First Steps in Programming: A Rationale for Attention Investment Models

Alan F. Blackwell
University of Cambridge Computer Laboratory
Alan.Blackwell@cl.cam.ac.uk

Abstract

Research into the cognitive aspects of programming originated in the study of professional programmers (whether experts or students). Even "end-user" programmers in previous studies have often worked in organizations where programming is recognized to be demanding professional work – the term "power-user" recognizes this technical kudos. But as personal computers become widespread, and most new domestic appliances incorporate microprocessors, many people are engaging in programming-like activities in domestic or non-professional contexts. Such users often have less motivation and more obstacles to programming, meaning that they may be unlikely even to take the first steps. This paper analyses the generic nature of those first steps, and identifies the cognitive demands that characterize them. On the basis of this analysis we propose the Attention Investment model, a cognitive model of programming that offers a consistent account of all programming behaviour, from professionals to end-users.

1. What is Programming?

Goodell's excellent website devoted to end user programming [10] offers definitions of "end user" and "end user programming", but not of "programming". This may not seem a serious omission, as most researchers in computer science use implicit definitions that seem quite adequate for their professional work. Nevertheless, this paper considers the possibility that challenging the implicit professional definitions of programming may generate important insights for the study of end user programming.

Programming is in fact seldom defined in modern research publications. An earlier programming textbook from 1959 gives a typical formulation for that time: "This sequence [of basic operations] is called the program and the process of preparing it is called programming" [32, p. 4]. ? Programming is the "spadework" of finding a precise mathematical formulation and method of solution, possibly notated in a "convenient problem-oriented language" whose symbols are "more closely related to the mathematical problem to be solved".

The major changes since this was written are a) that many computer users do not now consider themselves programmers (when Weinberg wrote his early monograph "The Psychology of Computer Programming" [30], it was assumed that serious users of computers would be programmers), and b) that most programming deals with problems that we would not now consider to be mathematical.

As programming applications have moved away from the mathematical domain typical of early computing, the nature of the basic operations, the symbols in programming languages, and the formulations or methods of solution have all evolved. An introductory chapter to the book "Psychology of Programming" [17] notes that the programming has changed from "describing calculations" to "defining functions", and then to "defining and treating objects". Several contributors to that book broadly describe the cognitive challenges of programming. Examples include "Programming is a human activity that is a great challenge" (p. 3), or "Programming is an exceedingly diverse activity" (p. 21). A section entitled "what is programming?" concludes that "The crucial dimensions in the activity of programming are processing and representation" (p. 160). But this final definition could refer to almost any human cognitive skill – it no longer provides a basis for investigating the distinctive problems of programming.

This paper asks instead what is distinctive about the cognitive tasks involved in programming, and in particular which distinctive cognitive tasks are shared between all programmers, whether professional or end-users. This is particularly challenging in the case of systems that are described as "programming" by their users ("programming" HTML, "programming" a VCR or a microwave oven), but do not appear to meet the criteria that would make them suitable as a professional programming language.

2. Three definitional questions

2.1. Who is a programmer?

The lines of professional demarcation within the software development community have always been fluid as a result of changing programming tools. For example, the distinction between "analysts" and "programmers" blurred when 4GLs enabled programming at a level of abstraction that was comparable to the vocabulary of the analyst. Analysts thus became analyst/programmers in the 1980s, and were simply called programmers again by the 90s. The same trends occurred in other specialist jobs. Unit test engineers were initially programmers (who wrote test harnesses), then simple operators of regression test tools, then programmers again as the testing tools became programmable.

Recent trends have been to increase the number of people who might do programming in the course of their work. Almost all major software applications include scripting or macro languages of some sort, usable to configure and customize the behaviour of the application. Most operating systems include scripting languages. One of the most widely used classes of software application, the spreadsheet, is itself a declarative programming language. Many people who are not professional programmers use spreadsheets to create large and complex applications, thus inheriting all the software engineering problems of specification, design, testing and maintenance.

End-user development, end-user customization and end-user software engineering have all been proposed as terms expressing the challenges faced by users encountering these new tools. Some of those terms apparently deemphasise the sophistication of the programming required ("customization"), while others emphasise the fact that large and complex design projects are difficult whatever the tools used ("engineering").

