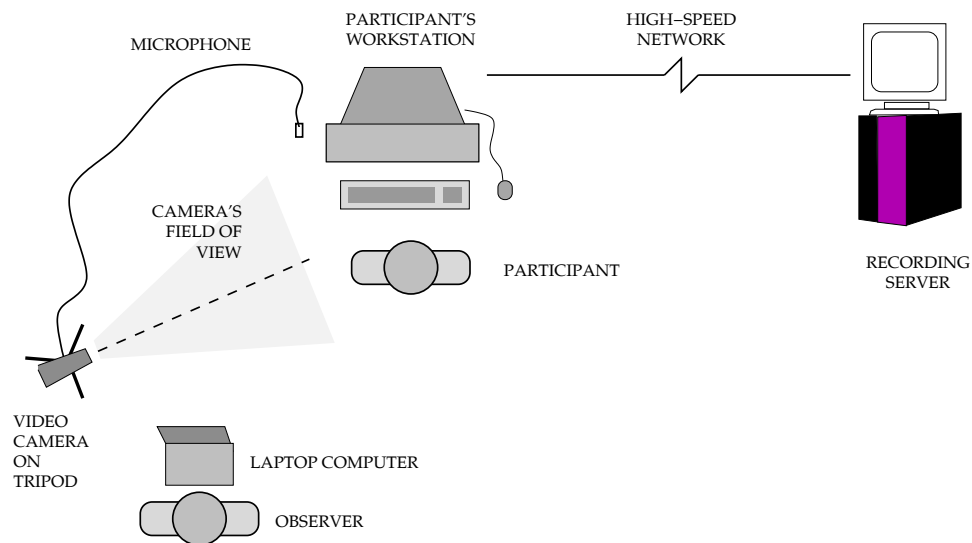


Figure 9.1: Typical observation configuration

9.1.3 Observation Methods

Data were collected using automated recording techniques, in combination with notemaking. Figure 9.1 illustrates the typical configuration for each study session. I was positioned to the side of the video camera, and both myself and the camera were outside the participant's line of sight.

Video and Audio

A video camera was trained on the screen of the participant's workstation, with a wide enough field of view to capture both the screen and participant during normal interaction with the computer. Usually the user's screen filled roughly 1/3 the resulting field of view. The intent of the video was to capture activities such as looking to read printed documentation, pointing to the screen, etc. Although some basic features of the computer screen could be made out, the goal of the video record was to focus on the participant rather than the computer screen. A sensitive and separate microphone was placed nearby the individual. It was able to create a suitably crisp recording of the subject's keystrokes, mouse clicks, and verbalizations—all except for the softest mumblings. It was a lapel-type microphone (but it was not clipped on), so it was unobtrusive.

Computer Screen Recording

A detailed recording of the computer screen was made by using specially modified instrumentation software. The recording software is reminiscent of other commercial screen capturing products which have been utilized in similar observational settings (e.g. Graham *et al.* [252]). This verbatim capture of the screen was critical to have for later study. Screen sizes for the participants ranged from 1024x768 to 1152x884, and the monitors often had high refresh rates. It is impossible to capture details of screens with this resolution well on standard videotape—certainly not unobtrusively. It is worthwhile describing this software

since it is a key enabling factor in this sort of field study.

The recording software was a specially modified version of the free (GNU-licensed) software called VNC [540]. “VNC” stands for “Virtual Network Computing”. Its original purpose was to remotely view or control one computer from another, including over relatively slow network links. This software runs on multiple platforms, including Unix, Windows, and Macintosh computers. The portability of the software was important for this study since the participants used a mixture of Unix and Windows NT. The software consists of two parts, a *server* and a *client*. The server is the machine to control or view; in this case, the server was the participant’s machine. Standard, unmodified VNC server software was installed on each of the participants’ machines. The VNC software is small, and it imparts only a small load on the machine in many common computer activities.

A Unix version of the client software was modified to record the server’s (participant’s) screen data. The recording software was kept running on a networked Linux machine with a large hard disk (see Figure 9.1). All data from the participants’ machines were thus saved automatically to central location. This obviated the need to recover the captured data from the participants’ machines; it made it impossible to forget the data there, and it made it extremely difficult to run out of disk space while recording. All recorded screen updates (e.g., a mouse movement) were stamped with the date and time. Although time stamping was done by the recording server and therefore had inaccuracies¹, the advantage was that off-the-shelf VNC server software could be installed on the participant’s machines, and that it did not matter if the participant’s machine’s clock was inaccurate.

Reaction to installing the software was uniformly favourable. The server software was easily installable (and removable, although no participants asked me to remove it afterwards). All six participants were interested in the functionality of this software. Three participants were glad that I was able to install the software and were interested in controlling their own machines from home and from the desks of their colleagues. After one study session, one of the participants eagerly requested that I show him how it worked across the network, and to replay for him his recorded session. He also rather excitedly called one of his co-workers in to discuss it and its possibilities to transform their existing work practices.

The VNC recording software is efficient for the type of work the participants were doing. With the exception of one participant, none of them appeared to notice any appreciable slow downs in their machines (I asked them to report any slow-downs when I was installing the software). Partly this is because of the way the server software operates (for example, only screen updates are sent, and the update frequency is throttled by the effective network bandwidth). In addition, the generated data files are quite small. The largest of the 10 computer protocols collected is 55 megabytes in length; this for over 40 minutes of computer recording at a screen resolution of 1152x884x24bit (about 23 kilobytes/second on average, but it has high burst rates). Most computer logs hovered between 20-30 megabytes for 35-45 minutes. This small data footprint made it feasible to capture a large quantity of computer protocols. With the size

¹Recording on the client side instead of the server side caused certain errors in the resolution for accurately stamping the time. The problem is that networking latencies caused stochastic skew in the arrival times of the updates. Although the arrival times were recorded to the machine’s clock resolution (about a microsecond) their arrival times were randomly shifted. Test runs suggested that the time skew would not be a problem for the intranet being used during the study. My subjective experience using this specific network was that the time skews were perceptible but not significant—much better than, say, IP telephony delays.

#	TIME	CODE	TEXT
11	14:20:57	COMM	does the split screen represent search and its context (two context/plan state)?
12	14:22:19	COMM	note that he doesn't use RCS or undo as a revision checkpointing system
13	14:22:43	COMM	a ritualized backup when he thinks he's going to do massive changes (checkpointing)
14	14:24:27	COMM	temporary memory for plan: he was about to write something complex and took time to try documenting

Figure 9.2: Snippet of coded field nodes (verbatim).

of inexpensive modern hard drives, it makes it reasonable to consider automatically collecting computer recordings for several months of average work days in this manner!

In addition to the recording software, a small conversion program was written to convert the resulting data into an "AVI" file. An AVI file is a standard Windows file format for encoding time-synchronized data streams such as audio and video. A compressor/decompressor component ("codec") was also written for Windows so that any Windows-based machine could use off-the-shelf display and editing software to display and manipulate the corresponding AVI file. My experience was that it was best for me to post-process the resulting AVI file to re-compress it using another free codec. A few free codecs were tried; all reduced the quality of the recording and expanded the file size, but both of these tradeoffs were manageable. The main advantage of recoding the computer logs was that it made it easier to use them for exploratory viewing and protocol coding. Partly this was due to the fact that the large displays of the participants could be modestly resized in the recoding process without sacrificing legibility. This made it possible to view the large screens on modestly sized computer screens. This in turn made it possible to add a synchronized display of a transcribed verbal protocol or action codes. This was accomplished by a simple Perl script that created closed-captioning commands (playing closed-captioning commands is a standard feature of Windows Media Player). Another advantage of post processing was that the recoding could support efficient seeking and skipping.² I also digitized the recorded audio and added it to this AVI file. The result of all this post-processing is a full recording of the computer screen, with synchronized playback of the verbal protocol and synchronized display of its transcription. These AVIs can be played on virtually any Windows-based machine (and Mac and Linux machines as well). In the future, it would probably be better to be able to record this verbal protocol on the participant's machine at the same time as the screen data is recorded. It would also be useful to have a streamlined way of producing the AVIs.

Notemaking Techniques

I recorded notes during and after the study session. During the study session, notes were written on a laptop computer. The notemaking technique was an adaptation of the "shadowing" field technique used by Singer *et al.* [596]. In this technique, a real-time "coding" of the activities is performed by observers

²"Keyframes" could be introduced in the video stream.

CODE	DESCRIPTION	SAMPLE(S) FROM LOGS
COMM	comment; usually used to record thoughts on the participant's work, or questions for later research	"so did the program work the way he wanted to, or is it just the answer he wanted?"
		"fluid motion of finding within the debugger—skilled."
		"note that the calling relationships are incredibly simple, yet he didn't know it."
STRUCT	follow structure in the external environment (an external plan)	"note the use of the history mechanism"
TALKBAC	instance of talkback from tool or system	"so he didn't think that it would work as it did"
EXTERN	externalizing information or structures	"the testing environment is explicitly being structured so that he can quickly set up the state that he needs to test the new code."
EXPERT	example of strategic expertise for using a tool	"scrolling to gain context in forward search (function name)"
		"did stacking by splitting window—it represented history"

Table 9.2: Codes for event types in "shadowing" observation technique

with laptops.³ The laptop runs a word processor with a few simple macros that help enter time-stamped notes in a stylized manner. Each time-stamped note is invoked by typing in a simple control-key sequence. A time-stamp is written to the record and the cursor is left on a comment field for the observer to type in notes to associate with the event. A different key sequence is defined for each event type or "code". The codes used in this study are replicated in Table 9.2 (examples are taken verbatim from logs of a single session). During observations, the observer watches the action and tries to interpret what the participants are doing. The observer then records events only when they seemed significant for cognition distribution and tool support. A sample of these notes is reproduced in Figure 9.2.

Although I had practiced this coding in the test run, I found that after a few sessions the coding method was not working as well as first envisioned. I felt that the notemaking was productive, but the coding technique was not directed enough to be useful for this work. For instance, the "COMM" code was by far the most common noted event (around 70% of all events logged), but frequently I felt that a more directed coding scheme would have used more specific codes rather than the "catch-all" "COMM" code. Moreover, after reviewing the notes, it became clear that assigning codes could likely be done *post hoc* just as easily as in situ. After three sessions I changed to use only comment style notes tagged by time. Although the result is an in-situ, time-stamped log of comments, they were not coded in-situ. Some comments will be made about this experience below.

In addition to the in-situ notes, free-form notes were also generated *post hoc*. Immediately after each session, I sat down alone and wrote out notes about my impressions and ideas without trying to deeply

³The work of Singer *et al.* is a "synchronized" shadowing technique in which two (or more?) observers code different aspects of the interaction using this shadowing technique. In this work, only one shadowing coder is used. Instead of a second observer coding the computer, automated computer logging was used instead.

PARTICIPANT ID	TASK DESCRIPTION	# OF SESSIONS	TOTAL MINUTES	NOTES IN SITU (ENTRIES)	NOTES POST HOC (PAGES)
A	adaptive maintenance	4	120	167	5.5
B	unit testing	2	33	17	2.0
C	coding	2	80	41	0.5
D	source change merging	1	40	46	2.5
E	design and coding	2	80	34	4.0
TOTALS:		10	353	305	14.5

Table 9.3: Summary of data collected during the study

interpret them. This two-staged notemaking technique was tried for both theoretical and practical reasons. On the theoretical front, ethnographers rightly place a great deal of emphasis on making notes in the field. The observer in the field has access to a great deal of context that is simply unavailable from the limited slice of data collected automatically. Thus one can view the situated observer as a type of highly sensitive and intelligent instrument; their notes are secondary data for aiding *post hoc* interpretation. On the practical front, the notes were considered a possible way of allowing researchers to quickly zero in on interesting observations. Generally speaking not all of a protocol will contain activity of interest to a tools researcher. A highly-directed observer may notice particularly interesting parts of the protocol right away. Notes made to this effect can direct post hoc coding to the relevant data quickly. I wished to see if the notes could have been used in this way.

9.1.4 Test Run (Pilot)

Before the study began, a test run of the protocol was conducted with a volunteer. The volunteer was a student but otherwise the conditions for observation closely resembled the conditions for the participants in the study. The test run was performed primarily to ensure that the study session scheduling, video taping, and notemaking techniques ran as expected. The test run was also taken as an opportunity to test out and subsequently tweak the note coding scheme used. In addition, the test run served as a rehearsal for the session introductions, and provided some practice for making useful field notes. No computer protocols were recorded for the test run.

9.1.5 First Stage Summary of Study

Before continuing on with the analysis of the data, an overview of it will be given, and a summary of notemaking experiences are reported.

Overview of Data Collected

Table 9.3 presents a summary of the observational data collected. Only a small fraction of it will be directly used in the subsequent analysis. Partly this is due to the exploratory nature of the study: only a few questions can be investigated within the scope of this work. Moreover, the types of activities and qualities

of tool interactions vary considerably between participant, and also between individual study sessions. The activities in any one session may not be useful for investigating any given theory-application scenario. At this point, however, I should point out that the terms of the study were such that open-ended analysis was allowed for, and permission was granted to archive the collected data for a number of years. This could permit a return to the data in the future as new analysis methods are proposed. An example of such an approach is Flor's studies of programmer pairs [223,224]. It should be noted that there was at first some resistance to obtaining ethical approval for longer-term data archival. It helped to point out that such methods are typical for other fields such as those from cultural anthropology (they could argue for archiving data permanently). In this study, a limited time frame was instituted; this seemed appropriate for this type of exploratory study with open-ended analysis requirements.

Only one observation session is examined in the remainder. This limited scope of analysis reflects primarily the limited amount of time and space that can be devoted to the exploration. In addition, the limits are partly a function of the reporting techniques that appear necessary for this work. The problem is that many details of the individual contexts need to be conveyed if the activity is to be understood. This requirement makes it difficult to use more than one participant since the length of the chapter would balloon considerably. For these reasons, one particular participant's observation session was chosen early on in the analysis stage as the one to analyze. The session is relatively simple, and yet it appears to demonstrate several types of cognitive support well. This should not be taken to mean that none of the other sessions contained interesting observations.

Notes on Notemaking

The "shadowing" technique that was attempted met with mixed success. In the "synchronized shadowing" work of Singer *et al.* [596] a key goal was to reduce the reliance on tedious and time-consuming post-observation coding. The coding was done in-situ and in real-time in the field. This means also that the observer can use the rich context to help interpret the activity. The main problem encountered with applying a similar coding technique in this case was that I was not looking to extract any particular observations or data from the observations. The codes that were tried were generic, and they did not seem to be helpful at the time (and they do not now). In other cases, a tools researcher may have a specific hypothesis to evaluate, or a specific issue to investigate. Then a highly-specific coding technique may be helpful. Further comments are made on this possibility in Section 9.3.4.

9.2 Context for Research Scenarios

In the sections following this one, the observations from one particular participant are analyzed in some detail. These observations all come from the first study session with participant E from Table 9.3. The purpose of the present section is to present some of the background and context needed in the following sections. In Section 9.2.1, the participant, his motivations, and his work context are described. In Section 9.2.2, an analysis is conducted of the cognitive support provided by one of E's tools, *Visual Café*. Specifically, *Visual Café* is examined for its ability to distribute planning and support plan-following.

This type of cognitive support analysis was presented briefly in Chapter 6. The intent of this analysis is that it should simulate a claims analysis of cognitive support—as if the `Visual Café` design or quality assurance team could have generated it. Later in this chapter field study scenarios will be presented in which this claims analysis will be used as a background to the scenario.

9.2.1 Description of Participant Context

The following description is a summary of the information known about E and his work context. The description is based on the recorded data (computer record, verbal reports), as well as the notes from informal interviews.

E was developing a software system in Java⁴ while using a UML-based object-oriented (OO) design method. His main tools were the modeling tool `Rational Rose` for UML-based design, and the integrated development environment `Visual Café` for the Java coding. In the jargon of CASE (Computer-Aided Software Engineering) tool vendors, `Rational Rose` was the “upper-CASE” tool, and `Visual Café` the “lower-CASE” tool. E developed this code under Windows. The system he was working on was part of a large, ongoing research and development project that contained a web-based interface to a complicated AI-based information system. He was familiar with OO development. Throughout all of his recorded observations he has shown a high degree of familiarity with OO design concepts, UML-based design (sequence diagrams, use cases, etc.), and the Java language and its coding idioms (exception throwing, memory management, etc.). It was my opinion that he could be classified as an expert developer. In this judgment of expertise I am not considering whether or not he demonstrated behaviour consistent with currently understood “best practices” of software engineering. It is wise to reserve judgment on expertise in engineering (which is largely based on normative or idealized models of behaviour); his development skills and experience seemed advanced enough to warrant the term “expert developer” within common programmer expertise classification schemes (e.g., Adelson *et al.* [6]).

Since the following deals with his skilled use of his tools, some prefacing notes need to be made regarding the qualities of his tool use and tool setup. E kept a number of applications running throughout the sessions, and he frequently switched between them. These applications include `Visual Café`, `Rational Rose`, and `Netscape Navigator`. He switched between these applications both by clicking on their icons in the Windows task bar, and by using the alt-TAB keyboard shortcut to navigate the application MRU (most-recently used) stack that Windows maintains. The `Rational Rose` and `Visual Café` applications were kept “maximized” (full-screen) during the full time for which I observed him (he did not display them side-by-side or overlapping).

E displayed a high level of expertise and familiarity with the applications he used. He never had to look up tool usage information in a printed help manual or online help system, and he never appeared to search the menus for actions to perform. His use of navigation panes, button bars, and menus, in fact, seemed practiced, skilled, and effortless. In the episodes reported in the following, no verbalizations as to how to perform actions on the tools were evident.

⁴Java, `Rational Rose`, `Visual Café`, `Netscape Navigator`, and Windows are all registered trademarks.

9.2.2 Distributed Planning in Visual Café

Visual Café is a Java compiler and editing environment. The interface of this environment is a multi-paneled environment that has become common for “integrated development environments” (IDEs) since the 1980s (e.g., see Teitelman [636]). Although Visual Café is the particular tool being used by the participant, the following analysis is not intended to evaluate the tool as good or bad in any respect. Neither is the analysis specific to this tool (the version of the tool is not even mentioned), but rather to a broad class of tools with similar functionality.

The main activity of interest in this analysis is the interaction between automated program checking and manual program repair. The typical working context is the common edit–compile–debug iterative development cycle. Visual Café (the compiler portion) can generate a list of warnings and errors regarding violations in a program’s (partial) correctness. Each of these violations are associated with a program position which is thought to be the likely location for the cause of the violation. For simplicity, let us call each error or warning simply an “error”. The programmer can invoke a check of the program’s correctness⁵ at (nearly) any time by hitting a key sequence, pressing a button on a button bar, or selecting a menu item. The output of the correctness check is, in part, a list of reports of errors. These reports are “specified” using a fixed list of schematized report categories (“undeclared identifier”, “type mismatch”, etc.). This output is displayed in a special “error list” panel that can be scrolled. The error list panel is not a simple text display panel: it is effectively an action menu. The user can click on a error report which forces an editor window to jump to display the location for which the report is associated. The error list panel also maintains an internal state of the “currently visited report”. A key sequence or menu command can be invoked to step this internal state and invoke the corresponding display jump.

The above description is straightforward, but it does not highlight the cognitive support offered to the developer. A simplified CoSTH analysis of a similar sort of environment was already given in Section 6.5.1 for the `compile` mode in Emacs. That prior analysis is revisited below, related specifically to Visual Café, and then expanded slightly to consider how the support relates into the code fixing process. As with the analyses of `RMTTool` and `Rigi`, the trick is to perform a bit of task analysis to figure out how the tools reengineer cognition. The portion of the development task in question here concerns generating code within correctness constraints. The Java language defines partial correctness criteria for valid programs (syntax, access control, scoping rules, typing rules, etc.). These constraints can be checked completely automatically. The automated checking of Visual Café distributes these constraint checks (D/P/CF). In cases where the programmer is not assisted in this manner (e.g., using only a pen and a pad of paper on a long flight from Australia), the programmer must check for such violations herself.

The partial automation of program correctness checking is well understood and appreciated in the community. What is somewhat less well appreciated is the *partiality* of the distribution, and the subsequent human–tool coordination design issues that crop up. A claims analysis using CoSTH can bring it into sharper focus. Like in the `RMTTool` and `Rigi` examples, the correctness checking is a fractional part

⁵Note that for historical and efficiency reasons correctness checking and object code generation (compilation) are performed at the same time. In common usage both the correctness checking and code generation are called “compilation”. There is no fundamental reason for requiring these to be performed simultaneously.

SUPPORT TYPE	TOOL FEATURE	DESCRIPTION
D/P/CF	correctness checking	compiler
D/P/CF/planning	error list generator	generates repair plan
D/D/plans	error list	plan of steps to repair program
D/D/PB/options	error reports	cue repair actions possible
D/P/VH/VM	clickable reports	automatically pages in program location
D/D/PG/state	error iteration count	holds state in sequential following order
D/D/goals	next-error sequence	holds sequence of future goals

Table 9.4: A list of cognitive support claims for Visual Café

of the overall programming task. In the case of `RMTool`, a human and automated checking cooperatively refined a shared representation of a hypothesized model of the system being studied. Part of this cooperation consisted of cooperative planning of which hypotheses to check. In the case of automated program correctness, this cooperative planning is also given high prominence.

In the programming context being studied, a programmer can use the correctness checker to partially distribute program repair planning and plan following. Automated correctness checking in the context of interactive programming environments offers more than merely a check for correctness: the checker can be reasonably interpreted as a *repair plan* generator. That is, it partially distributes the planning of repair work (D/P/CF/planning). To see why it is not merely correctness testing, notice that Visual Café (1) attempts to find many errors, (2) tries to determine likely source line locations for the cause of each error, and (3) renders an ordered list of messages that are intended to be diagnostic so as to cue repair actions. If the purpose were merely to test correctness, a simple “yes” or “no” answer would suffice. Instead, the purpose is to create a list of the errors in such a way that the programmer can effectively determine and execute reparations. This is an external resource that can structure the programmer’s subsequent repair activities. In such cases the programmer can refer to the external plan instead of an internal one (D/D/plans). Moreover, the diagnostic reports suggest the possible actions that could be performed to repair the program, that is, the reports reify a portion of the space of possible moves (D/D/PB/options). If the error reports merely listed line numbers, this supportive role would not be granted.

In combination with generating such external plans, Visual Café contains some features that allow for opportunistic or systematic plan following. Like Emacs’ `compile` mode, Visual Café makes it so that when an error report is clicked on, the appropriate external memory location is paged in (D/P/VH/VM).⁶ The paging in of the code does not remove the error list panel from the display, so its contents remain visible, making it unnecessary to remember them. These features allow for random access to the plan, enabling flexible and *ad hoc* exploration of the error list by clicking on various reports. In addition, Visual Café provides facilities for a more systematic traversal of the error list. Visual Café maintains an internal iterator state for traversing the linear list of errors, and a key sequence to advance the state. This allows for a systematic iteration of the plan steps. In this case, the iterator state distributes plan step following state (D/D/PG/state). Since the programmer does not need to decide the next goal to select, the “next-error” state of Visual Café implicitly serves as an externalized agenda of repair goals to perform

⁶This is a relatively new capability of the `compile` mode in Emacs.

in sequence (D/D/agenda).

The plan step iteration facilities make it easier to systematically step through the repair plan. However, in many cases there is little need to do so since the correctness checker can be re-invoked at low cost to regenerate and refresh the error list. In this sense, frequent invocation of the correctness checker amounts to a type of *incremental re-planning*. As in other incremental planning accounts (e.g., see Young *et al.* [722]) the external plan is updated as a result of new knowledge or as a result of changes to the task context. In fact, in the case of Visual Café, unless the plan step iteration facilities are used, it is sensible to follow a strategy of frequently updating the repair plan by re-compiling. One reason this strategy is sensible stems from the fact that Visual Café has no way of visually indicating plan completion status. Frequently recompiling gets around this limitation by removing errors that were repaired. This makes it simpler to follow the plan since the programmer need not remember the plan steps already performed. Another reason is that repairs made may generate a cascade of newly necessary repairs, or may expose new errors that could not be checked automatically before (e.g., semantic errors are usually not caught until syntax errors are fixed). Thus incremental re-planning in Visual Café rewards the developer by maintaining an updated external plan that can be referred to for determining subsequent action. This allows the programmer to adopt a shallow individual planning strategy which maintains a minimum agenda internally: pick an error or two and repair them, re-run the checker, and then iterate until the errors disappear.

Note that the error list can only be reasonably interpreted as a repair plan in the context of the user's goals for invoking the correctness check. To illustrate this, consider an alternative context: performing impact analysis. An example, due to Cardelli, was cited earlier (page 298). Cardelli's example is of a developer using a compiler to create a list of all uses of a particular type so as to be able to make a systematic change to these uses. This requires the use of a trick of changing the name of the type in question so that the uses of the type will generate errors. In this case, the goal of running the check is not to perform a set of repair operations, but to perform a sequence of manual checks of the uses of the type. In such a situation, systematic and complete iteration through the error list is an important desideratum.

A quick recap is in order for the tool usage being contemplated. The context of use occurs when a programmer has generated or modified code, and then wishes to engage in a cycle of program fixing. The programmer invokes the checker to find errors (D/P/PF), and thus construct a partial repair plan (D/P/CF/planning). This plan structures her subsequent activities. She consults the external plan, and then engages in a sequence of repair actions indicated by the plan. Thus the error list plays the role of an external plan for action (D/D/plans). To use the repair plan, the programmer can follow a systematic iteration strategy using a next-error sequence operator, or she can follow a more flexible exploration by clicking on items in the error list. Re-invoking the checker updates the external plan.

The above analysis contains plausible support claims that may be made for Visual Café by an imaginary researcher or engineer. These claims are listed in Table 9.4. They will be used in the following as a basis for generating scenarios in which researchers perform field studies to test and examine these claims.

9.3 An Exploration of Data Analysis Techniques

If a theory-based cognitive support research stream is to blossom in SE, a critical component of the stream will undoubtedly be ways of leveraging the theories for interpreting and analyzing observations. One obvious context is when trying to validate a claim about cognitive support, that is, when trying to verify that a proposed support exists as predicted. For instance, if memory is thought to be offloaded, such offloading could be looked for in real uses. Another possible context is measuring properties of the support. For instance, one may wish to know how much memory is offloaded so it can be compared to the cost of doing so. The purpose of this section is to explore ways of interpreting and analyzing field observations so these sorts of questions can be answered.

Consequently, in this section observations of participant E are analyzed using the claims analysis of the previous section. Once this analysis is done, it will form the “base” analysis for Section 9.4. In that section, scenarios similar to the ones described above will be investigated. These will use the “base” analysis of this section as a starting point.

The section proceeds as follows. First, an overview of the general data analysis approach is outlined in Section 9.3.1. Next, a coding scheme is described in Section 9.3.2 for coding up the collected protocols. The coding scheme is a key technology for making focused observations, and for relating these to cognitive support theory. The coding scheme is tailored specifically for the cognitive support claims of Visual Café which were identified earlier. The relevant episodes from E’s protocol were coded up using this coding scheme. The results from this coding process are described in Section 9.3.3. The results indicate that E utilized the support in more or less the expected manner. A visualization is shown to illustrate several aspects that are explained well only as an instance of distributed planning. Several simple statistical measures are applied to these results. A short discussion is included in Section 9.3.4 concerning the potential for applying similar techniques in a more lightweight manner in the future.

9.3.1 Analysis Methods

The technique used here for analyzing the observational data is similar to prior verbal protocol analysis methods. It consisted of a sequence of 5 main steps:

1. **Develop a Coding Scheme.** Identify possible activities relating to the claims and develop a specialized coding scheme to identify and tag such activities. The goal during this phase is to generate codes for the activities related to the claim, and to ignore all other activities. This goal is adopted so that the analysis can be as focused as possible.
2. **Extract Activity Episodes.** Isolate the activity episodes that contain claim-relevant behaviour. First, start out by reviewing the notes and protocols in order to home in on the activity of interest. Then review the protocols one or more times to become familiar with them.
3. **Transcribe and Segment Episodes.** Make transcriptions of the protocols for the selected episodes according to perceived phrasing and timing of the activity. These transcriptions are made both for the verbal reports and for the computer logs. In the case of verbal reports, the transcriptions are

annotated English. In the case of computer logs, the transcriptions are short descriptions of human-computer interaction activity. Segmentation is the process of breaking up the protocol stream into “atomic” events.

4. **Code Episodes (Generate Trace).** Code up the selected portions of the protocols using the coding scheme. The result of the coded protocols is a coded sequence of events. These are aptly called at *trace* of the distributed computation.
5. **Visualize/Perform Analyses on Trace.** Extract measurements from the coded protocols relating to the cognitive support predicted. In exploratory situations, this is a mining exercise. The trace could also be visualized for comprehension.

In classic verbal protocol analysis, steps 1–4 are often iterated, especially when a good coding scheme is not known in advance. In the case reported in the following, the claims analysis was able to generate an effective coding scheme. The only tweaking of the coding scheme that needed to be done was to change some of the code names, and to remove codes for sequential plan exploration (they were not used).

Notes on Transcription and Segmentation

A short note is required regarding the transcription and segmentation techniques that were used. Transcription and segmentation of verbal reports is a relatively well-studied problem, with many prior examples to consider [207, 256, 518]. Transcription of verbal protocols is typically just written English (or whichever language is used) possibly augmented with phrasing punctuation (commas, etc.). Segmentation can be done on a lexical or grammatical basis, or based on the content or inferred content of the verbal protocols (goals, questions [382], type of information being studied [373], etc.). The techniques are similar in many domains, even in “lightweight” applications of verbal protocol analysis (e.g., Nielsen [458]). Transcribing and segmenting human-computer interaction activities are rather more varied and *ad hoc*. Some comments need to be made as to how this was performed here.

