Software Qual J (2006) 14:187–208
DOI 10.1007/s11219-006-9216-4

Theories, tools and research methods in program comprehension: *past, present and future*

Margaret-Anne Storey

Abstract Program comprehension research can be characterized by both the theories that provide rich explanations about how programmers understand software, as well as the tools that are used to assist in comprehension tasks. In this paper, I review some of the key cognitive theories of program comprehension that have emerged over the past thirty years. Using these theories as a canvas, I then explore how tools that are commonly used today have evolved to support program comprehension. Specifically, I discuss how the theories and tools are related and reflect on the research methods that were used to construct the theories and evaluate the tools. The reviewed theories and tools are distinguished according to human characteristics, program characteristics, and the context for the various comprehension tasks. Finally, I predict how these characteristics will change in the future and speculate on how a number of important research directions could lead to improvements in program comprehension tool development and research methods.

Keywords Program comprehension · Cognitive models · Software tools · Research methods · Software navigation · Software visualization · Collaborative software engineering

1. Introduction

Challenges in understanding programs are all too familiar from even before the days of the first software engineering workshop (Nato, 1968). Since that time, the field of program comprehension as a research discipline has evolved considerably. The goal of our community is to build an understanding of these challenges, with the ultimate objective of developing more effective tools and methods. From these early days we have come to accept that there is no silver bullet (Brooks, 1987), but the community has made advances which have helped software engineers tackle important problems such as the Y2K problem and the integration of heterogeneous distributed systems.

M.-A. Storey (✉)
Department of Computer Science, University of Victoria, Canada
e-mail: mstorey@uvic.ca

There are two key strands of program comprehension research. The first is empirical research which strives for an understanding of the cognitive processes that programmers use when understanding programs. The second involves technology research with a focus on developing semi-automated tool support to improve program comprehension. This paper provides a meta-analysis of how these two strands of research are related.

Empirical research in program comprehension has culminated in a wide variety of theories that provide rich explanations of how programmers understand programs and can provide advice on how program comprehension tools and methods may be improved. In response to these theories, and in some cases in parallel to theory development, many powerful tools and innovative software processes have evolved to improve comprehension activities.

The field of program comprehension research has been rich and varied, with various shifts in paradigms and research cultures during the last few decades. A multitude of differences in program characteristics, programmer ability and software tasks have led to many diverse theories, research methods and tools. In this paper, I provide a review of this work to create a landscape of program comprehension research. Such a view emphasizes how the theories and tools are related and should reveal if parts of the landscape have not received much attention. This review, combined with an excursion to newer areas of software engineering theory and practice, directs us to specific areas for the future of program comprehension research.

This paper is organized as follows. In Section 2, I provide an overview of comprehension theories and describe how programmer, program and task variability can impact comprehension strategies. In Section 3, the implications of cognitive theories on tool requirements are considered and several theories that specifically address tool support are reviewed. In Section 4, I briefly describe comprehension tools and refer back to the theories about tool support. In Section 5, I look to the future and predict how programmer and program characteristics are likely to vary in the near term. Building on these predicted changes, I then suggest in Section 6 how research methods, theories and tools will evolve in the future. The paper concludes in Section 7.

2. A review of program comprehension cognitive theories

Francois Détienne's book, "Software Design—Cognitive Aspects" (Détienne, 2001), provides an excellent review of the history of cognitive models and related experiments over the past twenty or so years. She delves back to a time, in the early 1970's, when experiments were done without theoretical frameworks to guide the evaluations. Consequently, it was neither possible to understand nor to explain to others why one tool or technique might be superior to other options.

The lack of theories was recognized as being problematic. As the field of program comprehension matured, research methods and theories were borrowed from other areas of research, such as text comprehension, problem solving and education. Using these theoretical underpinnings, cognitive theories about how programmers understand programs and how tools could support comprehension were developed. Perceived benefits of having these models include having rich explanations of behaviour that would lead to more efficient processes, methods and tools, as well as improved education procedures (Hohmann, 1996).

In this section, I review some of the influential cognitive theories in program comprehension research. I then present some results on how differences between programmers, programming paradigms and task characteristics impact comprehension. First some terminology is defined.

2.1. Concepts and terminology

A *mental model* describes a developer's mental representation of the program to be understood whereas a *cognitive model* describes the cognitive processes and temporary information structures in the programmer's head that are used to form the mental model. *Cognitive support* refers to support provided for cognitive tasks such as thinking or reasoning (Walenstein, 2003).

Programming plans are generic fragments of code that represent typical scenarios in programming. For example, a sorting program will contain a loop to compare two numbers in each iteration (Soloway, 1984). Programming plans are also often referred to as clichés and schemas (Rich, 1990). *Delocalized plans* occur when a programming plan is implemented in disparate areas of the program (Soloway, 1984).

Beacons are recognizable, familiar features in the code that act as cues to the presence of certain structures (Brooks, 1983). For example, a procedure name can indicate the implementation of a specific feature. *Rules of programming* discourse capture the conventions of programming, such as coding standards and algorithm implementations (Soloway, 1984). Rules of discourse invoke expectations in the mind of the programmer.

2.2. Top-down comprehension

Brooks theorizes that programmers understand a completed program in a top-down manner where the comprehension process is one of reconstructing knowledge about the domain of the program and mapping this knowledge to the source code (Brooks, 1983). The process starts with a hypothesis about the general nature of the program. This initial hypothesis is then refined in a hierarchical fashion by forming subsidiary hypotheses. Subsidiary hypotheses are refined and evaluated in a depth-first manner. The verification (or rejection) of hypotheses depends heavily on the absence or presence of beacons (Brooks, 1983).

Soloway and Ehrlich (Soloway, 1984) observed that top-down understanding is used when the code or type of code is familiar. They observed that expert programmers use beacons, programming plans and rules of programming discourse to decompose goals and plans into lower-level plans. They noted that *delocalized plans* complicate program comprehension.

2.3. Bottom-up comprehension

The bottom-up theories of program comprehension assume that programmers first read code statements and then mentally chunk or group these statements into higher level abstractions. These abstractions (chunks) are aggregated further until a high-level understanding of the program is attained (Shneiderman, 1979). Shneiderman and Mayer's cognitive framework differentiates between *syntactic* and *semantic* knowledge of programs. Syntactic knowledge is language dependent and concerns the statements and basic units in a program. Semantic knowledge is language independent and is built in progressive layers until a mental model is formed which describes the application domain.

Pennington also describes a bottom-up model (Pennington, 1987). She observed that programmers first develop a control-flow abstraction of the program which captures the sequence of operations in the program. This model is referred to as the *program model* and is developed through the chunking of microstructures in the text (statements, control constructs and relationships) into macrostructures (text structure abstractions) and by cross-referencing these structures. Once the program model has been fully assimilated, the *situation model* is developed. The situation model encompasses knowledge about data-flow abstractions and functional abstractions (the program goal hierarchy). The programmer's understanding is further enhanced through the cross referencing of artifacts in the program model and situation model.