Some of these differing emphases in terminology can seem more suited to software product marketing, rather than seriously contending that "customization" means no programming is involved. If a conventional programming language were used to carry out the same tasks, there would certainly be no doubt that these applications were conventional pieces of programming work. But the one aspect in which end-user programming tasks always differ from conventional programming tasks is precisely that the software tools used for development or customization have the potential to be so unlike conventional programming languages.

Which of the following can unambiguously be categorised with respect to the boundary between programming languages and other forms of software: Scripting languages? Spreadsheets? Macro languages? Keyboard macros? Configuration files? Java programs? Javascript programs? Server Side Include macros? Cascading Style Sheets? HTML pages? Microsoft Word documents? From the perspective of a non-programmer or end-user, the distinctions between these technologies are not at all clear-cut. All of them are able to produce dynamically modified text documents, and several can potentially be used to create apparently identical results. Yet some of them are classified in professional contexts as being programming languages, and some are not, with the result that when carried out by an end user, some of them may be classified as end-user programming and some not, even though the user may feel that he or she is doing a single task in different ways.

This ambiguity becomes acute when the programming is taking place in a context that is completely separate from the workplace. An end-user programmer at work is quite likely to realise that the things he or she are doing could have been achieved by a professional programmer working within the organization. Previous academic studies have emphasized the roles these end users play on the boundaries of professional programming activity within an organization [21]. The term "power-user" acknowledges the fact that these people have valuable technical skills.

In contrast, a domestic programmer is not usually described as a "power-user". There may still be good reasons why somebody would create a Word macro to save time on a lengthy task at home. We have proposed elsewhere a system suitable for enhancing domestic remote controls with programming abilities [4]. And a person learning to program a VCR certainly does not qualify for the description "power-user", even if he or she does meet the criterion for an "end-user".

In order to avoid these inconsistencies, the first proposal of this paper is that all computer users ought to be regarded as potential programmers, whose tools differ only in their usability for that purpose. The social approval accorded to such skills may increase with time, but this is not a fundamental indicator of inherent cognitive challenge in the task. If it is possible to find interesting programming-like attributes in other kinds of computer use, programming research could be universal and inclusive in its scope, rather than restricted to the experience of "professional" programmers. The next section therefore considers a classification of programming languages that takes account of these user perspectives, rather than conventional definitions.

2.2. What is a programming language?

What aspect of programming languages makes them different to other kinds of computer usage? Consider some of the examples presented above. A web page generated by a Javascript script or Server Side Include macro, when seen in a browser, may appear indistinguishable from a web page written directly in HTML. The difference resulting from the script or macro is that a different viewer, or the same viewer at another place or time, will see a different web page. The author writing the page specifies these differences by adding control information (in the scripting or macro language) to be interpreted by the computer rather than by the viewer.

These simple variations might be seen as conflicting with some ideals of modern design for usability. What the user sees when authoring the page is not necessarily what he or she gets when viewing it – a conflict with the ideal of WYSIWYG. What he or she is manipulating is not a concrete instance of the desired result, but an abstract notation defining required behavior in different circumstances – a conflict with the ideal of direct manipulation. Of course these departures from the "ideal" are not a bad thing – they are necessary in order to achieve the task. But the additional challenges to the user are typical of the challenges that distinguish programming activities from those activities that do allow direct manipulation and WYSIWYG.

When we consider other examples given above, similar properties are apparent. The distinction between writing an HTML document and a Word document is that the HTML document may look different to different viewers (depending on the size and shape of the browser window, the browser version, platform, available fonts etc). A single decision by the author can thus have multiple consequences – different results each time the document is viewed in a different situation. Once again, this range of effects is produced in HTML by the use of an abstract notation to define required behavior in different circumstances (here, the markup language). As with the use of JavaScript, even the abstractions of HTML provide the opportunity for syntax errors, runtime errors, or bugs in the form of unintended or exceptional behaviors.