Computer actions and user-computer interactions were transcribed using simple descriptions at a level that seemed suitably course grained for the activities of interest. For instance, a transcription of “types snausage” would be made instead of a stream “types s, types n, types a, ...”. Attending to information on the screen was also transcribed as an interaction. Self-generated computer events (e.g., popup menus) were coded as necessary. Relatively similar types of codings of external actions can be found elsewhere, however frequently the external action is intermixed with the sequential flow of verbal reports (as it is done by Flor *et al.* [224]). For clarity purposes and for segmentation purposes, the verbal transcriptions were placed in parallel with the interaction transcriptions (see e.g., Figure F.1). The main reason for doing so was for segmentation: segmentation made heavy use of both verbal reports and computer logs. The verbal reports were treated as evidence for a stream of internal events that occurred asynchronously but in a coordinated manner with action and computer events. A single coherent event might consist of a user reading something aloud, or expressing surprise at something the computer did. Given that the issue was planning and plan following for a program repair task, the segmentation took this into account. Segments were constructed based on the types of codes that were to be used. For instance,

CODE	ACTION DESCRIPTION
Gi. <i>gp</i>	push goal/plan <i>gp</i> onto stack (from internal)
Ge. <i>p</i>	push goal/plan <i>gp</i> onto stack (from external)
Pi. <i>g</i>	push goal <i>g</i> from plan onto stack (from internal)
Pe. <i>g</i>	push goal <i>g</i> from plan onto stack (from external)
Fi	no internal goal, derive or page in
Xe	planning/replanning (external)
repair	code repairing action
enable	plan-related enabling action
other	other episodes or actions

Table 9.5: Coding scheme for Visual Café example

if E added an `import` line to resolve a missing declaration error, it was considered a single repair event even if much backspacing and fumble-fingered typing occurred.

9.3.2 Coding Scheme

There are few considerations to bear in mind when developing a coding scheme for cognitive support. First, one must have codes for both internal and external cognitive action within the overall distributed cognitive task. For instance, if the use of an external memory is being examined for offloading, one may need to code for memory storage, retrieval, and management activities. Second, the codes must often be couched in cognitivist terms. For instance, if an error list is acting as an external plan, then actions on the error list are coded as planning, plan following, and so on.

In the context of the Visual Café cognitive support claims being examined, a coding scheme focusing on distributed planning was decided upon. Several of the claims in Table 9.4 fell outside this particular scope. The coding scheme is quite simple, containing only nine codes. They are listed in Table 9.5. Such a small coding scheme should probably be taken to signify the tight focus of study rather than theoretical weakness [518]. The naming scheme for the codes are in part hierarchically defined based on an internal versus external resource dichotomy, and by using the CoSTH naming scheme. This naming and code organizing scheme is similar to other published coding schemes [373]. The coding scheme can be broken down into three distinct sections as indicated by dividing lines in Table 9.4.

The top six are the primary codes. These codes relate to planning activity. They are divided into four categories: goal selection, plan selection, plan generation (i.e., planning), and “faulting”. These are labelled “G”, “P”, “X”, and “F”, respectively. “Faulting” refers to conditions where the developer runs out of internal goals to accomplish (new goals are then “paged in”). Each of G, P, and X categories can be divided into internal or external action types, so each action category is suffixed with “i” or “e”, correspondingly. Since no internal planning actions were observed, category Xi is not presented in the coding scheme. This type of plan coding scheme is consonant with published accounts of goal/plan action decompositions (e.g., Black *et al.* [51]).

The code “repair”, denotes error fixing activities. These could include code fixing activities not related to the repair plans being considered. In the observation session being examined, however, all of

the repairs were directly associated with a goal in a repair plan associated with the studied episodes.⁷ repair actions are interpreted quite loosely to correspond to any reasonable activity related to fixing a code flaw (removing a syntax error, adding missing declarations, and so on). repair actions—more so than the prior codes—may be relatively macroscopic. Specifically, a single repair event may be decomposed into a relatively extended sequence of low-level actions. For instance, in the episode being examined, E adds a missing `import` statement. Adding the `import` statement is a single repair action even though it is performed using an extended sequence of keystrokes, mouse movements, and selections from menus.

The last two code are intended to capture activity which is not directly related to planning and executing a sequence of repairs. The `enable` action is intended to label actions which are clearly a part of utilizing the cognitive support in question, but which do not contribute directly to the support in question. Examples of these sorts of `enable` actions would include moving a window into position to follow a plan, saving a file to disk so that it can be analyzed, or putting a tool into a mode where planning actions can be invoked. The other code is assigned to any other discrete event. This is a common “catch-all” category that is normally included in protocol coding schemes [256]. Distinct other events are identified through segmentation of the protocol just as the previously described event types are. This ensures that several other events are not merged into a single macroscopic other event. Sometimes when many interesting events end up being assigned an `other` code, it is hint that the coding scheme misses important aspects of behaviour [256] (e.g., metacognition [17]). In this case, other coded activities are more likely to signify activities that are outside of our highly focused attention on repair planning and plan following. Nevertheless, an analyst that is even more pressed for time might decide to ignore both `enable` and `other` events completely, and continue on as if the events in question did not occur. These codes were included in this scheme for the purposes of certain analyses that will be made later on. In reference to this, note that the `enable` actions could have been coded as `other` events. However `enable` actions indicate strategic expertise in using the toolset to enable the cognitive support to be used effectively. More will be said about these actions in the discussion section.

9.3.3 Results

The main activity in E’s first session was the design and coding of a method that generated a web page. E planned out how to write the method, coded an initial (skeletal) implementation, and then began a repair sequence to get it into runnable form. Only a fraction of the entire 40 minute session used the relevant features of `Visual Café`. The `Visual Café` use being studied here consisted of a sequence of three incremental code generation episodes (about 2 minutes in length total). These three episodes were coded according to the coding scheme described above. The complete contents of these codings are reported in Appendix F. The results of analyzing these episodes are described below. The results are based on analyzing a *trace* of the distributed planning state and how it evolves. The technique for constructing this trace is described below, and its analysis follows.

⁷These repairs may also be planned. For instance, to output a line of code, the programmer may need to have a plan (either planned, or retrieved from memory) for the sequence of tokens to write. Planning decomposition, however, is not broken down further here.

Methods for Generating Distributed Planning Trace

The coding scheme focuses on distributed planning activities—activities revolving around the construction and manipulation of plans, following plans, adopting goals, and so on. From the DC point of view, they are a trace of events in a distributed computational system. In this sense, the sequence of coded events is precisely analogous the “interesting events” of program visualization. In program visualization it is typical (and often difficult) to first identify classes of “interesting events” [516] in order to somehow generate a *trace* of these events. This is the function served by the coding scheme: to focus on and identify interesting events. In the present case, the events in question are ones that manipulate goal and plan states. An important objective in support analysis is to understand how plan and goal states are generated, manipulated, and pursued. In traditional DC terminology, the aim is to trace the “propagation of representational state” [224]. Here the simple term “state” (meaning *computational* state), is more than sufficient. Overall then, a problem for analyzing coded protocols is to analyze or visualize the path of states that the distributed computational system traces through, and the events that generate this path. This state transition path has been called the system’s “trajectory” [224]. How to adequately represent and understand such a trajectory is an imperfectly solved question.

The technique explored here is an adaptation of standard techniques for visualizing AI algorithms. These techniques are often used in AI textbooks to show planning and search algorithms. To understand these algorithms, one must understand aspects such as how plans are generated or updated and how goals are managed or adopted. For instance, a planning visualization could show the contents of a goal stack at each point in the execution of the algorithm. In the present case, the only essential difference is that the planning is distributed between the programmer and the tools. The trick, then, is to display the programmer’s internal state in combination with the state presented externally. In other words, one is careful to display the distributed system state. Then one could visualize how and where goals and plans are generated, stored, and modified.

In order to do this, a way of determining the plan-related computational state at any point in time is required. Also, the coded event sequence must be matched against this to determine how events serve to modify it. The programmer’s internal goal stack (i.e., agenda) could not, obviously, be directly observed. It had to be inferred from the verbal reports, the visible activities of E and his computer, and from E’s ability to perform certain activities. For instance, at one point (28:00), E was able to repair two errors in a row without appearing to refer to the compiler’s error list between repair events. It is presumed that somehow E internally maintained (or generated) a plan for performing those repair events in turn. This is indicated by his comments while reading the error list previously, saying “oh we may also need to um,,, import our,,, own DTD interface stuff” (27:34-27:40). Here it is reasonable to assume that E knows the “interface stuff” includes two classes, so that he internally maintained a plan to import one and then the other. External planning state was determined by examining the contents of the computer logs at each relevant point, and then interpreting them in terms of cognitive state.

Figure 9.6 shows a simple (tabular) visualization of the trace of repair planning activity that was extracted from the three repair episodes. The overall scheme is reminiscent of how a UML sequence diagram [554] is laid out in two dimensions. The trace events are laid out on the vertical axis with the flow

TIME	CODE	INTERNAL		ACTION	EXTERNAL
		GOAL STACK	PLAN		PLAN
26:36	Gi.G0	G0			
26:36	Xe	G0		\Rightarrow	P1(G1,G2,G3,G4,G5,G6)
26:43	Ge.P1	G0, P1		\Leftarrow P1	P1(G1,G2,G3,G4,G5,G6)
26:48	Gi.fault	G0, P1, !			P1(G1,G2,G3,G4,G5,G6)
26:50	Pe.G1	G0, P1, G1	G1	\Leftarrow G1	P1(G1,G2,G3,G4,G5,G6)
27:13	Gi.fault	G0, P1, !			P1(G1,G2,G3,G4,G5,G6)
27:17	Pe.G6	G0, P1, G6	G6	\Leftarrow G6	P1(G1,G2,G3,G4,G5,G6)
27:30	Gi.G0	G0			
26:36	Xe	G0		\Rightarrow	P2(G2,G3,G7)
27:34	Ge.P2	G0, P2		\Leftarrow P2	P2(G2,G3,G7)
27:37	Pe.<G2,G3>	G0, P2, G2	<G2,G3>	\Leftarrow <G2,G3>	P2(G2,G3,G7)
28:00	Pi.G3	G0, P2, G3	G3		P2(G2,G3,G7)
28:21	Xe	G0		\Rightarrow	P3(G7)
28:25	Ge.P3	G0, P3		\Leftarrow P3	P3(G7)
28:26	Pe.G7	G0, P3, G7	G7	\Leftarrow G7	P3(G7)

bold = active goal, ! = faulted goal, <, > = tuple, \Rightarrow = compiler invocation, \Leftarrow = read goal

Table 9.6: Trace table for distributed planning activity in episodes V1 and V2

of time going from top to bottom. On the horizontal axis are laid out “timelines” for internal and external components storing state. In this table, the joint system state is composed of the external plan in combination with E’s internal control (plan) and agenda (goal) panels. When events generate or modify this joint system state, the “actions” are represented in another column. These actions include commanding the computer to generate the external plans, and reading plan elements off the external display. Event times are indicated in the first column so that they can be cross-referenced to the coded protocols in Appendix F. Distinct goals and plans are given different labels. The labels used in Figure 9.6 are shown in Table 9.7. Plans are represented as a flat structure of goals to achieve. For instance “P2(G2,G3,G7)” denotes a plan labeled P2 which indicates the intention to solve three goals G2, G3, and G7 (in that order). For simplicity,

LABEL	DESCRIPTION
G0	check to see if repairs are needed
G1	fix error missing include <code>HttpServletResponse</code>
G2	fix error missing include <code>WebMakerClass</code>
G3	fix error missing include <code>WebDataClass</code>
G4	fix error undeclared method in call <code>makeMethod</code>
G5	fix error missing include <code>PrintWriter</code>
G6	fix error missing include <code>PrintWriter</code>
G7	fix error uncaught exception <code>java.io.IOException</code>
P1	fix sequence in first error list (Figure F.1)
P2	fix sequence in second error list (Figure F.2)
P3	fix sequence in third error list (Figure F.3)

Table 9.7: Description of goal and plan labels used in Figure 9.6

a plan label is allowed to be on the goal stack. In that case, it is meant to represent an intention to follow a plan. Internally E may be thought to store only a reference to the plan (a “pointer”), either to internal or external memory (e.g., “lets fix these errors”, where “these” is effectively a pointer to the external plan).

Simple Measures On The Trace

Given the coded protocol, and a trace of the joint system’s trajectory (Figures F.3 and 9.6), measures and statistics can be taken to help analyze their contents. Table 9.8 shows the frequency of codes within the three episodes. In this table, the different types of shared and internal memory uses are collected together under the “shared” and “internal” categories. Using this table, several points can be backed by simple statistical measures:

- 15 out of 31 events (48%) of the coded events in the episode are related to generating or manipulating plans and goals. This rather high number gives some confidence that these episodes are good examples of planning activity.
- There is no evidence to suggest that any significant internal planning of repair activities is performed by E. Yet from a joint performance point of view the code was repaired quite systematically. In addition, 10 out of 15 plan related events (66.6%) are generated externally. In other words, 2/3 of all of the action structuring activity has an external locus. This gives a rough indication of the extent that external planning resources are relied upon. Although deep planning is obviously not required for this particular task (error fixes are relatively independent), the complexity of the plan is not the issue at stake. Thus one can still say that the 66.6% statistic is a measure of the cognitive support that is provided and relied upon. The statistic immediately following bolsters this argument.
- The trace in Figure 9.6 shows that shallow plan and goal knowledge is maintained internally by E. E may have remembered multiple repairs once, but otherwise he relied upon the external plan—“paging” repair goals (error messages) into internal memory as needed.

9.3.4 Discussions

There are two distinct issues to discuss: (1) the results from the analysis and what they say about cognitive support, and (2) the experience gained in performing this analysis and what it implies for lightweight theory application techniques. These two sets of issues are described below.







CODE TYPE	FREQ.	FREQUENCY CHART
Xe	3	
shared memory	7	
internal memory	5	
repair	4	
enable	5	
other	7	

Table 9.8: Frequencies of coded actions by type

Discussion of Cognitive Support

Points about joint system activity and cognitive support are briefly outlined below.

1. **Skilled Interaction.** No verbal reports are given for most computer interactions. The notable exception is that E does verbalize much of his typing, although many of the typed words are missing from the report, and they are often mumbled under his breath. Otherwise his interactions appear to be consistent with the “cognitive skill” levels of task execution noted by Card *et al.* [94].
2. **Dependencies and Step Ordering.** The first goal in the external plan is chosen by E after each compilation invocation. One likely way of explaining this is that E is generally biased to select the first elements in the plan, possibly by habitually starting at the top of the list and scanning down. This may be a winning heuristic for repair ordering. The reason is that in Java programs most of the dependencies between program elements within a single file are concentrated on the initial portion of the file, where declarations (e.g., imports) are concentrated. Repairing errors at the top may remove many dependent errors within the list. Planning is most effective in conditions where there are dependencies or constraints on the ordering of actions. In this sense, the compiler’s error ordering technique and E’s bias towards selecting the first element is an example of good planning. The fact that the compiler’s error listing ordering seems “obvious” is irrelevant. In other similar situations (e.g., exploring hits from a search engine), ranking techniques are more convoluted, but not fundamentally more significant.

This raises the possibility of using similar studies to determine the efficacy of alternative error report orderings.

3. **Coping Strategies for Plan Following.** In the first repair episode the last goal is selected after the first one, indicating that errors may be opportunistically followed (i.e., opportunistically selected from the list). Issues regarding visual search and visual popout may affect the ordering. In this case, it is unclear why E selected G6 after G1. G2 and G3 were overlooked completely (E exhibited some surprise at seeing these errors at 27:34). E may have pursued a habit of frequently re-compiling to refresh the repair plan regularly instead of systematically going through the error list. So long as correctness checks are quick, this may be a winning heuristic for two reasons.

First, the automated error checking generally⁸ keeps any errors that are not resolved. This means that there is usually little penalty for missing a plan step since it will be retained during re-planning. In other circumstance this is not so, and it is much more important to be thorough in executing plans.

Second, each repair can remove many of the dependent errors in the list, and it is effortful to figure out which errors are dependent. It may be better to let the re-compilation prune the error list. This removes the errors already repaired (and checks the repairs early), making it easier to select a new goal. The goal selection is made easier in this manner because E does not need to remember or determine which errors were already selected. In this sense, frequent re-compilation is a *coping*

⁸It is possible to “mask” an error by making a repair, especially an erroneous one. For instance, a globally declared variable might mask an error where a local variable of the same name is undeclared.

strategy used to avoid the need for tracking state internally. Visual indication of repair status would reduce the importance of this strategy. So this example illustrates an instance where a coping strategy can be identified, and where the need for this coping strategy may be reduced by the provision of cognitive support.

- 4. System-level Understanding of Planning.** The external planning and plan resources effectively remove repair planning from E's personal workload. His goal stack and internal plans are very shallow indeed. They are more characteristic of display-based problem solving, where actions are cued by display state, and no explicit planning is performed. Nevertheless, from the system point of view the action *is* structured by incremental planning. The difference is that the evolving plan is constructed, maintained, and evolved externally (this point will be brought up again in Section 9.5 while discussing theory validity). This is a case where distributed processing cannot be understood fully by looking at only one processing element: a system-level understanding is required (see Section 4.2.3).

It is obviously impossible to generalize from such a limited set of observations, but the above comments are consistent with experience with these sorts of tools.

Theory Application Techniques Evaluation

The preceding analysis is fairly typical of *post hoc* verbal protocol analysis, at least in terms of overall procedure and coding techniques. These are acknowledged to be onerous. Some previous techniques have been discussed elsewhere of reducing the burdens of this form of work. For instance, there exist techniques for automating or guiding aspects of protocol analysis, including segmentation [125] and trace analysis and visualization [217]. But what does the current experience imply as to how to make the above sorts of analyses more lightweight? The experiences from this test case suggest four potential ways of realizing speedups, and two limitations for the techniques.

One way of realizing a speedup over traditional techniques is by being extremely targeted in the investigation. In this particular example, once it was determined which cognitive support claims would be investigated, a highly targeted search for exemplary episodes was invoked. Out of a total of 40 minutes of recorded data, only 2 minutes of data needed to be transcribed and coded. The target episodes could easily be identified by a pass through the recorded logs. If I had known in advance that I would be pursuing this aspect of cognitive support, I would almost certainly have made a note of the relevant activities in my notemaking. Note that the session initialization method (participant-initiated observations) play an important role in this overall strategy. Another aspect of being highly targeted is that a small set of relatively macroscopic codes can be used. In the above analysis, only 6 "interesting" codes were used. This makes coding relatively simple because aspects of cognitive support falling outside the scope are more or less ignored. Overall, this suggests that at least some of the time the transcription and coding burdens will be tolerable in some research and development contexts. The key is the tight focus on a particular aspect of cognitive support. For comparison, notice that it is unlikely that a controlled experiment testing two variations on the tools would be a faster way of determining similar sorts of support issues.

A second way of realizing a speedup is to avoid the transcription and coding completely. This might be feasible if a particular issue is being considered. For instance, consider the case where a plan-following cognitive support is being pursued, and the question being pondered is whether or not plan steps will be accidentally skipped (e.g., because the state is not sufficiently visible). In such specific contexts it may be sufficient to simply watch the recorded tapes and note instances where errors are not explored systematically. The argument is that with a sufficiently narrow focus, the targeted issue is relatively easy to pick out, and so the transcription and coding techniques need not be so systematic. This argument is similar to Nielsen's argument for why protocol analysis could be useful for usability engineering:

In most realistic development situations, each hour of thinking aloud observation probably only needs to be followed by half an hour of combined analysis and report writing, since the experimenter will have noticed all the important usability catastrophes as they occurred during the thinking aloud experiment. [458, pg. 70]

The differences in this case are that (1) aspects of support are being examined, not usability blunders, and (2) theory is used to narrow down the focus enough so that one does not have to rely so heavily on the luck and insight of the experimenter.

The above point brings up a third way of realizing speedups in analysis: focused coding. The coding and analysis in this example was, for the most part, set up from the beginning during the claims analysis. In other protocol analysis work, this is not so. Often times iterative coding and re-coding is performed [207]. This is especially required when a novel aspect of cognition is being studied (e.g., see Gray *et al.* [255]), or when a holistic and fully bottom-up approach is being pursued (e.g., see Purcell *et al.* [518]). In this experience, much of the interesting work was already done by applying the cognitive support theory in the claims analysis. What was left to do for analysis was to establish codes and execute a straightforward analysis. Thus the argument being advanced is that as a discovery tool, protocol analysis is often intensive, but as a theory-honed search tool, it can be done much more nimbly.

Another way of realizing a speedup is to avoid verbal report recording *and* post hoc transcription and coding altogether. This might be done by using a shadowing technique. As it was reported in Section 9.1.5, the in-situ coding technique attempted was not successful. One possible reason for the failure was that the codes were not specific enough so that the coder could be properly prepared. If I had known to specifically look for programmer goals and actions to look for, it might have been feasible to perform in situ coding of the verbal protocol. Depending upon the aspect of cognitive support being studied, it might be easier or harder to do in situ coding. In the present experience one aspect suggests that in situ coding might be possible: assigning codes to the recorded protocol was found to be easy. In other cases of protocol analysis, coding is often iterative because it is unclear what codes to assign. In addition to verbal report coding, sometimes the synchronized shadowing technique of Singer *et al.* [596] could be feasible too, and in that way remove the need for recording low-level data completely (video, audio, computer logging).

One clear limitation of the techniques tried was that it would make certain forms of cognitive support difficult to observe or measure in the field. For example, much of E's lower-level interactions (reading, graph manipulation, etc.) with tools were highly skilled and no verbal reports were generated. Thus

determining if a specialization substitution is occurring (like a visual operator substitution) may be challenging without more specialized data like that from gaze tracking hardware. Otherwise it may require some manipulation of the task context (e.g., providing different visualizations) to be able to determine the impact of such skilled activity. In other words, one of the main limitations of this sort of field study is the type of data that is easily observable and recordable in the field. Only certain facts can be ascertained by verbal protocols, computer logs, and ordinary video taping. These limit the types of cognitive support that can be readily analyzed.

Another limitation suggested by the present experience is that in many circumstances the “shadowing” technique will not be sufficient, and *post hoc* coding from protocols will be necessary. The problem is that many activities are rapid and could occur faster than they can be noted and written down. In the synchronized shadowing of Singer *et al.* [596], the goals were generated at a relatively slow rate. The present experience leaves the issue unresolved. Only 15 events of interest were generated in roughly a 2 minute span (see Figure 9.6). A trained observer may or may not have been able to accurately pick out and write down the sequence of goals pursued by E. It can be argued that all users are limited processors who can pursue only so many verbalizable goals per minute. Trained coders might therefore have a chance of matching this limited rate of goal pursuit (contrast, for example, a “twitch” video game [353]). This is a potentially testable supposition in many cases. Regardless, the rub occurs if one tries to follow skilled activities such as visual search.

9.4 Theory Application Scenarios

If a theory-based tool evaluation and design stream is to become a reality in SE, it will likely be necessary to be able to efficiently test and explore cognitive support claims. This section proposes two plausible research scenarios and then looks to the results of the above field study to determine the potential of field research in similar scenarios. The first scenario is of a claims check. The second scenario is of attempting to measure support. These two scenarios are important contexts for future SE tools researchers. In these scenarios the above field investigation suggests potential ways of using empiricism to answer important support questions.

9.4.1 A Claims Check

In the future, cognitive claims analysis could be an important aspect of tools research and design. When claims are generated, they may need to be evaluated in many different contexts. One obvious context is when trying to validate a claim for the purpose of publishing a scientific report about the tool. Another important context is during the normal course of tool analysis and design. This latter evaluation context was implied by Scenarios 4 and 7 from Chapter 2. During the design iteration cycle, a type of cognitive support may be proposed, a tool may be designed, and the claim may need to be checked. In any of these sorts of cases, the researcher might wish to go to the field or lab to determine if they are on the right track. This is the sort of claim evaluation context that will be explored in this section.

Claims Check

April is a “usefulness engineer”⁹ on the design team of Visual Café. The design team has performed a cognitive support claims analysis on Visual Café, and the results are similar to Table 9.4. April is concerned about the claim for offloading plan steps. Although she does not doubt the theory that memory can be usefully offloaded, she has uncertainties about the claims. She is unsure if her design team has done a good job of matching Visual Café features to the support they presume they provide (e.g., whether error lists really are used as plans). She is also uncertain if the various features (windowing, key bindings, etc.) of Visual Café will make the offloading effective. And even if these are true, is the offloading used in realistic development situations? Before continuing on with design and analysis, it seems important to April to lay some of these uncertainties to rest.

April decides on a short user study. She knows of a capable programmer from another department within the company who uses Visual Café to support the company’s web presence. This programmer, Kylee, is willing to help in a study. First she generates a coding scheme to code events related to error step offloading: when error list steps are internalized, and when they are attended to. Then she adds a tracing statement to an internal development version of Visual Café so that it logs all error lists generated and window events. She then sets her laptop coding system to code for the error internalization and use events. She has Kylee phone her when she’s doing some code writing. During the development she records the errors that are fixed. She also records the error messages she thinks Kylee reads off the screen, and the errors she appears to recall from memory. Back in her office, April matches up the computer log to the timestamped codes from her notemaking. Rarely does Kylee appear to use internal memory for error messages. In the instances where Kylee does, she notes interesting extenuating circumstances (perhaps simplicity or regularity in fixes, screen crowding, etc.). Overall, though, April comes to appreciate the importance of error list offloading and the progression of fixes that occur. She brings this knowledge back to the design team, who can then better evaluate likely design choices (highlighting errors, etc.)

This scenario is of a simple claims check. The empirical work suggested was a modest extension of the current field study. In this scenario, it may be considered far-fetched to think that an experienced designer of a tool like Visual Café would not know if the error list offloads memory or not. Most compiler environment designers have a good understanding of the importance of frequently consulting the error list. However this is not the point of the scenario. The scenario is meant to be illustrative of the possibilities of claims checks without prejudice to whether the claim is obvious to the reader. In this case, a reasonable argument can be made that a simple and tightly focused user study could be performed to gain confidence in the claims being made during design. In circumstances where the claim appears less obvious, the same basic approach could be tried. Furthermore, the empirical approach is most important when the claim is wrong. For instance, it may be the case that seemingly reasonable ideas for offload memory may turn out to be wishful thinking. Even experienced designers may be wrong. Thus the main use of the scenario is to envision a case where an empirical can be cheap and effective.

⁹This is an alternate spin on “usability engineer”, a common title for HCI practitioners. As far as I know, there is no such title in existence, but I would argue it makes equal sense.

The scenario is a speculative extension of the techniques tried for participant E. The claims analysis is not unusual, since it was assumed to be substantially as the one given already. Thus the main speculative part of the scenario is the observation and analysis techniques. A few notes about these differences are therefore in order. The most notable difference between April's techniques and the ones used here are that it is assumed that April can use a lightweight observation and coding technique. The assumption is perhaps reasonable. It is unlikely that Kylee will be able to read an error list item and then make a code fix in less than a few seconds. In E's protocol, it was readily apparent on most occasions when he read an error list and verbalized his intention to make a corresponding fix. Even if this assumption is not borne out, April has the option of videotaping Kylee and performing more traditional coding. The computer logs should help in this regard for determining which sequences of activity to look at. The main point to note is that a clearly defined cognitive support claim is the factor that enabled a tightly focused approach to observation and analysis. In addition, note that it seems unlikely that a controlled experiment involving two different design options could have yielded similarly informative results.

9.4.2 Measurement Scenario

CoSTH does not quantify cognitive support. This is a shortcoming common to broad-brush theories, however it is one that might be counteracted using empirical techniques.

Offloading Measure

Rico is a usefulness engineer in the quality assurance team for Visual Café. He is in charge of making sure the cognitive support requirements are met, and for measuring quality of the delivered product. One of the goals of the Visual Café team was to make an environment that offloaded repair memory effectively. Many factors—including usability factors—can affect this. Rico examined the requirements and decided to measure the offloading achieved in order to quantify the usefulness targets met.

Rico contacts three beta testers of Visual Café who are located nearby and are willing to be used as a focus group. He makes sure one is a new user to Visual Café, another is a heavy developer who makes extensive use of Visual Café and is always willing to test the new improvements, and the third is an occasional developer. He brings two of them into the company's usefulness testing lab and visits the heavy user at her work site. He collects the observational techniques deposited by April in the design team's repository of design documents. Rico uses the same shadowing methods used by April, although in the usefulness lab there are video cameras set up, and he uses these also. In this way he double-checks his codings. Rico then enters his codes into a spreadsheet. He calculates the frequency of internal versus external plan step usage (as in Table 9.8), and the total internal load (goal stack depth plus plan step count) carried by the developers. Armed with these measures, Rico reports back to the quality assurance team.

One complaint that has been raised about usability engineering is that it too infrequently resembles quantifiable, theory-backed engineering practices [391]. One argument that has been put forward in response to this charge is that it may be possible to be more quantifiably systematic in design and engineering

of interfaces. One place to start is in “usability specifications” [108]. These would be usability goals that are to be met in design, and they could be included in a system’s specification just as functional requirements are. Once specified, they can guide development. During development goals are set to meet the specification, and early prototypes are evaluated as to how well they meet the specification. For usability issues, quantification and measurement has been problematic [108,309]. The wrinkle added here is that usefulness in the form of cognitive support may be specified, quantified, and measured. In this context, it means more than measuring an intermediate performance factor like cognitive load (e.g., Chandler *et al.* [118]). In the context of measuring a cognitive support claim, measuring support principles such as offloading binds the quantification to the features thought to produce the benefit.

9.4.3 Summary and Discussion

Cognitive support theories like CoSTH can be used to generate and explore claims during tool development or evaluation. However in many circumstances it might be beneficial to perform a field or laboratory study in order to examine the claims. In this section two scenarios were proposed for which empirical examinations of cognitive support appeared important. These were for testing a claim to see if it was well founded, and in measuring the support claimed of a tool. In both these scenarios, the field study described earlier in the chapter was employed to argue the possibilities for empirical studies of the claims. Techniques similar to the ones used in this study were proposed. It might be reasonable to use the exact same field study techniques as were used here. Nonetheless in the spirit of exploring the possibilities of lightweight techniques, more “discount” investigation methods were suggested. These methods could not be tried in the context of this study, however they suggest future lines of research.