2.4. Knowledge-base model

Letovsky views programmers as opportunistic processors capable of exploiting both bottom-up and top-down cues (Letovsky, 1986). There are three components to his model:

- The **knowledge base** encodes the programmer's expertise and background knowledge. The programmer's internal knowledge may consist of application and programming domain knowledge, program goals, a library of programming plans and rules of discourse.
- The **mental model** encodes the programmer's current understanding of the program. Initially it consists of a specification of the program goals and later evolves into a mental model which describes the implementation in terms of the data structures and algorithms used.
- The **assimilation process** describes how the mental model evolves using the programmer's knowledge base together with program source code and documentation. The assimilation process may be a bottom-up or top-down process depending on the programmer's initial knowledge base

Letovsky describes the observation of two recurring patterns: asking questions and conjecturing answers (Letovsky, 1986). He calls such activities *inquiries*. An inquiry may consist of a programmer asking a question (for example, what is the purpose of variable x?), conjecturing an answer (x stores the maximum of a set of numbers), and then searching through the code and documentation to verify the answer (the conjecture is verified if x is in an assignment statement where two values are compared to see which is greater). According to Letovsky there are three major types of hypotheses:

- **why** conjectures (questioning the role of a function or piece of code),
- **how** conjectures (what is the method for accomplishing a goal),
- and **what** conjectures (what is a variable or one of the program functions).

2.5. Opportunistic and systematic strategies

Littman *et al.* observed programmers enhancing a personnel database program (Littman, 1986). They observed that programmers either systematically read the code in detail, tracing through the control-flow and data-flow abstractions in the program to gain a global understanding of the program, or that they take an as-needed approach, focusing only on the code relating to a particular task at hand. Subjects using a systematic strategy acquired both static knowledge (information about the structure of the program) and causal knowledge (interactions between components in the program when it is executed). This enabled them to form a mental model of the program. However, those using the as-needed approach only acquired static knowledge resulting in a weaker mental model of how the program worked. More errors occurred since the programmers failed to recognize causal interactions between components in the program.

2.6. The Integrated Metamodel

The Integrated Metamodel, developed by von Mayrhauser and Vans, builds on the models just summarized. Their model consists of four major components (von Mayrhauser, 1993). The first three components describe the comprehension processes used to create mental representations at various levels of abstraction and the fourth component describes the knowledge base needed to perform a comprehension process:

- The **top-down (domain) model** is usually invoked and developed using an as-needed strategy, when the programming language or code is familiar. It incorporates domain knowledge as a starting point for formulating hypotheses.
- The **program model** may be invoked when the code and application is completely unfamiliar. The program model is a control-flow abstraction.
- The **situation model** describes data-flow and functional abstractions in the program. It may be developed after a partial program model is formed using systematic or opportunistic strategies.
- The **knowledge base** consists of information needed to build these three cognitive models. It represents the programmer's current knowledge and is used to store new and inferred knowledge.

Understanding is formed at several levels of abstraction simultaneously by switching between the three comprehension processes.

2.7. Impact of program characteristics

Programs that are carefully designed and well documented will be easier to understand, change or reuse in the future. Pennington's experiments showed that the choice of language has an effect on the comprehension processes (Pennington, 1987). COBOL programmers consistently fared better at answering questions related to data-flow than FORTRAN programmers, and FORTRAN programmers consistently fared better than COBOL programmers for control-flow questions.

Programming paradigms will also impact comprehension strategies. Object-oriented (OO) programs are often seen as a more natural fit to problems in the real world because of 'is-a' and 'is-part-of' relationships in a class hierarchy and structure, but others argue that objects do not always map easily to real world problems (Détienne, 2001). In OO programs, additional abstractions that capture domain concepts are achieved through encapsulation and polymorphism (Détienne, 2001). Message-passing is used for communication between class methods and hence programming plans are dispersed (i.e. scattered) throughout classes in an OO program. Corritore *et al.* compared how experts comprehend object-oriented and procedural programs (Corritore, 1999). They noted that experts make more use of domain concepts in the initial understanding of object oriented programs.

2.8. Influence of individual programmer differences

There are many individual characteristics that will impact how a programmer tackles a comprehension task. These differences also impact the requirements for a supporting tool. There is a huge disparity in programmer ability and creativity which cannot be measured simply by their experience (Curtis, 1981). Vessey presents an exploratory study to investigate expert and novice debugging processes (Vessey, 1985). She classified programmers as expert or novice based on their ability to chunk effectively. She notes that experts used breadth-first approaches and at the same time were able to adopt a system view of the problem area, whereas novices used breadth-first and depth-first approaches but were unable to think in system terms.

Détienne also notes that experts make more use of external devices as memory aids (Détienne, 2001). Experts tend to reason about programs according to both functional and object-oriented relationships and consider the algorithm, whereas novices tend to focus on objects. Burkhardt *et al.* (Burkhardt 1998) performed experiments to study how object-oriented programs are comprehended. They observed that novices are less likely than experts to use inheritance and compositional relationships to guide their comprehension. Finally, Davies (Davies, 1993) notes that experts tend to externalize low level details, but novices externalize higher levels of representation related to the problem.

2.9. Effects of ask variability in program comprehension

According to Clayton there is no real agreement on what it is about a program that needs to be understood (Clayton, 1998). This is not surprising as program comprehension is not an end goal, but rather a necessary step in achieving some other objective. Tasks that involve comprehension steps include maintenance such as fixing an error, reusing code, refactoring, impact analysis of proposed changes and changing the functionality of the program. Other tasks include software inspection, evaluating quality metrics (such as reusability, performance, security), and testing.

Obviously the type and scope of the ultimate programming task has have an impact on the comprehension process followed. If a task is a simple one, the change probably only affects a small portion of the code. For more complex changes, the programmer has to consider global interactions thus requiring that the programmer obtain a thorough understanding of the causal relationships in the entire program.

Pennington's research indicates that a task requiring recall and comprehension resulted in a programmer forming a program model (control-flow abstraction) of the software whereas a task to modify the program resulted in a programmer forming a situation model containing data-flow and functional information (Pennington, 1987).

For a programmer, a reuse task requires that she first understand the source problem, retrieve an appropriate target solution, and then adapt the solution to the problem. The mapping from the problem to the solution is often done using analogical reasoning (Détienne, 2001) and may involve iterative searching through many possible solutions. Davies *et al.* note that the task, whether it is a development or comprehension task, influences which information is externalized by experts (Davies, 1993).

2.10. Discussion: *Implications for tool research*

Many of the researchers that developed the traditional cognitive theories for program comprehension discuss the implications of the developed theories on tool design and in some cases also discuss how education and program design could be improved to address program understanding challenges. However, in many cases, the connection to tools and how they could be improved or evaluated according to the theories could be stronger. Moreover, some of these results were also criticized because the researchers studied novice programmers doing fabricated tasks (Curtis, 1986). Vans and von Mayrhauser are notable exceptions. Despite these criticisms, program comprehension research does contain many gems of advice on how tools can be improved. The advantage for tool designers is that they can use these theories to help them understand, not only what features are needed, but also why some features may neither be appropriate nor sufficient to assist comprehension tasks. Moreover, the research which explores differences in programmers, programs and tasks can be used to guide tool designers to meet specific needs.

3. Current theories and tool support

What features should an ideal tool to support program comprehension have? Program comprehension tools play a supporting role in other software engineering activities of design, development, maintenance, and redocumentation. As discussed in the previous section of this paper, there are many characteristics which influence the cognitive strategies the programmers use and in turn influence the requirements for tool support. In this section, I look in general terms at tool requirements but these should be refined for a tool to be deployed in a particular context. In Section 3.1, I review specific research that recommends tool features. I then extract and synthesize tool requirements based on the prior discussion on cognitive theories and their implications for tools in Section 3.2.

