What is Programming?

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

Keywords: POP-II.A. end users POP-V.A. attention investment
POP-V.B. phenomenology POP-VI.A. definition of programming

Abstract

Research into the cognitive aspects of programming originated in the study of professional programmers (either experts or those learning to program). As personal computers become widespread, and most new domestic appliances incorporate microprocessors, many more people are engaging in programming-like activities. Some of these are studied as "end-user" programmers, by analogy to professional programming, but many encounter tasks and contexts completely unlike conventional programming. This paper analyses the generic nature of these new kinds of programming, identifies the cognitive demands that characterize them, and presents one possibility for a cognitive model of programming whose development was driven by these concerns.

Introduction

The nature and extent of programming activity is changing rapidly. In the domain of professional programming, new tools continue to change the day-to-day nature of design and coding. But more radically, programming-like tasks are entering many non-professional domains. This paper explores the characteristics of those domains, and proposes changes to the way we define programming in order to explore its cognitive demands.

Historic definitions

Early publications in computing were written by practitioners, building and applying computers to practical tasks (though they often worked in a university context). These practitioners offered clear definitions of programming, without necessarily intending them as deep analyses of human activity. Nevertheless, these early publications are now seldom cited, and it is instructive to quote from them. Hartree (1950) explained "The process of preparing a calculation for a machine can be broken down into two parts, 'programming' and 'coding'. Programming is the process of drawing up the schedule of the sequence of individual operations required to carry out the calculation" (Hartree 1950, p. 111). Coding was of course extremely time-consuming before the development of assembler languages, but programming soon became the predominant activity. Wilkes (1956) made the distinction between the programme and other forms of calculation as follows: "The sequence of orders is known as the programme, and the machine performs it automatically without intervention from the user" (Wilkes 1956, p. 2).

The fact that a notation or language was developed to express the programme led to metaphors of programming as communication: "Programming … is basically a process of translating from the language convenient to human beings to the language convenient to the computer" (McCracken 1957). Note that this definition appears to subsume the earlier activity of coding, already less significant. In fact the definition could even be reversed: "The process of organizing a calculation can be divided into two parts – the mathematical formulation and the actual programming … translating … into the language of the computing machine" (Booth 1958). By the end of the first decade of computing research, the notions of a programme as an automatic sequence, programming as mathematical specification, and linguistic translation between the two were firmly established. "This sequence [of basic operations] is called the program and the process of preparing it is called programming" (Wrubel 1959, 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".

Research definitions

As programming applications have moved away from the mathematical domain of these early practitioners, the nature of the basic operations, the symbols in the language, and the formulations or methods of solution have all evolved. An introductory chapter to the book "Psychology of Programming" notes that the programming has changed from "describing calculations" to "defining functions", and then "defining and treating objects". These terms reflect the changes in programming from mathematical to more general data processing issues. Within mainstream computer science (i.e. outside the psychology of programming field), these advances may be considered more or less elegant or intuitive by computer scientists (Blackwell & Hague 2001b), but the criteria for defining these terms are often (as in mathematics), a matter for polite agreement rather than empirical investigation. The agreed basis of the human activity of programming is still, as defined in Collins dictionary, "to write a sequence of coded instructions fed into a computer … to arrange data in a suitable form so that it can be processed by a computer … to feed a program into a computer".

So far as this conventional activity of programming itself is concerned, the observed nature of the task is influenced more than anything by the skill of the programmer - as an anonymous reviewer of this paper noted, this is very frequently discussed in many online forums. A typical example was the thread "How hard is programming?" on the PPIG-discuss list in which Brooks observed that the "factor of 4-5 for differences in programmer performance is already substantially greater than almost any other human skill" (Brooks 2001). Among published research, several contributors to the book "Psychology of Programming" (Hoc, Green, Samurcay & Gilmore 1990) 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 (like this paper) "what is programming?" concludes that "The crucial dimensions in the activity of programming are processing and representation" (p. 160). But this definition could refer to almost any human cognitive skill.

Further analysis leads also to subtasks outside the scope of coding, such as "understanding the problem, design, coding and maintenance" (p. 46), and thus to the observation that "programming is a complex cognitive and social task" (p. 47). Many skills of a professional programmer are related to social context rather than the technical one – writing and interpreting specification documents, participating in design meetings, estimating effort and so on. In fact the activities described in the early programming texts above (now called "design" and "coding") are a small component of professional programmers' work. One reaction to this in the research community has been to broaden the remit of investigation to include various social aspects (rather than simply cognitive ones) of the task of the professional programmer. This leads to research questions such as "What can we assume about skills, work practices, background, exchange of information with each other? Do they work singly or together? etc." (Green 1998)

This paper suggests moving in the other direction – away from conventional programming and software engineering to focus on the characteristic cognitive tasks of programming, whether or not they occur in a social context that would normally be called programming.