The attention here was on the techniques explored in this field study. Nonetheless, other claim evaluation techniques are possible. For instance it could be proposed that various forms of intervention and control could be effectively used to explore the claims. For instance, it is possible to manipulate task demands in order to see how the cognitive supports are used in various circumstances. An example might be in study memory offloading. In difficult tasks, a user’s short term memory may be in great demand, so offloading may be more important, and the user may make more systematic use of externally stored state. For instance it might be useful to increase cognitive load by having the users perform concurrent tasks such as an “articulatory suppression” [164] activity. Such interventions might be used to test the efficacy of the offloading by varying the task demands. Similarly, changing the cost structure of operations [164,478] in the tool may help expose aspects of cognitive support. For instance, it might be fruitful to see whether raising the delay for compilation would cause users to make more systematic passes through error lists.

9.5 Validity and Evaluation

The excerpt of cockpit activity presented above is only approximately 1.5 minutes in duration, yet it is very rich. It contains within it illustrations of many of the central concepts of a theory of distributed cognition.

– Hutchins and Klausen, “Distributed Cognition in an Airline Cockpit” [323], pg. 15.

In science, when a theory is proposed there is naturally a call to validate the theory by setting up an experiment or otherwise collecting evidence for its veracity. In cognitive science, there are a variety of criteria for establishing the validity of any cognitive model (sufficient, necessary, psychologically plausible, neurologically plausible, etc., see e.g., Thagard [639]). But the theories (CoSTH) and models (HASTI-based) being proposed here are meant to be broad-brush approximations. Instead of being considered valid models in any strong sense, these and other similar sorts of models should probably be thought of as tool for inquiry. Thus they should be evaluated primarily on their merits as tools for real investigation. Real debate over scientific merit should be reserved for the basic science literature upon which they draw. It seems to a degree incongruous to question the validity of a purposefully simplified account built from previously tested theories.

Nonetheless, CoSTH and HASTI are theories and models of DC activity and cognitive support, so establishing parameters for their validity is desirable. At minimum, one should wish to make sure that all validity is not lost by purposeful simplification. Yet question remains as to what type of evaluation is appropriate. In works with similar aims, the key issue is whether the model or theory “says” something interesting about the behaviour it is used to analyze (see Whitefield [700], Vinze *et al.* [660], Wright *et al.* [719]). Validating models and theories built for insight normally consists of relatively informally evaluating how consistent they are to observed behaviours. Dillon [183] provides a good argument for this approach in the context of his framework for modeling reading behaviours. He argues that his framework

is not intended to provide a precise model of human mental activity during reading. To test for this would therefore be pointless. In its form as a generic description of the reading process at a level appropriate for design however, it is proposed as valid, and a test of this would be relevant. One test of suitable form would be to examine readers’ behaviour and verbal protocols when using a document, parse them into their various components and then relate these to the components in the framework. If the framework is valid, such protocols should provide clear examples of the behavioural and cognitive elements that constitute the framework. If it is an invalid description, the protocols should fail to provide such a match or should indicate the presence of activities not accounted for in the elements of the framework. [183, pg. 138]

HASTI can be evaluated¹⁰ in a similar manner. If one were to take Dillon’s procedure and apply it to

¹⁰I hesitate to follow Dillon’s use of the term “valid”. Although his usage can be defended, I think that it is appropriate to limit the interpretation of the term herein. Thus I shall restrict the use of the term “valid” to stronger

HASTI and CoSTH, one should hope to find that: (1) events are well accounted for by a distributed computational interpretation consistent with HASTI, and (2) attributions of cognitive benefit are consistent with the arguments made by CoSTH.

To this end, let us revisit the small segment of *Visual Café* use that was analyzed in Section 9.3. This is an example taken from uncontrolled field observations. The analysis of claims involved in *Visual Café* were a minor elaboration of support arguments made for a different tool. The analysis of joint action was based on HASTI modeling techniques for distributed planning and plan following. Therefore it is reasonable to start with this analysis of *Visual Café*, and merely consider the merits of the explanation here. Unfortunately, since this sample analysis uses a limited portion of HASTI and CoSTH in its construction, it is possible to evaluate only parts of them.

Several aspects of the analysis are consistent with HASTI. The main aspects of HASTI applied in the example are the control and agenda facets of the Agent model. The protocol shows some direct evidence that internal plans and a goal stack drove part of E's behaviour. For example, it is fairly evident that E would maintain a goal (P2 and P3 of Figure 9.6) of trying to fix up a number of repairs in the external repair plan. Another aspect of HASTI demonstrated incidentally in the episodes are the limitations in memory encoded by the Hardware model of HASTI. The protocols contain indirect evidence for this limitation in the form of the minimal use of internal memory by E. Except for one plan step, E relied on external memory to furnish the goal to work on next. This is not a strong proof by any means that short term memory is limited, but he did not show any strong memory capability such as recalling all the error messages after reading them once. Although this presents a weak set of evidence in favour of a complicated model, it shows that the basic description is consistent with observable behaviour.

Perhaps more important is the fact that HASTI is a *distributed* model. It assumes that plans or goals can be located externally or internally and still organize behaviour. The analysis in Section 9.3 reveals quite strongly that this is the case. Although internally E maintained little goal or plan state, the three episodes can beneficially be viewed as being organized by planning. It may be opportunistically followed and incrementally generated, but the activity was structured. Goals G1, G6, G2, and G3 (Figure 9.6) were attended to in that order. This shows rather limited opportunism. External plans were updated twice during performance, but only one goal was added: the other changes were limited to the removal of goals that were achieved (so that state need not be remembered). This shows limited incremental planning due to feedback from actions. As a joint system, E and his computer acted in a systematic and planful manner. The system set up a goal to fix the program and systematically determined a course of action and followed it.

The main point to notice is not necessarily how systematic and goal-driven the activity was, but the fact that only from the distributed planning point of view does this understanding become apparent. If just the programmer was examined, it would seem that virtually no planning was performed—just shallow cue-directed activity. It could be pointed out that the programmer's behaviour was "structured" by external artifacts. But what "structure"? Without a realization of the role of the external planning, the contributions of the compiler towards systematic activity are lost. However when one looks at the

concepts associated with traditional cognitive science validation, and speak merely of "evaluation" in the remainder of this section.

joint system, one sees a joint system state that evolves by first generating a plan of action which is quite routinely followed and updated. This point helps validate the importance of HASTI's DC view of activity.

The episode is also consistent with CoSTH. To take stock, let us consider in turn the cognitive supports analyzed in Table 9.4. Regarding constraint processing distribution, it is difficult to doubt that the automated checking reduced cognitive work on the part of the programmer. Regarding distributed planning, it is fairly easy to see that the repair steps taken by E (i.e., G1, G6, G2, and G3) were a direct result of the repair plan. The result is that E's verbal reports contained virtually no planning. Thus it is reasonable to say that the planning was offloaded. The third item in Table 9.4 refers to the location of control information. Again, the protocol strongly suggests that E did not maintain the control information (plan) internally, but kept it externally, referring to it whenever no more internal goals were available. This is comfortably construed as plan offloading. The remaining items in Table 9.4 were not examined or used during the observed activity. From this review it seems clear that the account of cognitive assistance made by CoSTH is consistent with the observed activity.

From the above short review, it can be argued reasonably well that the account made by HASTI and CoSTH are consistent with real cognitive system behaviour. To use Dillon's words, they "provide clear examples of the behavioural and cognitive elements that constitute the framework[s]". This is not a validation, but it does help evaluate how reasonable HASTI and CoSTH are.

9.6 Conclusions

As of this moment, the empirical evaluation of existing tools has not benefited from usefulness theories. When theories have been used, they have been relatively unsuccessful in systematically identifying tool benefits. Design is similarly affected, since it has not been possible to state and test for usefulness design goals. A cognitive support theory like CoSTH has a chance to change this state of affairs. To make inroads, however, the empirical techniques must be lightweight. Methods of effectively deploying such techniques in realistic scenarios must be explored to ensure that systematic engineering of cognitive support is possible. To date, little is known about the ways to do such empirical investigations in practical tools research and engineering settings.

In response, this chapter has presented an exploratory attempt at investigating relatively lightweight field techniques. An example of analyzing a recorded set of field observations was worked through. Overall, this analysis showed that useful answers can be obtained relatively cheaply by a field study. It did this for a professional tool in real practice. Because the observed behaviour matched HASTI and CoSTH explanations well, the field study also provided some assurance that HASTI and CoSTH are reasonable and trustworthy, and can be expected to work in real-world situations.

This analyzed example served to exercise poorly understood data gathering techniques to gauge their promise and efficacy. These two data gathering techniques were an in-situ shadowing technique and a computer logging technique that is generic and reasonably lightweight enough to be deployed in many field scenarios. The in-situ shadowing technique was not effective for this study. Nonetheless, suggestions were made as to how theory-based support analyses might be used to make the observations focused enough to make the technique work.

The example observation session also served to highlight two possibilities for employing empirical studies to answer support related questions. First, it was argued that claims checking or validation might be performed efficiently. This capability could be important for checking that the claims made are reasonable. Second, it was suggested that certain aspects of cognitive support could be measured. This capability could be valuable in many software development settings, including quality assurance settings. As a whole, the scenarios bring into focus the question of usefulness engineering—could real-life cognitive support engineers be using such techniques? The results here were tentative and weak, but they should be evaluated within the larger context of the aims of this dissertation. The overarching goal has been to provide a theoretical tool in the form of cognitive support theories—to integrate it into working research and (eventually) development practices. This chapter was a step in that direction.

Chapter 10

Conclusions

Throughout this work, the guiding thesis was that a solid, theory-based understanding of cognitive support is possible, and that time has come for SE research to begin developing and using applied cognitive support theories in earnest. There is a strong need for them, for we build tools with undeniable cognitive consequences, and trade in cognitive support ideas. There already exists a psychological and scientific base which, so far, has been underutilized in part due to its fragmentary nature. As of now, craft knowledge and folk psychology fill the theoretical void. Evaluation and design suffers from the inability to systematically explore cognitive support, and to do so at a level above the details and features of individual tools. It would be a shame to continue leaving the existing theory untapped; it should be collected together and simplified for use. Methods for applying these must be made practical, for informal contexts as well as more rigorous studies. A critical contribution of this dissertation is therefore a vision for transforming SE research from by putting it on a firmer theoretical and science-backed foundation, and a set of core theoretical tools and methods needed to do so. This is basic, necessary equipment for a research programme investigating cognitive support in SE. A core infrastructure.

Bound up tightly in this overarching thesis is a second thesis: that RODS, HASTI, and CoSTH are a suitable initial basis for starting the project of injecting existing theory into SE research. They should be a good first step towards crystallizing scientific knowledge about cognitive support into a form that is suitable for use in SE. They may be useful in more broad contexts, but the domain of interest here was SE, and in particular software comprehension and reverse engineering. The second major contribution of this work is therefore a toolkit of theories, models, and methods for investigating cognitive support in SE. The overarching thesis and this more specific thesis are complementary. It would be suspicious to argue that SE ought to embark on a theory-based research programme without being able to point to a credible basis for doing so. RODS, HASTI, and CoSTH are an initial proof of concept in support of the primary thesis. Furthermore, the theoretical toolkit might be interesting enough alone, however within a vision for transforming SE research, the toolkit becomes an agent for evolution of the field. This has been a dissertation concerning the SE research infrastructure as much as it was an investigation into applied theories of cognitive support.

The main conclusion to be drawn from this study is that a theory-driven research programme in SE is currently possible, and that RODS, HASTI, and CoSTH are resources to lead the way with. The vision and the applied theories were the main contributions, but there were also a number of secondary ones along the way. The following recapitulates these contributions and summarizes the implications for future work in the field.

10.1 Summary of Contributions

For exposition purposes this dissertation was organized as a sequence of topics suitable for individual chapters. Although this groups the contributions according to those topics (phenomena, design theory, etc.), it is also possible to categorize the contributions according to what types of advances they provide (basic framework, model, etc.). This latter method of organization is used here. Using this method of decomposition there were four primary contributions made, and three supporting contributions. Figure 10.1 depicts an incidence matrix showing how the contributions are spread out across the chapters. Short summations of these contributions follow.

Primary Contributions

1. **Basis for Applied Theorizing.** A theoretical basis for researching and designing cognitive support in SE tools was provided.
 - (a) *Basic Analytic Framework.* A DC framework was detailed for explaining cognitive support in computational terms. This basic framework includes:
 - i. DCAF: a set of DC convictions for understanding human–computer systems in joint cognitive terms. Such a framework is needed in order to interpret tool contributions in cognitive terms
 - ii. RODS: a set of support principles. These provided the core principles for explaining cognitive support as computational advantage.
 - iii. Claims Method: a way of making cognitive support claims. This explains the steps needed for claiming that a tool feature supports cognition.
 - iv. Modeling Methods: DC architectures and virtual hardware models were proposed as modest extensions of current DC theorizing. These establish a basis for generalizing cognitive support arguments, and for raising the analysis level above low-level interaction.

For the most part, this framework is a collection and unification of prior work on DC theory, modeling techniques, and claims analysis. Even so, the selection, collection, and integration adds value. Psychological research in SE appears to have become fixated on cognitive theories and not support theories, and little is currently known in the field as to how to apply cognitive theory to explain tool value. This dissertation provides a solid foundation for doing so.

- (b) *Research Vision.* A vision was provided of a cognitive support oriented research stream in SE tools research. The vision established roles and boundaries for tools research as separate from

CONTRIBUTION	CHAPTER								
	2	3	4	5	6	7	8	9	
<i>primary</i>									
Basis for Applied Theorizing									
basic analytic framework			■						
research vision	■								
Model/Theory Principles									
model decomposition framework				■					
support (de-)composition framework					■				
Models and Theories									
HASTI				■					
CoSTH					■				
Theory Application Principles									
design representations						■			
empirical techniques								■	
<i>secondary</i>									
Reviews and Summaries									
cognitive support phenomena		■							
applied model building principles				■					
Debates and Clarifications									
inadequacies of non-theoretical SE	■								
role for theory in design	■					■			
Generated Resources/Byproducts									
design idea cookbook					■				
theory application examples					■		■		
design stances						■		■	

Chapter Contents	
2	vision
3	phenomena
4	RODS
5	HASTI
6	CoSTH
7	design
8	test drive
9	field study

Table 10.1: Diagram of how contributions are spread across the chapters

cognitive support theory research, and it provided a glimpse at what theory-based research might do for SE tools research.

2. **Principles of Applied Models and Theories of Cognitive Support.** Principles and rules were described for constructing useful models and theories for cognitive support.

(a) *Model Decomposition Framework.* The key principle for constructing HASTI was to usefully decompose the modeling issues of interest, and to match the computational structure of HASTI to this decomposition. The framework is a critical structure for deciding on how to integrate the disparate theoretical content found in the literature.

(b) *Support (De-)Composition Framework.* There are many different variations of cognitive support ideas but these were shown to be hierarchically decomposable into three main families.

3. **Models and Theories.** Integrative, broad-brush models and theories were proposed for analyzing and explaining cognitive support.

- (a) *HASTI*. This was a modeling framework constructed from a variety of different prior models. Based on the main modeling components, one could view this as a kind of integration of four well-known modeling methods: the Model Human Processor [94], blackboard models for opportunism [298], the Resources model [719], and the Skills–Rules–Knowledge framework [526].
- (b) *CoSTH*. This is a hierarchically refined collection of theories of cognitive support. At a fundamental level, the hierarchy is merely an elaboration of the tree generated by applying the support factors (RODS) to the DC model framework (HASTI).

Both of these are substantially based on existing theory. This shows that integrative accounts are not premature. The collection and hierarchical structuring brings added value.

4. **Theory Application Principles.** Principles and techniques were outlined for applying cognitive support theory in tools research.

- (a) *Design Representations*. Three methods (vocabulary and concepts, design stances, and reified design space) were explored for making cognitive support theory directly usable in design.
- (b) *Empirical Techniques*. Ways of cheaply applying theories in experimental and field contexts were explored. Lightweight methods are required if cognitive support theories are to be effectively deployed in relatively informal or commercial contexts.

Secondary Contributions

In order to be able to make the primary contributions of the dissertation, some minor and secondary advances had to be made to fill in some missing details, and to solidify a position from which to argue the main points.

1. **Reviews and Summaries.** This work rests on collected wisdom rather than hinging on specific experiments, theories, or papers. Reviewing and summarizing many prior works was necessary to bring the diffuse elements into a coherent arrangement that was strong enough to hang the required arguments on.

- (a) *Review Of Support Phenomena*. A broad range of cognitive support phenomena were collected, organized, and described. This breadth helped ensure that the applied theorizing was not too limited of scope to be especially useful.
- (b) *Applied Model Building Principles*. Principles for building useful applied models for SE research were summarized. These principles are generally quite different from typical principles from cognitive science. This review is important because it appears that many times the assumptions and ideals from cognitive science have been imported into SE research without much explicit consideration. This review of contrasting principles act as a foil, as much as it acts as an organizing force for defining the applied theories.

2. **Debates and Clarifications.** Making progress in the presence of many conflicting positions and views on the subject matter can be difficult. Debates and clarifications of positions were engaged in to deflect unnecessary conflicts. In these cases the issues were too complex to be briefly refuted in a few introductory paragraphs: the various positions needed to be unfolded, the unimportant challenges needed to be pruned, and the critical ones needed to be singled out.
 - (a) *Inadequacy of Non-theoretical Work in SE.* The central importance and possibilities of cognitive support theories was debated. This debate countered contentions that the current course of research is adequate. Essentially, the debate served to establish an argument that pursuing explicit cognitive support theories may be the only realistic hope of answering many questions that are important to SE tools research.
 - (b) *Role for HCI Theory in Design.* Using the metaphors of a fitness landscape and a gulf of synthesis, the essential requirements and qualities of design theories in HCI-related design were considered. A useful taxonomy of theory applications was also described, and this was used to emphasize the importance of FP-reasoning (forward reasoning about positive consequences). This summarized the crucial role for broad-brush, summative approximations suitable for reasoning about positive consequences. CoSTH was built to exhibit these qualities: this summary makes it possible to fully appreciate the design decisions underlying the construction of HASTI and CoSTH.
3. **Generated Resources (“byproducts”).** A thesis about how to improve tools research using cognitive support theories was investigated. Along the way, other artifacts of value were created which could be useful in their own right.
 - (a) *Design Idea “Cookbook” and Index.* CoSTH was described using a combination of abstract explanations and concrete examples. Collectively, the examples sample and illustrate a diverse range of supportive techniques and technologies. In a sense, they provide a simple (although very preliminary) catalogue of design ideas that might be useful to thumb through for inspiration. Moreover, they are hierarchically organized according to the types of support that they offer. That is, they are design knowledge indexed by solution type. This is a promising area of development for building reusable design knowledge repositories.
 - (b) *Examples of Theory Applications.* Another source of useful reusable knowledge consists of examples of applying HASTI and CoSTH to analyze cognitive support in existing tools. Similar tools should beget similar analyses, which may mean being able to reuse much of the reasoning, modeling methods, or protocol coding techniques. Prime examples from the field of reverse engineering (RMTool and Rigi) were analyzed, and a number of smaller analyses of various tools (including the commercial tool Visual Café) were also performed.
 - (c) *Design Stances.* A small collection of design stances were supplied for building reverse engineering and software comprehension tools. These were developed primarily from the basic DC commitment to viewing engineers as one part of a joint computational system. They provided useful contrasts to the currently dominant—but limited—design stances.

10.2 Future Work

It is not much of an exaggeration to say that at the end of this work, what we are left with is not a resolution of a simple question, but a new beginning for research. In a very real sense what was done here was to wrestle the very complicated issues of cognitive support theory development into a position where research into applied cognitive support can begin in earnest. There are many ways of expanding the resources and analyses given in this dissertation. Also, there are many possibilities for applying them to improve SE research. Here, I have listed some of the possibilities that, to me, seem most promising at the present time.

Expanding The Current Work

In many places I have made it clear that I consider the theories and models developed here to be mere starting points—as works in progress. There are a number of promising ways of expanding the current resources in the future:

1. **Expanded Support Analysis.** Once a theory is proposed there is often a tendency to begin pointing out its inaccuracies, and then to begin fixing these by tweaking the theory. This is almost always a very inward looking process. My instinct says that the main advantage of CoSTH comes primarily from its ability to tackle a variety of different types of support at once. Although CoSTH provides explanations for a very interesting variety of cognitive support, it is not nearly exhaustive enough. Some of these shortcomings were pointed out in Section 6.7, but doubtlessly more will emerge. A key challenge is to temporarily resist the temptation to quibble about minor aspects of the framework, and to try to expand the frameworks to encompass other important support concepts and issues.
2. **Design Tradeoffs.** One limitation of CoSTH is that the design tradeoffs are not made explicit. For instance, there is a cost to externalizing data, so data redistribution is only sometimes cost effective. Somehow integrating an analysis of design tradeoffs into CoSTH seems very important. It may help to be able to quantify and measure important costs and benefits (below).
3. **Quantification and Measurement of Costs and Benefits.** External memories expand the effective size of problem solving memory: how large is a joint system's effective working memory size? An external memory creates external memory maintenance costs: what is the maximum average external memory maintenance cost that typical code refactoring tasks will bear? A tool is being developed to meet certain cognitive support requirements: can the cognitive support offered be measured to assure quality? How much memory is offloaded by a tool? To create an engineering discipline for cognitive support, some quantification and measurement methods are needed. With support explanations firmly in place, this possibility can be entertained. What it may take to make this a reality is to extend CoSTH or related theories with measurements regarding important system variables like effective working memory size. This basic route has been very successfully taken by Card *et al.* [94] for building engineering models of performance. The approximation and calculation which result appear important for engineering applications [309,448]. Analogous approximated theories for measurable aspects of cognitive support may be feasible.

4. **Groups and Multi-Agent Systems.** To make this work even half decently tractable, multi-person systems have not been considered. However DC as a theoretical framework has attracted a fair share of attention for being able to focus on social and group issues, and being able to integrate them into a holistic account. It may be fruitful to expand HASTI (and possibly CoSTH, if needed) to consider multi-agent systems.

Many of these sorts of lists are written “in no particular order”. Not so this list. The order I have listed these indicate what I consider to be an appropriate priority list. Nothing prevents future work from exploring all at once, but the priorities, I feel, are as listed.

Applying The Current Work

In Chapter 2, I recalled the aphorism stating that there is nothing so useful as a good theory. RODS, HASTI and CoSTH are effectively new weapons in the arsenal that an investigator of SE tools can apply to research and design problems. Ultimately, their most important impact might be realized if they are taken up and integrated into a discipline of cognitive claims analysis and testing. Even before then, however, one can look to attack the level of craft knowledge in the field by applying the theoretical toolkit in a number of ways. Some of the especially promising lines of attack include:

1. **Generalizable and Reusable DC Architectures.** Although humans are highly adaptable, and tasks are numerous and varied, it is unlikely that in practice more than a smallish number of basic tool and interaction types will be in common use at once. For instance, many software development IDEs are very similar, and many windowing systems have a very similar underlying logical structure even if the particulars for interaction change. When one combines this fact with the assumption that these tools are part of a DC system, it strongly suggests that it will be fruitful to search for common DC structures that apply to many different tools or tool types. If true, a number of common HASTI analyses would tend to crop up. These might be codified as “standard” DC architecture that can be pulled out of a textbook as required. Virtual architectures may also need to be defined in order to be able to reason about these architectures at a higher level.
2. **Design Catalogs and Patterns.** The presentation of CoSTH in Chapter 6 effectively provides an indexing structure that matches needs (e.g., offloading knowledge) with examples of means for achieving them (e.g., wizards). This is a fundamental design resource. In addition, Section 6.5.2 presented examples where cognitive support appear to compose into particularly harmonious combinations; they seem to do so in a patterned way. The analysis suggests that the CoSTH theoretical apparatus can be used to *mine* the existing craft knowledge base for examples of support techniques and excellent patterns of combination. These could spark a fruitful research programme of mining and codifying design knowledge for cognitive support.
3. **Evaluating and Comparing Software Tools.** Even before new tools are created, our current crop of tools are poorly understood from a cognitive support point of view. It may help to expand the analysis and empirical investigations started in Chapters 8 and 9.

10.3 Coda

Research involving SE tools needs to begin importing and using cognitive support theories instead of avoiding the topic or trying to build them from within. RODS provides a basic foundation for doing this. HASTI and CoSTH are a modeling framework and a cognitive support theory framework that are suitable starting points. They collect together and abstract many existing theories from cognitive science. They are demonstrably applicable to analyzing current reverse engineering tools and even commercial development tools. Cognitive support *is* an important concern for SE. For far too long, we have lived with an itch we just could not effectively scratch. My hope is that SE can use this toolkit to explore the nearly uncharted research stream of applying cognitive support theories in SE tools.

Appendix A

Invitation to Participate

INVITATION TO PARTICIPATE IN A STUDY

You are invited to participate in a study of programmer tool use. We are studying how programmer tools help or hinder the work of those who use them. We're hoping to be able to help people build better programmer tools such as the ones you use. These studies could eventually help us do that.

STUDY DESCRIPTION

In short (no more than 40 minute) study sessions I will come to your work place and watch and record how you use computer tools such as your compiler and editor. I'll be asking you to talk about what you're thinking while you work and will be recording you work, either with a video camera or audio-only. I would like to do as many as five of these sessions, but you are not obligated in any way to participate in five sessions. For practical reasons we can do at most one of these per day. You'll control when I should come to your office, so it might take no more than a week, but if you only occasionally call me in I might take a lot longer so we will allow up to 3 months.

You'll also be given a short questionnaire about your background and how you've set up your work environment. You'll probably finish the questionnaire in 5 to 10 minutes. After the observation sessions you'll be given a chance to meet with me so that we can discuss what I observed, and so we can both ask questions about what you did and how you used your tools. This last session is entirely optional. I expect that should take no more than half an hour.

VOLUNTEER REQUIREMENTS

Volunteers must meet the following criteria:

1. You must be a relatively experienced programmer. Specifically you must have had at least two years of programming experience.
2. You must be currently working on a project involving programming. This should involve understanding or maintaining programs of at least 1500 lines of code.
3. You should have significant (at least 4 months) experience in the programming environment that you are using. Preference is made for Unix-based programmers, such as those working with Emacs, vi, grep, or other more involved Integrated Development Environments such as Centerline C++.

Appendix B

Research Description

STUDY DESCRIPTION

Why?

I wish to study how programming tools are used by programmers to do their work. Programmers and tools have been studied independently, but very little is known about how programmers and tools interact in “in the field”, or as some call it, in the “real world.” I am trying to develop a model and theory of how programmer tools help programmers understand and modify their programs. I believe that this study will help me develop these models and theories. These theories may in turn may help other researchers build better tools that make programming easier or reduce the number of bugs in the software that is developed.

How?

The main part of the study is actually watching how you use your computer tools and your working environment to solve the problems you face. In order to understand what you’re doing, I need to understand what you’re thinking, so I’ll be asking you to talk aloud while you work, and recording what you say and what you do. If feasible, I’ll be using a video camera, but may have to use only an audio tape record. In similar work most people quickly become accustomed to the camera and have no trouble ignoring it as if its not there. Most people also find talking while working unproblematic. I’ll also be recording some of what you do using programs on your computer. How I do this will depend upon your setup and what tools you use, but I will figure this out and you will not have to concern yourself with it. You’ll be deciding when these observation sessions occur since only you know when you’ll be the type of program maintenance work I’m interested in watching. These study sessions will likely last around a half hour but won’t go past 40 minutes.

Although watching your use of tools is the main goal, this type of observation-based research frequently requires help from questionnaires and interviews. Consequently I’ll want you to fill in a short questionnaire about yourself and your environment. This should take between 5 and 10 minutes. Also, after the observation sessions, both you and I may be curious about what you did and how the tools helped you do them. So I’ll want to set up an short (typically half hour) interview session in which you can ask me questions and I can ask questions back. Throughout all of this your participation is voluntary, and you may choose to cut any of our meetings short, stop the study at any time, and

ask me to erase any part of the video or other records. You can also ask to destroy these records any time after the study is finished.

Privacy and Confidentiality

All of our conversations and the observations I make will be considered private and confidential. No one other than myself and qualified researchers in my research group will have direct access to the observations that I gather. The data I gather will be held confidential to the full extent of the law and destroyed after I have finished studying it. Mind you, I hope to be able to publish models or theories that were derived from me observing you work. For scientific purposes certain facts about the observations may need to be published. However none of the specific details that could identify you or your work will be published. I will do this by referring to you in code and disguising the actual details of your work where it is needed. This is a common practice in scientific reporting and it works extremely well. Before publishing any work that directly uses your data I will contact you and give you a preprint and ask for your consent.

Likelihood Of Harm

There is very little risk of harm to yourself if you choose to participate in this study. There is a chance that you might feel somewhat uncomfortable with me observing what you do. However most people quickly become quite accustomed to being observed. Also, I will be monitoring your computer work by running programs that record your keystrokes and the responses by the computer. There is a small possibility that these programs will interfere with your computer or its operations. You are free to examine these programs before they are installed and operated.

Withdrawing

You may withdraw from participation at absolutely any time, in whole or in part. You can ask me to destroy any of the observations of your work that I gather. There will be no obligation for you to continue.

