Usability Issues in the Design of Novice Programming Systems

John F. Pane
Brad A. Myers

1 August 1996
CMU-CS-96-132

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891

Also appears as:
Human-Computer Interaction Institute Technical Report CMU-HCII-96-101

Abstract

This report reviews and organizes research about novice programmers. Over the past two decades, many aspects of novice programming have been investigated, resulting in the discovery of important facts and tradeoffs about what makes programming difficult to learn, and about the effectiveness of existing languages, environments, and methods of instruction. However, because this research is dispersed throughout the literature, it is difficult for designers of new programming systems to consider all of the issues collectively. The result is that most new systems are built primarily around technical objectives, perhaps considering only a subset of the usability issues summarized here. In addition to providing a checklist of issues that should be considered in the design of future systems, this report can be used to help researchers identify fruitful topics of future novice programming research.

The authors can be contacted by electronic mail at:
John Pane <pane+@cs.cmu.edu>
Brad Myers <bam+@cs.cmu.edu>

This research was partially sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326, and partially by NSF under grant number IRI-9319969. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: Novice Programming Environments, Empirical Studies of Programmers, End-User Programming, Programming Languages, Human-Computer Interaction, Usability, Computer Science Education

Table of Contents

1. Introduction

This report summarizes research about novice programming. Over the past twenty years many research studies have discovered useful information about novice programmers, and identified good and bad aspects of today's programming systems[1], both visual and textual. However, this body of research is widely distributed throughout the literature and is not well organized, making it difficult to use in guiding the design of new systems. The result is that these research results generally have not been systematically fed back into the design of new programming systems. Instead, the design of new languages and environments has most often been driven by technical objectives, such as ease of parsing, ease of generating fast code, closeness to the machine, ease of proving correctness, etc. Even systems that were designed for novice users or for teaching have not attempted to broadly survey this body of research before making critical decisions about the metaphor or model that the language is based on, the notation that is used in the language, and the environment. For example, the Turing language [Cordy 1992] was designed, in part, as a teaching language for children, with attempts to resolve perceived difficulties with Pascal. However, the list of perceived difficulties is focused primarily on missing features: lack of string handling facilities, type-safe variant records, modularity, concurrency, and type-safe compilation.

The focus of this report is research about the novice programmer. Research that compares novices and experts is included, but research that focuses exclusively on experts is not included here unless it offers general insight into all programmers. Of course, many of the issues mentioned here are applicable to experts as well.

Inspired by the Smith & Mosier user interface guidelines [Smith 1986b], we have gathered and organized this information so that it can be used in the design of new programming systems. The information is organized into topics that span the issues that researchers have explored. While the topics are interrelated in many ways, we have organized them using a subset of the usability principles called *heuristics* defined by the *Heuristic Evaluation* usability engineering method [Nielsen 1994]:[2]

- Visibility of system status.
- Match between system and the real world.
- User control and freedom.
- Consistency and standards.
- Recognition rather than recall.
- Aesthetic and minimalist design.
- Help users recognize, diagnose, and recover from errors.
- Help and documentation.

Each of these heuristics is represented by a section of this document. The introduction to each section contains Nielsen's definition of the heuristic, followed by our interpretation of the heuristic in the domain of novice programming. Following each introductory section are topics summarizing the novice programming issues that fall into the heuristic.

1. In this report, *system* will be used to mean the programming language as well as the programming environment (set of tools) in which programs are developed.
2. Note that, although this report adopts a subset of the usability *principles* defined in [Nielsen 1994], it does not make use of the usability engineering *method* that is described there.