3.1. Cognitive models and tool implications

Documentation: Brooks suggests that documentation should be designed to support top-down comprehension strategies (Brooks, 1983). He advocates that it is as important to document the problem domain as it is to document programming concepts. Also stress the importance of explicitly documenting domain knowledge to improve comprehension (Clayton *et al.*, 1998). Brooks suggests that different styles of languages require different styles of documentation. Observations from Laitenberger *et al.*'s research corroborate that different people read the documentation from different points of view and how they read the documentation will depend on their comprehension goal (Laitenberger, 1995). Prechelt *et al.* (Prechelt, 2002) report from their experiments that redundant design pattern documentation, i.e. program level documentation, improves maintenance tasks. The additional documentation results in faster task completion times and fewer errors.

Browsing and navigation support: The top-down process requires browsing from high-level abstractions or concepts to lower level details, taking advantage of beacons (Brooks, 1983) in the code. Flexible browsing support also helps to offset the challenges from de-localized plans. Bottom-up comprehension requires following control-flow and data-flow links as well as cross-referencing navigation between elements in the program and situation models defined by Pennington. The Integrated Metamodel suggests that frequent switching between top-down and bottom-up browsing should be supported (von Mayrhauser, 1993). Since both experts and novices switch between top-down and bottom-up strategies (Burkhardt, 1998), they may both benefit from tools that support breadth-first and depth-first browsing.

Searching and querying: Tool support may be beneficial for searching of code snippets by analogy and for iterative searching. Inquiry episodes (Letovsky, 1986) should be supported by allowing the programmer to query on the role of a variable, function etc. A program slice is a subcomponent of the program that may potentially impact the value of some variables at another part of the program (Tip, 1995). Weiser conducted empirical research that demonstrated that programmers use slices during debugging (Weiser, 1982). Francel *et al.* (Francel, 1999) also determined that slicing techniques can be useful during debugging to help the programmers find the location of a program fault. This research suggests that comprehension tools should therefore have query support for program slices, especially for tools that support debugging.

Multiple views: Due to the diversity of comprehension strategies and the variability of questions programmers need to answer during comprehension, programming environments need to provide different ways of representing programs. One view could show a textual representation of the code in the program, an additional view may show the message call graph providing insight into the programming plans, while a third view could show a representation of the classes and relationships between them to show an object-centric or data-centric view of the program. These orthogonal views, if easily accessible, should facilitate comprehension, especially if effectively combined and cross-referenced. Petre *et al.* discuss some of the challenges with using and bridging between multiple views (Petre, 1998).

Context-driven views: The size of the program and other program metrics influence which view is the preferred one to show to support a programmer's task. For example, to support browsing of an unfamiliar object-oriented program, it is usually preferable to show the inheritance hierarchy as the initial view. However, if the inheritance hierarchy is flat, it may be more appropriate to show a call graph as the default view.

Cognitive support: Experts need external devices and scratchpads to support their cognitive tasks, whereas novices need pedagogical support to help them access information about the programming language and the corresponding domain. Robbins' research (Robbins, 1996) indicates the importance of having tool support to manage "to do" activities during design. More cognitive support could also benefit comprehension for sequential tasks such as code inspections and debugging.

3.2. Tool requirements explicitly identified

Several researchers studied expert programmers in industrial settings and consequently recommended specific requirements for improving tools to support comprehension. Others built on their own personal or colleagues' experiences to recommend needed tool features. These efforts and the requirements they determined are summarized here.

Concept assignment problem: Biggerstaff notes that one of the main difficulties in understanding comes from mapping what is in the code to the software requirements – he terms this the concept assignment problem (Biggerstaff, 1983). Although automated techniques can help locate programming concepts and features, it is challenging to automatically detect human oriented or domain based concepts. The user may need to indicate a starting point and then use program slicing techniques to find related code. It may also be possible for an intelligent agent (that has domain knowledge) to scan the code and search for candidate starting points. From experimenting with research prototypes, Biggerstaff found that queries, slicing, graphical views and hypertext were important tool features.

Reverse engineering tool needs: Von Mayrhauser and Vans, from their research on the Integrated Metamodel, make an explicit recommendation for tool support for reverse engineering (von Mayrhauser, 1993). They determined basic information needs according to cognitive tasks and suggested the following tool capabilities to meet those needs:

- Top-down model: on-line documents with keyword search across documents; pruned call trees; difference tools; history of browsed locations; and entity fan-in.
- Situation model: a view showing a list of domain sources including non-code related sources; and a view displaying a visual representation of major domain functions.
- Program model: pop-up declarations; on-line cross-reference reports and function count.

Wong also discusses reverse engineering tool features (Wong, 2000). He specifically mentions the benefits of using a "notebook" to support continuous comprehension processes.

Importance of search and history: Singer and Lethbridge also observed the work practices of software engineers (Singer, 1997). They explored the activities of a single engineer, as well as a group of engineers, and considered company-wide tool usage statistics. Their study led to the requirements for a tool that was implemented and successfully adopted by the company. Specifically they suggested tool features to support "just-in-time comprehension of source code". They noted that engineers after working on a specific part of the program quickly forget details when they move to a new location. This forces them to rediscover information at a later time. They suggest that tools need the following features to support rediscovery:

- Search capabilities so that the user can search for code artifacts by name or by pattern matching.
- Capabilities to display all relevant attributes of the items retrieved as well as relationships among items.
- Features to keep track of searches and problem-solving sessions, to support the navigation of a persistent history.

Information needs for maintainers: Erdös and Sneed designed a tool to support maintenance following many years of experience in the maintenance and reengineering industry. They proposed that a programmer maintaining an unfamiliar program will need to answer the following seven questions (Erdos, 1998):

1. Where is a particular subroutine/procedure invoked?
2. What are the arguments and results of a function?
3. How does control flow reach a particular location?
4. Where is a particular variable set, used or queried?
5. Where is a particular variable declared?
6. Where is a particular data object accessed?
7. What are the inputs and outputs of a module?

Software visualization tool needs: In (Storey, 1999) we presented a cognitive framework of design elements for improving software exploration tools. The framework provides an overview of various tool features which could provide cognitive support for the various comprehension models such as top-down, bottom-up and knowledge based approaches. The emphasis was on tool support for browsing static software structures and architectures.

Pacione *et al.* (Pacione, 2004) provide a more extensive software visualization model to support program comprehension. Although their model was specifically designed for evaluating software visualization tools, it is also usable as a mechanism for evaluating more general purpose program comprehension tool requirements. Their paper gives a detailed list of both specific and general comprehension tasks. General tasks include determining how the high-level components of a system interact and determining how interactions occur between objects. More specific tasks include asking how the problem domain functionality is implemented in the software systems. Pacione further maps these tasks to specific activities and determines how the information needs of these activities can be met through static and dynamic software visualizations.

3.3. Discussion: *Methods for determining program comprehension tool requirements*

Determining which new features should be added or which features should be replaced or improved in a program comprehension environment is not trivial. On the one hand, the empirical approach results in the construction of theories about program comprehension strategies and proposed tools. Despite some criticisms, these predominantly formal experiments nevertheless have contributed to the wide body of knowledge we now have about comprehension theories and tool requirements. Moreover, some researchers, such as von Mayrhauser and Vans, and Singer and Lethbridge, have conducted more qualitative experiments using professional programmers on industrially-relevant programs. Techniques such as the analysis of think-aloud protocols (Letovsky, 1986) and contextual inquiry (Beyer, 1997) proved useful in many of these ecologically valid experiments.