Appendix C

Questionnaire

1. Approximately how many years of programming experience have you had?
 - (a) less than 2 years
 - (b) 2-4 years
 - (c) 4-10 years
 - (d) more than 10 years
2. List the programming language or languages you feel most familiar with.
3. Approximately how long have you been using the editor or editors that you regularly use for programming (or similar ones)?
 - (a) less than 2 months
 - (b) 2-12 months
 - (c) 12-24 months
 - (d) more than 24 months
4. Approximately how long have you been using the operating system you primarily program on (or similar ones)?
 - (a) less than 2 months
 - (b) 2-12 months
 - (c) 12-24 months
 - (d) more than 24 months
5. Did you customize your program editor (for instance, by adding macros or scripts)?
6. Did you customize your windowing environment (if any)?
7. Imagine a computer error occurs and the customizations you have made to your editor and windowing environment were destroyed and could not be restored automatically. Approximately how long do you think it would take you to re-establish your customizations?
 - (a) 0-2 hours
 - (b) 2-10 hours
 - (c) probably more than 10 hours
 - (d) I might not be able to do it
 - (e) I don't know

Appendix D

Instruction Card

Instructions for Study Volunteers

1. I am trying to study computer tool use during program maintenance and program understanding. I do not expect that you will be understanding or maintaining your programs continually, so to minimize the interference with your own work, I will request that you find parts of your work that you will allow me to observe. We'll arrange for me to come in and observe how you use tools in your work. I am especially interested in situations where you don't understand your program well, such as in debugging or understanding how something works.
2. Please take time to occasionally monitor your work. *If you become aware that you might be starting a program maintenance task of interest* to the experimental study, you are asked to hold off on that work, if you can, until I can come in and observe it. At that point you can email me (Andrew Walenstein) at the email address `walenste@cs.sfu.ca`. We can then set up a time that is convenient for you. I will endeavour to be near your workplace during normal working hours for the duration of my study.
3. Generally speaking I am interested in how tools are used to understand and modify programs during relatively difficult program maintenance or program modification. You should use the following points as a guideline in evaluating whether your activity might be of interest to this study. Your work is generally of interest if the following hold:
 - (a) You anticipate doing some work either modifying your program or trying to understand some aspect of it that you don't understand (e.g. debugging).
 - (b) You expect your work to take somewhere between five or ten minutes and an hour.
 - (c) You have some uncertainty as to exactly how you will accomplish your task. For instance you might be uncertain as to whether or not you will be able to successfully modify your program as you wish to.
4. *Please do not begin working on your maintenance problem before I arrive to observe you doing it.* This includes planning what you are about to do to perform that work. It is important for me to understand how tools are used right from the beginning and working on it beforehand makes it hard for me to understand all of this process.

Appendix E

Instructions For Producing Verbal Reports

In this study, I'm interested in how you use computer tools, and am interested in what you think about when you work on your program. In order to find this out, I am going to ask you to **THINK ALOUD** as you work on your program. What I mean by "think aloud" is that I want you to tell me **EVERYTHING** you are thinking from the time you first start your work until the end of our session. I would like you to talk aloud **CONSTANTLY** while you work. I don't want you to plan out what you say or try to explain to me what you are saying. Just act as if you are alone in this room speaking to yourself. It is most important that you keep talking. If you are silent for any long period of time, I will ask you to talk. Please try to speak as clearly as possible, as I shall be recording you as you speak. Do you understand what I want you to do?

Notes:

- phrases in **CAPITALS** mean that emphasis will be placed on those words when they are spoken.
- practicing the report generation for any session was optional

Appendix F

Coding for Participant E

This appendix contains a listing of relevant observational data for the tool use episodes studied in Chapter 9. The main contents are a series of figures. These figures are briefly listed here with notes describing their contents and notational conventions.

Visual Café **Protocols**

Figures F.1, F.2, and F.3 are transcribed and coded protocols of participant E's three repair episodes. The following notes apply to these figures:

- For each of the episodes, a few lines of protocol transcripts are included before and after the actual episode in question. This is intended to help provide some context for the activity in the episode. For reference, all three episodes are presented consecutively in Figure F.4. In that figure, single horizontal lines indicate the start of an episode and double lines indicate an end.
- Computer interaction and responses are “transcribed” by short descriptions of the action at the computer interface. The aim of these transcriptions is to convey the content of the actions without trying to interpret the significance or meaning of them. Interpretation of action is accomplished elsewhere by using the coding scheme.
- Individual events are a combination of verbalizations and computer interactions. Each individual event is identified by its start time in the protocol. Times are reported only to the second. In the case that two events are labelled with the same time, the events can be identified uniquely by the time and code (no events with the same code are labelled with the same time). Note that using a finer time resolution is will not generally solve the issue of multiple events per time label. The system being observed is a joint multiprocessing system in which multiple events can occur effectively simultaneously. Furthermore, humans are multiprocessing. The participants were observed to verbalize their thought while performing other (usually skilled) activities at the same time.
- Italicized interaction transcriptions indicate inferred actions. In these cases, there is evidence that points to the described action occurring, but the action could not be detected with a sufficient level

```

1 | Error C:\src\...\servingClass.cpp[47]:
   |     Class org.HttpServletResponse not found in type declaration
2 | Error C:\src\...\servingClass.cpp[47]:
   |     Class org.WebMakerClass(int) not found in type declaration
3 | Error C:\src\...\servingClass.cpp[47]:
   |     Class org.WebDataClass(int) not found in type declaration
4 | Error C:\src\...\servingClass.cpp[49]:
   |     Class org.HttpServletResponse not found in void
   |     makeMethod(org.WebDataClass, org.WebMakerClass
5 | Error C:\src\...\servingClass.cpp[52]:
   |     Class org.PrintWriter not found in type declaration
6 | Error C:\src\...\servingClass.cpp[52]:
   |     Class org.PrintWriter not found in type declaration
7 | 6 errors, 0 warning(s)
8 | Build Failed

```

Figure F.1: Fascimilie of Participant E's first error list

of certainty. For instance, usually the videotape indicated the general area that E was glancing at, but it is impossible to tell precisely what he gazes at.

- No special transcribing codes are used in the verbal protocols except two minor points: (1) emotions and inflections are noted within “[]” brackets, and (2) multiple commas indicate a pause. Two commas indicate a short pause, three indicate a longer one. The lengths of pauses were not a concern, so no more
- Computer interactions are occasionally annotated with the interaction method used in parentheses. For instance, the compiler may be invoked by a menu selection or a key sequence, and these are denoted with “(menu)” and “(keys)”, respectively.
- The symbol “↓” is not an action code, but a denotation that the immediately preceding code applies to the current event. ↓ typically indicates that a composite action spans several unit events.
- The term “dropdown” refers to a context-sensitive menu that “drops down” near the point of typing. The dropdowns in the protocol allow a form of word completion by listing the identifiers known to be valid at the current cursor insertion point.

Visual Café **Error Lists**

Figures F.1, F.2, F.3 contain replicas of the compiler error messages that are generated in each of the three repair episodes. These are included so that E's action context can be better appreciated. As in the facsimiles, no special visual indications were given in Visual Café (bold facing, colouring, etc.). The names of the files and classes involved have been modified to preserve anonymity. In addition, because of space limitations the error lines have been reformatted to span several lines. On the participant's screen, each error message appears on a single line. Line numbers for each output line are displayed at the left to help avoid confusion.

TIME	VERBAL REPORT	COMPUTER INTERACTION	CODE
..... start of episode			
26:36	ready to compile,, see what we got		Gi.G0
26:36		invokes compiler (menu)	Xe
26:43	see compile		other
26:44		error list displayed	other
26:45	[with surprise] woah,,okaya		Ge.P2
26:47		resizes error list	enable
26:48	gotta import some stuff	<i>reads error list (~2 sec.)</i>	Gi.fault
26:50		<i>notices first item</i>	Pe.G1
26:50	oyakay, so here we go back up to the top	scrolls manually to top of file	enable
26:55		types import	repair
26:57	import uh javax dot Servlet	types javax.Servlet.	↓
27:05	http dot	clicks http on dropdown menu	↓
27:09	HttpServletResponse that we need to import	types HttpS & clicks dropdown	↓
27:12		starts new import line	enable
27:13	we need to import	<i>looking at error list (~3 sec.)</i>	Gi.fault
27:17		<i>notices last item(s)</i>	Pe.G5
27:17	ah java dot io dot	types java.io.	repair
27:21	I think it's print	types p & scrolling dropdown	↓
27:25	PrintWriter class	selects PrintWriter	↓
..... end of episode			
27:27		<i>saves file (keys)</i>	enable
27:30	so we'll try that again		Gi.G0
27:30		<i>invokes compiler (keys)</i>	Xe
27:33		error list refreshed	other
27:34	oh we may also need to um		Ge.P2

Table F.1: Protocol and codes for first Visual Café episode (Episode V1)

```

1 | Error C:\src\...\servingClass.cpp[50]:
  |     Class org.WebMakerClass(int) not found in type declaration
2 | Error C:\src\...\servingClass.cpp[50]:
  |     Class org.WebDataClass(int) not found in type declaration
3 | Error C:\src\...\servingClass.cpp[52]:
  |     Exception java.io.IOException must be caught, or it must be
  |     declared in a throws clause of this method
4 | 3 errors, 0 warning(s)
5 | Build Failed
    
```

Figure F.2: Fascimilie of Participant E’s second error list

```

1 | Error C:\src\...\servingClass.cpp[57]:
  |     Exception java.io.IOException must be caught, or it must be
  |     declared in a throws clause of this method
2 | 1 errors, 0 warning(s)
3 | Build Failed
    
```

Figure F.3: Fascimilie of Participant E’s third error list

TIME	VERBAL REPORT	COMPUTER INTERACTION	CODE
27:27		<i>saves file (key)</i>	enable
..... start of episode			
27:30	so we'll try that again		Gi.G0
27:30		<i>invokes compiler (keys)</i>	Xe
27:33		error list refreshed	other
27:31		<i>reads error list (~2 sec.)</i>	Ge.P2
27:34	oh we may also need to um	<i>notices first item</i>	Pe.G3
27:37	import our,,, own DTD		repair
27:40	interface stuff so it's a	starting new import line	↓
27:44	import org,,package,,uh,,	types org . , uses dropdowns	↓
27:52	anonML,, storage,,	↓	↓
28:00	and the storage interface		Pi.G3
28:00		begins new import line	repair
28:06	org dot,, package,, storage,,, and,,,the interface	types org . clicking dropdowns	↓
..... end of episode			
28:21		<i>saves file (key)</i>	enable
28:21		<i>invokes compiler (key)</i>	Xe
28:23		error list refreshed	other
28:25	so what error do we get?	<i>scanning errors</i>	Ge.P3

Table F.2: Protocol and codes for second Visual Café episode (Episode V2)

TIME	VERBAL REPORT	COMPUTER INTERACTION	CODE
28:21		<i>saves file (key)</i>	enable
..... start of episode			
28:21		<i>invokes compiler (key)</i>	Xe
28:23		error list refreshed	other
28:25	so what error do we get?	<i>scanning errors</i>	Ge.P3
28:26	oh we have to catch the io exception	<i>reading error</i>	Pe.G7
28:30	throws clause for the method um	reading error report aloud	other
..... end of episode			
28:35		<i>rereading? (~4 sec.)</i>	other
28:37	yeah now it's the interesting thing what		other
28:40	how should we handle this now?		↓

Table F.3: Protocol and codes for third Visual Café episode (Episode V3)

TIME	VERBAL REPORT	COMPUTER INTERACTION	CODE
26:36	ready to compile,, see what we got		Gi.G0
26:36		invokes compiler (menu)	Xe
26:43	see compile		other
26:44		error list displayed	other
26:45	[with surprise] woah,,okaya		Ge.P2
26:47		resizes error list	enable
26:48	gotta import some stuff	<i>reads error list (~2 sec.)</i>	Gi.fault
26:50		<i>notices first item</i>	Pe.G1
26:50	oyakay, so here we go back up to the top	scrolls manually to top of file	enable
26:55		types <code>import</code>	repair
26:57	import uh javax dot Servlet	types <code>javax.Servlet.</code>	↓
27:05	http dot	clicks <code>http</code> on dropdown menu	↓
27:09	HttpServletResponse that we need to import	types <code>HttpS</code> & clicks dropdown	↓
27:12		starts new <code>import</code> line	enable
27:13	we need to import	<i>looking at error list (~3 sec.)</i>	Gi.fault
27:17		<i>notices last item(s)</i>	Pe.G5
27:17	ah java dot io dot	types <code>java.io.</code>	repair
27:21	I think it's print	types <code>p</code> & scrolling dropdown	↓
27:25	PrintWriter class	selects <code>PrintWriter</code>	↓
27:27		<i>saves file (key)</i>	enable
27:30	so we'll try that again		Gi.G0
27:30		<i>invokes compiler (keys)</i>	Xe
27:33		error list refreshed	other
27:31		<i>reads error list (~2 sec.)</i>	Ge.P2
27:34	oh we may also need to um	<i>notices first item</i>	Pe.G3
27:37	import our,, own DTD		repair
27:40	interface stuff so it's a	starting new <code>import</code> line	↓
27:44	import org,,package,,uh,,	types <code>org.</code> , uses dropdowns	↓
27:52	anonML,, storage,,	↓	↓
28:00	and the storage interface		Pi.G3
28:00		begins new <code>import</code> line	repair
28:06	org dot,, package,, storage,, and,,the interface	types <code>org.</code> clicking dropdowns	↓
28:21		<i>saves file (key)</i>	enable
28:21		<i>invokes compiler (key)</i>	Xe
28:23		error list refreshed	other
28:25	so what error do we get?	<i>scanning errors</i>	Ge.P3
28:26	oh we have to catch the io exception	<i>reading error</i>	Pe.G7
28:30	throws clause for the method um	<i>reading error report aloud</i>	other

Table F.4: Full Visual Café protocol for the three episodes

Bibliography

- [1] *SNiFF+ Release 2.2 User's Guide and Reference*, July 31, 1996 ed. , Product Number SNiFF-URG-022.
- [2] Ackerman, M. S., and Halverson, C. A. Reexamining organizational memory. *Communications of the ACM*, 43(1), Jan. 2000, pp. 59–66.
- [3] Ackermann, D., and Tauber, M. J., Eds. *Mental Models and Human-Computer Interaction 1*, vol. 3 of *Human Factors in Information Technology*. North Holland, Amsterdam, The Netherlands, 1990.
- [4] Adelson, B. When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 10(3), 1984, pp. 483–495.
- [5] Adelson, B. Modeling software design within a problem-space architecture. In *Program of the Tenth Annual Conference of the Cognitive Science Society* (Montreal, Québec, Aug 17–19 1988), Lawrence Erlbaum Associates, 1988, pp. 174–180.
- [6] Adelson, B., and Soloway, E. M. A model of software design. In Chi *et al.* [123], pp. 185–208.
- [7] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
- [8] Allen, B., and Brown, S. W. Introduction to the special issue on multimedia and interactivity. *Instructional Science*, 25(2), Mar. 1997, pp. 75–77.
- [9] Allen, B. L. *Information Tasks: Toward a User-Centered Approach to Information Systems*. Academic Press Limited, 1996.
- [10] Altmann, E. M. Episodic memory for external information. Tech. Rep. CMU-CS-96-167, Carnegie Mellon University, School of Computer Science, Aug. 1996. Phd Thesis.
- [11] Altmann, E. M., Larkin, J. H., and John, B. E. Display navigation by an expert programmer: A preliminary model of memory. In CHI'95 [733], pp. 3–10.
- [12] Anderson, J. R. Methodologies for studying human knowledge. *Behavioural and Brain Sciences*, 10, 1987, pp. 467–505.
- [13] Anderson, J. R. *Cognitive Science and its Implications*, 3rd ed. Freeman, 1990.
- [14] Anderson, J. R., Boyle, C. F., Farrell, R., and Reiser, B. J. Cognitive principles in the design of computer tutors. In *Modelling cognition*. Wiley, New York, 1996, ch. 4, pp. 93–134.
- [15] Arias, E., Eden, H., Fischer, G., Gorman, A., and Scharff, E. Transcending the individual human mind—creating shared understanding through collaborative design. *ACM Transactions on Computer-Human Interaction*, 7(1), Mar. 2000, pp. 84–113.

- [16] Arnold, R. S., and Bohner, S. A. Impact analysis—towards a framework for comparison. In *Proceedings of the IEEE Conference on Software Maintenance – 1993* (Montreal, Québec, Sep 27–30 1993), D. Card, Ed., IEEE Computer Society Press, 1993, pp. 292–301.
- [17] Arunachalam, V., and Sasso, W. Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering. *The Journal of Systems and Software*, 34(2), 1996, pp. 177–189.
- [18] Bacon, D. F., Graham, S. L., and Sharp, O. J. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), Dec. 1997, pp. 345–420.
- [19] Baecker, R. M. Experiments in on-line graphical debugging: The interrogation of complex data structures (summary only). In *Proceedings of the First Hawaii International Conference on the System Sciences*, 1968, pp. 128–129.
- [20] Baecker, R. M., and Buxton, W. A. S. Design principles and methodologies. In *Readings in Human-Computer Interaction: A Multidisciplinary Approach* [21], ch. 11, pp. 483–491.
- [21] Baecker, R. M., and Buxton, W. A. S., Eds. *Readings in Human-Computer Interaction: A Multidisciplinary Approach*. Morgan Kaufmann, Los Altos, CA, 1987.
- [22] Baecker, R. M., Nastos, D., Posner, I. R., and Mawby, K. L. The user-centred iterative design of collaborative writing software. In INTERCHI'93 [757], pp. 399–405.
- [23] Baker, M. J., and Eick, S. G. Visualizing software systems. In ICSE'94 [754], pp. 59–67.
- [24] Ball, L. J., and Ormerod, T. C. Applying ethnography in the analysis and support of expertise in engineering design. *Design Studies*, 21(4), July 2000, pp. 403–423.
- [25] Balmas, F. Query by outlines: A new paradigm to help manage programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toulouse, France, Sep 6 1999), Association for Computing Machinery, 1999, pp. 86–94.
- [26] Bannon, L. J., and Bødker, S. Beyond the interface: Encountering artifacts in use. In Carroll [101], ch. 12, pp. 227–253.
- [27] Barnard, P., May, J., Duke, D., and Duce, D. Systems, interactions, and macrotheory. *ACM Transactions on Computer-Human Interaction*, 7(2), June 2000, pp. 222–262.
- [28] Barnard, P. J. Bridging between basic theories and the artifacts of human-computer interaction. In Carroll [101], ch. 7, pp. 103–127.
- [29] Barnard, P. J., and Harrison, M. D. Integrating cognitive and system models in human computer interaction. In HCI'89 [747], pp. 87–103.
- [30] Barnard, P. J., and May, J. Cognitive modelling for user requirements. In Byerley *et al.* [90], ch. 2.2, pp. 101–145.
- [31] Barwise, J., and Shimojima, A. Surrogate reasoning. *Cognitive Studies: Bulletin of the Japanese Cognitive Science Society*, 2(4), Nov. 1995, pp. 7–27.
- [32] Basalla, G. *The Evolution of Technology*. Cambridge University Press, 1988.
- [33] Basili, V. R. A plan for empirical studies of programmers. In ESP'86 [741], pp. 252–255.
- [34] Basili, V. R., and Mills, H. D. Understanding and documenting programs. *IEEE Transactions on Software Engineering*, SE-8(3), May 1982, pp. 270–283.
- [35] Basili, V. R., Shull, F., and Lanubile, F. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4), July 1999, pp. 456–473.

- [36] Bass, L., Kazman, R., and Little, R. Toward a software engineering model of human-computer interaction. In *Engineering for Human-Computer Interaction, Proceedings of the IFIP WG2.7 Working Conference* (Ellivuori, Finland, Aug 10–12 1993), North Holland, 1993, pp. 131–153.
- [37] Bates, M. J. The design of browsing and berrypicking techniques for the online search interface. *Online Review*, 13(5), 1989, pp. 407–424.
- [38] Baya, V., and Leifer, L. J. Understanding information in conceptual design. In Cross *et al.* [153], pp. 151–168.
- [39] Bellamy, R. K. E. Strategy analysis: An approach to psychological analysis of artifacts. In Gilmore *et al.* [242], pp. 57–67.
- [40] Bellamy, R. K. E. What does pseudo-code do? A psychological analysis of the use of pseudo-code by experienced programmers. *Human Computer Interaction*, 9(2), 1994, pp. 225–246.
- [41] Bellamy, R. K. E., and Gilmore, D. J. Programming plans: Internal or external structures? In *Lines of Thinking: Reflections on the Psychology of Thought*, K. J. Gilhooly, M. T. G. Keane, R. H. Logie, and G. Erdos, Eds., vol. 2. John Wiley and Sons, 1990, ch. 4, pp. 59–72.
- [42] Bellay, B., and Gall, H. An evaluation of reverse engineering tool capabilities. *Software Maintenance: Research and Practice*, 10(5), 1998, pp. 305–331.
- [43] Bellotti, V., Shum, S. B., MacLean, A., and Hammond, N. Multidisciplinary modeling in HCI design ...in theory and in practice. In CHI'95 [733], pp. 146–153.
- [44] Berlin, L. M. Beyond program understanding: A look at programming expertise in industry. In ESP'93 [744], pp. 6–25.
- [45] Beynon, M., Roe, C., Ward, A., and Wong, A. Interactive situation models for cognitive aspects of user-artefact interaction. In CT'2001 [738], pp. 356–372.
- [46] Bhavnani, S. K. Designs conducive to the use of efficient strategies. In DIS'2000 [739], pp. 338–345.
- [47] Bhavnani, S. K., and John, B. E. From sufficient to efficient usage: An analysis of strategic knowledge. In CHI'97 [735], pp. 91–98.
- [48] Bhavnani, S. K., and John, B. E. Delegation and circumvention: Two faces of efficiency. In CHI'98 [736], pp. 273–280.
- [49] Bhavnani, S. K., and John, B. E. The strategic use of complex computer systems. *Human-Computer Interaction*, 15(2/3), 2000, pp. 107–137.
- [50] Bibby, P. A. Distributed knowledge: in the head, in the world or in the interaction? In Rogers *et al.* [548], ch. 7, pp. 93–99.
- [51] Black, J. B., Kay, D. S., and Soloway, E. M. Goal and plan knowledge representations: From stories to text editors and programs. In Carroll [100], ch. 3, pp. 36–60.
- [52] Blackler, F. Activity theory, CSCW and organizations. In Monk and Gilbert [415], ch. 10, pp. 223–249.
- [53] Blackwell, A. F. Metacognitive theories of visual programming: What do we think we are doing? In *12th International IEEE Symposium on Visual Languages* (Boulder, Colorado, Sep 3–6 1996), IEEE Computer Society Press, 1996, pp. 240–246.
- [54] Blackwell, A. F., Britton, C., Cox, A., Green, T. R. G., Gurr, C., Kadoda, G., Kutar, M. S., Loomes, M., Nehaniv, C. L., Petre, M., Roast, C., Roe, C., Wong, A., and Young, R. M. Cognitive dimensions of notations: Design tools for cognitive technology. In CT'2001 [738], pp. 325–341.

- [55] Blackwell, A. F., and Engelhardt, Y. A taxonomy of diagram taxonomies. In *Proceedings of Thinking with Diagrams 98: Is there a science of diagrams?*, 1998, pp. 60–70.
- [56] Blaha, M., and Benson, I. Teaching database reverse engineering. In WCRE'2000 [765], pp. 79–85.
- [57] Blandford, A. E., Buckingham Shum, S. J., and Young, R. M. Training software engineers in a novel usability evaluation technique. *International Journal of Human-Computer Studies*, 49(3), 1998, pp. 245–279.
- [58] Blandford, A. E., Harrison, M. D., and Barnard, P. J. Using Interaction Framework to guide the design of interactive systems. *International Journal of Human-Computer Studies*, 43(1), 1995, pp. 101–130.
- [59] Block, L. G., and Morwitz, V. G. Shopping lists as an external memory aid for grocery shopping: Influences on list writing and list fulfillment. *Journal of Consumer Psychology*, 8(4), 1999, pp. 343–375.
- [60] Blomberg, J. L. Ethnography: Aligning field studies of work and system design. In Monk and Gilbert [415], ch. 8, pp. 175–198.
- [61] Boehm-Davis, D. A. Software comprehension. In Helander [302], ch. 5, pp. 107–133.
- [62] Bonnardel, N. Creativity in design activities: The role of analogies in a constrained cognitive environment. In *Proceedings of the Third Conference on Creativity and Cognition* (Loughborough, UK, Oct 11–13 1999), Association for Computing Machinery, 1999, pp. 158–160.
- [63] Bowdidge, R. W., and Griswold, W. G. Automated support for encapsulating abstract data types. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (New Orleans, Louisiana, Dec 6–9 1994), Association for Computing Machinery, 1994, pp. 97–110.
- [64] Bowdidge, R. W., and Griswold, W. G. How software engineering tools organize programmer behavior during the task of data encapsulation. *Empirical Software Engineering*, 2(3), Sept. 1997, pp. 221–267.
- [65] Boy, G. A. *Intelligent Assistant Systems*, vol. 6 of *Knowledge-Based Systems*. Academic Press Limited, 1991.
- [66] Boy, G. A. Cognitive function analysis for human-centered automation of safety-critical systems. In CHI'98 [736], pp. 265–272.
- [67] Brade, K., Guzdial, M., Steckel, M., and Soloway, E. M. Whorf: A visualization tool for software maintenance. In *Proceedings of the 1992 IEEE Workshop on Visual Languages* (Seattle, WA, Sep 15–18 1992), IEEE Computer Society Press, 1992, pp. 148–154.
- [68] Brooks, A., Miller, J., Roper, M., and Wood, M. Criticisms of an empirical study of recursion and iteration. Tech. Rep. EFoCS-1-92, University of Strathclyde, Empirical Studies of Computer Science, Department of Computer Science, 1992.
- [69] Brooks, R. A. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1), Mar. 1986.
- [70] Brooks, R. A. Intelligence without reason. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence* (Sydney, Australia, Aug 20–24 1991), J. Mylopoulos and R. Reiter, Eds., Morgan Kaufmann, 1991, pp. 569–595.
- [71] Brooks, R. E. A model of human cognitive behavior in writing code for computer programs. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (Tbilisi, Georgia, USSR), William Kaufman, 1975, pp. 878–884.
- [72] Brooks, R. E. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6), 1977, pp. 737–751.

- [73] Brooks, R. E. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd International Conference on Software Engineering*, 1978, pp. 196–201.
- [74] Brooks, R. E. Studying programmer behaviour experimentally: the problems of proper methodology. *Communications of the ACM*, 23(4), 1980, pp. 207–213.
- [75] Brooks, R. E. A theoretical analysis of the role of documentation in the comprehension of computer programs. In *Proceedings of Human Factors in Computer Systems* (Gaithersburg, Maryland, Mar 15–17 1982), M. Schneider, Ed., Association for Computing Machinery, 1982, pp. 125–129.
- [76] Brooks, R. E. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6), 1983, pp. 543–554.
- [77] Brooks, R. E. Comparative task analysis: An alternative direction for human-computer interaction science. In Carroll [101], ch. 4, pp. 50–61.
- [78] Brooks Jr., F. P. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4), Apr. 1987, pp. 10–19.
- [79] Brooks Jr., F. P. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison Wesley, 1995.
- [80] Brouwer-Janse, M. D., and Harrington, T. L., Eds. *Human-Machine Communication for Educational Systems Design*, vol. 129 of NATO ASI Series. Series F, Computer and System Sciences. Springer-Verlag, 1994.
- [81] Brown, A., and Wallnau, K. A framework for systematic evaluation of software technologies. *IEEE Software*, 13(5), Sept. 1996.
- [82] Brown, A. W., Earl, A. N., and McDermid, J. A. *Software Engineering Environments: Automated Support for Software Engineering*. McGraw-Hill, 1992.
- [83] Brown, P. J. Integrated hypertext and program understanding tools. *IBM Systems Journal*, 30(3), 1991, pp. 363–391.
- [84] Buckingham Shum, S. Practise what we preach: Making HCI design techniques usable. In *STIMDI-95: Annual Conference of the Swedish Interdisciplinary Interest Group for Human-Computer Interaction* (University of Uppsala, Sweden, May 22–23 1995), 1995.
- [85] Buckingham Shum, S., and Hammond, N. Argumentation-based design rationale: What use at what cost? *International Journal of Human-Computer Studies*, 40(4), Apr. 1994, pp. 603–652.
- [86] Buckingham Shum, S., and Hammond, N. Delivering HCI modelling to designers: A framework and case study of cognitive modelling. *Interacting With Computers*, 6(3), 1994, pp. 314–341.
- [87] Bush, V. As we may think. *Interactions*, 3(2), Mar. 1996. Originally published in “The Atlantic Monthly”, July, 1945.
- [88] Butler, S. *Erewhon, or Over the Range*, vol. 2 of *The Shewsbury Edition of the Works of Samuel Butler*. AMS Press, 1968.
- [89] Butterworth, R., Blandford, A., and Duke, D. Using formal models to explore display-based usability. *Journal of Visual Languages and Computing*, 10(5), 1999, pp. 455–479.
- [90] Byerley, P. F., Barnard, P. J., and May, J., Eds. *Computers, Communication and Usability: Design Issues, Research and Methods for Integrated Services*. Elsevier Science Ltd., Amsterdam, 1993.
- [91] Byrne, E. J. A conceptual foundation for software re-engineering. In *Proceedings of the IEEE Conference on Software Maintenance – 1992*, IEEE Computer Society Press, 1992, pp. 226–235.