The same is true even of a keyboard macro. Pressing a key when composing a regular document is a fairly direct manipulation – the character that was written on the key appears on the screen, and can be viewed and retained or deleted in a direct feedback process. But pressing a key when composing a keyboard macro has other effects beyond those that are directly visible. When the macro is executed again in a new context, the results will be different. The user must anticipate this, and use abstract commands rather than direct manipulation commands (e.g. using the "end of line" key rather than pressing the right arrow key until the cursor reaches the end of the line, which would fail with a bug the first time it was executed in a line with a different number of characters).

All of these examples, although rather trivial by comparison to the challenges of large software projects, do share important characteristics of conventional programming. The user must:

- Decide the intended result of executing the program (requirements);
- Identify when it will be executed, and allow for variation in different circumstances (specification);
- Choose from a set of technical features that may support this behaviour (design);
- Enter abstract control commands as well as data (coding) and anticipate; and
- Account for departures from the intended behavior (debugging).

All of these things are intellectually challenging, and they increasingly arise in all aspects of computer use. Consider, for example, the definition of a document template, or even a paragraph style in a word processor. Even quite mundane user tasks can involve requirements gathering, specification, design, coding and debugging.

In order to account for these experiences, the second proposal of this paper is that almost all major software applications could now be recognized as including programming languages. If this were the case, research into programming could focus on programming experiences independent of language, especially those which result when abstract notation replaces direct manipulation.

2.3. What is programming activity?

The word programming is often used in common speech to describe activities that might seem trivial by comparison to large-scale software application development. People do not in general say they are "programming" their Word document when defining a paragraph style. But many people do say (even on their resumes, I have found), that they have been "programming" in HTML. Furthermore, people say they are "programming" their VCR, their microwave oven, their car radio or their boiler controls.

Is there any value in extending our attention to these common uses of the term when we do research into the cognitive demands of programming? If we consider the user experience of marginal programming technologies, as addressed in the previous section, we see that even these mundane activities share many of the same properties. They all have the basic character that the user is not directly manipulating observable things, but specifying behaviour to occur at some future time.

In order to address these experiences, the third proposal of this paper is that when people say they are programming, we should not question whether this activity is genuine programming, but instead analyse their experience in order to understand the general nature of programming activity.

3. Cognitive features of programming

What are the cognitive implications of this broader domain of programming tasks? The common features of the various programming tools described so far are a) loss of the benefits of direct manipulation and b) introduction of notational elements to represent abstraction. It is possible to relate both of these to relevant topics in cognitive science.

3.1. Loss of direct manipulation

The cognitive benefits of direct manipulation arise partly from the fact that image-based representations mitigate the "frame problem" in cognitive science [18]. If a planning agent maintains a mental representation of the situation in which it acts, the process of planning relies on the agent being able to simulate updates to the situation model, and thereby anticipate the effects of potential courses of action.

Such planning is only possible if the scope of effects of a given action can be constrained. In other words there must be a defined boundary beyond which the action will not have further effects. If there is no basis for setting such a boundary, any action may potentially have infinite consequences, and it will not be possible to place bounds on the planning algorithm. This is known as the frame problem.

In direct manipulation systems, many constraints on causality are made directly available via the user's perception of the apparently physical situation. This is less true of linguistic representations, where there is no limit on the abstract expressive power of the representation system [29], and hence no boundary that can be exploited to constrain reasoning during planning.

These considerations lead to the well-known cognitive benefits of direct manipulation [27]. In a direct manipulation system, the current status of the system should be continuously represented to the user, a single action should have a single visible effect in the representation, and restoring the state of the representation to that before the action should restore the situation.

In programming systems, none of these things is necessarily true. The situation in which the program is to be executed may not be available for inspection, because it may be in the future, or because the program may be applied to a greater range of data situations than are currently visible to the programmer. Where acting on a single situation is concrete (actions have visible effects in the current situation), programming is abstract (effects may occur in many different situations, not currently visible). Multiple effects of an action will be distributed either in space, in time or in both (if two events occur in the same place at the same time, they are the same event). In previous publications, we have described these fundamental non-direct manipulations of programming as "abstraction over time" and "abstraction over a class of situations" [2,3].

3.2. Use of notation

The second universal characteristic of programming situations is that the program is represented using some notation. This is also a universal characteristic of abstract thought. According to one perspective in the philosophy of mind, concrete action is that in which there is a causal relation between the action and a perceivable state of the world [20]. Abstraction results from forming some representation of the state of the world – either a mental representation, a linguistic representation or some other representational system. The correspondence between a representation and the state of the world is one of convention, not of causality. (This is true even in the case of pictorial representations, which differ in their information content rather than in any fundamental kind of resemblance [11]).