Technical drivers for reassessment

There are three relatively recent developments in computer science and related fields that encourage reassessment of these definitional questions. The first of these is the development of end-user programming as a field of interest. This field has devoted considerable attention to definition of the term "end-user", but perhaps less to definition of the term "programming", as noted below. The second development is of programming techniques that do not appear to be languages by the normal analogy to human language. Programming by example or demonstration is one such technique. The third development is the increasingly common use of the word "programming" to refer to activities that do not fall within the remit of computer science definitions of the term (insofar as they do not offer Turing-equivalent formalisms) – many people say that they are "programming" in HTML, not to mention "programming" a VCR or a microwave oven.

The aim of this paper is to establish common ground across these current and future challenges to the definition of programming. But definition of terminology is not the main reason for doing this (it may be no more practical to define exactly what is programming than, as observed by Wittgenstein, to define exactly what is a game). The main objective is to ask more universal questions about the cognitive activity of programming, which can then form a basis for future empirical research as well as for the design, development and evaluation of programming systems.

Definition A: Who is a programmer?

When Weinberg wrote his early monograph "The Psychology of Computer Programming" (Weinberg 1971), it was assumed that every serious user of a computer would be a programmer. He used the term "programmer" almost as a synonym for user (some people performed mundane operations, such as data entry or job control – but these existed solely to serve the programmers). It was the development of standardized packaged software (which did not require programmers at the point of use), and of more sophisticated interactive applications (which needed special training and experience outside of programming) that led to classes of professional computer user who were not programmers.

The lines of 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 products, 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.

This phenomenon is increasingly being described as "end user programming". End users are normally defined in terms such as "people for whom programming is not their main job." This also makes the distinction between analysts, who once represented the requirements of users to the programmers but were not users themselves, and end-users, who create programs for their own use. These aspects of the term end-user are becoming clear, but use of the word programming in this context less so. Goodell's excellent website devoted to the topic (Goodell 1999) offers definitions of "end user" and "end user programming", but not of "programming". End-user development, end-user customization and end-user software engineering have also been proposed as expressions of the challenges faced by users encountering these new tools. Some of these terms apparently deemphasise the sophistication of the programming required ("customization"), while others emphasise the fact that large or complex application development is difficult whatever the tools used ("software engineering").

These alternate emphases in terminology often seem intended to influence the features demanded by the software tools market, rather than seriously to contend that programming is not involved. Programming would certainly be recognized as an important component of software engineering, application development and application customization in those situations where conventional programming languages are used for these purposes.

The one aspect in which "end-user" programming differs from these is that the programming techniques used for development or customization can potentially be so unlike conventional programming languages. Is a scripting language really a programming language? How about a spreadsheet? Or a macro language? Or a keyboard macro? A configuration file? A Java program? A Javascript program? A Server Side Include macro? A Cascading Style Sheet? An HTML page? A Microsoft Word document? From the perspective of a non-programmer or end-user, the distinctions

are scarcely important – all of these technologies have similar capabilities, and several can potentially be used to create apparently identical results. Yet some of them are classified by computer scientists as being programming languages, and some are not.

The first proposal of this paper is that all computer users may now be regarded as programmers, whose tools differ only in their usability for that purpose. If this is the case, programming research should be universal and inclusive in its scope, rather than restricted to the experience of "professional" programmers. The next section considers a classification of programming languages that takes account of these user perspectives, rather than the conventional theoretical perspective.

Definition B: 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 distinctions are in conflict with some ideals of modern design for usability. What the user sees when authoring the page in HTML is not 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 behaviour in different circumstances – a conflict with the ideal of direct manipulation. Of course these departures from the "ideal" are necessary here in order to achieve the task. 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 many different consequences. Once again, this range of effects is produced by the use of an abstract notation to define behaviour in different circumstances (the markup language). As with the use of JavaScript, this may result in syntax errors, runtime errors, or bugs in the form of unintended or exceptional behaviours.

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).

All of these examples, though trivial by comparison to 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 behaviour (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. The second proposal of this paper is that almost all major software applications now include programming languages. If this is the case, research into programming should focus on typical programming experiences, especially those which result when abstract notation replaces direct manipulation.

Definition C: When should we use the word programming?

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 point in considering these common uses of the term when we do research into the cognitive demands of programming? When 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 property that the user is not directly manipulating observable things, but specifying behaviour to occur at some future time.