- [92] Card, S., and Moran, T. User technology: From pointing to pondering. In HPW'86 [751], pp. 183–198. Reprinted in "Readings in Human–Computer Interaction: Toward the Year 2000", R. M. Baecker, J. Grudin, W. A. S. Buxton and S. Greenberg, ed., 2nd ed., Morgan-Kaufmann, 1995.
- [93] Card, S. K. Theory-driven design research. In *Applications of Human Performance Models to System Design* (Proceedings of the NATO Reserach Study Group 9 Workshop, Orlando, Fla, 1988), G. R. McMillan, D. Beevis, E. Salas, M. H. Strub, and R. Sutton, Eds., Plenum, 1989, pp. 501–509.
- [94] Card, S. K., Moran, T. P., and Newell, A. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.
- [95] Card, S. K., Robert, J. M., and Keenan, L. N. On-line composition of text. In INTERACT'84 [755], pp. 51–56.
- [96] Cardelli, L. Type systems. In *Handbook of Computer Science and Engineering*. CRC Press, 1997, ch. 103, pp. 2208–2236.
- [97] Carpenter, P. A., and Just, M. A. Computational modeling of high-level cognition versus hypothesis testing. In Sternberg [617], ch. 8, pp. 245–292.
- [98] Carroll, J., and Mack, R. Learning to use a word processor: By doing, by thinking, and by knowing. In *Human Factors in Computer Systems*, J. C. Thomas and M. L. Schneider, Eds., Ablex Publishing Corporation, 1984, pp. 13–51.
- [99] Carroll, J. M. Psychology and the user interface: Science is soft at the frontier. In *Proceedings of Graphics Interface '86 and Vision Interface '86* (Vancouver, BC, May 26–30 1986), Canadian Information Processing Society, 1986, pp. 186–187.
- [100] Carroll, J. M., Ed. *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. MIT Press, Cambridge, MA, 1987.
- [101] Carroll, J. M., Ed. *Designing Interaction: Psychology at the Human-Computer Interface*. Cambridge University Press, 1991.
- [102] Carroll, J. M. Artifacts and scenarios: An engineering approach. In Monk and Gilbert [415], ch. 6, pp. 121–144.
- [103] Carroll, J. M. Human–computer interaction: Psychology as a science of design. *International Journal of Human-Computer Studies*, 46(4), 1997, pp. 501–522.
- [104] Carroll, J. M., and Campbell, R. L. Artifacts as psychological theories: The case of human-computer interaction. *Behaviour and Information Technology*, 8(4), 1989, pp. 247–256.
- [105] Carroll, J. M., and Kellogg, W. A. Artifact as theory-nexus: Hermeneutics meet theory-based design. In CHI'89 [730], pp. 7–14.
- [106] Carroll, J. M., Kellogg, W. A., and Rosson, M. B. The task-artifact cycle. In Carroll [101], ch. 6, pp. 74–102.
- [107] Carroll, J. M., Mack, R. L., Robertson, S. P., and Rosson, M. B. Binding objects to scenarios of use. *International Journal of Human-Computer Studies*, 41(1/2), 1994, pp. 243–276.
- [108] Carroll, J. M., and Rosson, M. B. Usability specification as a tool in iterative development. In *Advances in Human–Computer Interaction*, H. R. Hartson, Ed., vol. 1 of *Human/Computer Interaction Series*. Ablex Publishing Corporation, Norwood, NJ, 1985, pp. 1–15.
- [109] Carroll, J. M., and Rosson, M. B. Paradox of the active user. In Carroll [100], ch. 5, pp. 80–111. Online at <http://www.winterspeak.com/columns/paradox.html>, Checked 2001/09/04.
- [110] Carroll, J. M., and Rosson, M. B. Getting around the task-artifact cycle: How to make claims and design by scenario. *ACM Transactions on Information Systems*, 10(2), 1992, pp. 181–212.

- [111] Carroll, J. M., Singer, J. A., Bellamy, R. K. E., and Alpert, S. R. A view matcher for learning smalltalk. In CHI'90 [731], pp. 431–437.
- [112] Carswell, C. M. Choosing specifiers: An evaluation of the basic task models of graphical perceptions. *Human Factors*, 34(5), 1993, pp. 535–544.
- [113] Carver, N., and Lesser, V. The evolution of blackboard control architectures. Tech. Rep. TR-92-71, Department of Computer Science, University of Massachusetts, 1992. Expanded version of paper in *Expert Systems with Applications*, 7(1), pg. 1–30, 1994.
- [114] Cary, M., and Carlson, R. A. External support and the development of problem-solving routines. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 25(4), 1999, pp. 1053–1070.
- [115] Casner, S., and Larkin, J. H. Cognitive efficiency considerations for good graphic design. In *Proceedings of the 11th Annual Conference of the Cognitive Science Society (Aug 16–19 1989)*, Lawrence Erlbaum Associates, 1989, pp. 275–282.
- [116] Casner, S. M. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2), 1991, pp. 111–151.
- [117] Chalmers, D. J. A computational foundation for the study of cognition. Cogprint ID cog000000319, <http://cogprints.soton.ac.uk>. An extended version of the paper “On Implementing a Computation”, *Minds and Machines*, 4, 1994, pp. 391–402.
- [118] Chandler, P., and Sweller, J. Cognitive load while learning to use a computer program. *Applied Cognitive Psychology*, 10(2), 1996, pp. 151–170.
- [119] Chandrasekaran, B. Design problem solving: A task analysis. *AI Magazine*, 11(4), 1990, pp. 59–71. Reprinted in “Knowledge Aided Design”, vol. 10 of *Knowledge Based Systems*, Marc Green, ed., Academic Press, 1992, pp. 25–46.
- [120] Chen, C., and Rada, R. Interacting with hypertext: A meta-analysis of experimental studies. *Human-Computer Interaction*, 11(2), 1996, pp. 125–156.
- [121] Chen, Z. Toward a better understanding of idea processors. *Information and Software Technology*, 40(10), 1998, pp. 541–553.
- [122] Cheng, P. C.-H. Functional roles for the cognitive analysis of diagrams in problem solving. In *Proceedings of the Eighteenth Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum Associates, 1996, pp. 207–212.
- [123] Chi, M. T. H., Glaser, R., and Farr, M. J., Eds. *The Nature of Expertise*. Lawrence Erlbaum Associates, 1988.
- [124] Chignell, M., Hancock, P. A., and Takeshita, H. Human—computer interaction: The psychology of augmented human behavior. In Hancock [287], ch. 11, pp. 291–327.
- [125] Chignell, M. H., Motoyama, T., and Melo, V. Discount video analysis for usability engineering. In HCII'95 [750], pp. 323–328.
- [126] Chikofsky, E. J., and Cross II, J. H. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1), Jan. 1990, pp. 13–17.
- [127] Chin, D. N., and Quilici, A. DECODE: A cooperative program understanding environment. *Software Maintenance—Research and Practice*, 8(1), 1996, pp. 3–34.
- [128] Chipman, S. F., Schraagen, J. M., and Shalin, V. L. Introduction to cognitive task analysis. In Schraagen *et al.* [570], ch. 1, pp. 3–23.

- [129] Choi, E. M., and von Mayrhauser, A. Assessment of support for program understanding. In *Proceedings of the Second Symposium on Assessment of Quality Software Development Tools* (May 27–29 1992), E. Nahouraii, Ed., 1992, pp. 102–111.
- [130] Christensen, S., Jorgensen, J. B., and Madsen, K. H. Design as interaction with computer based materials. In *DIS'97* [740], pp. 65–71.
- [131] Christie, J. M., and Just, M. A. Remembering the location and content of sentences in a prose passage. *Journal of Educational Psychology*, 68(6), 1976, pp. 702–710.
- [132] Cioch, F. A., Palazzolo, M., and Lohrer, S. A documentation suite for maintenance programmers. In *Proceedings of the IEEE Conference on Software Maintenance – 1996* (Monterey, California, Nov 4–8 1996), IEEE Computer Society Press, 1996, pp. 286–295.
- [133] Clancey, W. J. AI: Inventing a new kind of machine. *ACM Computer Surveys*, 27(8), Sept. 1995.
- [134] Clancey, W. J. *Situated Cognition: on Human Knowledge and Computer Representations*. Cambridge University Press, Cambridge, UK, 1997.
- [135] Clancey, W. J. Interactive coordination processes: How the brain accomplishes what we take for granted in computer languages—and then does it better. In Pylyshyn [521], ch. 9, pp. 165–189.
- [136] Clark, A. *Being There: Putting Brain, Body, and World Together Again*. MIT Press, 1997.
- [137] Clark, A., and Chalmers, D. The extended mind. *Analysis*, 58, 1998, pp. 10–23. Reprinted in *The Philosopher's Annual*, 1998.
- [138] Clarke, C., Cox, A., and Sim, S. Searching program source code with a structured text retrieval system. In *Proceedings on the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Berkeley, CA, Aug 15–19 1999), Association for Computing Machinery, 1999, pp. 307–308.
- [139] Clayton, R., Rugaber, S., Taylor, L., and Wills, L. M. A case study of domain-based program understanding. In *WPC'97* [770].
- [140] Clegg, C. Psychology and information technology: The study of cognition in organizations. *The British Journal of Psychology*, 85, Nov. 1994, pp. 449–477.
- [141] Cole, E., and Dehdashti, P. Computer-based cognitive prosthetics: Assistive technology for the treatment of cognitive disabilities. In *Proceedings of the Third International ACM Conference on Assistive Technologies (ASSETS'98)* (Marina del Rey, CA, Apr 15–17 1998), 1998, pp. 11–18.
- [142] Cole, M., and Engeström, Y. A cultural-historical approach to distributed cognition. In Salomon [558], ch. 1, pp. 1–46.
- [143] Cole, M. I. Algorithmic skeletons: A structured approach to the management of parallel computations. Tech. Rep. CST-56–88, University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, Oct. 1988. PhD. Thesis.
- [144] Collberg, C., Thomborson, C., and Low, D. A taxonomy of obfuscating transformations. Tech. Rep. TR-148, Department of Computer Science, The University of Auckland, Auckland, N.Z., 1997.
- [145] Consens, M. P. Creating and filtering structural data visualizations using Hygraph patterns. Tech. Rep. CSRI-302, Computer Systems Research Institute, University of Toronto, Toronto, Canada M5S 1A1, Feb. 1994. PhD. Thesis.
- [146] Consens, M. P., and Mendelzon, A. O. Hy+: A hypergraph-based query and visualization system. *ACM SIGMOD Record*, 22(2), 1993, pp. 511–516.

- [147] Cooper, R., and Shallice, T. Soar and the case for unified theories of cognition. *Cognition*, 55(2), 1995, pp. 115–149.
- [148] Coplien, J. O. *Software Patterns*. SIGS Books & Multimedia, NY, NY, 1996.
- [149] Corbi, T. A. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2), 1989, pp. 294–306.
- [150] Corkill, D. D. Blackboard systems. *AI Expert*, 9(6), Sept. 1991, pp. 40–47.
- [151] Craig, I. D. From blackboards to agents. In *Online Proceedings of the VIM Project Spring Workshop on Collaboration Between Human and Artificial Societies* (Lanjarón, Spain, May 1–3 1997), 1997. Retrieved from <http://www.maths.bath.ac.uk/~jap/VIM/lanjaron.html>, 2000/08/31.
- [152] Crosby, M. E., and Stelovsky, J. How do we read algorithms? A case study. *Computer*, 23(1), Jan. 1990, pp. 24–35.
- [153] Cross, N., Christiaans, H., and Dorst, K., Eds. *Analysing Design Activity*. John Wiley and Sons, 1996.
- [154] Cummaford, S., and Long, J. Towards a conception of HCI engineering design principles. In *Proceedings of ECCE-9: Ninth European Conference on Cognitive Ergonomics* (Limerick, Ireland, Aug 24–26 1998), Published on-line at <http://www.cs.vu.nl/eace/ECCE9/>, 1998.
- [155] Curtis, B. By the way, did anyone study any real programmers? In ESP'86 [741], pp. 256–268.
- [156] Curtis, B. Five paradigms in the psychology of programming. In Helander [302], ch. 4, pp. 87–105.
- [157] Curtis, B. Empirical studies of the software design process. In INTERACT'90 [756], pp. xxxv–xl.
- [158] Curtis, B. Techies as non-technological factors in software engineering? In ICSE'91 [753], pp. 147–148.
- [159] Curtis, B., Krasner, H., and Iscoe, N. A field study of the software design process for large systems. *Communications of the ACM*, 31(11), Nov. 1988, pp. 1268–1287.
- [160] Curtis, B., and Walz, D. The psychology of programming in the large: Team and organizational behaviour. In Hoc *et al.* [307], ch. 4.1, pp. 253–270.
- [161] Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. An empirical study evaluating depth of inheritance on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2), 1996, pp. 109–132.
- [162] Daly, J. W. *Replication and a Multi-Method Approach to Empirical Software Engineering Research*. PhD thesis, Department of Computer Science, University of Strathclyde, Glasgow, 1996.
- [163] Davies, S. P. Characterizing the program design activity: Neither strictly top-down nor globally opportunistic. *Behaviour and Information Technology*, 10(3), 1991, pp. 173–190.
- [164] Davies, S. P. The role of expertise in the development of display-based problem solving strategies. In *Proceedings of the 14th Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum Associates, 1992, pp. 797–802.
- [165] Davies, S. P. Externalising information during coding activities: Effects of expertise, environment and task. In ESP'93 [744], pp. 42–61.
- [166] Davies, S. P. Display-based skills in a complex domain: The use of external information sources in computer programming. In HCII'95 [750], pp. 641–646.
- [167] Davies, S. P. Display-based problem-solving strategies in computer programming. In ESP'96 [745].
- [168] Davies, S. P. External information sources in computer programming. the role of textual and graphical representations in support of complex problem solving activities. In *IEE Colloquium on Thinking with Diagrams* (Jan 18 1996), Institute for Electrical Engineers, 1996, pp. 21–22.
- [169] Davies, S. P., and Castell, A. M. Linking theory with ITS implementation: Models of programming and the development of programming tutors. In NATO.ASI.111 [759], pp. 172–184.

- [170] Dawkins, R. *The Blind Watchmaker*. Longmans, London, 1986.
- [171] de Greef, H. P., and Neerincx, M. A. Cognitive support: Designing aiding to supplement human knowledge. *International Journal of Human-Computer Studies*, 42(5), 1995, pp. 531–571.
- [172] de Vries, E., and de Jong, T. The design and evaluation of hypertext structures for supporting design problem solving. *Instructional Science*, 27(3), 1999, pp. 285–302.
- [173] Decker, K. S., and Lesser, V. R. Coordination assistance for mixed human and computational agent systems. In *Proceedings of the Second International Conference on Computational Agents* (McLean, Virginia, Aug 23–25 1995), 1995. Available as tech. rep. CSTR-95-31, Computer Science Dept., U. of Massachusetts, 1995.
- [174] Demirors, E., and Coyle, F. Behavioral factors in software development. In HCII'95 [750], pp. 665–670.
- [175] Denley, I., and Long, J. A planning aid for human factors evaluation practice. *Behaviour and Information Technology*, 16(4/5), 1997, pp. 203–219.
- [176] Dennett, D. C. Real patterns. *Journal of Philosophy*, LXXXVIII(I), Jan. 1991, pp. 25–51. Reprinted in *Brainchildren: Essays on Designing Minds*, MIT Press, 1998, ch 5.
- [177] Dennett, D. C. *Darwin's Dangerous Idea: Evolution and the Meanings of Life*. Simon and Schuster, New York, NY, 1995.
- [178] Derry, S. J., and Lajoie, S. P., Eds. *Computers as Cognitive Tools*. Lawrence Erlbaum Associates, 1993.
- [179] Devanbu, P. T., Brachman, R. J., Selfridge, P. G., and Ballard, B. W. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5), May 1991, pp. 34–49.
- [180] Dillenbourg, P. Distributing cognition over humans and machines. In Vosniadou *et al.* [684], ch. 9, pp. 165–183.
- [181] Dillenbourg, P., and Self, J. A. A computational approach to socially distributed cognition. *European Journal of Psychology of Education*, 7(4), 1992, pp. 352–373.
- [182] Dillon, A. Reading from paper versus screens: A critical review of the empirical literature. *Ergonomics*, 35(10), 1992, pp. 1297–1326.
- [183] Dillon, A. *Designing Usable Electronic Text: Ergonomic Aspects of Human Information Usage*. Taylor and Francis, London, 1994.
- [184] Dillon, A. TIMS: A framework for the design of usable electronic text. In van Oostendorp and de Mul [652], ch. 5, pp. 99–120.
- [185] Dillon, A., and Watson, C. User analysis in HCI: the historical lessons from individual differences research. *International Journal of Human-Computer Studies*, 45(6), 1996, pp. 619–637.
- [186] diSessa, A. A. Local sciences: Viewing the design of human–computer systems as cognitive science. In Carroll [101], ch. 10, pp. 162–202.
- [187] Dishaw, M. T., and Strong, D. M. Assessing software maintenance tool utilization using task–technology fit and fitness-for-use models. *Software Maintenance—Research and Practice*, 10(3), 1998, pp. 151–179.
- [188] Dishaw, M. T., and Strong, D. M. Supporting software maintenance with software engineering tools: A computed task–technology fit analysis. *The Journal of Systems and Software*, 44, 1998, pp. 107–120.
- [189] Dowell, J., and Long, J. Conception of the cognitive engineering design problem. *Ergonomics*, 41(2), 1998, pp. 126–139.
- [190] Draper, S. W. The nature of expertise in UNIX. In INTERACT'84 [755], pp. 465–471.

- [191] Draper, S. W. The notion of task in HCI. In INTERCHI'93 [757], pp. 207–208.
- [192] Duke, D. J., Barnard, P. J., Duce, D. A., and May, J. Syndetic modelling. *Human Computer Interaction*, 13(4), 1998, pp. 337–393.
- [193] Dunson, P. J., and Ridgeway, V. G. The effect of graphic organizers on learning and remembering information from connected discourse. *Forum for Reading*, 22(1), 1990, pp. 15–23.
- [194] Eastman, C. M. Cognitive processes and ill-defined problems: A case study from design. In *Proceedings of the International Joint Conference on Artificial Intelligence: IJCAI-69* (Washington, DC, May 9 1969), 1969, pp. 669–690.
- [195] Ecker, C., Kelly, I., and Stacey, M. Cognitive foundations for interactive generative systems in early design. In *Proceedings of the 12th International Conference on Engineering Design, ICED 99: Communication and Cooperation of Practice and Science* (Munich, Germany, Aug 24–26 1999), U. Lindemann, H. Birkhofer, H. Meerkamm, and S. Vajna, Eds., Technische Universität München, 1999.
- [196] Edwards, W., and Fasolo, B. Decision technology. *Annual Review of Psychology*, 52, 2001, pp. 581–606.
- [197] Egan, D. E. Individual differences in Human–Computer Interaction. In Helander [302], ch. 24, pp. 543–568.
- [198] Eick, S. G., Steffen, J. L., and Sumner Jr., E. E. Seesoft - A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11), Nov. 1992, pp. 957–968.
- [199] Engelbart, D. The augmented knowledge workshop. In HPW'86 [751], pp. 73–83.
- [200] Engelbart, D. C. Special considerations of the individual as a user, generator, and retriever of information. *American Documentation*, 12(2), Apr. 1961, pp. 121–125. Presented at the Annual Meeting of the American Documentation Institute, Berkeley, CA, October 23–27, 1960.
- [201] Engelbart, D. C. Toward augmenting the human intellect and boosting our collective IQ. *Communications of the ACM*, 38(8), Aug. 1995.
- [202] Engeström, Y., and Middleton, D., Eds. *Cognition and Communication at Work*. Cambridge University Press, 1996.
- [203] Engeström, Y., Miettinen, R., and Punamäki, R.-L., Eds. *Perspectives on Activity Theory*. Cambridge University Press, 1999.
- [204] Erdem, A., Johnson, W. L., and Marsella, S. User and task tailored software explanations. Retrieved from <http://www.isi.edu/isd/I-DOC/ASE99J.ps>, 2001/03/01.
- [205] Erickson, T. *Lingua Francas* for design: Sacred places and pattern languages. In DIS'2000 [739], pp. 357–368.
- [206] Ericsson, K. A., and Chase, W. G. Exceptional memory. *American Scientist*, 6, 1982, pp. 607–612.
- [207] Ericsson, K. A., and Simon, H. A. *Protocol Analysis*. MIT Press, 1984.
- [208] Fagan, M. E. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 1976, pp. 182–211. Reprinted in *IBM Systems Journal*, 38(2/3), 1999, pp. 258–287.
- [209] Fensel, D., and Motta, E. Structured development of problem solving methods. In *Eleventh Workshop on Knowledge Acquisition, Modeling and Management* (Banff, Canada, Apr 18–23 1998), 1998.
- [210] Fields, B., Wright, P. C., and Harrison, M. D. Designing human-system interaction using the resource model. In *Human Factors of IT: Enhancing Productivity and Quality of Life (Proceedings of the 1st Asia-Pacific Conference on Human Computer Interaction)* (Singapore), June 1996.
- [211] Fields, B., Wright, P. C., and Harrison, M. D. Objectives, strategies and resources as design drivers. In *Human Computer Interaction: INTERACT'97* (Sidney, Australia, Jul 14–18 1997), S. Howard, J. Hammond, and G. Lin-gaard, Eds., Chapman & Hall, 1997, pp. 164–171.

- [212] Findlay, J. M., Davies, S. P., Kentridge, R., Lambert, A. J., and Kelly, J. Optimum display arrangements for presenting visual reminders. In *HCI'88* [746], pp. 453–464.
- [213] Finnigan, P. J., Holt, R. C., Kalas, I., Kerr, S., Kontogiannis, K., Müller, H. A., Mylopoulos, J., Perelgut, S. G., Stanley, M., and Wong, K. The software bookshelf. *IBM Systems Journal*, 36(4), Nov. 1997.
- [214] Fischer, G., Grudin, J., McCall, R., Ostwald, J., Redmiles, D., Reeves, B., and Shipman, F. Seeding, evolutionary growth and reseeding: The incremental development of collaborative design environments. In *Coordination Theory and Collaboration Technology*, G. M. Olson, T. W. Malone, and J. B. Smith, Eds. Lawrence Erlbaum Associates, Mahwah, NJ, 2001, pp. 447–472.
- [215] Fischer, G., Henninger, S., and Redmiles, D. F. Cognitive tools for locating and comprehending software objects for reuse. In *ICSE'91* [753], pp. 318–328.
- [216] Fischer, G., Lemke, A. C., McCall, R., and Morch, A. I. Making argumentation serve design. In Moran and Carroll [420], ch. 9, pp. 267–293.
- [217] Fisher, C. Advancing the study of programming with computer-aided protocol analysis. In *ESP'87* [742], pp. 198–216.
- [218] Fisher, J. Defining the novice user. *Behaviour and Information Technology*, 10(5), 1991, pp. 437–441.
- [219] Flach, J., Hancock, P., Caird, J., and Vicente, K. J., Eds. *Global Perspectives on the Ecology of Human–Machine Systems*. Lawrence Erlbaum Associates, 1995.
- [220] Flach, J., Hancock, P., Caird, J., and Vicente, K. J. Preface. In *Global Perspectives on the Ecology of Human–Machine Systems* [219], pp. xi–xiv.
- [221] Flach, J. M. The ecology of human-machine systems: A personal history. In Flach *et al.* [219], ch. 1, pp. 1–13.
- [222] Fleishman, E. A., and Quaintance, M. K. *Taxonomies of Human Performance: The Description of Human Tasks*. Academic Press, Boston, MA, 1984.
- [223] Flor, N. V. Side-by-side collaboration: Case study. *International Journal of Human-Computer Studies*, 49(3), 1998, pp. 201–222.
- [224] Flor, N. V., and Hutchins, E. L. Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In *ESP'91* [743], pp. 36–64.
- [225] Flower, L. *Problem-solving Strategies for Writing*, 4th ed. Harcourt Brace Jovanovich College Publishers, Fort Worth, 1993.
- [226] Fogel, D. B. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, 1995.
- [227] Foltz, P. W. Comprehension, coherence and strategies in hypertext and linear text. In Rouet *et al.* [552], ch. 6, pp. 109–136.
- [228] Ford, L., and Tallis, D. Interacting visual abstractions of programs. In *Proceedings of the 1993 IEEE Symposium on Visual Languages* (Bergen, Norway, Aug 24–27 1993), E. P. Glinert and K. A. Olsen, Eds., IEEE Computer Society Press, 1993, pp. 93–99.
- [229] Freed, M. A., and Shafto, M. G. Human-system modeling: Some principles and a pragmatic approach. In *Design, Specification and Verification of Interactive Systems '97: Proceedings of the Eurographics Workshop* (Grenada, Spain, Jun 4–6 1997), M. D. Harrison and J. C. Torres, Eds., Springer-Verlag, 1997. Retrieved preprint from <http://human-factors.arc.nasa.gov/cognition/papers/freed/dsvi97.html>, 2000/08/31.
- [230] Furnas, G. W. Generalized fisheye views. In *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems* (Boston, MA, Apr 13–17 1986), Association for Computing Machinery, 1986, pp. 16–23.

- [231] Gaines, B. R., and Shaw, M. L. G. Concept maps as hypermedia components. *International Journal of Human-Computer Studies*, 43(3), 1995, pp. 323–361.
- [232] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable, Object-Oriented Software*. Addison Wesley, 1994.
- [233] Gannod, G., and Cheng, B. A framework for classifying and comparing software reverse engineering and design recovery tools. In *Proceedings of the Sixth Working Conference on Reverse Engineering* (Atlanta, Georgia, Oct 6–8 1999), IEEE Computer Society Press, 1999, pp. 389–398.
- [234] Garg, P. K., and Scacchi, W. On designing intelligent hypertext systems for information management in software engineering. In *Proceedings of the ACM Conference on Hypertext* (Chapel Hill, NC, Nov 13–15 1987), Association for Computing Machinery, 1987, pp. 409–432.
- [235] Gedenryd, H. *How Designers Work*. PhD thesis, Cognitive Studies, Lund University, 1998.
- [236] Gibson, J. J. *The Ecological Approach to Visual Perception*. Houghton-Mifflin, Boston, MA, 1979.
- [237] Gillies, A. C. Making information systems fit user needs. In *Proceedings of the Fifth International Conference on Human-Computer Interaction (HCI International '93)* (Orlando, Fla, Aug 8–13 1993), M. J. Smith and G. Salvendy, Eds., vol. 19A–19B of *Advances in Human Factors/Ergonomics*, Elsevier Science Ltd., 1993, pp. 391–396.
- [238] Gilmore, D. J. Methodological issues in the study of programming. In Hoc *et al.* [307], ch. 1.5, pp. 83–98.
- [239] Gilmore, D. J. Visibility: A dimensional analysis. In HCI'91 [748], pp. 317–329.
- [240] Gilmore, D. J., and Green, T. R. G. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, 21(1), 1984, pp. 31–48.
- [241] Gilmore, D. J., and Green, T. R. G. Programming plans and programming expertise. *Quarterly Journal of Experimental Psychology*, 40A(3), 1988, pp. 423–442.
- [242] Gilmore, D. J., Winder, R. L., and D tienne, F., Eds. *User-Centred Requirements for Software Engineering Environments*, vol. 123 of *NATO ASI Series F*. Springer-Verlag, 1994.
- [243] Goel, V. *Sketches of Thought*. MIT Press, Cambridge, MA, 1995.
- [244] Goel, V., and Pirolli, P. I. The structure of design problem spaces. *Cognitive Science*, 16(3), 1992, pp. 395–429.
- [245] Goguen, J. Tossing algebraic flowers down the great divide. In *People and Ideas in Theoretical Computer Science*, C. S. Calude, Ed. Springer, New York, 1999, pp. 93–129.
- [246] Goldman, S. R. Reading, writing, and learning in hypermedia environments. In van Oostendorp and de Mul [652], ch. 2, pp. 7–42.
- [247] Goldman, S. R., Zech, L. K., Biswas, G., Noser, T., and The Cognition and Technology Group at Vanderbilt. Computer technology and complex problem solving: Issues in the study of complex cognitive activity. *Instructional Science*, 27(3), 1999, pp. 235–268.
- [248] Golovchinsky, G. Hypertext interfaces for programmers. In SEHCIW'94 [762].
- [249] Good, J. The 'right' tool for the task: An investigation of external representations, program abstractions, and task requirements. In ESP'96 [745].
- [250] Good, J. *Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension*. PhD thesis, The University of Edinburgh, 1999.
- [251] Gould, J. D., and Lewis, C. Designing for usability: Key principles and what designers think. *Communications of the ACM*, 28(3), Mar. 1985, pp. 300–311.

- [252] Graham, M., Kennedy, J., and Benyon, D. Towards a methodology for developing visualizations. *International Journal of Human-Computer Studies*, 53(5), 2000, pp. 789–807.
- [253] Gray, W. D., and Altmann, E. M. Cognitive modeling and human-computer interaction. In Karwowski [341], pp. 387–391.
- [254] Gray, W. D., John, B. E., and Atwood, M. E. Project Ernestine: Validating a GOMS analysis for predicting and explaining real-world task performance. *Human Computer Interaction*, 8(3), 1993, pp. 237–309.
- [255] Gray, W. D., and Kirschenbaum, S. S. Analyzing a novel expertise: An unmarked road. In Schraagen *et al.* [570], ch. 17, pp. 275–290.
- [256] Green, C., and Gilhooly, K. Protocol analysis: Practical implementation. In *Handbook of Qualitative Research Methods for Psychology and the Social Sciences*, J. T. E. Richardson, Ed. British Psychological Society, 1996, ch. 5, pp. 55–74.
- [257] Green, T. R. G. Cognitive dimensions of notations. In HCI'89 [747], pp. 443–460.
- [258] Green, T. R. G. The cognitive dimension of viscosity: A sticky problem for HCI. In INTERACT'90 [756], pp. 79–86.
- [259] Green, T. R. G. Limited theories as a framework for human-computer interaction. In Ackermann and Tauber [3], ch. 1, pp. 3–40.
- [260] Green, T. R. G. The nature of programming. In Hoc *et al.* [307], ch. 1.2, pp. 21–44.
- [261] Green, T. R. G. Programming languages as information structures. In Hoc *et al.* [307], ch. 2.2, pp. 117–137.
- [262] Green, T. R. G. Describing information artifacts with cognitive dimensions and structure maps. In HCI'91 [748], pp. 297–315.
- [263] Green, T. R. G. Why software engineers don't listen to what psychologists don't tell them anyway. In Gilmore *et al.* [242], pp. 323–333.
- [264] Green, T. R. G. Looking through HCI. In HCI'95 [749], pp. 21–36.
- [265] Green, T. R. G. Commentary: The conception of a conception. *Ergonomics*, 41(2), 1998, pp. 143–146.
- [266] Green, T. R. G. Instructions and descriptions: Some cognitive aspects of programming and similar activities. In *Proceedings of Working Conference on Advanced Visual Interfaces (AVI 2000)* (Palermo, Italy, May 23–26 2000), V. D. Gesù, A. Levialdi, and L. Tarantino, Eds., ACM Press, 2000, pp. 21–28.
- [267] Green, T. R. G., Bellamy, R. K. E., and Parker, J. M. Parsing and gnirap: A model of device use. In ESP'87 [742], pp. 132–146.
- [268] Green, T. R. G., and Benyon, D. R. The skull beneath the skin: Entity-relationship models of information artifacts. *International Journal of Human-Computer Studies*, 44(6), 1996, pp. 801–828.
- [269] Green, T. R. G., and Blackwell, A. *Cognitive Dimensions of Information Artefacts: a Tutorial*, Oct. 1998. From <http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/>, retrieved 2000/11/30.
- [270] Green, T. R. G., Davies, S. P., and Gilmore, D. J. Delivering cognitive psychology to HCI: the problems of common language and knowledge transfer. *Interacting With Computers*, 8(1), 1996, pp. 89–111.
- [271] Green, T. R. G., Gilmore, D. J., Blumenthal, B. B., Davies, S. P., and Winder, R. L. Towards a cognitive browser for OOPS. *International Journal of Human-Computer Interaction*, 4(1), 1992, pp. 1–34.
- [272] Green, T. R. G., and Petre, M. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2), 1996, pp. 131–174.