In contrast with the empirical approach, some researchers and tool designers may rely on practical experience and intuition to propose what is needed in a comprehension tool. Given the variability in comprehension settings, all of these approaches contribute to the required body of knowledge.

4. Tool research

The field of program comprehension research has resulted in many diverse tools to assist in program comprehension. Program understanding tool features can be roughly categorized according to three categories: extraction, analysis and presentation (Tilley, 1996).

Extraction tools include parsers and data gathering tools to collect both static and dynamic data. Static data is obtained by extracting facts from the source code. A fact extractor should be able to determine what artifacts the program defines, uses, imports and exports, as well as the relationships between those artifacts. The technologies underlying fact extractors are based on techniques from compiler construction (Aho, 2000). Example modern fact extractors include CAN, a fast C/C++ extractor, from the Columbus reverse engineering tool (Ferenc, 2004) and CPPX (Dean, 2001). Moonen *et al.* discuss the benefits of island grammars for generating parsers that can handle incomplete source code and different language dialects (Moonen, 2001). Dynamic data is obtained by examining and extracting data from the run time behaviour of the program. Such data can be extracted through a wide variety of trace exploration tools and techniques (Hamou-Lhadj, 2004).

Analysis tools support activities such as clustering, concept assignment, feature identification (Eisenbarth, 2003), transformations, domain analysis, slicing and metrics calculations. There are numerous software clustering techniques that can be used during reverse engineering to identify software components (Koschke, 2000). Frank Tip provides a survey of program slicing techniques (Tip, 1995) and Mathias *et al.* discuss the role of various software metrics on program comprehension (Mathias, 1999).

Dynamic analysis usually involves instrumentation of the source code. With dynamic analysis, only a subset of the program code may be relevant but dynamic traces can be very large posing significant challenges during the analysis of the data. Systä *et al.* describe how static analysis can be used to prune the amount of information looked at during dynamic analysis (Systä, 2001).

Presentation tools include code editors, browsers, hypertext viewers, and visualizations. Integrated software development and reverse engineering environments usually have some features from each category. The set of features they support is usually determined by the purpose for the resulting tool or by the focus of the research.

It is possible to examine a selection of these environments and to recover the motivation for the features they provide by tracing back to the tool and cognitive theories. For example, the well known Rigi system (Muller, 1988) has support for multiple views, cross-referencing and queries to support bottom-up comprehension. The Reflexion tool (Murphy, 1995) has support for the top-down approach through hypothesis generation and verification. The Bauhaus tool (Eisenbarth, 2001) has features to support clustering (identification of components), slicing and concept analysis. The SHriMP tool (Storey, 2003) provides navigation support for the Integrated Metamodel, i.e. frequent switching between strategies. And the Codecrawler tool (Lanza, 2001) uses visualization of metrics to support understanding of an unfamiliar system and to identify bottlenecks and other architectural features.

4.1. Discussion: Methods for evaluating comprehension tools

Evaluating program comprehension tools can be a formidable task for a researcher. In many cases the comprehension tools are evaluated by the researchers using case studies. There have been some usability experiments conducted to evaluate program comprehension tools (Storey, 2000). Bellay and Gall conducted a comparative evaluation of five reverse engineering tools using a case study and an evaluation framework (Bellay, 1995).

Tools which fall into the extraction and analysis categories are inherently easier to evaluate if there are well established benchmarks and associated criteria (Sim, 2003). However, for many of the tools that present information to the user and offer cognitive support, it is not always that easy to determine what to measure and when to take such measurements.

The comprehension theories can play a role in the evaluation of these environments. They can be used as a first step in performing a heuristic evaluation of the environment and in describing what the environment does and does not do. Secondly, they can be used to help guide evaluations and to assist in presenting results. The issue of evaluation is discussed further in Section 6 after a discussion of how the characteristics of programs and programmers are currently evolving.

5. Programmer and program trends

Any research paper or talk that attempts to predict the future, always discusses how risky and difficult such an endeavour is. Fortunately, it is less risky to closely examine current trends and predict what will occur in the very near future. Tilley and Smith, in 1996, wrote a thought-provoking paper entitled “Coming Attractions in Program Understanding” (Tilley, 1996). I follow a similar tack to that of Tilley and Smith, and keep the distance to the horizon short by looking at the near future. To guide theory and tool predictions, I first consider how some of the programmer and program characteristics will evolve.

5.1. Programmer characteristics

More diversified programmers: The need to use computers and software intersects every walk of life. Programming, and hence program comprehension, is no longer a niche activity.

Scientists and knowledge workers in many walks of life have to use and customize software to help them do science or other work. In our research alone, we have already worked alongside scientists from a variety of knowledge domains including forestry, astronomy, business and medical science domains. We have encountered that these scientists are using and developing sophisticated software through web services and other mechanisms despite the fact that they lack a formal education in computer science. Consequently, there is a need for techniques to assist in non-expert and end-user program comprehension. Fortunately, there is much work in this area (especially at conferences such as Visual Languages and by the PPIG group—www.ppig.org), where they investigate how comprehension can be improved through tool support for spreadsheet and other end user applications.

Sophisticated users: Currently, advanced visual interfaces are not often used in development environments. A large concern by many tool designers is that these advanced visual interfaces require complex user interactions. However, tomorrow's programmers will be more familiar with game software and other media that displays information rapidly and requires sophisticated user controls. Looking at today's industry trends, many programmers are using larger screens and multiple monitors to help with programming and other complex tasks. Consequently, the next generation of users will have more skill at interpreting information presented visually and at manipulating and learning how to use complex controls on larger displays and on multiple monitors. Grudin (Grudin, 2001) discusses some of the implications for multiple monitor use for programmers as well as other classes of users. Modern developers are also more likely to have had much exposure to social software such as blogging, wikis and social bookmarking (Hammond, 2005). Such features are be expected to be in the next generation of development environments.

Globally distributed teams: Advances in communication technologies have enabled globally distributed collaborations in software development. Distributed open source development is having a significant impact on industry. The most notable examples are Linux and Eclipse. Some research has been conducted on studying collaborative processes in open source projects (Mockus, 2002, Gutwin, 2004, German, 2006), but more research is needed to study how distributed collaborations impact comprehension.

Agile developers: Agile development which incorporates techniques such as test driven development and pair programming has an impact on how programs are comprehended within an organization. To date there have been few studies to investigate how these practices improve or possibly hinder how programmers comprehend programs.

5.2. Program characteristics

Distributed applications and web-based applications are becoming more prevalent with technologies such as .NET, J2EE and web services. One programming challenge that is occurring now and is likely to increase, is the combination of different paradigms in distributed applications, e.g. a client side script sends XML to a server application. Service oriented architectures are also becoming popular and may require different comprehension tool support.

Higher levels of abstraction: Visual composition languages for business applications are also on the increase. As the level of abstraction increases, comprehension challenges are shifting from the understanding of source code in a single program to more abstract concepts such as understanding the services provided by and the interfaces of components in libraries and how to make use of software frameworks.