Is there any kind of programming that does not use notation? It would be possible to do programming using representations not usually regarded as notational (e.g. by speaking to a computer, or drawing a picture of the required situation in the world), but these alternatives can be regarded for our purpose as impoverished notational systems. The cognitive benefits of various notational options have been analysed at length by Green with various collaborators in the Cognitive Dimensions of Notations framework [14,15]. It is not necessary to review those analyses in any depth here, other than to note the main conclusions – that notational systems are designed rather than being prescribed by any necessary constraints, and that the design choices made are subject to tradeoffs between factors that will facilitate some kinds of cognitive task while inhibiting others. There is thus no ideal notation for any programming situation, only designs that are more or less well suited to the activities of the people doing the programming.

3.3. Abstraction as a tool for complexity

Tools for processing abstractions provide a further benefit beyond those of defining actions in the future, or in multiple situations where the actor need not be present. Conceptual abstractions can also be defined and combined in order to manage complexity. This results from a further confluence of the two primary characteristics of abstract action, indirect effects and notation use.

In a simple programming activity such as programming a VCR, the user is defining some abstract behaviour which is not directly observable because it will take place in the future. This is done with the assistance of a simple notation – perhaps a display of start time and channel identification. But the user manipulates this notation directly – there is no higher order mechanism by which the user can specify changes to the programme other than those defined directly with the VCR controls.

In contrast to this very simple programming situation, more complex situations can be approached by defining changes to the notation itself, so that the user can extend the vocabulary with which he or she will then express required behaviour. An example of this in a domestic context is a sophisticated boiler control (such as those common in Central Europe) in which the user can define one or more modes of operation, then specify that a particular mode should operate at a particular time of day. This makes programming itself more efficient by allowing the user to refer to a new abstraction (the mode) rather than repeating all the notational elements defining time and temperature for every occasion on which that mode of operation is required.

Abstraction use in which the user conceptualizes common features of complex behaviour, then formulates notational abstractions in which to express them, completes the range of generic programming behaviours for which we propose a common description of cognitive challenges. In the domain of professional programming, this type of abstraction use is still a very active topic of research.

One way of answering the question "what is programming" from a computer science context (proposed by Tony Hoare [16]), is that programming is the process of describing a situation, then refactoring that description in accordance with a set of computational formalisms. The process of refactoring is itself critical to the professional design of software systems and to the refinement of designs in recent system development methodologies such as aspect-oriented programming [8].

4. A cognitive model for abstraction use

We have been working on a theoretical model called "Attention Investment" that accounts for the cognitive challenges arising from these essential features of programming activity: loss of direct manipulation, notation use, and development of abstractions. We intend this model to be sufficiently well-defined that it can be implemented in a cognitive simulation (thus providing a degree of scientific rigor), while also being sufficiently close to subjective user experience that it can be used as a design guide by people developing new programming tools.

This is not an easy combination to achieve. There are other cognitive models that can be used to provide fine-grained descriptions of relatively well defined HCI tasks (GOMS [6] is an example). It is straightforward to use such models for small tasks, but very difficult to model tasks that involve complex, unconstrained user interfaces and many possible solution styles (as is the case in programming).

There are also models for usability evaluation that provide design advice at a heuristic level, without attempting to derive this advice from a model of low-level cognitive phenomena. Heuristic Evaluation [22] and Cognitive Dimensions of Notations [13,15] are two such methods. The Cognitive Dimensions framework is designed specifically with this intention, described as avoiding "death by detail", and offering designers a "broad-brush" description of relevant usability characteristics.

Attention Investment is not a perfect cognitive model, nor a design method that can be used in isolation without support from other methods. However it does offer broad-brush advice relevant to designers of end user programming systems, and is also sufficiently rigorous that it can be verified (for small tasks) by implementation of a cognitive simulation.