- [273] Green, T. R. G., Petre, M., and Bellamy, R. K. E. Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. In *ESP'91* [743], pp. 121–146.
- [274] Green, T. R. G., Schiele, F., and Payne, S. J. Formalisable models of user knowledge in human–computer interaction. In *Working With Computers: Theory Versus Outcome*, G. C. van der Veer, T. R. G. Green, J.-M. Hoc, and D. Murray, Eds. Academic Press Limited, 1989, pp. 3–46.
- [275] Greenberg, S. *The Computer User as Toolsmith: The Use, Reuse, and Organization of Computer-based Tools*. Cambridge University Press, 1993.
- [276] Greenberg, S., and Thimbleby, H. The weak science of human–computer interaction. In *CHI'92 Research Symposium on Human–Computer Interaction* (Monterey, CA), 1992.
- [277] Greeno, J. Situations, mental models, and generative knowledge. In Klahr and Kotovsky [355], ch. 11, pp. 285–318.
- [278] Griswold, W. G. Coping with software change using information transparency. Tech. Rep. CS98–585, University of California, San Diego, Department of Computer Science and Engineering, Apr. 1998.
- [279] Griswold, W. G., Chen, M. I., Bowdidge, R. W., and Morgenthaler, J. D. Tool support for planning the restructuring of data abstractions in large systems. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE-4)* (San Francisco, CA, Oct 16–18 1996), Association for Computing Machinery, 1996.
- [280] Grudin, J. The computer reaches out: The historical continuity of interface design. In *CHI'90* [731], pp. 261–268.
- [281] Guarino, N. Formal ontology, conceptual analysis and knowledge representation. *International Journal of Human-Computer Studies*, 43(5–6), 1995, pp. 625–640.
- [282] Guinan, P. J., Coopriider, J. G., and Sawyer, S. The effective use of automated application development tools. *IBM Systems Journal*, 36(1), 1997, pp. 124–139.
- [283] Guindon, R. Designing the design process: Exploiting opportunistic thoughts. *Human Computer Interaction*, 5(2-3), 1990, pp. 305–344.
- [284] Guindon, R. Requirements and design of DesignVision, an object-oriented graphical interface to an intelligent software design assistant. In *CHI'92* [732], pp. 499–506.
- [285] Guindon, R., and Curtis, B. Control of cognitive processes during software design: What tools are needed? In *Proceedings of ACM CHI'88 Conference on Human Factors in Computing Systems* (Washington, D.C., May 15–19 1988), Association for Computing Machinery, 1988, pp. 263–268.
- [286] Hale, D. P., and Haworth, D. A. Towards a model of programmers' cognitive processes in software maintenance: A structural learning theory approach for debugging. *Software Maintenance—Research and Practice*, 3(1), 1991, pp. 85–106.
- [287] Hancock, P. A., Ed. *Human Performance and Ergonomics*. Academic Press Limited, 1999.
- [288] Hansen, W. J. User engineering principles for interactive systems. In *AFIPS Conference Proceedings 39*, AFIPS Press, 1971, pp. 523–532.
- [289] Harnad, S. Interactive cognition: Exploring the potential of electronic quote/commenting. In *Cognitive Technology: In Search of a Humane Interface*, B. Gorayska and J. L. Mey, Eds. Elsevier, 1995, ch. 25, pp. 397–414.
- [290] Harrison, M. D., Fields, B., and Wright, P. C. Supporting concepts of operator control in the design of functionally distributed systems. In *ALLFN'97: Revisiting the Allocation of Functions Issue*. IEA Press, 1997, pp. 215–225.
- [291] Hartman, H. J. Metacognition in teaching and learning: An introduction. *Instructional Science*, 26(1), 1998, pp. 1–3.

- [292] Hatcliff, J., Magensen, T. A., and Thiemann, P., Eds. *Partial Evaluation: Practice and Theory*, vol. 1706 of *Lecture Notes in Computer Science*. Springer, New York, 1999.
- [293] Hayes, J. R., and Nash, J. G. The nature of planning in writing. In Levy and Ransdell [383], ch. 2, pp. 29–55.
- [294] Hayes, P. Aristotelian and platonic views of knowledge representation. In CS'94 [737], pp. 1–10.
- [295] Hayes-Roth, B. BB1: An architecture for blackboard systems that control, explain, and learn about their own behavior. Tech. Rep. CS-TR-84-1034, Stanford University, Department of Computer Science, Dec. 1984.
- [296] Hayes-Roth, B. A blackboard architecture for control. *Artificial Intelligence*, 26(3), 1985, pp. 251–322.
- [297] Hayes-Roth, B. Architectural foundations for real-time performance in intelligent agents. *Real-Time Systems: The International Journal of Time-Critical Computing*, 2, 1990, pp. 99–125.
- [298] Hayes-Roth, B. Architectural foundations for real-time performance in intelligent agents. In *Second Generation Expert Systems*, J.-M. David, J.-P. Krivine, and R. Simmons, Eds. Springer-Verlag, 1993, pp. 643–672.
- [299] Hayes-Roth, B., and Hayes-Roth, F. A cognitive model of planning. *Cognitive Science*, 3, 1979, pp. 275–310.
- [300] Hayes-Roth, B., Hewett, M., Washington, R., Hewett, R., and Seiver, A. Distributing intelligence within an individual. Tech. Rep. CS-TR-88-1229, Stanford University, Department of Computer Science, Nov. 1988.
- [301] Hearst, M., Kopec, G., and Brotsky, D. Research in support of digital libraries at Xerox PARC: Part II: Paper and digital documents. *D-Lib Magazine*, June 1996. Retrieved from <http://www.dlib.org/dlib/june96/06hearst.html>, 2000/09/08.
- [302] Helander, M., Ed. *Handbook of Human-Computer Interaction*. North Holland, 1988.
- [303] Henderson, Jr., D. A., and Card, S. K. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Transactions on Graphics*, 5(3), July 1986, pp. 211–243.
- [304] Hertel, P. T. Implications of external memory for investigations of mind. *Applied Cognitive Psychology*, 7(7), Dec. 1993, pp. 665–674.
- [305] Hoc, J.-M., Ed. *Expertise and Technology: Cognition and Human-Computer Cooperation*. Lawrence Erlbaum Associates, 1995.
- [306] Hoc, J.-M. From human-machine interaction to human-machine cooperation. *Ergonomics*, 43(7), 2000, pp. 833–843.
- [307] Hoc, J.-M., Green, T. R. G., Samurcay, R., and Gilmore, D. J., Eds. *Psychology of Programming*. Academic Press Limited, San Diego, 1990.
- [308] Hoc, J.-M., Green, T. R. G., Samurcay, R., and Gilmore, D. J. Theoretical and methodological issues. In Hoc *et al.* [307], ch. 1.0, pp. 1–7. Introduction to Chapter 1.
- [309] Hockey, G. R. J., and Westerman, S. J. Commentary: Advancing human factors involvement in engineering design: a bridge not far enough? *Ergonomics*, 41(2), 1998, pp. 147–149.
- [310] Holbrook, S. H. *The Golden Age of Quackery*. Macmillan, New York, 1959.
- [311] Hollan, J., Hutchins, E., and Kirsh, D. Distributed cognition: Toward a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction*, 7(2), June 2000, pp. 174–196.
- [312] Hollnagel, E., Cacciabue, P. C., and Hoc, J.-M. Work with technology: Some fundamental issues. In Hoc [305], ch. 1, pp. 1–42.
- [313] Holt, P. O., and Williams, N., Eds. *Computers and Writing: State of the Art*. Kluwer Academic Publishers, 1992.

- [314] Howes, A. An introduction to cognitive modelling in human–computer interaction. In Monk and Gilbert [415], ch. 5, pp. 97–119.
- [315] Howes, A., and Young, R. M. The role of cognitive architecture in modelling the user: Soar’s learning mechanism. *Human Computer Interaction*, 12(4), 1997, pp. 311–343.
- [316] Hubka, V., and Eder, W. E. *Design Science: Introduction to the Needs, Scope and Organization of Engineering Design Knowledge*. Springer-Verlag, London, 1996.
- [317] Hunt, E. What is a theory of thought? In Sternberg [617], ch. 1, pp. 3–49.
- [318] Hunter, I. M. L. Memory in everyday life. In *Applied Problems in Memory*, M. M. Gruneberg and P. E. Morris, Eds. Academic Press Limited, 1979, ch. 1, pp. 1–13.
- [319] Hutchins, E. L. Learning to navigate. In *Understanding Practice: Perspectives on Activity and Context*, S. Chaiklin and J. Lave, Eds. Cambridge University Press, 1993, ch. 2, pp. 35–63.
- [320] Hutchins, E. L. *Cognition in the Wild*. MIT Press, 1995.
- [321] Hutchins, E. L. How a cockpit remembers its speed. *Cognitive Science*, 19, 1995, pp. 265–288.
- [322] Hutchins, E. L., Hollan, J. D., and Norman, D. A. Direct manipulation interfaces. In Norman and Draper [475], ch. 5, pp. 87–124.
- [323] Hutchins, E. L., and Klausen, T. Distributed cognition in an airline cockpit. In Engestroöm and Middleton [202], ch. 2, pp. 15–34.
- [324] Imagix. *Imagix-4D User’s Manual*.
- [325] Intons-Peterson, M. J. External memory aids and their relation to memory. In *Cognitive Psychology Applied*, C. Izawa, Ed. Lawrence Erlbaum Associates, 1993, ch. 6, pp. 135–158.
- [326] Intons-Peterson, M. J., and Fornier, J. External and internal memory aids: When and how often do we use them? *Journal of Experimental Psychology: General*, 115(3), Mar. 1986, pp. 267–280.
- [327] Jackson, D. Aspect: Detecting bugs with abstract dependencies. *ACM Transactions on Software Engineering and Methodology*, 4(2), Apr. 1995, pp. 109–145.
- [328] Jackson, D., and Rollins, E. J. Abstraction mechanisms for pictorial slicing. In WPC’94 [768], pp. 82–88.
- [329] Jackson, S. L., Krajcik, J., and Soloway, E. M. The design of guided learner-adaptable scaffolding in interactive learning environments. In CHI’98 [736], pp. 187–194.
- [330] Jahnke, J. H. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, Fachbereich Mathematik-Informatik, Universität Paderborn, Aug. 1999.
- [331] Jahnke, J. H., and Walenstein, A. Reverse engineering tools as media for imperfect knowledge. In WCRE’2000 [765], pp. 22–31.
- [332] Jarvenpaa, S. L., and Dickson, G. W. Graphics and managerial decision making: Research-based guidelines. *Communications of the ACM*, 31(6), June 1988, pp. 764–774.
- [333] Jennings, N. R. Coordination techniques for distributed artificial intelligence. In *Foundations of Distributed Artificial Intelligence*, G. M. P. O’Hare and N. R. Jennings, Eds. John Wiley and Sons, 1996, ch. 6, pp. 187–210.
- [334] Jennings, N. R., Sycara, K., and Wooldridge, M. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1, 1998, pp. 275–306.
- [335] John, B. E., and Kieras, D. E. Using GOMS for user interface design and evaluation: Which technique? *ACM Transactions on Computer-Human Interaction*, 3(4), Dec. 1996, pp. 320–351.

- [336] Johnson, P., Johnson, H., and Wilson, S. Rapid prototyping of user interfaces driven by task models. In *Scenario-based Design: Envisioning Work and Technology in System Development*, J. Carroll, Ed. John Wiley, 1995, pp. 209–246.
- [337] Jones, W. P. On the applied use of human memory models: The memory extender personal filing system. *International Journal of Man-Machine Studies*, 25(2), 1986, pp. 191–228.
- [338] Kaindl, H. How to identify binary relations for domain models. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, Germany, Mar 25–29 1996), IEEE Computer Society Press, 1996, pp. 28–36.
- [339] Kaptelinin, V. Human computer interaction in context: The activity theory perspective. In *East-West International Conference on Human-Computer Interaction: Proceedings of the EWHCI'92* (St.-Petersburg, Russia, Aug 4–8 1992), J. Gornostaev, Ed., International Centre for Scientific and Technical Information (ICSTI), 1992, pp. 7–13.
- [340] Kaptelinin, V., Nardi, B. A., and MacAulay, C. The activity checklist: A tool for representing the space of context. *Interactions*, 6(4), 1999, pp. 27–39.
- [341] Karwowski, W., Ed. *International Encyclopedia of Ergonomics and Human Factors*. Taylor and Francis, New York, 2001.
- [342] Kellner, M. I. Non-technological issues in software engineering: Panel session overview. In ICSE'91 [753], pp. 144–146.
- [343] Kieras, D. E., and Meyer, D. E. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human Computer Interaction*, 12(4), 1997, pp. 391–438.
- [344] Kieras, D. E., and Meyer, D. E. The role of cognitive task analysis in the application of predictive models of human performance. In Schraagen *et al.* [570], ch. 15, pp. 237–260.
- [345] Kim, J., Lerch, F. J., and Simon, H. A. Internal representation and rule development in object-oriented design. *ACM Transactions on Computer-Human Interaction*, 2(4), 1995, pp. 357–390.
- [346] Kiper, J. D., Howard, E., and Ames, C. Criteria for evaluation of visual programming languages. *Journal of Visual Languages and Computing*, 8(2), Apr. 1997, pp. 175–192.
- [347] Kirlik, A. Modeling strategic behavior in human-automation interaction: Why an aid “can” (and should) go unused. *Human Factors*, 35(2), 1993, pp. 221–242.
- [348] Kirlik, A. Requirements for psychological models to support design: Toward ecological task analysis. In Flach *et al.* [219], ch. 4, pp. 68–120.
- [349] Kirlik, A. The ecological expert: Acting to create information to guide action. In *Proceedings of the Fourth Symposium on Human Interaction with Complex Systems* (Dayton, Ohio, Mar 22–25 1998), 1998, pp. 15–28.
- [350] Kirlik, A. Everyday life environments. In *A Companion to Cognitive Science*, W. Bechtel and G. Graham, Eds. Blackwell, Malden, MA, 1998, ch. 56, pp. 702–712.
- [351] Kirlik, A., and Bisantz, A. M. Cognition in human-machine systems: Experiential and environmental aspects of adaptation. In Hancock [287], ch. 2, pp. 47–68.
- [352] Kirsh, D. Complementary strategies: Why we use our hands when we think. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum Associates, 1995.
- [353] Kirsh, D., and Maglio, P. P. On distinguishing epistemic from pragmatic actions. *Cognitive Science*, 18(4), 1994, pp. 513–549.
- [354] Kjaer-Hansen, J. Unitary theories of cognitive architectures. In Hoc [305], ch. 3, pp. 45–54.

- [355] Klahr, D., and Kotovsky, K., Eds. *Complex Information Processing: The Impact of Herbert A. Simon*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
- [356] Knuth, D. E. Computer-drawn flow charts. *Communications of the ACM*, 6(9), Sept. 1963, pp. 555–563.
- [357] Koenemann, J., and Robertson, S. P. Expert problem solving strategies for program comprehension. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems* (New Orleans, LA, Apr 27–May 2 1991), Association for Computing Machinery, 1991, pp. 125–130.
- [358] Kontogiannis, K. A. Partial design recovery using dynamic programming. In *Proceedings of the 1994 CAS Conference* (Toronto, Ontario, Oct 1994), IBM Centre for Advanced Studies, 1994.
- [359] Koschke, R. *Atomic Architectural Component Recovery for Program Understanding and Evolution: Evaluation of Automatic Re-Modularization Techniques and Their Integration in a Semi-Automatic Method*. PhD thesis, Institut für Informatik, Universität Stuttgart, Oct. 1999.
- [360] Kosslyn, S. M. Visual mental images as re-presentations of the world: A cognitive neuroscience approach. In VSRD'99 [764].
- [361] Koubek, R. J., and Salvendy, G. Cognitive performance of super-experts on computer program modification tasks. *Ergonomics*, 34(8), 1991, pp. 1095–1112.
- [362] Kraemer, E., and Stasko, J. T. Issues in visualization for the comprehension of parallel programs. In WPC'94 [768], pp. 116–127.
- [363] Kremer, R. Concept mapping: Informal to formal. In CS'94 [737].
- [364] Kruchten, P. Architectural blueprints—the “4+1” view model of software architecture. *IEEE Software*, 12(6), Nov. 1995, pp. 42–50.
- [365] Kulpa, Z. Diagrammatic representation and reasoning. *Machine Graphics and Vision*, 3(1–2), 1994, pp. 77–103.
- [366] Kuutti, K. Activity theory as a potential framework for human–computer interaction research. In Nardi [436], ch. 2, pp. 17–44.
- [367] Kuutti, K., and Kaptelinin, V. Rethinking cognitive tools: From augmentation to mediation (extended abstract). In *Proceedings of the Second International Conference on Cognitive Technology: Humanizing the Information Age* (Aizu, Japan, Aug 25–28 1997), 1997, pp. 31–32.
- [368] Laitenberger, O., El Emam, K., and Harbich, T. G. An internally replicated quasi-experimental comparison of checklist and perspective-based reading of code documents. *IEEE Transactions on Software Engineering*, 27(5), May 2001, pp. 387–420.
- [369] Landauer, T. K. Relations between cognitive psychology and computer science. In Carroll [100], ch. 1, pp. 1–25.
- [370] Landauer, T. K. Let's get real: A position paper on the role of cognitive psychology in the design of humanly useful and usable systems. In Carroll [101], ch. 5, pp. 60–73.
- [371] Landauer, T. K. *The Trouble with Computers: Usefulness, Usability, and Productivity*. MIT Press, 1995.
- [372] Landauer, T. K., Foltz, P. W., and Laham, D. Introduction to latent semantic analysis. *Discourse Processes*, 25, 1998, pp. 259–284.
- [373] Lang, S., and von Mayrhauser, A. Building a research infrastructure for program comprehension observations. In WPC'97 [770], pp. 165–170.
- [374] Larkin, J. H. Display-based problem solving. In Klahr and Kotovsky [355], ch. 12, pp. 319–341.

- [375] Larkin, J. H., and Simon, H. A. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1), 1987, pp. 65–99.
- [376] Lave, J. *Cognition in Practice: Mind, Mathematics and Culture in Everyday Life*. Cambridge University Press, 1988.
- [377] Law, L.-C. A situated cognition view about the effects of planning and authorship on computer program debugging. *Behaviour and Information Technology*, 17(6), 1998, pp. 325–337.
- [378] Lehman, M. M. Process improvement—the way forward. In *Advanced Information Systems Engineering; 7th International Conference, CAiSE'95* (Jyväskylä, Finland, Jun 12–16 1995), J. Iivari, K. Lyytinen, and M. Rossi, Eds., Springer-Verlag, 1995, pp. 1–11.
- [379] Lehrer, R. Authors of knowledge: Patterns of hypermedia design. In Derry and Lajoie [178], ch. 7, pp. 197–227.
- [380] Lethbridge, T. C., and Singer, J. A. What's so great about 'grep'? implications for program comprehension tools". Online experience report. Retrieved from <http://wwwsel.iit.nrc.ca/seldocs/eassedocs/grepSinger.pdf>, 2000/08/01.
- [381] Lethbridge, T. C., and Singer, J. A. Understanding software maintenance tools: Some empirical research. In *Proceedings of the 1997 IEEE Workshop on Empirical Studies of Software Maintenance (WESS 97)* (Bari, Italy, Oct 3 1997), 1997, pp. 157–162.
- [382] Letovsky, S. Cognitive processes in program comprehension. In ESP'86 [741], pp. 58–79.
- [383] Levy, C. M., and Ransdell, S., Eds. *The Science of Writing: Theories, Methods, Individual Differences, and Applications*. Lawrence Erlbaum Associates, 1996.
- [384] Lewis, C. Inner and outer theory in human-computer interaction. In Carroll [101], ch. 9, pp. 154–162.
- [385] Licklider, J. C. R. Man–computer symbiosis. *IRE Transactions on Human Factors in Electronics*, HFE-1, Mar. 1960, pp. 4–1.
- [386] Lieberman, H., Ed. *Your Wish is my Command: Programming by Example*. Morgan Kaufmann, San Francisco, CA, 2001.
- [387] Linos, P. K., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P., and Tulula, P. Visualizing program dependencies: An experimental study. *Software–Practice and Experience*, 24(4), Apr. 1994, pp. 387–403.
- [388] Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E. M. Mental models and software maintenance. In ESP'86 [741], pp. 80–98.
- [389] Lloyd, P., Lawson, B., and Scott, P. Can concurrent verbalization reveal design cognition? In Cross *et al.* [153], pp. 437–463.
- [390] Long, J. Specifying relations between research and the design of human-computer interactions. *International Journal of Human-Computer Studies*, 44(6), 1996, pp. 875–920.
- [391] Long, J., and Dowell, J. Conceptions of the discipline of HCI: Craft, applied science, and engineering. In HCI'89 [747], pp. 9–32.
- [392] Long, J., and Dowell, J. Cognitive engineering human–computer interactions. *The Psychologist*, 9, July 1996, pp. 313–317.
- [393] Macaulay, C., Benyon, D., and Crerar, A. Ethnography, theory and systems design: From intuition to insight. *International Journal of Human-Computer Studies*, 53(1), 2000, pp. 35–60.
- [394] Mackay, W. E. Responding to cognitive overload: Co-adaptation between users and technology. *Intellectica*, 30(1), 2000, pp. 177–193.

- [395] Maes, P. Modeling adaptive autonomous agents. In *Artificial Life, An Overview*, G. Langton, Ed. MIT Press, Cambridge, 1995.
- [396] Marchionini, G. *Information Seeking in Electronic Environments*. Cambridge University Press, 1995.
- [397] Marshall, C. C., and Irish, P. M. Guided tours and on-line presentations: How authors make existing hypertext intelligible for readers. In HT'89 [752], pp. 15–26.
- [398] Mathewson, J. H. Visual-spatial thinking: An aspect of science overlooked by educators. *Science Education*, 83(1), 1999.
- [399] Mathieson, K., and Keil, M. Beyond the interface: Ease of use and task/technology fit. *Information and Management*, 34(4), Nov. 1998, pp. 221–230.
- [400] Mayer, R. E. Learners as information processors: Legacies and limitations of educational psychology's second metaphor. *Educational Psychologist*, 1996, pp. 151–161.
- [401] Mayhew, D. J. *Principles and Guidelines in Software User Interface Design*. Prentice Hall, Englewood Cliffs, NJ, 1992.
- [402] McCarthy, J. Applied experimental psychology. In Monk and Gilbert [415], ch. 4, pp. 75–97.
- [403] McKeithen, K. B., Reitman, J. S., Rueter, H. H., and Hirtle, S. C. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13, 1981, pp. 307–325.
- [404] McKim, R. H. *Thinking Visually: a Strategy Manual for Problem Solving*. Lifetime Learning Publications, Belmont, Calif., 1980.
- [405] McKim Jr., J. C. Programming by contract: Designing for correctness. *Journal of Object-Oriented Programming*, 9(2), May 1996, pp. 70–74.
- [406] McPhee, K. Design theory and software design. Tech. Rep. TR 96–26, Department of Computing Science, University of Alberta, Oct. 1996.
- [407] Meyers, S. D., and Reiss, S. P. An empirical study of multiple-view software development. In SIGSOFT'92 [763], pp. 47–57. Published Proceedings of the ACM SIGPLAN/SIGSOFT Conference, 1992.
- [408] Millen, D. R. Rapid ethnography: Time deepening strategies for HCI field research. In DIS'2000 [739], pp. 280–286.
- [409] Miller, J. Applying meta-analytical procedures to software engineering experiments. Tech. Rep. EFOCS–30–98, Empirical Foundations of Computer Science, University of Strathclyde, 1998.
- [410] Miller, J., Daly, J., Wood, M., Roper, M., and Brooks, A. Statistical power and its subcomponents—missing and misunderstood concepts in empirical software engineering research. *Information and Software Technology*, 39, 1997, pp. 285–295.
- [411] Miller, J., and Macdonald, F. Statistical analysis of two experimental studies. Tech. Rep. EFOCS–31–98, Empirical Foundations of Computer Science, University of Strathclyde, 1998.
- [412] Miller, J., and Macdonald, F. An empirical incremental approach to tool evaluation and improvement. *The Journal of Systems and Software*, 51(1), 2000, pp. 19–35.
- [413] Moher, T., and Schneider, G. M. Methodology and experimental research in software engineering. *International Journal of Man-Machine Studies*, 16(1), 1982, pp. 65–87.
- [414] Monk, A. F. Modeling cyclic interaction. *Behaviour and Information Technology*, 18(2), 1999, pp. 127–139.
- [415] Monk, A. F., and Gilbert, G. N., Eds. *Perspectives on HCI: Diverse Approaches*. Academic Press Limited, 1995.

- [416] Monk, A. F., Walsh, P., and Dix, A. J. A comparison of hypertext, scrolling and folding as mechanisms for program browsing. In *HCI'88* [746], pp. 421–435.
- [417] Moore, J. L., and Rocklin, T. R. The distribution of distributed cognition: Multiple interpretations and uses. *Educational Psychology Review*, 10(1), 1998, pp. 97–113.
- [418] Moran, T. P. The command language grammar: A representation for the user interface of interactive computer systems. *International Journal of Man-Machine Studies*, 15(1), 1981, pp. 3–50.
- [419] Moran, T. P. Guest editor's introduction: An applied psychology of the user. *ACM Computing Surveys*, 13(1), Mar. 1981, pp. 1–11.
- [420] Moran, T. P., and Carroll, J. M., Eds. *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associates, Mahwah, NJ, 1996.
- [421] Moriconi, M., and Hare, D. The PegaSys system: Pictures as formal documentation of large programs. *ACM Transactions on Programming Languages and Systems*, 8(4), Oct. 1986, pp. 524–546.
- [422] Mulholland, P. *A Framework for Describing and Evaluating Software Visualization Systems: A Case-Study in Prolog*. PhD thesis, Knowledge Media Institute, The Open University, 1994.
- [423] Müller, H. A. Rigi – A model for software system construction, integration, and evolution based on module interface specifications. Tech. Rep. TR86–36, Department of Computer Science, Rice University, Houston, Texas, Aug. 1986. PhD. Thesis.
- [424] Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R., and Wong, K. Reverse engineering: A roadmap. In *The Future of Software Engineering*. ACM, 2000, pp. 47–60.
- [425] Müller, H. A., and Klashinsky, K. Rigi—a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering* (Singapore, Apr 11–15 1988), IEEE Computer Society Press, 1988, pp. 80–85.
- [426] Müller, H. A., Orgun, M. A., Tilley, S. R., and Uhl, J. S. A reverse engineering approach to subsystem structure identification. *Software Maintenance—Research and Practice*, 5(4), Oct. 1993, pp. 181–204.
- [427] Müller, H. A., Tilley, S. R., Orgun, M. A., and Corrie, B. D. A reverse engineering environment based on spatial and visual software interconnection models. In *SIGSOFT'92* [763], pp. 88–98. Published Proceedings of the ACM SIGPLAN/SIGSOFT Conference, 1992.
- [428] Murphy, G. C. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1996.
- [429] Murphy, G. C., and Notkin, D. Reengineering with reflexion models: A case study. *Computer*, 30(8), Aug. 1997, pp. 29–36.
- [430] Murphy, G. C., Notkin, D., and Sullivan, K. J. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the Third SIGSOFT Symposium on the Foundations of Software Engineering* (Washington, DC, Oct 12–15 1995), G. E. Kaiser, Ed., Association for Computing Machinery, 1995, pp. 18–28.
- [431] Murphy, G. C., Notkin, D., and Sullivan, K. J. Extending and managing software reflexion models. Tech. Rep. TR–97–15, University of British Columbia, Department of Computer Science, Sept. 1997.
- [432] Myers, G. J. *The Art of Software Testing*. John Wiley & Sons, New York, NY, 1979.
- [433] Narayanan, N. H., and Hübscher, R. Visual language theory: Towards a human-computer interaction perspective. In *Visual Language Theory*, K. Marriott and B. Meyer, Eds. Springer-Verlag, 1998, ch. 3, pp. 87–128.