Dynamically configured systems: The prevalence of model driven development and autonomic systems are leading to complex systems that are dynamically configured at run time. Such systems will pose significant comprehension challenges. Pluggable frameworks such as Eclipse are also posing development challenges as it is difficult to understand and manage the dependencies resulting from different versions of many plug-ins. More advanced tool support for such environments will be required as these technologies mature.

Aspect-oriented programming: The introduction of aspects as a construct to manage scattered concerns (delocalized plans) in a program has created much excitement in the software engineering community. Aspects have been shown to be effective for managing many programming concerns, such as logging and security. However, it is not clear how aspects written by others will improve program understanding, especially in the long term. The focus of the work currently is on developing techniques and tools to support aspect oriented programming and on how to identify aspects in legacy code for future refactorings (Hannemann, 2001, Tonella, 2004, van Deursen, 2004). More empirical work is needed to validate and demonstrate the assumed benefits of aspect oriented programming for the long term comprehension of software programs.

Improved and newer software engineering practices: The more informed processes that are used for developing software today will hopefully lead to software that is easier to comprehend in the future. Component-based software systems are designed using familiar design patterns, and other conventions. Future software may have traceability links to requirements, and improved documentation such as formal program specifications. Test driven development methods may also lead to code that is easier to comprehend due to the implicit requirements captured by test cases.

Diverse sources of information: The program comprehension community, until quite recently, mostly focused on how static and dynamic analyses of source code, in conjunction with documentation, could facilitate program comprehension. Modern software integrated development environments, such as the Eclipse Java development environment, also manage other kinds of information such as bug tracking, test cases and version control. This information, combined with human activity information such as emails and instant messages, will be more readily available to support analysis in program comprehension. Domain information should also be more accessible due to model driven development and the semantic web.

6. Future methods, theories and tools

In this section, I consider how research methods, theories and tools may evolve in response to the predicted programmer and program changes.

6.1. Research methods

In recent years there has been a high expectation in the research community that tools should be subject to some sort of evaluation. Case studies of industrial systems are often used as a mechanism for demonstrating that a technique can efficiently and robustly perform the expected analysis. In some cases, we may also need evidence that the approach is useful for and usable by less sophisticated developers.

Many techniques, especially those utilized in presentation tools, are evaluated using experiments. Conducting empirical work and experiments in program comprehension is always a challenging endeavour. There are two main research paradigms, the *quantitative* and the *qualitative* (Cresswell, 1994).

The quantitative approach, the traditional approach in program comprehension, assumes that reality is objective and is independent from the researcher. Quantitative studies are formal and factors are isolated before the study. They are usually performed in context free situations, and the studies are seen as reliable and repeatable.

The qualitative approach, the constructivist approach, assumes that the researcher interacts with what is observed and that context is important. These studies are more informal and hypotheses about the results are formed inductively. The key results from qualitative research are patterns and theories which lead to initial or further understanding.

For the most part, cognitive models and tool evaluations in program comprehension have been determined through quantitative approaches. However, the conditions for program comprehension are so complex and varied, that many researchers are recognizing that qualitative approaches conducted in more ecologically valid settings may be very insightful. This shift to conducting research in industrial settings brings with it many logistical challenges. Observations can be hard to do in industry and may result in large data sets that are difficult to analyze. Observations can also be disruptive and are subject to the Hawthorne effect (e.g. a programmer may change her behaviour because she is observed).

To address these issues, several researchers are also collecting instrumented data after the tool is deployed in an industrial setting. This data collection technique shows much promise as it captures information on the context of tool use as well as accurate information about how a tool is used over a longer period of time. However, such results can also be misleading. A lack of adoption is not enough to indicate that a tool is not useful as there are many barriers to adoption (e.g. seemingly trivial usability issues can impede usage of a tool). There have been ICSE workshops (Balzer, 2003, Balzer, 2004) that have hosted discussion on the adoption of software tools.

As a community, there have been some shared evaluation efforts—the use of exemplar case studies, i.e. benchmarks, (Sim, 2003) and collaborative tool demonstrations (Storey, 2003). These efforts give researchers a mechanism to compare their tools with others, and learn more about different and similar approaches to research.

Irrespective of the evaluation technique used, theoretical underpinnings will benefit the evaluations as the results will be easier to interpret. Although our long term goal may be to build better tools, we need to be able to articulate *why* they are better than other approaches.

6.2. Theories

As the programming workforce and technology changes, learning theories (Exton, 2002) will become more relevant to end-users doing programming-like tasks. Theories are currently being developed to describe the social and organizational aspects of program comprehension (Gutwin, 2004). The impact of open source processes and agile methods have yet to be fully understood. Richer cognitive theories about how aspect oriented programming will impact comprehension in the longer term need to be further developed. More theories about the collaborative nature of program comprehension, both co-located and distributed, are needed.

It is becoming clear to many in this research area that developing theories on program comprehension is an ambitious undertaking. This is mostly due to the large variability in the possible conditions for any experiment. It is important as a community to have many data points; this will enable future researchers to do a meta-analysis of results from several experiments so that common trends and issues can be extracted. This phenomenon can be compared to efforts in the clinical trial community where many studies have to be done to understand how a drug interacts with other drugs, different kinds of individuals and environmental conditions.

In our research community, we need to document and present results in such a way that others can make sense of our data and conclusions. Researchers evaluating presentation tools and user interfaces for program comprehension tools could benefit from the work of Walenstein and Green *et al.:* Walenstein proposes a methodology for evaluating cognitive support (Walenstein, 2003) and Green's Cognitive Dimensions provides a language and framework that can be used for evaluating presentation tools (Green, 1996). More work is needed to understand how we can combine results and benefit from collaborative efforts in empirical work.

6.3. Tools

Faster tool innovations: The use of frameworks as an underlying technology for software tools is leading to faster tool innovations as less time needs to be spent reinventing the wheel. A prime example of how frameworks can improve tool development is the Eclipse framework (see www.eclipse.org). Eclipse was specifically designed with the goal of creating reusable components which would be shared across different tools. The research community benefits from this approach in several ways. Firstly, they are able to spend more time writing new and innovative features as they can reuse the core underlying features offered by Eclipse and its plug-ins; and secondly, researchers can evaluate their prototypes in more ecologically valid ways as they can compare their new features against existing industrial tools.

Plug 'n play tools: In Section 5, I described understanding tools according to three categories: extraction, analysis and presentation. Given a suite of tools that all plug in to the same framework, together with a standard exchange format (such as GXL—http://www.gupro.de/GXL/), researchers will be able to more easily try different combinations of tools to meet their research needs. This should result in increased collaborations and more relevant research results. Such integrations will also lead to improved accessibility to repositories of information related to the software including code, documentation, analysis results, domain information and human activity information. Integrated tools will also lead to fewer disruptions for programmers.

Recommenders and search: Software engineering tools, especially those developed in research, are increasingly leveraging advances in intelligent user interfaces (e.g. tools with some domain or user knowledge). Recommender systems are being proposed to guide navigation in software spaces. Examples of such systems include Mylar (Kersten, 2005) and NavTracks (Singer, 2005). Mylar uses a degree of interest model to filter non-relevant files from the file explorer and other views in Eclipse. NavTracks provides recommendations of which files are related to the currently selected files. Deline *et al.* also discuss a system to improve navigation (Deline, 2005). The FEAT tool suggests using concern graphs (explicitly created by the programmer) to improve navigation efficiency and enhance comprehension (Robillard, 2003). This work can be further inspired by research from the HCI (Human Computer Interactionz) community on tracking interaction histories to inform navigation (Chalmers, 1998, Wexelblat, 1998, Wexelblat, 1999).