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. If this is the case, it will be necessary to adopt research methods that analyse users' reports of their own experience. In other disciplines where there is a concern with the significance of experience reports over prior analytical expectations, phenomenological research methods are employed, and these may also be appropriate to the investigation of programming in these broader contexts.

Consequences: Why is programming hard?

The reason for extending the research definition of programming is in order to identify uniformities across a wider range of human activity, and thus achieve greater generality of research results. Many practices of professional programmers are determined by the cultural context of the programming profession, as well as by analytic conventions from the study of computer science and management science. If we can exploit commonality between conventionally recognized programming activity and other contexts of human tool use, this offers the opportunity both to apply insights from the world of programming to other fields of tool design, and also to adopt creative solutions for programming tools that have been identified in atypical programming situations.

One productive starting point for such research is to consider the common cognitive demands of programming tasks across this wider range of programming activities. The argument presented so far leads directly to observations about what makes programming hard. The common features of the various programming tools described are a) loss of the benefits of direct manipulation and b) introduction of notational elements to represent abstraction.

Loss of Direct Manipulation

The cognitive benefits of direct manipulation arise partly from the fact that image representations mitigate the "frame problem" in cognitive science (Lindsay 1988). This states that 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, in order to evaluate the results of potential actions. The problem is that planning is only possible if the scope of effects of a given action can be constrained, meaning that there must be a strictly defined boundary beyond which the action will not have any effects. If this is not the case, any action may potentially have infinite consequences, and it is not feasible to constrain the planning process. In direct manipulation systems, many constraints on causality are directly available in the actor's perception of the situation. This is less true of linguistic representations, where there is no limit on the abstract expressive power of the representation system (Stenning & Oberlander 1995), and hence no boundary that can be exploited to constrain reasoning during planning.

These cognitive considerations lead to the well-known benefits of direct manipulation (Shneiderman 1983). 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. While 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" (Blackwell & Green 1999, Blackwell & Hague 2001a).

Use of Notation

The second universal characteristic of programming situations is that the program is represented in 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 (Mellor 1988). 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 - Goodman 1969).

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 (Green 1989, Green & Petre 1996, Blackwell & Green forthcoming). 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.

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 integrated in order to manage complexity. This results from a combination 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 situation, more complex programs can be designed by making changes to the notation itself, so that the user can extend the vocabulary with which to 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 of 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, and 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" in the conventional computer science context (an answer for which I am indebted to Tony Hoare (personal communication)), 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 (Dias-Pace & Campo 2001).

Proposal: A cognitive model for abstraction use in programming

Definitions of programming from a computational perspective have often settled on the attributes of the program rather than those of the user, such as: "My personal minimal definition of programming is 'writing a program', where 'program' is a description containing a 'while' loop" (Prechelt 1998). The argument of this paper leads to the alternative proposal that a definition of programming should instead focus on the experience of the user. From the perspective of a user, the crucial factor is not whether the problem is intrinsically complex (suitable software can make even complex problems seem simple), but rather the cognitive resources that the user is prepared to devote to solving the problem.

We have developed a basic cognitive model to predict usability characteristics of programmable systems, based on these observations. The attention investment model is a decision-theoretic account of programming behaviour. It offers a cost/benefit analysis of abstraction use that allows us to predict the circumstances in which users will 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. The effort invested in attention can be described as a measure of "concentration" - cumulative attentional effort over time. Creating a program requires some amount of concentration - an investment of attention. The payoff if the program works correctly is that it will automate some task in the future, thereby saving attentional cost. 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.

The relationship between these factors in the professional programming context has been discussed previously (Green & Blackwell 1996, Blackwell & Green 1999). Many programming activities promise, through automation, to save attentional effort in the future. This has always been a primary goal of programming, as noted by Wilkes, whose definition of a computer programme is precisely that "the machine performs it automatically without intervention from the user" (Wilkes 1956). The irony of this abstract approach is that the activity of programming may involve more effort than the manual operation being automated. Some people choose to write a program anyway (perhaps with an additional benefit – acquiring programming skills for the future), but most decisions to start programming are based on an implicit cost-benefit analysis. The attentional measures that we have defined above constitute a basis for cost-benefit analyses regarding investment of attention in programming.

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" - Russel & Wefald 1991). In the attention investment model, the costs of mental activity must be accounted as an attentional effort. This is completely consistent both with the common understanding of the nature of concentration, and also with cognitive models of attention in which locus of attention can be applied to internal as well as external objects.