2. Definitions

Beacon	A common code fragment that serves to indicate the probable presence of a certain high-level operation.
Environment	The collection of tools used in viewing, constructing, running, and debugging programs.
Expressiveness	A measure of the ease in translating a plan into a program. A high-level language is more expressive than assembly language.
Guiding Knowledge	A description of everything a naive user needs to know about the system. This can be the contents of a manual, tutorial, or verbal instructions.
Heuristics	A set of recognized usability principles from [Nielsen 1994].
Inner World	The details of programming that are not directly related to the high level task, such as variable declarations.
Locality	A measure of the proximity of related items, e.g. the declaration and use sites of a variable.
Match-mismatch	A phenomenon where different notations are better depending on the task. Performance is best when the structure of the information sought matches the structure of the notation. Mismatch leads to poor performance.
Metaphor	An real world system that the programmer can use as a reference for how the programming system will work. For example, a stack of dishes is a metaphor for the stack data structure; a variable is often described as a box.
Natural Language	A language that originated in human spoken form, e.g. English.
Notation	The symbols of a programming language and the syntactic rules for combining them into a program.
Plan	A high-level mental strategy for accomplishing a particular programming goal. In order for the plan to be implemented, it must be elaborated and translated into the programming language.
Secondary Notation	Information that is embedded in the program text that is not part of the syntactic structure that is meaningful to the system. For example, comments and indentation are forms of secondary notation in most systems.

Signalling	A secondary notation that uses color and typographic styles to emphasize certain ideas or to clarify the organization of the program. For example, keywords are often shown in boldface or a different color than surrounding text.
Superlativism	An expectation that a particular kind of programming system is superior to another for <u>all</u> programming tasks. Some people expect superlativism of functional languages over imperative languages.
Viscosity	A measure of how much effort is required to make a small change to the program, e.g. how difficult it is to change a conditional into a loop.
Visibility	A measure of how much effort is required to expose desired information.

3. Organization of this Document

Each novice programming usability topic is formatted like this page into a number of parts. The first line contains the item number and title of the issue described. It is followed by a prose description of the issue, definition of terminology, implications or guidelines, and motivations.

Context of Use:	This area indicates where the issue applies, such as notation, metaphor, environment, education, etc.
Justified by:	This area lists how these findings were determined, such as through formal human-factors empirical studies, observation of individual users, or the opinion of experts.
Examples:	This area contains positive and/or negative examples of this issue.
Exceptions:	This area lists exceptions to any guidelines suggested above.
Cross References:	This area lists any other related issues with their section numbers.
References:	This area contains the full list of references for this section; they are also embedded above where appropriate.

4. Visibility of System Status

“The system should always keep users informed about what is going on, through appropriate feedback within reasonable time [Nielsen 1994].”

The research in this section identifies information that is important to the programming process, and suggests ways that the language and the programming environment can help make this information visible, such as by keeping related items close together, by revealing or highlighting important items, by avoiding potentially confusing appearances, and by providing immediate feedback.

4-1. Use Signalling to Highlight Important Information

Typographic signalling is a secondary notation that uses color or typographic styles to emphasize certain ideas or to clarify the organization of the program. Signalling improves comprehension of the signalled material, at the expense of non-signalled material. [Fitter 1979] cites the principles of *redundant recoding* and *relevance* in good notational schemes: in addition to symbolic information there are perceptual cues, thus both perceptual and symbolic characteristics highlight important information; but only information that is actually useful to the user is highlighted.

Many environments automatically signal keywords of the language, even though this may not be the most important information that the reader needs [Baecker 1986]. Instead, the environment should signal semantically important information, or facilitate the signalling of those items that the user feels are important [Gellenbeck 1991a]. Secondary notation should be used to improve access to information that is needed but obscured [Green 1990b]. The user should be informed in advance about the meaning of the signals. Of course, these signals should be correct and not misleading.

Context of Use:	Environment, notation
Justified by:	Empirical studies, expert opinion
Examples:	Many environments highlight syntactic structures of the language, such as keywords. An intelligent environment could use color to highlight semantic information about the program that is less obvious in the syntactic structure, such as the bindings of identifiers. In visual languages, all visible symbols are interpreted by the user as relevant, even if they are incidental [Green 1991]. [Baecker 1986] and [Baecker 1990] describe an elaborate automated system that uses graphic design principles to enhance text in C program printouts, in an attempt to make them more readable, understandable, appealing, memorable and maintainable. This system highlights information that the authors judged to be fundamental, such as: the relationships between comments and the code they discuss; certain tokens such as identifiers; the correct parsing of complex expressions and statements; use of global variables, which are a frequent source of errors; and unusual flow of control. This system improved readability as measured by performance on a comprehension test.
Cross References:	4-2. “Beacons” 4-4. “Beware of Misleading Appearances” 6-3. “Support Secondary Notation”
References:	[Baecker 1986, Baecker 1990, Fitter 1979, Gellenbeck 1991a, Green 1990b, Green 1991]