- [434] Nardi, B. Studying context: A comparison of activity theory, situated action models, and distributed cognition. In Nardi [436], ch. 4, pp. 7–16.
- [435] Nardi, B. A. Activity theory and human-computer interaction. In *Context and Consciousness: Activity Theory and Human-Computer Interaction* [436], ch. 1, pp. 7–16.
- [436] Nardi, B. A., Ed. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, Cambridge, MA, 1996.
- [437] Nardi, B. A., and Zamer, C. L. Beyond models and metaphors: Visual formalisms in user interface design. *Journal of Visual Languages and Computing*, 4, 1993, pp. 5–33.
- [438] Naur, P. Program development studies based on diaries. In *The Psychology of Computer Use*, A. stephen j), and G. C. van der Veer, Eds. Associated Press, 1983, pp. 159–170.
- [439] Naur, P. *Computing: A Human Activity*. Association for Computing Machinery, 1990.
- [440] Naur, P. The place of strictly defined notation in human insight. In *Computing: A Human Activity* [439], ch. 7.5. Originally published in the *Proceedings of the Workshop on Programming Logic*, ed. P. Dybjer, L. Hallnäs, B. Nordstrom, K. Petersson, and J. M. Smith. Report 54, Programming Methodology Group, Univ. of Göteborg and Chalmers University of Technology, Göteborg, Sweden, May 1989.
- [441] Neuwirth, C. M., and Kaufer, D. S. The role of external representations in the writing process: Implications for the design of hypertext-based writing tools. In HT'89 [752], pp. 319–341.
- [442] Newell, A. Some problems of the basic organization in problem-solving programs. In *Proceedings of the Second Conference on Self-Organizing Systems*, M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, Eds., Spartan Books, 1962, pp. 393–423.
- [443] Newell, A. Heuristic programming: Ill-structured problems. In *Progress in Operations Research: Relationship Between Operations Research and the Computer*, J. Aronofsky, Ed., vol. 3. John Wiley & Sons, New York, 1969, pp. 360–413.
- [444] Newell, A. You can't play '20 questions' with nature and win: Projective comments on the papers in this symposium. In *Visual Information Processing*, Academic Press, 1972, pp. 283–308.
- [445] Newell, A. The knowledge level. *Artificial Intelligence*, 18(1), 1982, pp. 87–127.
- [446] Newell, A. *Unified Theories of Cognition*. Harvard University Press, 1990.
- [447] Newell, A. Précis of unified theories of cognition. *Behavioral and Brain Sciences*, 15, 1992, pp. 425–492.
- [448] Newell, A., and Card, S. K. The prospects for psychological science in human-computer interaction. *Human Computer Interaction*, 1(3), 1985, pp. 209–242.
- [449] Newell, A., and Simon, H. A. *Human Problem Solving*. Prentice-Hall, Inc., 1972.
- [450] Newell, A., and simon-herbert a. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3), Mar. 1976, pp. 113–126.
- [451] Newell, A. F., and Gregor, P. Extra-ordinary human-machine interaction: What can be learned from people with disabilities? *Cognition, Technology and Work*, 1(2), 1999, pp. 78–85.
- [452] Newman, W. A preliminary analysis of the products of HCI, using *pro forma* abstracts. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems* (Boston, MA, Apr 24–28 1994), B. Adelson, S. Dumais, and J. Olson, Eds., vol. 1, Association for Computing Machinery, 1994, pp. 278–284.

- [453] Nickell, J. Peddling snake oil. *Skeptical Briefs Newsletter*, Dec. 1998. Retrieved from <http://www.csicop.org/sb/9812/snakeoil.html>, 2001/01/10.
- [454] Nickerson, R. S. On the distribution of cognitions: Some reflections. In Salomon [558], ch. 8, pp. 229–262.
- [455] Nickerson, R. S. Basic versus applied research. In Sternberg [617], ch. 12, pp. 409–442.
- [456] Nielsen, J. A virtual protocol model for computer-human interaction. *International Journal of Man-Machine Studies*, 24(3), 1986, pp. 301–312.
- [457] Nielsen, J. The matters that really matter for hypertext usability. In HT'89 [752], pp. 239–248.
- [458] Nielsen, J. Evaluating the thinking aloud technique for use by computer scientists. In *Advances in Human-Computer Interaction*, H. R. Hartson and D. Hix, Eds., vol. 3. Ablex, Norwood, NJ, 1992, pp. 69–82.
- [459] Nielsen, J. A layered interaction analysis of direct manipulation. Unpublished online paper. Retrieved from http://www.useit.com/papers/direct_manipulation.html, 2000/09/28, 1992.
- [460] Nielsen, J. *Usability Engineering*. Academic Press, Boston, MA, 1993.
- [461] Nielsen, J., and Mack, R. L., Eds. *Usability Inspection Methods*. John Wiley & Sons, New York, NY, 1994.
- [462] Nii, H. P. Blackboard systems. Tech. Rep. CS-TR-86-1123, Stanford University, Department of Computer Science, June 1986.
- [463] Nii, H. P. Blackboard systems. *AI Magazine*, 7(2), 1986, pp. 38–53. Two parts, second in v7 n3, pp. 82–106. Revised version published in *Handbook of Artificial Intelligence*, vol 4, A. Barr, P. Cohen, E. Feigenbaum, ed., Addison-Wesley, 1989, p. 1–82.
- [464] Nix, R. P. Editing by example. *ACM Transactions on Programming Languages and Systems*, 7(4), Oct. 1985, pp. 600–621.
- [465] Norman, D. A. Twelve issues for cognitive science. *Cognitive Science*, 4, 1980, pp. 1–32. Reprinted in Chapter 11, *Perspectives on Cognitive Science*. Norman, D. A., ed., Norwood, NJ: Ablex., 1981.
- [466] Norman, D. A. Four stages of user activities. In INTERACT'84 [755], pp. 507–511.
- [467] Norman, D. A. Cognitive engineering. In Norman and Draper [475], ch. 3, pp. 31–65.
- [468] Norman, D. A. Cognitive engineering—cognitive science. In Carroll [100], ch. 12, pp. 325–336.
- [469] Norman, D. A. *The Psychology of Everyday Things*. Basic Books, 1988.
- [470] Norman, D. A. Cognitive artifacts. In Carroll [101], ch. 2, pp. 17–38.
- [471] Norman, D. A. Cognition in the head and in the world: An introduction to the special issue on situated action. *Cognitive Science*, 17, 1993, pp. 1–6.
- [472] Norman, D. A. *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine*. Addison-Wesley, Reading, Massachusetts, 1993.
- [473] Norman, D. A. *The Invisible Computer: Why Good Products Can Fail, The Personal Computer is So Complex, and Information Appliances are the Solution*. The MIT Press, 1998.
- [474] Norman, D. A. Affordance, conventions, and design. *Interactions*, 6(3), 1999, pp. 38–42.
- [475] Norman, D. A., and Draper, S. W., Eds. *User Centered System Design: New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.
- [476] Oberg, B., and Notkin, D. Error reporting with graduated color. *IEEE Software*, Nov. 1992, pp. 33–38.

- [477] O'Hara, K. Towards a typology of reading goals. Tech. Rep. EPC-1996-107, Rank Xerox Research Centre, Cambridge UK, 1996.
- [478] O'Hara, K. P., and Payne, S. J. Planning and the user interface: The effects of lockout time and error recovery cost. *International Journal of Human-Computer Studies*, 50(1), 1999, pp. 41-59.
- [479] Olson, G. M., and Moran, T. P. Commentary on "damaged merchandise?". *Human Computer Interaction*, 13(3), 1998, pp. 263-323.
- [480] Orhun, E. Design of computer-based cognitive tools. In NATO.ASI.146 [761], pp. 305-320.
- [481] Ormerod, T. Human cognition and programming. In Hoc *et al.* [307], ch. 1.4, pp. 63-82.
- [482] Paige, J. M., and Simon, H. A. Cognitive processes in solving algebra word problems. In *Problem Solving: Research, Method, and Theory* (Carnegie Institute of Technology, Apr 15-16 1966), B. Kleinmuntz, Ed., 1966, pp. 51-118.
- [483] Palincsar, A. S. Social constructivist perspectives on teaching and learning. *Annual Review of Psychology*, 49(1), 1998, pp. 345-375.
- [484] Parasuraman, R. Designing automation for human use: empirical studies and quantitative models. *Ergonomics*, 43(7), 2000, pp. 931-953.
- [485] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), Dec. 1972, pp. 1053-1058.
- [486] Parnas, D. L. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1), 1976, pp. 1-9.
- [487] Parnas, D. L. Software aspects of strategic defense systems. *Communications of the ACM*, 28(12), Dec. 1985, pp. 1326-1335.
- [488] Parnas, D. L., and Clements, P. C. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2), Feb. 1986, pp. 251-257.
- [489] Patalano, A. L., and Seifert, C. M. Opportunistic planning: Being reminded of pending goals. *Cognitive Psychology*, 34(1), Oct. 1997, pp. 1-36.
- [490] Paul, S., and Prakash, A. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6), June 1994, pp. 463-475.
- [491] Payne, S. J. Looking HCI in the I. In INTERACT'90 [756], pp. 185-191.
- [492] Payne, S. J. Interface problems and interface resources. In Carroll [101], ch. 8, pp. 128-153.
- [493] Payne, S. J. On mental models and cognitive artefacts. In Rogers *et al.* [548], ch. 8, pp. 103-118.
- [494] Payne, S. J. Cognitive psychology and cognitive technologies. *The Psychologist*, 9, July 1996, pp. 309-312.
- [495] Pea, R. D. Practices of distributed intelligence and designs for education. In Salomon [558], ch. 2, pp. 47-87.
- [496] Pemberton, L., Shurville, S., and Sharples, M. External representations in the writing process and how to support them. In *Artificial Intelligence in Education (EuroAIED-96)* (Lisbon, Portugal, Sep 30-Oct 2 1996), 1996.
- [497] Pennington, N. Comprehension strategies in programming. In ESP'87 [742], pp. 100-113.
- [498] Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19, 1987, pp. 295-341.
- [499] Pennington, N., and Grabowski, B. The tasks of programming. In Hoc *et al.* [307], ch. 1.3, pp. 45-62.

- [500] Perkins, D. N. The fingertip effect: How information-processing technology shapes thinking. *Educational Researcher*, 14, Aug. 1985, pp. 11–17.
- [501] Perkins, D. N. Person-plus: A distributed view of thinking and learning. In Salomon [558], ch. 3, pp. 88–110.
- [502] Perry, M. Cognitive artefacts and collaborative design. In *IEE Colloquium on Design Systems with Users in Mind: The Role of Cognitive Artefacts*, 1995, pp. 2/1–2/2.
- [503] Perry, M., and Thomas, P. Externalising the internal: Collaborative design through dynamic problem visualisation. In HCI'95 [749], pp. 149–154. Adjunct Proceedings.
- [504] Perry, M. J. *Distributed Cognition and Computer Supported Collaborative Design: The Organisation of Work in Construction Engineering*. PhD thesis, Department of Information Systems and Computing, Brunel University, U.K., 1997.
- [505] Peterson, D., Ed. *Forms of Representation: An Interdisciplinary Theme for Cognitive Science*. Intellect, Exeter, UK, 1996.
- [506] Petre, M. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6), June 1995, pp. 33–44.
- [507] Petre, M., and Blackwell, A. F. Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51(1), 1999, pp. 7–30.
- [508] Petre, M., Blackwell, A. F., and Green, T. R. G. Cognitive questions in software visualization. In *Software Visualization: Programming as a Multimedia Experience*, J. T. Stasko, J. Domingue, B. A. Price, and M. H. Brown, Eds. MIT Press, 1998, pp. 453–480.
- [509] Petre, M., and Green, T. R. G. Learning to read graphics: Some evidence that 'seeing' an information display is an acquired skill. *Journal of Visual Languages and Computing*, 4(1), 1993, pp. 55–70.
- [510] Petroski, H. *The Evolution of Useful Things*. A. Knopf, New York, NY, 1992.
- [511] Pirolli, P. I. Towards a unified model of learning to program. In NATO.ASI.111 [759], pp. 34–48.
- [512] Pohthong, A., and Budgen, D. Reuse strategies in software development: an empirical study. *Information and Software Technology*, 43(9), Aug. 2001, pp. 561–575.
- [513] Polya, G. *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed. Doubleday, Garden City, NJ, 1957.
- [514] Poon, J., and Maher, M. L. Emergent behaviour in co-evolutionary design. In *International Conference on Artificial Intelligence in Design (4th, 1996)* (Stanford, CA), J. S. Gero and F. Sudweeks, Eds., Kluwer Academic Publishers, 1996, pp. 703–722.
- [515] Preston, B. Cognition and tool use. *Mind & Language*, 13(4), 1998, pp. 513–547.
- [516] Price, B. A., Baecker, R. M., and Small, I. S. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3), 1993, pp. 211–266.
- [517] Purcell, A. T., and Gero, J. S. Drawings and the design process. *Design Studies*, 19(4), 1998, pp. 389–430.
- [518] Purcell, T., Gero, J., Edwards, H., and McNeill, T. The data in design protocols: The issue of data coding, data analysis in the development of models of the design process. In Cross *et al.* [153], pp. 151–168.
- [519] Purchase, H. C. Performance of layout algorithms: Comprehension, not computation. *Journal of Visual Languages and Computing*, 9(6), 1998, pp. 647–657.
- [520] Pylyshyn, Z. The role of cognitive architecture in theories of cognition. In *Architectures for Intelligence*, K. VanLehn, Ed. Lawrence Erlbaum Associates, Hillsdale, NJ, 1988, pp. 189–223.

- [521] Pylyshyn, Z., Ed. *Constraining Cognitive Theories: Issues and Options*. Ablex Publishing Corporation, 1998.
- [522] Pylyshyn, Z. W. Some remarks on the theory-practice gap. In Carroll [101], ch. 3, pp. 39–49.
- [523] Pylyshyn, Z. W. Introduction: Cognitive architecture and the hope for a science of cognition. In Pylyshyn [521], ch. 1, pp. 1–8.
- [524] Quilici, A. Reverse engineering of legacy systems: A path towards success. In *17th International Conference on Software Engineering* (Seattle, Washington, Apr 24–28 1995), IEEE Computer Society Press, 1995, pp. 333–336.
- [525] r m), A., and Buxton, W. A. S. Design and evaluation (introduction). In Baecker and Buxton [21], ch. 2, pp. 73–91.
- [526] Rasmussen, J. Skills, rules, knowledge: Signals, signs, and symbols and other distinctions in human performance models. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3), 1983, pp. 257–267.
- [527] Rasmussen, J. A framework for cognitive task analysis in system design. Tech. Rep. RISØ-M-2519, RisøNational Laboratory, DK-4000 Roskilde, Denmark, Aug. 1985.
- [528] Rasmussen, J. *Information Processing and Human–Machine Interaction: An Approach to Cognitive Engineering*. North Holland, New York, 1986.
- [529] Rasmussen, J. Mental models and the control of action in complex environments. In Ackermann and Tauber [3], ch. 2, pp. 41–72.
- [530] Rasmussen, J., and Pejtersen, A. M. Virtual ecology of work. In Flach *et al.* [219], ch. 5, pp. 121–156.
- [531] Rasmussen, J., Pejtersen, A. M., and Goodstein, L. P. *Cognitive Systems Engineering*. John Wiley & Sons, Inc., New York, NY, 1994.
- [532] Redmiles, D. F. Observations on using empirical studies in developing a knowledge-based software engineering tool. In *Proceedings of the Eighth Knowledge Based Software Engineering Conference* (Chicago, Illinois, Sep 20–23 1993), IEEE Computer Society Press, Sept. 1993, pp. 170–177.
- [533] Redmiles, D. F. Reducing the variability of programmers’ performance through explained examples. In INTERCHI’93 [757], pp. 67–73.
- [534] Retkowsky, F. Software reuse from an external memory: the cognitive issues of support tools. In *PPIG Workshop’98 Proceedings* (Open University, Milton Keynes, UK, Jan. 1998), Psychology of Programming Interest Group, 1998. Retrieved from <http://ppig.org/papers/10th-retkowsky.pdf>, 2000/05/18.
- [535] Retkowsky, F. P. *A Cognitive Approach to Supporting Reuse*. PhD thesis, University of Sussex, June 2000.
- [536] Rettig, M. Hat racks for understanding. *Communications of the ACM*, 35(10), Oct. 1992, pp. 21–25.
- [537] Rheingold, H. *Tools for Thought: The History and Future of Mind-Expanding Technology*. MIT Press, 2000.
- [538] Rich, C. A formal representation for plans in the Programmer’s Apprentice. In *Proceedings of the 7th International Joint Conference on AI* (Vancouver, BC, Aug 1981), A. Drinan, Ed., 1981, pp. 1044–1052.
- [539] Rich, C., and Waters, R. C. *The Programmer’s Apprentice*. Association for Computing Machinery, 1990.
- [540] Richardson, T., Stafford-Fraser, Q., Wood, K. R., and Hopper, A. Virtual network computing. *IEEE Internet Computing*, 2(1), Jan. 1998, pp. 33–38.
- [541] Rittel, H. W. J., and Webber, M. M. Dilemmas in a general theory of planning. *Policy Sciences*, 4, 1973, pp. 155–169.
- [542] Ritter, F. E., and Larkin, J. H. Developing process models as summaries of HCI action sequences. *Human Computer Interaction*, 9(3), 1994, pp. 345–383.

- [543] Robbins, J. E., Hilbert, D. M., and Redmiles, D. F. Extending design environments to software architecture design. In *KBSE'96* [758].
- [544] Robbins, J. E., and Redmiles, D. F. Software architecture design from the perspective of human cognitive needs. In *Proceedings of the California Software Symposium (CSS'96)* (University of Southern California, Los Angeles, CA, Apr 17 1996), 1996, pp. 16–27.
- [545] Robson, D. J., Bennett, K. H., Cornelius, B. J., and Munro, M. Approaches to program comprehension. *The Journal of Systems and Software*, 14, Feb. 1991, pp. 79–84.
- [546] Rogers, Y., Bannon, L. J., and Button, G. Report on the INTERCHI-93 workshop “Rethinking Theoretical Frameworks for HCI”, Amsterdam, 24-25th April, 1993. *ACM SIGCHI Bulletin*, 26(1), Jan. 1994, pp. 28–30.
- [547] Rogers, Y., and Ellis, J. Distributed cognition: an alternative framework for analysing and explaining collaborative working. *Journal of Information Technology*, 9(2), 1994, pp. 119–128.
- [548] Rogers, Y., Rutherford, A., and Bibby, P. A., Eds. *Models in the Mind: Theory, Perspective and Application*. Academic Press Limited, 1992.
- [549] Rosenblum, D. S. Towards a method of programming with assertions. In *14th IEEE International Conference on Software Engineering* (Melbourne, Australia, May 11–15 1992), IEEE Computer Society Press, May 1992, pp. 92–104.
- [550] Rosson, M. B. The role of experience in editing. In *INTERACT'84* [755], pp. 45–50.
- [551] Rosson, M. B., Maass, S., and Kellogg, W. The designer as user: Building requirements for design tools from design practice. *Communications of the ACM*, 31(11), Nov. 1988, pp. 1288–1298.
- [552] Rouet, J.-F., Levonen, J. J., Dillon, A., and Spiro, R. J., Eds. *Hypertext and Cognition*. Lawrence Erlbaum Associates, Mahwah, NJ, 1996.
- [553] Rouet, J.-F., and Tricot, A. Task and activity models in hypertext usage. In van Oostendorp and de Mul [652], ch. 11, pp. 239–264.
- [554] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, 1999.
- [555] Rumelhart, D. E., and Norman, D. A. Representation in memory. In *Stevens' Handbook of Experimental Psychology*, R. C. Atkinson, R. J. Herrnstein, G. Lindzey, and R. D. Luce, Eds., 2nd ed., vol. 2: Learning and Cognition. John Wiley & Sons, New York, 1988, pp. 511–587.
- [556] Sagan, C. *The Demon-Haunted World: Science as a Candle in the Dark*. Random House, New York, NY, 1995.
- [557] Sajaniemi, J., and Niemelainen, A. Program editing based on variable plans: A cognitive approach to program manipulation. In *Proceedings of the Third International Conference on Human-Computer Interaction* (Boston, MA, Aug 18–22 1989), G. Salvendy and M. J. Smith, Eds., Elsevier Science Ltd., 1989, pp. 66–73.
- [558] Salomon, G., Ed. *Distributed Cognitions: Psychological and Educational Considerations*. Cambridge University Press, 1993.
- [559] Salomon, G. On the nature of pedagogic computer tools: The case of the writing partner. In Derry and Lajoie [178], ch. 6, pp. 179–196.
- [560] Sass, M. A., and Wilkinson, W. D., Eds. *Symposium on Computer Augmentation of Human Reasoning* (Washington, D.C., Jun 1965), Spartan Books, Inc.
- [561] Savage-Knepshield, P. A., and Belkin, N. J. Interaction in information retrieval: Trends over time. *Journal of the American Society for Information Science*, 50(12), Oct. 1999, pp. 1067–1082.

- [562] Scaife, M., and Rogers, Y. External cognition: How do graphical representations work? *International Journal of Human-Computer Studies*, 45(2), 1996, pp. 185–213.
- [563] Schauble, L., Raghavan, K., and Glaser, R. The discovery and reflection notation: A graphical trace for supporting self-regulation in computer-based laboratories. In Derry and Lajoie [178], ch. 11, pp. 319–337.
- [564] Schön, D. *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, New York, 1983.
- [565] Schön, D. *Educating the Reflective Practitioner: Toward a New Design for Teaching and Learning in the Professions*. Jossey-Bass, San Francisco, 1987.
- [566] Schön, D., and Bennett, J. Reflective conversation with materials: An interview with Donald Schön by John Bennett. In *Bringing Design to Software*, T. Winograd, J. Bennett, L. De Young, and B. Hartfield, Eds. Association for Computing Machinery, 1996, ch. 9, pp. 171–184.
- [567] Schönplflug, W. The trade-off between internal and external information storage. *Journal of Memory and Language*, 25(6), 1986, pp. 657–675.
- [568] Schönplflug, W. Retrieving texts from an external store: the effects of an explanatory context and of semantic fit between text and address. *Psychological Research*, 50, 1988, pp. 19–27.
- [569] Schönplflug, W., and Esser, K. B. Memory and its *graeculi*: Metamemory and control in extended memory systems. In *Discourse Comprehension: Essays in Honor of Walter Kintsch*, C. A. Weaver III, S. Mannes, and C. R. Fletcher, Eds. Lawrence Erlbaum, 1995, ch. 14, pp. 245–255.
- [570] Schraagen, J. M., Chipman, S. F., and Shalin, V. L., Eds. *Cognitive Task Analysis*. Lawrence Erlbaum, Mahwah, NJ, 2000.
- [571] Sen, A., Vinze, A., and Liou, S. Role of control in the model formulation process. *Information Systems Research*, 5(3), 1994, pp. 219–248.
- [572] Sengler, H. E. A model of the understanding of a program and its impact on the design of the programming language Grade. In *The Psychology of Computer Use*, T. R. G. Green, S. J. Payne, and G. C. van der Veer, Eds. Academic Press Limited, 1983, pp. 91–106.
- [573] Sharples, M. The development of a cognitive model for computer support of collaborative writing. *Journal of Computer Assisted Learning*, 7(3), 1991, pp. 203–204.
- [574] Sharples, M. Designs for new writing environments. In Sharples and van der Geest [577], ch. 7, pp. 97–115.
- [575] Sharples, M. Writing as creative design. In Levy and Ransdell [383], pp. 127–148.
- [576] Sharples, M., and Pemberton, L. Representing writing: External representations and the writing process. In Holt and Williams [313], ch. 21, pp. 319–336.
- [577] Sharples, M., and van der Geest, T., Eds. *The New Writing Environment: Writers at Work in a World of Technology*. Springer-Verlag, London, 1996.
- [578] Shaw, M. Larger scale systems require higher-level abstractions. In *Proceedings of the Fifth International Workshop on Software Specification and Design* (Pittsburgh, Pennsylvania, May 19–20 1989), Association for Computing Machinery, 1989, pp. 143–146. Published in *Software Engineering Notes*, 14(3).
- [579] Shaw, M. Prospects for an engineering discipline of software. *IEEE Software*, 7(6), Nov. 1990, pp. 15–24.
- [580] Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle Road, NJ, 1996.
- [581] Sheil, B. A. The psychological study of programming. *ACM Computing Surveys*, 13(1), Mar. 1981, pp. 101–120.

- [582] Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop, Cambridge, MA, 1980.
- [583] Shneiderman, B., and Carroll, J. M. Ecological studies of professional programmers: An overview. *Communications of the ACM*, 31(11), Nov. 1988, pp. 1256–1258.
- [584] Shneiderman, B., and Mayer, R. E. Syntactic/semantic interactions of programming behaviour: A model. *International Journal of Computer and Information Sciences*, 8(3), 1979, pp. 219–238.
- [585] Shneiderman, B., Shafer, P., Simon, R., and Weldon, L. Display strategies for program browsing. In *Proceedings of the IEEE Conference on Software Maintenance – 1985* (Washington, DC, Nov 11–13 1985), IEEE Computer Society Press, 1985, pp. 136–143.
- [586] Shukla, S. V., and Redmiles, D. F. Collaborative learning in a software bug-tracking scenario. In *Workshop on Approaches for Distributed Learning through Computer Supported Collaborative Learning* (Boston, MA, Nov 16–20 1996), Nov. 1996.
- [587] Shum, S. B., and Hammond, N. Transferring HCI modelling and design techniques to practitioners: A framework and empirical work. In *People and Computers IX, Proceedings of HCI'94* (Glasgow, Scotland, Aug 23–26 1994), G. Cockton, S. W. Draper, and G. R. S. Weir, Eds., Cambridge University Press, 1994, pp. 21–36.
- [588] Shurkin, J. *Engines of the Mind: A History of the Computer*. W. W. Norton & Company, New York, 1984.
- [589] Sidarkeviciute, D., Tyugu, E., and Kuusik, A. A knowledge-based toolkit for software visualisation. In KBSE'96 [758], pp. 125–133.
- [590] Sim, S. E., Clarke, C. L. A., and Holt, R. C. Archetypal source code searching: A survey of software developers and maintainers. In WPC'98 [771].
- [591] Sim, S. E., and Holt, R. C. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proceedings of the 20th International Conference on Software Engineering* (Kyoto, Japan, Apr 19–25 1998), IEEE Computer Society Press, Apr. 1998, pp. 361–370.
- [592] Simon, H. A. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106(6), Dec. 1962, pp. 467–482.
- [593] Simon, H. A. On the forms of mental representation. In *Perception and cognition: Issues in the foundations of psychology*, vol. 9 of *Minnesota Studies in the Philosophy of Science*, University of Minnesota Press, 1978, pp. 3–18.
- [594] Simon, H. A. *The Sciences of the Artificial*, 3rd ed. MIT Press, 1996.
- [595] Singer, J. A., and Lethbridge, T. C. Just-in-time-comprehension vs. the full-coverage strategy. In *Proceedings of the 1998 Workshop on Empirical Studies of Software (Online)* (Bethesda, MD, Nov 16 1998), 1998. Position Paper. Retrieved from <http://www.cs.umd.edu/~sharip/wess/papers/singer.html>.
- [596] Singer, J. A., and Lethbridge, T. C. Studying work practices to assist tool design in software engineering. In WPC'98 [771], pp. 173–179.
- [597] Singer, J. A., Lethbridge, T. C., Vinson, N., and Anquetil, N. An examination of software engineering work practices. In *Proceedings of the Seventh Centre for Advanced Studies Conference* (Toronto, Ontario, Nov 10–13 1997), IBM Centre for Advanced Studies, Nov. 1997, pp. 209–223.
- [598] Singley, M. K., and Carroll, J. M. Synthesis by analysis: Five modes of reasoning that guide design. In Moran and Carroll [420], ch. 8, pp. 241–265.
- [599] Singley, M. K., Carroll, J. M., and Alpert, S. R. Psychological design rationale for an intelligent tutoring system for Smalltalk. In ESP'91 [743], pp. 196–209.

- [600] Singley, M. K., Carroll, J. M., and Alpert, S. R. Incidental reification of goals in an intelligent tutor for Smalltalk. In NATO.ASI.111 [759], pp. 145–155.
- [601] Sinha, A., and Vessey, I. Cognitive fit in recursion and iteration: An empirical study. *IEEE Transactions on Software Engineering*, 18(5), May 1992, pp. 386–379.
- [602] Skillicorn, D. B., and Talia, D. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2), 1998, pp. 123–169.
- [603] Smith, J. B. *Collective Intelligence in Computer-Based Collaboration*. Lawrence Erlbaum Associates, Mahwah, NJ, 1994.
- [604] Smith, J. B., and Lansman, M. A cognitive basis for a computer writing environment. In *Computer Writing Environments: Theory, Research and Design*, B. K. Britton and S. M. Glynn, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989, ch. 2, pp. 17–56.
- [605] Smith, J. B., and Lansman, M. Designing theory-based systems: A case study. In CHI'92 [732], pp. 479–488.
- [606] Smolensky, P. On the proper treatment of connectionism. *Behavioral and Brain Sciences*, 11(1), 1988, pp. 1–23.
- [607] Snelting, G. Concept analysis — A new framework for program understanding. In *Proceedings of the 1998 ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)* (Montreal, Canada, Jun 1998), 1998, pp. 1–10.
- [608] Soloway, E. M. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9), Sept. 1986, pp. 850–858.
- [609] Soloway, E. M., Adelson, B., and Ehrlich, K. Knowledge and processes in the comprehension of computer programs. In Chi *et al.* [123], pp. 129–152.
- [610] Soloway, E. M., Bonar, J., and Ehrlich, K. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11), 1983, pp. 853–860.
- [611] Soloway, E. M., and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), Sept. 1984, pp. 595–609.
- [612] Soloway, E. M., Jackson, S. L., Klein, J., Quintana, C., Reed, J., Spitulnik, J., Stratford, S. J., Studer, S., Eng, J., and Scala, N. Learning theory in practice: Case studies of learner-centered design. In CHI'96 [734], pp. 189–196.
- [613] Sommerville, I. *Software Engineering*. Addison-Wesley, Reading, Massachusetts, 1989.
- [614] Spence, R., and Tweedie, L. The Attribute Explorer: Information synthesis via exploration. *Interacting With Computers*, 11(2), 1998, pp. 137–146.
- [615] Stacey, M., Clarkson, J., and Eckert, C. Signposting: an AI approach to supporting human decision making in design. In *20th Computers and Information in Engineering Conference, Proceedings of the ASME 2000 Design Engineering Technical Conferences*. (Baltimore, MD, Sep 10–14 2000), American Society of Mechanical Engineers, 2000. DETC2000/CIE-14617.
- [616] Star, S. L. Working together: Symbolic interactionism, activity theory, and information systems. In Engeström and Middleton [202], ch. 13, pp. 296–318.
- [617] Sternberg, R. J., Ed. *The Nature of Cognition*. MIT Press, 1999.
- [618] Storey, M.-A. D. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, School of Computing Science, Simon Fraser University, 1998.