Search technologies, such as Google, show much promise at improving search for relevant components, code snippets and related code. The Hipikat tool (Cubranic, 2006) recommends relevant software artifacts based on the developer's current project context and development history. The Prospector system recommends relevant code snippets (Mandelin, 2005). It combines a search engine with the content assist in Eclipse to help programmers use complex APIs. Zimmerman developed the eRose tool which generates recommendations on which files should be changed together (Zimmerman, 2004). eRose uses CVS data to find files that have frequently been changed together. Holmes and Murphy (Holmes, 2005) propose using structures to create code recommendations during evolution tasks.

Although this work in search and recommendations is relatively new, it shows much promise and it is expected to improve navigation in large systems while reducing the barriers to reusing components from large libraries.

Adaptive interfaces: Software tools typically have many features which may be overwhelming not only for novice users, but also for expert users. This information overload could be reduced through the use of adaptive interfaces. The idea is that the user interface can be tailored automatically, i.e. will self-adapt, to suit different kinds of users and tasks. Adaptive user interfaces are now common in Windows applications such as Word. Eclipse has several novice views (such as Gild (Storey, 2003) and Penumbra (Mueller, 2003)) and Visual Studio has the Express configuration for new users. However, neither of these mainstream tools currently have the ability to adapt or even be easily manually adapted to the continuum of novice to expert users. There is related work in the HCI community on adaptive interfaces (Findlater, 2004).

Visualizations have been the subject of much research over the past ten to twenty years. Many visualizations, and in particular graph-based visualizations, have been proposed to support comprehension tasks. Some examples of research tools include Seesoft (Ball, 1996), Bloom (Reiss, 2001), Rigi (Wong, 1995), Landscape views (Penny, 1992), sv3D (Marcus, 2003), and Codecrawler (Lanza, 2001). In our work, we developed the SHriMP tool (Storey, 2003) to provide support for the Integrated Metamodel of comprehension. Our goal was to facilitate navigation between different layers of abstractions, and to help navigation through delocalized plans. Graph visualization is used in many advanced commercial tools such as Klocwork, Imagix4D and Together. UML diagrams are also common place in mainstream development tools.

Challenges with visualizing software include the large amount of items that can be displayed, knowing at what level of abstraction details should be shown, as well as selecting which view to show. More details about the user's task combined with metrics describing the program's characteristics (such as inheritance depth) will improve how visualizations are currently presented to the user. A recommender system could suggest relevant views as a starting point. Bull proposes the notion of *model driven visualization* (Bull, 2005). He suggests creating a mechanism for tool designers and expert users that recommends useful views based on characteristics of the model and data.

Collaborative support: As software teams increase in size and become more distributed, collaborative tools to support distributed software development activities are more crucial. In research, there are several collaborative software engineering tools being developed such as Jazz and Augur (Hupfer, 2004, Froehlich, 2004). A review of how these and other tools mine human activities to improve comprehension through visualization is given in (Storey, 2005). There are also some collaborative software engineering tools deployed in industry, such as CollabNet, but they tend to have simple tool features to support communication and collaboration, such as version control, email and instant messaging. Current industrial tools lack more advanced collaboration features such as shared editors and awareness visualizations.

Although collaborative tools for software engineering have been a research topic for several years, there has been a lack of adoption of many of the approaches such as shared editors in industry and a lack of empirical work on the benefits of these tools. Another area for research that may prove useful is the use of large screen displays to support co-located comprehension. O'Reilly *et al.* (O'Reilly, 2005) propose a war room command console to share visualizations for team coordination. There are other research ideas in the CSCW (computer supported collaborative work) field that could be applied to program comprehension.

Domain and pedagogical support: The need to support domain experts that lack formal computer science training will necessarily result in more domain-specific languages and tools. Non-experts will also need more cognitive scaffolding to help them learn new tools and domain-specific languages (van Deursen, 2000) more rapidly. Tools to support the maintenance of programs written in domain specific languages will also be required. Pedagogical support, such as providing examples by analogy, will likely be an integral part of future software tools. The work discussed above on recommending code examples is also suggested at helping novices and software immigrants (i.e. programmers new to a project). Results from the empirical work also suggest that there is a need for tools to help programmers learn a new language. Technologies such as TXL (Cordy, 2002) can play a role in helping a user see examples of how code constructs in one language would appear in a another language.

Large screen and multiple displays: Modern tools should be designed to take advantage of the now ubiquitous use of multiple monitors and large screens. Little is understood about how these multiple monitors impact comprehension and how tool support could be improved to match such display configurations. Larger screens could also improve pair programming and the comprehension strategies used during agile development.

6.4. Discussion: *Back to the future*

It is an interesting exercise to travel back and look at Tilley and Smith's paper from 1996 on "Coming Attractions in Program Understanding" (Tilley, 1996). Some of the predictions they had then for technologies which would be available within five years did mature as they predicted. They suggested that mature technologies would be leveraged, which is now the case with mature off-the-shelf technologies such as Windows components and the Eclipse framework. They also predicted that web interfaces and hypertext would play a bigger role. Many modern tools now use web interfaces for navigating software resources. Tailorable user interfaces are now common place, as are more advanced pattern matching facilities.

Some of their predictions, however, are not fully realized and indeed overlap the predictions in this paper. These include: computer support for collaborative understanding; access to alternative sources of data through natural language processing; data filters to allow the programmer to focus on relevant information; conceptual and domain modeling techniques; use of intelligent agents in a "maintainer's handbook" and more advanced visual interfaces. These suggestions are still under development but today seem within closer reach. Whether these themes will reappear in another ‘prophecy’ paper, written ten years from now, only time will tell.

7. Conclusions

As a community, we can be proud of our achievements over the past thirty years. As the landscape of comprehension research evolves, the future looks bright. We can anticipate that advances in program comprehension will increase, in part due to recent enabling technologies, such as sophisticated frameworks to support the more rapid construction and integration of tools, and advanced technologies such as context-aware and history-aware search and intelligent user interfaces. In parallel to these tool advances, there are now more researchers from both within computer science and from other disciplines that are interested in understanding more about the cognitive and social aspects of program comprehension. These researchers are now equipped with more appropriate research methods that can help to reveal more understanding about program comprehension needs and how these needs may be met through tool and process support. As the field of software engineering matures and the possibilities for more advanced and pervasive software increase, the field of program comprehension promises to be an exciting area for future research.

Acknowledgements My thanks to Janice Singer, Hausi Muller, Robert Lintern, Adam Murray, and Jim Cordy for ideas, feedback and encouragement while writing this paper. The anonymous reviewers also provided very valuable and detailed comments that improved this paper.

References

Aho A.V., Sethi R., and Ullman J.D. 2000. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Ball T. and Eick S.G. 1996. Software visualization in the large. *IEEE Computer* 29(4): 33–43.

Balzer R., Jahnke J.H., Litoiu M., Müller H.A., Smith D.B., Storey M.-A.D., Tilley S.R. and Wong K. 2003. 3rd International Workshop on Adoption-centric Software Engineering (ACSE 2003). In *Proceedings of the 25th International Conference on Software Engineering*, pp. 789–790. Portland, Oregon, USA.