A case study in which Attention Investment has been used by a group of designers when building a new end user programming system is described in another paper at this conference [2]. That topic is not addressed any further here. The rest of this paper describes the cognitive model itself, gives an example of a small task that has been simulated using the model, and describes the relationship of the model to the kinds of programming activities characterized above.

4.1. Relationship to previous work

The ideal that usability theories should both be verifiable through cognitive simulation and provide qualitative guidance suited to use by designers has long been an ideal for HCI research. The recent research program most well known as exemplifying this approach is Pirolli and Card's decision theoretic Information Foraging theory [24]. Pirolli and Card have been highly successful in demonstrating that a relatively simple model can account for quantitative observations of human behavior, and can also be used in a predictive capacity for design. Of course Pirolli and Card's theory, although similar in approach to Attention Investment, deals with information search, rather than abstractions or programming.

The Attention Investment model is a decision-theoretic account of programming behavior. It offers a cost/benefit analysis of abstraction use that allows us to predict the circumstances in which users will choose to engage in programming activities, as well as helping tool designers to facilitate users' investment decisions and reduce the risks associated with those decisions. As with any decision theoretic account, this depends on the availability of some currency - a measure according to which cost, risk, pay-off etc can be calculated and compared.

Earlier papers on attention investment (Green & Blackwell [5,14] were influenced by discussion of the "attention economy" which argued that the scarcest economic resource on the Internet is human attention [9,25]. Since then the theory has been more influenced by models of attentional mechanisms in the cognitive science literature, especially those applied to HCI questions [23,12].

4.2. Cost in attentional units

For the purposes of our model, it is necessary to refine the concept of attention as it applies to programming behavior, in distinction to browsing or other computer activities. The effort invested in programming can be described as a nominal amount of "concentration", involving an integral of attentional effort over time. Creating a program requires some amount of concentration - an investment that has a cost in attention units. The payoff if the program works correctly is that it will automate some task in the future, thereby saving attentional cost (the user does not need to concentrate on a task that has been successfully automated). There is, however, a risk that the investment will not pay off (perhaps because there are bugs in the program). The decision to write a program can therefore be framed as an investment equation, in which the expected payoff is compared to the investment and risk.

Attention units provide the basis for modeling both micro (unconscious) and macro (conscious) decisions a user might make in attempting to minimize attention costs over longer timeframes. To summarize these descriptive factors at a qualitative level: Many programming activities promise, through automation, to save attentional effort in the future [3]. The irony of this abstract approach is that the activity of programming may involve more effort than the manual operation being automated [14]. Most decisions to start programming activities are based on an implicit cost-benefit analysis [1]. The variables involved in this cost-benefit analysis are:

Cost: attention units to get the work done. (Presumably the activity also has monetary costs, such as purchase of software, but this is external to the model.)

Investment: attention units expended toward a potential reward, where the reward can either be external to the model (such as payment for services) or an attention investment *pay-off*.

Pay-off: reduced future cost, also measured in attention units, that will result from the way the user has chosen to spend attention.

Risk: Probability that no pay-off will result, or even that additional future costs will be incurred from the way the user has chosen to spend attention.

In sophisticated decision-theoretic models, it is also necessary to account for the cost involved in making the decision. This is particularly relevant where there is some "prospecting" cost – costs involved in investigating the relative value of alternative courses of action. These costs might involve action in their own right (as in classical prospecting – digging a hole in the ground to find out whether it is worth siting a gold mine there), or might involve only mental activity while considering and evaluating available data. In the latter case, the mental decision process itself must be counted as a kind of action, and it is necessary to anticipate the costs of this activity (in cognitive science terms, "metareasoning" or "thinking as doing" [26]). In the Attention Investment model, meta-reasoning is also accounted for as an attentional cost.

4.3. Architecture of the model

The cognitive model we have developed simulates these phenomena using an agent architecture [28], in which all possible courses of action are represented by agents competing to be scheduled on a single processing agenda. Only one agent can be activated at a time, thereby simulating a unitary locus of human attention – we can only attend to a single location (usually by visual fixation) at any time. The agent that gets activated is selected according to a decision criterion that estimates the best cost-benefit return, subject to the quality of the information (observed or from previous experience) on which that estimate is based.