- [619] Storey, M.-A. D., Fracchia, F. D., and Müller, H. A. Cognitive design elements to support the construction of a mental model during software visualization. *The Journal of Systems and Software*, 44(3), Jan. 1999, pp. 171–185.
- [620] Storey, M.-A. D., Wong, K., Fong, P., Hooper, D., Hopkins, K., and Müller, H. A. On designing an experiment to evaluate a reverse engineering tool. In *Third Working Conference on Reverse Engineering (WCRE-3)* (Monterey, CA, Nov 8–10 1996), IEEE Computer Society Press, Nov. 1996.
- [621] Storey, M.-A. D., Wong, K., Fracchia, F. D., and Müller, H. A. On integrating visualization techniques for effective software exploration. In *Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)* (Phoenix, AZ, Oct 20–21 1997), 1997, pp. 38–45.
- [622] Storey, M.-A. D., Wong, K., and Müller, H. A. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2), Mar. 2000, pp. 183–207.
- [623] Suchman, L. A. *Plans and Situated Actions: The Problem of Human-Computer Communication*. Cambridge University Press, New York, 1987.
- [624] Sugiyama, K., and Misue, K. "Good" graphic interfaces for "good" idea organizers. In INTERACT'90 [756], pp. 521–526.
- [625] Sutcliffe, A. On the effective use and reuse of HCI knowledge. *ACM Transactions on Computer-Human Interaction*, 7(2), June 2000, pp. 197–221.
- [626] Sutcliffe, A. G., and Carroll, J. M. Designing claims for reuse in interactive systems design. *International Journal of Human-Computer Studies*, 50(3), 1999, pp. 213–241.
- [627] Suwa, M., Purcell, T., and Gero, J. Macroscopic analysis of design processes based on a scheme for coding designers' cognitive actions. *Design Studies*, 19(4), 1998, pp. 455–483.
- [628] Suwa, M., and Tversky, B. What architects see in their sketches: Implications for design tools. In CHI'96 [734], pp. 191–192.
- [629] Swetz, F. J. Bodily mathematics. In *From Five Fingers to Infinity: A Journey Through the History of Mathematics*. Open Court, Chicago, 1994, p. 52.
- [630] Szwillus, G., and Neal, L., Eds. *Structure-Based Editors and Environments*. Academic Press Limited, 1996.
- [631] Tallis, D. J. *Using Software Visualisation to Support Program Comprehension*. PhD thesis, University of Exeter, Department of Computer Science, July 1996.
- [632] Tateishi, A., and Walenstein, A. Applying traditional Unix tools during maintenance: An experience report. In WCRE'2000 [765], pp. 203–206.
- [633] Tauscher, L., and Greenberg, S. How people revisit web pages: Empirical findings and implications for the design of history systems. *International Journal of Human-Computer Studies*, 47(1), 1997, pp. 97–137.
- [634] Taylor, M. M. Layered protocols for computer-human dialogue. I: Principles. *International Journal of Man-Machine Studies*, 28(2/3), 1988, pp. 175–218.
- [635] Teasley, B. M. Program comprehension skills and their acquisition: A call for an ecological paradigm. In NATO.ASI.111 [759], pp. 71–79.
- [636] Teitelman, W. A display oriented programmer's assistant. *International Journal of Man-Machine Studies*, 11(2), 1979, pp. 157–187.
- [637] Tennyson, R. D., and Schott, F. Instructional design theory, research, and models. In *Instructional Design: International Perspectives*, R. D. Tennyson, F. Schott, N. M. Seel, and S. Dijkstra, Eds., vol. 1. Lawrence Erlbaum Associates, 1997, ch. 1, pp. 1–16.

- [638] Terrins-Rudge, D., and Jorgensen, A. H. Supporting the designers: Reaching the users. In Byerley *et al.* [90], ch. 1.4, pp. 87–98.
- [639] Thagard, P. *Mind: Introduction to Cognitive Science*. The MIT Press, Cambridge, MA, 1996.
- [640] Tichy, W. F. Should computer scientists experiment more? *Computer*, 31(5), May 1998, pp. 32–40.
- [641] Tilley, S. The canonical activities of reverse engineering. *Annals of Software Engineering*, 9, May 2000, pp. 249–271.
- [642] Tilley, S. R., Paul, S., and Smith, D. B. Towards a framework for program understanding. In WPC'96 [769], pp. 19–28.
- [643] Tilley, S. R., Whitney, M. J., Müller, H. A., and Storey, M.-A. D. Personalized information structures. In *ACM Eleventh International Conference on Systems Documentation* (Kitchener, ON, Oct 5–8 1993), Association for Computing Machinery, 1993, pp. 325–337.
- [644] Toleman, M. A., and Welsh, J. Systematic evaluation of design choices for software development tools. *Software—Concepts and Tools*, 19(3), 1998, pp. 109–121.
- [645] Tweedie, L. A. Interactive visualisation artifacts: How can abstractions inform design? In HCI'95 [749], pp. 247–265.
- [646] Tweedie, L. A. Characterizing interactive externalizations. In CHI'97 [735].
- [647] Tzerpos, V., and Holt, R. C. A hybrid process for recovering software architecture. In *Proceedings of CASCON 1996* (Toronto, Canada, Nov. 1996), 1996.
- [648] Tzerpos, V., and Holt, R. C. ACDC: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering, WCRE-2000* (Brisbane, Australia, Nov 23–25 2000), 2000, pp. 258–267.
- [649] Ulam, S. *Adventures of a Mathematician*. Scribner, New York, 1976.
- [650] Valkenburg, R., and Dorst, K. The reflective practice of design teams. *Design Studies*, 19(3), July 1998, pp. 249–271.
- [651] Van de Velde, W. Cognitive architectures—from knowledge level to structural coupling. In *The Biology and Technology of Intelligent Autonomous Agents*, L. Steels, Ed., vol. 144 of *NATO ASI Series F: Computer and Systems Science*. Springer-Verlag, 1995, pp. 197–221.
- [652] van Oostendorp, H., and de Mul, S., Eds. *Cognitive Aspects of Electronic Text Processing*, vol. LVIII of *Advances in Discourse Processes*. Ablex Publishing Corporation, Norwood, NJ, 1996.
- [653] van Welie, M., van der Veer, G. C., and Eliëns, A. An ontology for task world models. In *Proceedings of the 5th International Eurographics Workshop on Design Specification and Verification of Interactive Systems* (Abingdon, UK, Jun 3–5 1998), 1998.
- [654] Vans, A. M. *A Multi-Level Code Comprehension Model for Large Scale Software*. PhD thesis, Colorado State University, Department of Computer Science, Nov. 1996.
- [655] Vessey, I. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5), 1985, pp. 459–494.
- [656] Vessey, I. Cognitive fit: A theory-based analysis of the graphs versus tables literature. *Decision Sciences*, 22(2), 1991, pp. 219–240.
- [657] Vicente, K. *Cognitive Work Analysis: Toward Safe, Productive, and Healthy Computer-Based Work*. Lawrence Erlbaum Associates, Mahwah, NJ, 1999.

- [658] Vicente, K. J. A few implications of an ecological approach to human factors. In Flach *et al.* [219], ch. 3, pp. 54–67.
- [659] Vinze, A. S., Sen, A., and Liou, S. F. T. AEROBA: A blackboard approach to model formulation. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences* (Jan 7–10 1992), V. Milutinovic, B. D. Shriver, J. J. F. Nunamaker, and J. R. H. Sprague, Eds., vol. 3, 1992, pp. 551–562.
- [660] Vinze, A. S., Sen, A., and Liou, S. F. T. Operationalizing the opportunistic behavior in model formulation. *International Journal of Man-Machine Studies*, 38(3), Mar. 1993, pp. 509–540.
- [661] Visser, W. More or less following a plan during design: Opportunistic deviations in specification. *International Journal of Man-Machine Studies*, 33(3), 1990, pp. 247–278.
- [662] Visser, W. Planning and organization in expert design activities. In Gilmore *et al.* [242], pp. 25–39.
- [663] Visser, W., and Hoc, J.-M. Expert software design strategies. In Hoc *et al.* [307], ch. 3.3, pp. 235–249.
- [664] von Mayrhauser, A. Maintenance and evolution of software products. In *Advances in Computers, Volume 39*, M. C. Yovits and M. V. Zelkowitz, Eds., vol. 39 of *Advances in Computers*. Academic Press Limited, 1994, pp. 1–49.
- [665] von Mayrhauser, A., and Lang, S. Evaluating software maintenance support tools for their support of program comprehension. In *Proceedings of the 1998 IEEE Aerospace Conference* (Mar 21–28 1998), 1998, pp. 173–187.
- [666] von Mayrhauser, A., and Lang, S. A coding scheme to support systematic analysis of software comprehension. *IEEE Transactions on Software Engineering*, 25(4), July 1999, pp. 527–540.
- [667] von Mayrhauser, A., and Vans, A. Code comprehension model. Tech. Rep. CS-92-145, Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873, 1992.
- [668] von Mayrhauser, A., and Vans, A. M. From code comprehension model to tool capabilities. In *Proceedings of the 1993 International Conference on Computers and Information* (May 27–29 1993), Abou-Rabia, C. K. Chang, and W. W. Koczkodaj, Eds., IEEE Computer Society Press, 1993, pp. 469–473.
- [669] von Mayrhauser, A., and Vans, A. M. From code understanding needs to reverse engineering tool capabilities. In *Proceedings of the Sixth International Conference on Computer Aided Software Engineering* (Institute of Systems Science, National University of Singapore, Singapore, Jul 19–23 1993), H.-Y. Lee, T. F. Reid, and S. Jarzabek, Eds., IEEE Computer Society Press, 1993, pp. 230–239.
- [670] von Mayrhauser, A., and Vans, A. M. From program comprehension to tool requirements for an industrial environment. In *Proceedings of the Second Workshop on Program Comprehension* (Capri, Italy, Jul 8–9 1993), IEEE Computer Society Press, 1993, pp. 78–86.
- [671] von Mayrhauser, A., and Vans, A. M. Comprehension processes during large scale maintenance. In ICSE'94 [754], pp. 39–48.
- [672] von Mayrhauser, A., and Vans, A. M. Dynamic code cognition behaviors for large scale code. In WPC'94 [768], pp. 74–81.
- [673] von Mayrhauser, A., and Vans, A. M. Industrial experience with an integrated code comprehension model. *Software Engineering Journal*, Sept. 1995, pp. 171–182.
- [674] von Mayrhauser, A., and Vans, A. M. Program comprehension during software maintenance and evolution. *Computer*, 28(8), Aug. 1995, pp. 44–55.
- [675] von Mayrhauser, A., and Vans, A. M. Program understanding: Models and experiments. In *Advances in Computers, Volume 40*, M. C. Yovits and M. V. Zelkowitz, Eds., vol. 40 of *Advances in Computers*. Academic Press Limited, 1995, pp. 1–38.

- [676] von Mayrhauser, A., and Vans, A. M. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6), June 1996, pp. 424–437.
- [677] von Mayrhauser, A., and Vans, A. M. On the role of hypotheses during opportunistic understanding while porting large scale code. In WPC'96 [769], pp. 68–77.
- [678] von Mayrhauser, A., and Vans, A. M. On increasing our knowledge of large-scale software comprehension. *Empirical Software Engineering*, 2(2), 1997, pp. 159–163.
- [679] von Mayrhauser, A., and Vans, A. M. Program understanding behavior during debugging of large scale software. In *Empirical Studies of Programmers: Seventh Workshop* (Alexandria, VA, Oct 24–26 1997), S. Wiedenbeck and J. C. Scholtz, Eds., Association for Computing Machinery, 1997, pp. 157–179.
- [680] von Mayrhauser, A., and Vans, A. M. Program understanding during software adaptation tasks. In *Proceedings of the 1998 International Conference on Software Maintenance (CSM '98)* (Bethesda, MD, Nov 16–19 1998), IEEE Computer Society Press, 1998, pp. 316–416.
- [681] von Mayrhauser, A., Vans, A. M., and Howe, A. E. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance—Research and Practice*, 9(5), 1997, pp. 299–327.
- [682] von Mayrhauser, A., Vans, A. M., and Howe, A. E. Program understanding behaviour during enhancement of large-scale software. *Software Maintenance—Research and Practice*, 9(5), 1997, pp. 299–327.
- [683] Vosniadou, S. From cognitive theory to educational technology. In NATO.ASI.137 [760], pp. 11–18.
- [684] Vosniadou, S., De Corte, E., Glaser, R., and Mandl, H., Eds. *International Perspectives on the Design of Technology-Supported Learning Environments*. Lawrence Erlbaum Associates, 1996.
- [685] Walenstein, A. E. Developing the designer's toolkit with software comprehension models. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering* (Honolulu, Hawaii, Oct 13–16 1998), IEEE Computer Society Press, 1998, pp. 310–313.
- [686] Ware, C. The foundations of experimental semiotics: a theory of sensory and conventional representation. *Journal of Visual Languages and Computing*, 4(1), 1993, pp. 91–100.
- [687] Ware, C. *Information visualization: perception for design*. Morgan Kaufman, San Francisco, 2000.
- [688] Ware, C., and Franck, G. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics*, 15(2), Apr. 1996.
- [689] Warren, Jr., W. H. Constructing an econiche. In Flach *et al.* [219], ch. 8, pp. 210–237.
- [690] Waterson, P. E., Clegg, C. W., and Axtell, C. M. The interplay between cognitive and organizational factors in software development. In *Proceedings of Interact '95—the Fifth International Conference on Human-Computer Interaction* (Lillehammer, Norway, June 1995), K. Nordby, P. Helmersen, D. J. Gilmore, and S. A. Arnesen, Eds., Chapman and Hall, 1995, pp. 32–37.
- [691] Webb, S., and MacMillian, J. Cognitive bias in software engineering. *Communications of the ACM*, 38(6), June 1995, pp. 57–63.
- [692] Weiser, M. Programmers use slices when debugging. *Communications of the ACM*, 25(7), July 1982.
- [693] Weiser, M., and Lyle, J. R. Experiments on slicing-based debugging aids. In ESP'86 [741], pp. 187–197.
- [694] Weiser, M., and Shneiderman, B. Human factors of software design and development. In *Handbook of Human Factors/Ergonomics*, G. Salvendy, Ed. John Wiley and Sons, 1986, pp. 1398–1415.

- [695] Weizenbaum, J. *Computer Power and Human Reason: From Judgment to Calculation*. W. H. Freeman and Company, San Francisco, 1976.
- [696] Welty, C. Augmenting abstract syntax trees for program understanding. In *Proceedings of The 1997 International Conference on Automated Software Engineering (ASE'97)* (Lake Tahoe, CA, Nov 2–5 1997), Nov. 1997.
- [697] Welty, C. A. *An Integrated Representation for Software Development and Discovery*. PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, July 1995.
- [698] Wexelblat, A., and Maes, P. Footprints: History-rich tools for information foraging. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems* (Pittsburgh, PA, May 15–20 1999), Association for Computing Machinery, 1999, pp. 270–277.
- [699] Whitefield, A. A model of the engineering design process derived from hearsay-II. In INTERACT'84 [755].
- [700] Whitefield, A. Constructing appropriate models of computer users: The case of engineering designers. In *Cognitive Ergonomics and Human-Computer Interaction*, J. Long and A. Whitefield, Eds., vol. 1 of *Cambridge Series on Human-Computer Interaction*. Cambridge University Press, Cambridge, UK, 1989, ch. 3, pp. 66–94.
- [701] Whitefield, A., Esgate, A., Denley, I., and Byerley, P. On distinguishing work tasks and enabling tasks. *Interacting with Computers*, 5(3), 1993, pp. 333–347.
- [702] Whiteside, J., and Wixon, D. Discussion: Improving human-computer interaction—a quest for cognitive. In Carroll [100], pp. 353–365.
- [703] Whitley, K. N. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1), Feb. 1997, pp. 109–142.
- [704] Wiecha, C., and Henrion, M. Linking multiple program views using a visual cache. In *Proceedings of IFIP INTERACT'87: Human-Computer Interaction* (Stuttgart, Germany, Sep 1–4 1987), H. J. Bullinger and B. Shackel, Eds., IFIP, North-Holland, 1987, pp. 689–694.
- [705] Wilde, N. P. Using cognitive dimensions in the classroom as a discussion tool for visual language design. In CHI'96 [734].
- [706] Williges, R. C., Williges, B. H., and Han, S. H. Sequential experimentation in human computer interface design. In *Advances in Human-Computer Interaction*, H. R. Hartson and D. Hix, Eds., vol. 4. Ablex, Norwood, NJ, 1993, pp. 1–30.
- [707] Wills, L. M. Automated program recognition by graph parsing. Tech. Rep. TR-1358, MIT, Artificial Intelligence Laboratory, 1992. Phd Thesis.
- [708] Winn, W. An account of how readers search for information in diagrams. *Contemporary Educational Psychology*, 18, 1993, pp. 162–185.
- [709] Winograd, T. From programming environments to environments for designing. *Communications of the ACM*, 38(6), June 1995, pp. 65–74.
- [710] Winograd, T., and Flores, F. *Understanding Computers and Cognition: A New Foundation for Design*. Ablex, Norwood, NJ, 1986.
- [711] Wolf, C. G., Carroll, J. M., Landauer, T. K., John, B. E., and Whiteside, J. The role of laboratory experiments in HCI: Help, hindrance, or ho-hum? In CHI'89 [730], pp. 265–268.
- [712] Wong, K., Tilley, S. R., Müller, H. A., and Storey, M.-A. D. Structural redocumentation: A case study. *IEEE Software*, 12(1), Jan. 1995, pp. 46–54.

- [713] Woods, D. D. Commentary: Cognitive engineering in complex and dynamic worlds. In *Cognitive Engineering in Complex Dynamic Worlds*, E. Hollnagel, G. Mancini, and D. D. Woods, Eds. Academic Press Limited, 1988. Originally published in *International Journal of Man-Machine Studies*, 27(5-6), pp. 571-585.
- [714] Woods, S., Quilici, A., and Yang, Q. *Constraint-based Design Recovery for Software Reengineering: Theory and Experiments*. Kluwer Academic Publishers, 1997.
- [715] Wortham, S. Interactionally situated cognition: A classroom example. *Cognitive Science*, 25(1), 2001, pp. 37-66.
- [716] Wright, P. Cognitive overheads and prostheses: Some issues in evaluating hypertexts. In *Proceedings of the Third Annual ACM Conference on Hypertext* (San Antonio, TX, Dec 15-18 1991), Association for Computing Machinery, 1991, pp. 1-12.
- [717] Wright, P. The textbook of the future. In *Hypertext: A Psychological Perspective*, C. McKnight, A. Dillon, and J. Richardson, Eds. Ellis Horwood, 1993, pp. 137-152.
- [718] Wright, P. C., Fields, B., and Harrison, M. D. Distributed information resources: A new approach to interaction modelling. In *Proceedings of ECCE8: Eighth European Conference on Cognitive Ergonomics* (Grenada, Spain, Sep 10-13 1996), 1996.
- [719] Wright, P. C., Fields, R. E., and Harrison, M. D. Analyzing human-computer interaction as distributed cognition: The resources model. *Human Computer Interaction*, 15(1), Mar. 2000, pp. 1-41.
- [720] Young, R. M., and Barnard, P. J. The use of scenarios in human-computer interaction research: Turbocharging the tortoise of cumulative science. In *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface* (Toronto, Canada, Apr 5-9 1987), Association for Computing Machinery, 1987, pp. 291-296.
- [721] Young, R. M., Green, T. R. G., and Simon, T. Programmable user models for predictive evaluation of interface designs. In CHI'89 [730], pp. 15-19.
- [722] Young, R. M., and Simon, T. Planning in the context of human-computer interaction. In *Proceedings of the Conference on People and Computers III* (Exeter, UK, Sep 7-11 1987), D. Diaper and R. Winder, Eds., Cambridge University Press, 1987, pp. 363-370.
- [723] Zachary, W., Le Mentec, J.-C., and Ryder, J. Interface agents in complex systems. In *Human Interaction With Complex Systems: Conceptual Principles and Design Practice*, C. A. Ntuen and E. H. Park, Eds. Kluwer Academic Publishers, 1996, pp. 35-52.
- [724] Zelkowitz, M. V., and Wallace, D. R. Experimental models for validating technology. *Computer*, 31(5), May 1998, pp. 23-31.
- [725] Zhang, J. A representational analysis of relational information displays. *International Journal of Human-Computer Studies*, 45(1), 1996, pp. 59-74.
- [726] Zhang, J. The nature of external representations in problem solving. *Cognitive Science*, 21(2), 1997, pp. 179-217.
- [727] Zhang, J., and Norman, D. A. Representations in distributed cognitive tasks. *Cognitive Science*, 18, 1994, pp. 87-122.
- [728] Zimmermann, B., and Selvin, A. M. A framework for assessing group memory approaches for software design projects. In DIS'97 [740], pp. 417-426.
- [729] Ziv, H., and Osterweil, L. J. Research issues in the intersection of hypertext and software development environments. In SEHCIW'94 [762], pp. 268-279.

- [730] CHI'89. *Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems* (Austin, TX, Apr 30–May 4 1989), Association for Computing Machinery.
- [731] CHI'90. *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems* (Seattle, WA, Apr 1–5 1990), Association for Computing Machinery.
- [732] CHI'92. *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems* (Monterey, CA, May 3–7 1992), Association for Computing Machinery.
- [733] CHI'95. *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems* (Denver, Colorado, May 6–7 1995), Association for Computing Machinery.
- [734] CHI'96. Bilger, R., Guest, S., and Tauber, M. J., Eds. *Proceedings of CHI'96 Conference on Human Factors in Computing* (Vancouver, BC, Apr 13–14 1996), Association for Computing Machinery.
- [735] CHI'97. Pemberton, S., Ed. *Proceedings of CHI'97 Conference on Human Factors in Computing* (Atlanta, GA, Mar 22–27 1997), Association for Computing Machinery.
- [736] CHI'98. *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems* (Los Angeles, CA, Apr 18–23 1998), Association for Computing Machinery.
- [737] CS'94. Tepfenhart, W. M., Dick, J. P., and Sowa, J. F., Eds. *Conceptual Structure: Current Practices — Proceedings of the Second International Conference on Conceptual Structures* (College Park, Maryland, Aug 16–20 1994), vol. 835 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.
- [738] CT'2001. Benyon, M., Nehaniv, C. L., and Dautenhahn, K., Eds. *Instruments of Mind: Proceedings of The Fourth International Conference on Cognitive Technology*, vol. 2117 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 2001.
- [739] DIS'2000. *Proceedings on Designing Interactive Systems: Processes, Practices, Methods, and Techniques* (Brooklyn, NY, Aug 17–19 2000), Association for Computing Machinery.
- [740] DIS'97. *Proceedings of Designing Interactive Systems: Processes, Practices, Methods, & Techniques (DIS'97)* (The Netherlands, Aug 18–20 1997), Association for Computing Machinery.
- [741] ESP'86. Soloway, E. M., and Iyengar, S., Eds. *Empirical Studies of Programmers* (Washington, DC, Jun 5–6 1986), Ablex Publishing Corporation.
- [742] ESP'87. Olson, G. M., Sheppard, S., and Soloway, E. M., Eds. *Empirical Studies of Programmers: Second Workshop* (Washington, DC, Dec 7–8 1987), Ablex Publishing Corporation.
- [743] ESP'91. Koenemann-Bellinveau, J., Mohen, T. G., and Robertson, S. P., Eds. *Empirical Studies of Programmers: Fourth Workshop* (New Brunswick, NJ, Dec 7–9 1991), Ablex Publishing Corporation.
- [744] ESP'93. Cook, C. R., Scholtz, J. C., and Spohrer, J. C., Eds. *Empirical Studies of Programmers: Fifth Workshop* (Palo Alto, CA, Dec 3–15 1993), Ablex Publishing Corporation.
- [745] ESP'96. Gray, W. D., and Boehm-Davis, D. A., Eds. *Empirical Studies of Programmers: Sixth Workshop* (Alexandria, VA, Jan 5–7 1996), Ablex Publishing Corporation.
- [746] HCI'88. Jones, D. M., and Winder, R. L., Eds. *People and Computers IV: Proceedings of the Fourth Conference of the British Computer Society Human-Computer Interaction Specialist Group* (University of Manchester, Sep 5–9 1988), British Informatics Society, Cambridge University Press.
- [747] HCI'89. Sutcliffe, A., and Macaulay, L., Eds. *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society Human-Computer Interaction Specialist Group* (University of Nottingham, UK, Sep 5–8 1989), British Informatics Society, Cambridge University Press.

- [748] HCI'91. Diaper, D., and Hammond, N. V., Eds. *People and Computers VI: Proceedings of the HCI'91 Conference* (Edinburgh, Scotland, Aug 20–23 1991), British Informatics Society, Cambridge University Press.
- [749] HCI'95. Kirby, M. A. R., Dix, A. J., and Finlay, J. E., Eds. *People and Computers X, Proceedings of HCI'95* (Huddersfield, England, Aug 29–Sep 1 1995), Cambridge University Press.
- [750] HCII'95. Anzai, Y., Ogawa, K., and Mori, H., Eds. *Symbiosis of Human and Artifact: Future Computing and Design for Human-Computer Interaction* (Tokyo, Japan, Jul 9–14 1995), vol. 20A/20B of *Advances in Human Factors/Ergonomics*, Elsevier Science Ltd.
- [751] HPW'86. *Proceedings of the ACM Conference on History of Personal Workstations* (Palo Alto, CA, Jan 9–10 1986), Association for Computing Machinery.
- [752] HT'89. *Proceedings of the Second Annual ACM Conference on Hypertext* (Pittsburgh, PA, Nov 5–8 1989), Association for Computing Machinery.
- [753] ICSE'91. *14th IEEE International Conference on Software Engineering* (Austin, TX, May 13–17 1991), IEEE Computer Society Press.
- [754] ICSE'94. *IEEE International Conference on Software Engineering – 1994* (May 16–21 1994), IEEE Computer Society Press.
- [755] INTERACT'84. Shackel, B., Ed. *Proceedings of IFIP INTERACT'84: Human-Computer Interaction* (London, U.K., Sep 4–7 1984), IFIP, North-Holland.
- [756] INTERACT'90. Diaper, D., Ed. *Proceedings of IFIP INTERACT'90: Human-Computer Interaction* (Cambridge, U.K., Aug 27–31 1990), IFIP, North Holland.
- [757] INTERCHI'93. Ashlund, S., Ed. *INTERCHI '93 : Conference Proceedings : Bridges Between Worlds* (Amsterdam, The Netherlands, Apr 24–29 1993), Association for Computing Machinery.
- [758] KBSE'96. *Proceedings of the Eleventh Knowledge-Based Software Engineering Conference* (Syracuse, NY, Sep 25–28 1996), IEEE Computer Society Press.
- [759] NATO.ASI.111. Lemut, E., Dettori, G., and du Boulay, B., Eds. *Cognitive Models and Intelligent Environments for Learning Programming*, vol. 111 of *NATO ASI Series. Series F, Computer and System Sciences*, NATO, Springer-Verlag, 1993.
- [760] NATO.ASI.137. Vosniadou, S., Corte, E. D., and Mandl, H., Eds. *Technology-based Learning Environments: Psychological and Educational Foundations*, vol. 137 of *NATO ASI Series. Series F, Computer and System Sciences*, NATO, Springer-Verlag, 1994.
- [761] NATO.ASI.146. *Computers and Exploratory Learning*, vol. 146 of *NATO ASI Series. Series F, Computer and System Sciences*, Springer-Verlag, 1995.
- [762] SEHCIW'94. Taylor, R. N., and Coutaz, J., Eds. *Software Engineering and Human-Computer Interaction; ICSE '94 Workshop on SE-HCI: Joint Research Issues* (Sorrento, Italy, May 16–17 1994), Springer-Verlag.
- [763] SIGSOFT'92. *Proceedings of the ACM SIGPLAN/SIGSOFT Conference, 1992* (Washington, DC, Dec 9–11 1992), vol. 17, Association for Computing Machinery. Published Proceedings of the ACM SIGPLAN/SIGSOFT Conference, 1992.
- [764] VSRD'99. Gero, J. S., and Tversky, B., Eds. *Visual and Spatial Reasoning in Design* (MIT, MA, Jun 15–17 1999), Key Centre of Design Computing and Cognition, University of Sydney.
- [765] WCRE'2000. *Proceedings of the 7th Working Conference on Reverse Engineering* (Brisbane, Australia, Nov 23–25 2000), IEEE Computer Society Press.

- [766] WCRE'93. *Proceedings of the 1993 Working Conference on Reverse Engineering* (Baltimore, Maryland, May 21–23 1993), IEEE Computer Society Press.
- [767] WPC'92. *Proceedings of the Program Comprehension Workshop*, IEEE Computer Society Press, 1992.
- [768] WPC'94. Sipple, R. S., Ed. *Proceedings of the Third Workshop on Program Comprehension* (Washington, DC, Nov 14–15 1994), IEEE Computer Society Press.
- [769] WPC'96. Cimitile, A., and Müller, H. A., Eds. *Proceedings of the Fourth Workshop on Program Comprehension* (Berlin, Mar 29–31 1996), IEEE Computer Society Press.
- [770] WPC'97. *Proceedings of the Fifth Workshop on Program Comprehension* (Dearborn, Michigan, May 28–30 1997), IEEE Computer Society Press.
- [771] WPC'98. *Proceedings of the Sixth IEEE International Workshop on Program Comprehension* (Ischia, Italy, Jun 24–26 1998), IEEE Computer Society Press.