The model we have developed simulates these phenomena using an agent architecture (Staton 2002), 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. The agent to be activated is selected according to a decision criterion that estimates the best cost-benefit return, subject to the quality of the information on which that estimate is based. All "internal" cognitive activities are also represented by agents – these include decomposition of actions into component tasks, re-evaluation of the agenda, and decisions between prospecting or goal-reduction. When the system is initialized, it has no knowledge about the current situation, so immediately acts to reduce uncertainty by allocating attention to the gathering of information. Once enough information has been gathered to evaluate alternative potential courses of action, the system starts to act.

These simulated actions might involve "programming", or they might involve "direct manipulation". Direct manipulation tends to involve medium attentional cost, relatively low payoff and low risk. Programming tends to involve higher attentional cost (particularly in development of a specification, which involves additional prospecting), potentially high payoffs, but also high risks. A simulation of spelling correction has been implemented as a simple demonstration of these cognitive behaviours (Blackwell 2001). In this simulation, the "direct manipulation" alternative is manually to step through the document correcting each error. The "programming" alternative is to specify a search and replace operation, in which the results of this abstract operation may not be known in advance. We believe that these decision criteria can be generalized to many more complex opportunities for programming.

Conclusions

This paper has offered three proposals by which the definition of programming might be recast for research purposes. The first is that all computer users may now be regarded as programmers, whose tools differ only in their usability for that purpose. The second is that, as all major software applications now include programming languages, research into programming should focus on typical programming experiences rather than being constrained to the domain of conventional software engineering tools. The third is that when any device user says he or she is programming, we should not question whether this activity is "genuine" programming, but instead analyse the user's experience in order to develop a more generic understanding of programming activity.

These proposals lead to some initial observations about the cognitive tasks involved in this more general interpretation of programming. 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. Finally, the management of complexity as a cognitive task involves linguistic and representational strategies that can in themselves be viewed as notational.

Although these issues are highly generic, it is still possible to formulate useful research models that address them. The Attention Investment model is quantitative (in the usual decision theoretic manner). It is generic, in the sense that it offers a partial explanation of cognitive considerations for many users in many situations. 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.

Acknowledgments

The ideas in this paper have been developed over the course of many conversations, in which invaluable advice and ideas have been contributed by Margaret Burnett, Tony Hoare, Hugh Mellor, Howie Goodell and Thomas Green.

References

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

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.

Blackwell, A.F. and Green, T.R.G. (forthcoming). Notational systems – the Cognitive Dimensions of Notations framework. To appear in J.M. Carroll (Ed.) *Toward a multidisciplinary science of human-computer interaction*.

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

Blackwell, A.F. and Hague, R. (2001b). Designing a programming language for home automation. In G. Kadoda (Ed.) *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2001),* 85-103.

Booth, K.H.V. (1958). *Programming for an automatic digital computer*. Butterworth.

Brooks, R. (2001). Contribution to PPIG-discuss thread: *How hard is programming?* Wed, 2 May 2001 10:08:49.

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

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

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

Green, T.R.G. (1989). Cognitive dimensions of notations. In A. Sutcliffe & L. Macaulay (Eds.), *People and Computers V*. Cambridge University Press.

Green, T.R.G. (1998). Contribution to PPIG-discuss thread: *Who are END-USERS?,* Thu, 5 Nov 98 08:14:25.

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.

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

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.

Hartree, D.R. (1950). *Calculating instruments and machines*. Cambridge University Press.

Hoare, C.A.R. Personal communication, 30 October 2001.

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

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

McCracken, D.D. (1957). *Digital computer programming*. Wiley.

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.

Prechelt, L. (1998). Contribution to PPIG-discuss thread: *Who are programmers?* Wed, 14 Oct 1998 10:59:35.

Russell, S. & Wefald, E. (1991). *Do the right thing: Studies in limited rationality*. Cambridge, MA: MIT Press.

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

Staton, S. (2002). *Some notes on an agent-based model of cognition for programming-like tasks*. Paper presented at CHI 2002 workshop on Cognitive Models of Programming-Like Processes.

Stein, M.L. and Munro, W.D. (1964). *Computer programming: A mixed language approach*. Academic Press.

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

Weinberg, G. (1971). *The psychology of computer programming*. New York: Van Nostrand Reinhold.

Wilkes, M.V. (1956). *Automatic digital computers*. Cambridge University Press.

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

What is "programming". From Wikiversity. Jump to navigation Jump to search. Programming is coding, modeling, simulating or presenting the solution to a problem, by representing facts, data or information using pre-defined rules and semantics, on a computer or any other device for automation.Â Programming is the art and science of translating a set of ideas into a program - a list of instructions a computer can follow. The person writing a program is known as a programmer (also a coder).