All "internal" cognitive activities are also represented by agents – these activities include decomposition of actions into component tasks, re-evaluation of the agenda, and deciding between either further prospecting or further goal-reduction. There is a single attention resource that must be allocated either to these activities or to external perception [7].

When the system is initialized, it has no knowledge about the current situation, so immediately acts to reduce uncertainty (and thereby risk) by allocating attention resources to the gathering of information. Once enough information has been gathered to evaluate alternative potential courses of action, the system starts to act.

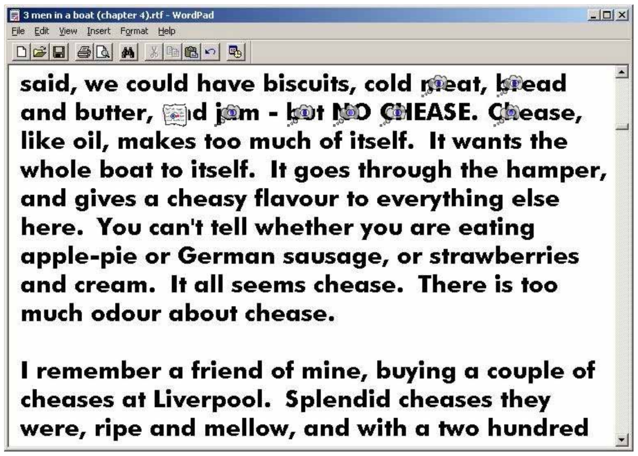
The transition between prospecting and acting is determined purely by a change in the expected utility of the agents responsible for each. The decision to proceed either with direct manipulation or programming actions is made on the same basis. This balance may change as the result of information gathered while acting – the model may start with a direct manipulation strategy, realise that it will be too costly, and change to a programming strategy. Similarly it may abandon a programming strategy if it appears too risky.

The model can also allocate attention units to re-evaluating its own agenda, in order to monitor and possibly change the current course of action. All actions occur when the agenda control processing decides that the expected utility of further consideration has fallen below the expected utility of acting. This means that the model is able to reason about situations where there is an infinite number of future actions or consequences of action: it is still able to act efficiently based on available information and estimates from past experience. This makes it well suited to modeling the programming situation, in which end-users may decide to act in a way that appears "irrational" to experienced programmers, yet may be rational based on that user's past experience or expected future utility.

4.4. Example Simulated Task

The task on which the model has been tested is a simple one, when compared to most programming tasks. However it does incorporate all the decision criteria of the Attention Investment model as described above. This simple task thus offers theoretical continuity with those more complex programming activities that would be too expensive to simulate, but can still benefit from the qualitative design perspectives offered by the Attention Investment model.

The sample task simulates a course of action in which the user has a choice between a "programming" strategy and a "direct manipulation" strategy. Direct manipulation tends to involve moderate attentional cost, relatively low payoff and low risk because results can be monitored as the user acts. Programming tends to involve higher attentional cost (particularly in development of a specification, which involves additional prospecting), potentially high payoffs, but also high risks.


3 men in a boat (chapter 4).rtf - WordPad
File Edit View Insert Format Help

said, we could have biscuits, cold meat, bread and butter, and jam - but NO CHEASE. Chease, like oil, makes too much of itself. It wants the whole boat to itself. It goes through the hamper, and gives a cheasy flavour to everything else here. You can't tell whether you are eating apple-pie or German sausage, or strawberries and cream. It all seems chease. There is too much odour about chease.

I remember a friend of mine, buying a couple of cheases at Liverpool. Splendid cheases they were, ripe and mellow, and with a two hundred

Figure 1. Snapshot of cognitive simulation based on the Attention Investment model

The task here is to correct spelling errors in a document (figure 1). The "direct manipulation" alternative is manually to step through the document correcting each error. The cost of doing this is uncertain, because the user does not know how many times the error occurs. The "programming" alternative is to specify a search and replace operation, which may have unexpected results (such as replacing text that was not intended, or missing some misspellings due to capitalization, different word endings etc.)

The simulation therefore starts by reading some of the document in order to gather evidence with regard to the best investment. It may scan over the text, look at the scrollbar to see how long the text is, and so on. This scanning phase is also driven by the agent architecture, so that multiple agents may propose that the reader directs attention to different locations on the screen in order to acquire information that will improve risk or payoff estimates. The proposed attention point with the best expected payoff rises to the top of the agenda, and once the payoff is higher than any possible benefit of further agenda reordering, attention is directed to that point.