4-2. Beacons

Beacons are common code fragments that serve to indicate the probable presence of certain high-level operations [Brooks 1983]. Although few have been identified, they are typically patterns that are not extremely common, and it is their infrequent appearance that makes them useful in confirming or suggesting a hypothesis about what the code does [Gellenbeck 1991b]. Thus they are useful for high-level understanding, but are less useful for detailed tasks like debugging [Wiedenbeck 1986b]. They have been shown to help experts in program comprehension [Boehm-Davis 1996]. These beacons are so strong that misleading ones can induce false comprehension in experts [Wiedenbeck 1989]. However, there is evidence that novices do not make effective use of beacons, probably because they have not learned the patterns yet [Gellenbeck 1991b, Wiedenbeck 1986a, Wiedenbeck 1986b]. This is supported in an analysis of eye movements of programmers, which found that experts spend more time viewing meaningful areas of the program while novices do not [Crosby 1990]. Even so, using colored cues to mark meaningful sections of code helps experts to find bugs in those sections [Gilmore 1988]. Perhaps the environment should highlight meaningful areas such as beacons with color or other cues, to draw novices' attention to them. This may also be instructional, because the beacons mark schemata that novices must learn anyway [Perkins 1986, Samurçay 1989].

Context of Use:	Environment
Justified by:	Empirical studies
Examples:	A beacon for sorting is the swapping operation, which exchanges the values of two variables:

```
temp := a;
a := b;
b := temp;
```

Cross References:	4-1. "Use Signalling to Highlight Important Information"
References:	[Boehm-Davis 1996, Brooks 1983, Crosby 1990, Gellenbeck 1991b, Gilmore 1988, Perkins 1986, Samurçay 1989, Wiedenbeck 1986a, Wiedenbeck 1986b, Wiedenbeck 1989]

4-3. Locality and Hidden Dependencies

Many researchers argue that locality is important in programming: physical proximity should be encouraged and remote references should be avoided [Cordy 1992]; strongly-related subcomponents should be kept together, not dispersed [Bonar 1990]; delocalized plans cause difficulty [Soloway 1988]; and hidden dependencies and poor visibility reduce understanding [Green 1996]. To the extent that information is visible on the screen, working memory limitations are reduced, allowing novices to perform better [Anderson 1985, Green 1987]. These observations are to some extent opposed to modularity and abstraction, which tend to hide details and make related pieces of code more distant from one another [Green 1996]. Part of this problem can be relieved by intelligent use of secondary notation [Green 1990b], or with modern environments that use powerful navigation features to make the remote code easily accessible [Goldenson 1991, Miller 1994].

[Lewis 1987] proposes replacing *programming by synthesis* with *programming by modification*, where a library of examples is provided, from which the programmer chooses an appropriate one for a starting point, identifies needed modifications, then modifies it to suit the current need. This elevates locality because it is an important factor in understanding the examples.

Context of Use:	Environment
Justified by:	Empirical studies, expert opinion

Examples:

The Turing language attempts to group things physically close to one another and avoid remote references where possible. Declarations are not distinct from statements or runtime constants. For example,

```
const temp := x; x := y; y := temp
```

is a valid swap operation at any point in the program, regardless of context or the type of values being swapped [Cordy 1992].

In Turing, control structures implicitly create a scope, encouraging locality [Cordy 1992].

Pascal programmers frequently forget initialization preconditions while coding, requiring them to go back later to insert them [Green 1987]. In contrast, this almost never happens in Prolog, where the preconditions are located adjacent to the focal line. FPL, a graphical language that is equivalent to Pascal, also seems to eliminate this problem, perhaps because the graphical layout prompts the programmer to remember the initialization [Cunniff 1989].