Balzer R., Litoiu M., Müller H.A., Smith D.B., Storey M.-A.D., Tilley S.R. and Wong K. 2004. 4th International Workshop on Adoption-Centric Software Engineering (ACSE 2004). In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pp. 748–774. Edinburgh, United Kingdom.

Bellay B. and Gall H. 1998. An evaluation of reverse engineering tool capabilities. *Journal of Software Maintenance*, 10(5): 305–331.

Beyer H. and Holtzblatt K. 1997. Contextual Design : A Customer-Centered Approach to Systems Designs. Morgan Kaufman Publishers, ISBN: 1558604111.

Brooks F.P. 1987. No Silver Bullet: Essence and accidents of software engineering. *Computer*, 20(4): 10–19.

Brooks R. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, 543–554.

Biggerstaff T.J., Mitbander B.W. and Webster D. 1993. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pp. 482–498.

Bull R.I. and Storey M.-A. 2005. Towards visualization support for the eclipse modeling framework, A Research-Industry Technology Exchange at EclipseCon, California.

Burkhardt J., Détienne F., and Wiedenbeck S. 1998. The effect of object-oriented programming expertise in several dimensions of comprehension strategies. 6th *International Workshop on Program Comprehension*, pp. 24–26. Ischia, Italy.

Chalmers M., Rodden K., and Brodbeck D. 1998. The order of things: Activity-centred information access,. In *Proceedings of 7th Intl. Conf. on the World Wide Web (WWW7)*. Brisbane, Australia.

Clayton R., Rugaber, S., and Wills L. 1998. On the knowledge required to understand a program. *The Fifth IEEE Working Conference on Reverse Engineering*, Honolulu, Hawaii.

Cordy J.R., Dean T.R., Malton A.J. and Schneider K.A. 2002. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13): 827–837.

Corritore C.L. and Wiedenbeck S. 1999. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, 50(1): 61–83.

Creswell J.W. 1994. *Research Design, Qualitative and Quantitative Approaches*, SAGE Publications.

Cubranic D., Murphy G.C., Singer J., and Booth K.S. 2006. Hipikat: A project memory for software development. IEEE Transactions on Software Engineering, to appear in the special issue on mining software repositories.

Curtis, B. 1981. Substantiating programmer variability. In *Proceedings of the IEEE*, pp. 846–846, vol. 69, Issue: 7, July.

Curtis B. 1986. By the way, did anyone study any real programmers? *Empirical Studies of Programmers*, 256–262.

Davies S.P. 1993.Externalising information during coding activities: effects of expertise, environment and task. In C.R. Cook, J.C. Scholtz, and J.C. Spohrer (eds.), *Empirical Studies of Programmers: Fifth Workshop*, pp. 42–61. Norwood, NJ: Ablex Publishing.

Dean T.R., Malton A.J., and Holt R.C. 2001. Union Schemas as a Basis for a C++ Extractor, WCRE 2001—8th Working Conference on Reverse Engineering, Stuttgart, Germany, pp. 59–67.

DeLine R., Khella A., Czerwinski M. and Robertson G. 2005. Towards understanding programs through wear-based filtering. In *Proceedings of Softvis*, pp. 183–192. Saint Louis, Missouri, USA.

Détienne F. 2001. *Software Design—Cognitive Aspects*. Springer Practitioner Series.

Eisenbarth T., Koschke R. and Simon D. 2001. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 602–611. Florence Italy.

Eisenbarth T., Koschke R. and Simon D. 2003. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3): 195–209.

Erdös K. and Sneed H.M. 1998. Partial comprehension of complex programs (enough to perform maintenance). In *Proceedings of the 6th International Workshop on Program Comprehension*, pp. 98–105. Ischia, Italy.

Exton C. 2002. Constructivism and program comprehension strategies. *10th International Workshop on Program Comprehension*, pp. 281–284. Paris, France.

Ferenc R., Beszédes A. and Gyimóthy T. 2004. Fact extraction and code auditing with columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance*, pp. 60–69. Chicago Illinois, USA.

Findlater, L., McGrenere, J. 2004. A comparison of static, adaptive, and adaptable menus. In *Proceedings of ACM CHI 2004*, pp. 89–96.

Francel, M.A. and Rugaber R. 1999. The relationship of slicing and debugging to program understanding. In *Proceedings of the International Workshop on Program Comprehension*, pp. 106–113. Pittsburgh, Pennsylvania.

Froehlich J. and Dourish P. 2004. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the 26th International Conference on Software Engineering*, pp. 387–396. Edinburgh, Scotland.

German D.M. 2006. Decentralized open source global software development, the GNOME experience. *Journal of Software Process: Improvement and Practice*, 8(4): 201–215.

Green T.R.G. and Petre M. 1996. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2): 131–174.

Grudin, J. 2001. Partitioning digital worlds: focal and peripheral awareness in multiple monitor use. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, United States). CHI '01, pp. 458–465. ACM Press, New York, NY.

Gutwin C., Penner R., and Schneider K. 2004. Group Awareness in Distributed Software Development. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pp. 72–81. Chicago.

Hammond T., Hannay T., Lund B., and Scott J. 2005. Social Bookmarking Tools (I): A General Review. D-Lib Magazine, Volume 11, Number 4, ISSN 1082–9873.

Hamou-Lhadj A. and Lethbridge T. 2004. A Survey of Trace Exploration Tools and Techniques. In *Proceedings of the 14th Annual IBM Centers for Advanced Studies Conferences (CASCON)*, pp.42–51. IBM Press, Toronto, Canada.

Hannemann J. and Kiczales G. 2001. Overcoming the Prevalent Decomposition in Legacy Code. ASOC Workshop at ICSE 2001, Toronto, Canada.

Hohmann L. 1996. Journey of the Software Professional: The Sociology of Software Development. Prentice Hall.

Holmes R.T. and Murphy G.C. 2005. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering*, pp. 117–125. St. Louis, Missouri.

Hupfer S., Cheng L.-T., Ross S. and Patterson J. 2004. Introducing collaboration into an application development environment. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pp. 444–454. Chicago.

Kersten M. and Murphy G. 2005. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the International Conference on Aspect Oriented Software Development*, pp 159–168. Chicago, IL.

Koschke, R. and Eisenbarth T. 2000. A Framework for Experimental Evaluation of Clustering Techniques. In *Proceedings of the 8th international Workshop on Program Comprehension*, pp. 201–210. Limerick, Ireland.

Laitenberger O. 1995. Perspective-based Reading: Technique, Validation and Research in Future. Technical Report, University of Kaiserslautern, Germany, ISERN-95-01.

Lanza M. and Ducasse S. 2001. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of ACM OOPSLA*, pp. 300–311. Tampa, Florida.

Letovsky S. 1986. Cognitive processes in program comprehension, In *Proceedings of Empirical Studies of Programmers*, pp. 58–79.

Littman D.C., Pinto J. Letovsky S., and Soloway E. 1986. “Mental models and software maintenance”, In Empirical Studies of Programmers, Washington, DC, pp. 80–98.

Mandelin D., Xu L., Bodik R. and Kimelman D. 2005. Mining Jungloids: Helping to Navigate the API Jungle. In *Proceedings of PLDI*, pp. 48–61. Chicago, IL.

Marcus A., Feng L., and Maletic J.I. 2003. Comprehension of Software Analysis Data Using 3D Visualization. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pp. 105–114. Portland, USA.