This is illustrated in figure 1, which shows a screenshot of the simulation running. The clouds indicate points on the screen which are being proposed by agents as attention fixation points. The agenda icon indicates the fixation point proposed by the currently preferred agenda item. At the moment when the currently preferred item is a programming action (raising a search and replace dialog), the equilibrium of the agenda will change from attention fixations contributing to the reading task to attention fixations contributing to raising the search and replace dialog.

This model architecture produces macro-level behaviour that emerges from micro-level processes based on the same decision factors. From the user's perspective, the micro-level decisions may not be accessible to introspection, but this model means that system designers are able to anticipate even unconscious factors leading to the choice of programming strategies.

This simulation starts with a single goal – to correct the spelling errors. It does not initially have a fixed strategy for achieving this. Decomposition of the top-level goal into subgoals and then specific actions is controlled, as with other agenda manipulations, by the meta-reasoning strategy. This means that some alternative actions may never be decomposed if they initially appear to have very high cost or very high risk. This aspect of the model simulates users who would never consider programming solutions to a task on the basis of their previous experience.

In most cases, the first results of goal decomposition will lead to activities that collect evidence to help refine risk and cost estimates. This will continue until the simulated user can make the decision to take a first step toward programming. This may not always happen, even where previous experience favours programming solutions – a short document can be corrected manually without any need to invest in more abstract alternatives. This is consistent with programming situations in the home, where it is generally possible to carry out an operation manually rather than programming. An example is staying up late to press the "record" button on the VCR, because you are not sure that programming will work (high degree of risk) or because you cannot be bothered learning how to operate it (high cost).

For this simple task, the Attention Investment model successfully simulates a range of observed user actions, accounting for them on the basis of previous experience, rather than assuming that different classes of user are either capable or incapable of programming because of their intellectual abilities. Removing this apparent discontinuity in the user population provides the opportunity to apply a single cognitive model of programming processes that can apply both to end-users and to everyday decisions made by professional programmers.

5. Conclusions

This paper has proposed that programming might be redefined to take in a wider range of computer usage contexts, all sharing certain cognitive features. According to this proposed definition:

- Programming involves loss of direct manipulation as a result of abstraction over time, entities or situations.
- Interaction with abstractions is mediated by some representational notation, and there are common properties of notations that determine the quality of that interaction.
- Management of complexity as a cognitive task involves linguistic and representational strategies that can in themselves be viewed as notational, and subject either to direct manipulation of the notation or more abstract interaction.

Although these issues are highly generic, it is still possible to formulate useful research models that address them. The Attention Investment model is quantitative, and can be implemented in a decision theoretic simulation. It is generic, in the sense that it offers a partial explanation of cognitive considerations for many users in many situations. It offers consistency between micro-level and macro-level cognitive mechanisms, making them accessible to designers as a basis for usability evaluation. Finally, it describes many situations that would not normally considered as varieties of programming, in a manner that clarifies the deep connections between programming and other kinds of human interaction with technology.

This model provides both a potentially rigorous description of constrained tasks, verifiable through cognitive simulation, and a qualitative design model that can be used to anticipate the consequences of certain types of interaction with design features without the need to do simulation. These features of the Attention Investment model include awareness of attention *costs*, assessment of *pay-off* that may result from abstract interaction, *risk* of failure, and the need to gather sufficient information to make appropriate *investment* decisions based on bounded reasoning assumptions. The results offer a description of first steps toward end-user programming in many situations, but in terms that are consistent with the cognitive demands in professional programming, providing a uniform basis for design of end-user programming systems.

6. Acknowledgments

Many people have contributed to the ideas in this paper. Margaret Burnett, Tony Hoare, Hugh Mellor, and Thomas Green have all given specific advice that has led to new perspectives.

7. References

[1] Blackwell, A.F. (2001). See What You Need: Helping end users to build abstractions. *Journal of Visual Languages and Computing*, 12(5), 475-499.

[2] Blackwell, A.F. & Burnett, M. (2002). Applying Attention Investment to end-user programming. In *Proceedings HCC'02*.