Mathias K.S., Cross J.H., Hendrix, T.D., and Barowski, L.A. 1999. The role of software measures and metrics in studies of program comprehension. In *Proceedings of the 37th Annual Southeast Regional Conference*, ACM-SE 37.

Mockus A., Fielding R., and Herbsleb J.D. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology, 11(3): 309–346.

Moonen L. 2001. Generating Robust Parsers using Island Grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pp. 13–24. Stuttgart, Germany.

Mueller F. and Hosking A.L. 2003. Penumbra: an Eclipse plugin for introductory programming. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology Exchange*, pp. 65–68. Anaheim, California.

Muller H.A. and Klashinsky K. 1988. Rigi: A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pp. 80–86. Singapore.

Murphy G.C., Notkin D. and Sullivan K. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *Proceedings of Foundations of Software Engineering*, pp. 18–28. Washington, DC.

NATO 1968 Software Engineering Conference, Garmisch, Germany.

O’Reilly C., Bustard D. and Morrow P. 2005. The War Room Command Console – Shared Visualizations for Inclusive Team Coordination. In *Proceedings of Softvis*, pp. 57–65. Saint Louis, Missouri, USA.

Pacione M.J., Roper M., and Wood M. 2004. A novel software visualisation model to support software comprehension. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 70–79. Delft.

Penny D.A. 1992. The Software Landscape: A Visual Formalism for Programming-in-the-Large, PhD thesis, University of Toronto.

Pennington N. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology, 19, pp. 295–341.

Petre M., Blackwell A.F. and Green T.R.G. 1998. Cognitive Questions in Software Visualization. In Software Visualization: Programming as a Multi-Media Experience. MIT Press, pp. 453–480.

Prechelt L., Unger-Lamprecht B., Philippsen M., & Tichy W. 2002. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance . IEEE Trans. Software Enginering 28(6): 595–606.

Reiss S.P. 2001. An overview of BLOOM. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 2–5.

Rich C, and Wills L.M. 1990. Recognizing a Program’s Design: A Graph-Parsing Approach. IEEE Software, vol. 07, no. 1, pp. 82–89, Jan/Feb.

Robbins J.E. and Redmiles D.F. 1996. Software Architecture Design From the Perspective of Human Cognitive Needs. In *Proceedings of the California Software Symposium, Los Angeles*, California.

Robillard M.P. and Murphy G. 2003. FEAT: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, pp. 822–823.

Shneiderman B. and Mayer R. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 8(3): 219–238.

Sim S.E., Holt R.C., and Easterbrook S. 2002. On Using a Benchmark to Evaluate C++ Extractors. In *Proceedings of the Tenth International Workshop on Program Comprehension*, pp. 114–123. Paris, France.

Sim S.E., Easterbrook S., and. Holt R.C. 2003. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *Proceedings of the 25th International Conference on Software Engineering*, pp. 74–83. Portland, Oregon.

Singer J., Lethbridge T., Vinson N. and Anquetil N. 1997. An Examination of Software Engineering Work Practices. In *Proceedings of CASCON '97*, pp. 209–223. Toronto.

Singer J., Elves R. and Storey M.-A. 2005. NavTracks: Supporting Navigation in Software Maintenance. In *Proceedings of International Conference on Software Maintenance*, pp. 325–334. Budapest, Hungary.

Soloway E., and Ehrlich K. 1984. Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, SE-10(5): 595–609.

Storey M.-A., Fracchia F.D. and Müller H.A. 1999. Cognitive Design Elements to support the Construction of a Mental Model During Software Exploration. Journal of Software Systems, special issue on Program Comprehension, 44, pp. 171–185.

Storey M.-A., Wong K. and Müller H.A. 2000. How do program understanding tools affect how programmers understand programs. Science of Computer Programming, 36 (2–3): 183–207.

Storey M.-A. 2003. Designing a Software Exploration Tool Using a Cognitive Framework of Design Elements. Software Visualization, Guest editor: Kang Zhang. Kluwer.

Storey M.-A., Sim S.E. and.Wong K. 2003. A Collaborative Demonstration of Reverse Engineering Tools. ACM Applied Computing Review, pp. 18–25.

Storey M.-A., Michaud J., Mindel M., Sanseverino M., Damian D., Myers D., German D. and Hargreaves E. 2003. Improving the Usability of Eclipse for Novice Programmers. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology Exchange*, pp. 35–39. Anaheim, California.

Storey M.-A., Cubranic D., and German D. 2005. On the use of visualization to support awareness of human activities in software development: A survey and a framewo Proceedings of Softvis, St. Louis, Missouri, pp. 193–202.

Systä T., Koskimies K., and Müller H.A. 2001. Shimba—An Environment for Reverse Engineering Java Software Systems. Software Practice & Experience, 31(4): 371–394.

Tilley S.R. and Smith D.B. 1996. Coming Attractions in Program Understanding. Technical Report CMU/SEI-96-TR-019.

Tip F. 1995. A survey of program slicing techniques, Journal of Programming Languages 3(3) , pp. 121–189.

Tonella P. and Ceccato M. 2004. Aspect Mining through the Formal Concept Analysis of Execution Traces. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 112–121. Delft, the Netherlands.

van Deursen A., Klint P., and Visser J. 2000. Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices, 35(6):26–36.

van Deursen A., Marin M., and Moonen L. 2003. Aspect Mining and Refactoring. First Intl. Workshop on REFactoring: Achievements, Challenges, Effects (REFACE).

Vessey I. 1985 Expertise in debugging computer programs: A process analysis. International Journal of Man-Machine Studies, 23, pp. 459–494.

von Mayrhauser A. and Vans A.M. 1993. From code understanding needs to reverse engineering tool capabilities. In *Proceedings of CASE'93*, pp. 230–239.

Walenstein A. 2003. Observing and Measuring Cognitive Support: Steps Toward Systematic Tool Evaluation and Engineering. In *Proceedings of 11th Intl. Workshop on Program Comprehension*, pp. 185–195. Portland, USA.

Weiser M. 1982. Programmers Use Slices When Debugging. CACM, 26, pp. 446–452.

Wexelblat A. 1998. Communities through time: Using history for Social Navigation. In Lecture Notes in Computer Science, vol. 1519, T. Ishida, Ed. Berlin: Spring Verlag, pp. 281–298.

Wexelblat A. and Maes P. 1999. Footprints: History-rich tools for information foraging. In *Proceedings of CHI, Pittsburgh, PA*.

Wong K., Tilley S.R., Muller H.A., and Storey M.-A. 1995. Structural redocumentation: A case study. IEEE Software, 12(1): 46–54.

Wong K. 2000. The Reverse Engineering Notebook. Ph.D. Thesis, University of Victoria.

Zimmermann T., WeiBgerber P., Diehl S., and Zeller A. 2004. Mining version histories to guide software changes. In *Proceedings of International Conference in Software Engineering*, Glasgow, Scotland.

Dr. Margaret-Anne Storey is an associate professor of computer science at the University of Victoria, a Visiting Scientist at the IBM Centre for Advanced Studies in Toronto and a Canada Research Chair in Human Computer Interaction for Software Engineering. Her research passion is to understand how technology can help people explore, understand and share complex information and knowledge. She applies and evaluates techniques from knowledge engineering and visual interface design to applications such as reverse engineering of legacy software, medical ontology development, digital image management and learning in web-based environments. She is also an educator and enjoys the challenges of teaching programming to novice programmers.