[3] Blackwell, A.F. and Green, T.R.G. (1999). Investment of Attention as an Analytic Approach to Cognitive Dimensions. In T. Green, R. Abdullah & P. Brna (Eds.) Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11), pp. 24-35.

[4] Blackwell, A.F. and Hague, R. (2001). AutoHAN: An Architecture for Programming the Home. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 150-157.

[5] Blackwell, A.F., Robinson, P., Roast, C, and Green, T.R.G. (2002). Cognitive models of programming-like activity. *Proceedings of CHI'02*, 910-911.

[6] Card, S.K., Moran, T.P. & Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum.

[7] Carlson, R.A., Wenger, J.L. & Sullivan, M.A. (1993). Coordinating information from perception and working memory. Journal of Experimental Psychology: Human Perception and Performance, 19(3), 531-548.

[8] Diaz Pace, J.A. & Campo, M.R. (2001). Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10), 67-73.

[9] Goldhaber, M.H. (1992). The attention society. In E. Dyson (ed.) *Release 1.0* number 3, New York, EDventure Holdings, pp. 1-20.

[10] Goodell, H. (1999). *End-User Programming* website. On-line proceedings and material from workshop at CHI 99 (Pittsburgh, PA May 17 1999) http://www.cs.uml.edu/~hgoodell/EndUser/

[11] Goodman, N. (1969). *Languages of art: An approach to a theory of symbols*. London: Oxford University Press.

[12] Gray, W.D. & Fu, W.-T. (2001). Ignoring perfect knowledge-in-the-world for imperfect knowledge-in-the-head: Implications of rational analysis for interface design. *CHI Letters* 3, 112-119.

[13] Green, T.R.G. and Blackwell, A.F. (1998). Design for usability using Cognitive Dimensions. Tutorial session at *British Computer Society conference on Human Computer Interaction HCI'98*.

[14] Green, T.R.G. and Blackwell, A.F. (1996). Ironies of Abstraction. In *Proceedings 3rd International Conference on Thinking*. British Psychological Society.

[15] Green, T.R.G. and Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' approach. *Journal of Visual Languages and Computing*, **7**,131-174.

[16] Hoare, A.J.P. Personal communication, 30 October 2001.

[17] Hoc, J.-M., Green, T.R.G., Samurcay, R. and Gilmore, D.J. (Eds) (1990). *Psychology of programming*. Academic Press..

[18] Lindsay, R.K. (1988). Images and inference. *Cognition*, **29**(3), 229-250.

[19] McCracken, D.D. (1957). Digital computer programming. Wiley.

[20] Mellor, D.H. (1988). 'How much of the mind is a computer? In P Slezak and W. R. Albury (Eds). Computers, Brains and Minds. Dordrecht: Kluwer, 47-69.

[21] Nardi, B.A. (1993). *A small matter of programming: Perspectives on end user computing*. MIT Press.

[22] Nielsen, J. & Molich, R. (1990). Heuristic evaluation of user interfaces, Proceedings of ACM CHI'90 Conf. (Seattle, WA, 1-5 April), 249-256.

[23] O'Hara, K.P. & Payne, S.J. (1998). The effects of operator implementation cost on planfulness of problem solving and learning. *Cognitive Psychology*, 35, 34-70.

[24] Pirolli, P. & Card, S.K. (1999). Information foraging. Psychological Review, 106, 643-675.

[25] Portante, T. & Tarro, R. (1997). Paying attention. *Wired* 5.09, 114-116.

[26] Russell, S. & Wefald, E. (1991) Do the right thing: Studies in limited rationality. MIT Press. (Book)

[27] Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. IEEE Computer, August, pp. 57-69.

[28] Staton, S. (2002). An agent architecture. Paper presented at CHI 2002 workshop on Cognitive Models of Programming-Like Processes.

[29] Stenning, K. & Oberlander, J. (1995). A cognitive theory of graphical and linguistic reasoning: Logic and implementation. *Cognitive Science*, **19**(1), 97-140.

[30] Weinberg, G. The psychology of computer programming. New York: Van Nostrand Reinhold. From (1971)

[31] Wrubel, M.H. (1959). A primer of programming for digital computers. McGraw Hill.