Hypercard seriously reduces the visibility of the program text. It takes too many steps to make a desired item visible [Green 1990a, Green 1996].

Textual languages have hidden dependencies in variables and parameters, as well as in side effects of functions. Visual languages such as LabView and Prograph do well at avoiding hidden dependencies at a local level, but not so well at a global level. Textual languages usually have better visibility than visual languages [Green 1996].

Cross References:	5-5. “Support for Planning” 5-7. “Visual vs. Textual” 5-13. “Modularity and Abstraction” 6-3. “Support Secondary Notation” 8-1. “Minimize Working Memory Load”
References:	[Anderson 1985, Bonar 1990, Cordy 1992, Cunniff 1989, Goldenson 1991, Green 1990a, Green 1990b, Green 1987, Green 1996, Lewis 1987, Miller 1994, Soloway 1988]

4-4. Beware of Misleading Appearances

[Fitter 1979] cites a principle of *restriction* in good notational schemes: the syntax prohibits the creation of code that could easily be confused with other closely-related forms. A typographical error or cognitive slip should result in an invalid program, so that the system can detect the error for the user [Green 1996]. If the error results in a valid but incorrect program, the error will not be detected until it causes the program to behave in a noticeably incorrect manner.

One common mistake among novices and experts is misunderstanding the program because its formatting invites an incorrect interpretation. For example, if objects look alike, kids expect them to behave alike [Smith 1995]. There are two obvious ways to deal with this: 1) the system can impose a formatting that is consistent with the true meaning of the code; or 2) the system can try to interpret the formatting of the code the same way that the user would. The first solution may interfere with allowing secondary notation.

Context of Use: Notation

Justified by: Empirical analysis of frequent novice errors

Examples: Incorrect indentation may lead the user to believe that code is part of a control structure when it is really outside the control structure [du Boulay 1989a].

Stray wires in graphical program cause confusion [Green 1991].

Some students expect that all statements in a program, including those inside procedures, are executed in the order they appear in a program; others think the procedures are executed when they are called *in addition* to executing in a top-to-bottom scan [Sleeman 1988]. Some students think that all statements in the program must be executed at least once, even those that may have been skipped due to branching: that any statements which have not yet been executed are executed before the program terminates [Putnam 1989].

Exceptions: Early versions of Fortran are notorious for their rigid column-based notation, where restriction was extreme. Some apparently harmless errors in alignment were flagged as errors, while others resulted in unexpected results.

Cross References: 4-1. “Use Signalling to Highlight Important Information”
4-5. “Avoid Subtle Distinctions in Syntax”
6-3. “Support Secondary Notation”

References: [du Boulay 1989a, Fitter 1979, Green 1996, Green 1991, Putnam 1989, Sleeman 1988, Smith 1995]

4-5. Avoid Subtle Distinctions in Syntax

Notation should avoid subtle distinctions in syntax which might be overlooked or confused by novices. As described above (see "Beware of Misleading Appearances" on page 10), the principle of *restriction* advises against the use of syntax that is so similar to other forms that it is easily confused [Fitter 1979]. Novices get confused when there are two different syntaxes to accomplish the same effect [Eisenberg 1987]. More planning is required when there are many different legal solutions to a goal, which leads to frequent code changes [Gray 1987].

Context of Use:	Notation
Justified by:	Empirical studies, observation of individual users

Examples:

The difference between the quoted string "123" and the numeric value 123 is a problem [du Boulay 1989a].

Novices have difficulty because Pascal does not permit a comma to be used as punctuation in a number. For example: 32,000 is illegal while 32000 is legal [Pane 1996].

A common problem in C is the distinction between the assignment operator =, and the equality operator ==. For example, sometimes the programmer will writes `if (a = 0)` instead of `if (a == 0)`, which results in an assignment where comparison was desired. Pascal has a similar distinction between `:=` for assignment and = for equality, but this is not so serious due to restriction: syntax prohibits one of them to be used where the other is legal. Some systems use a special symbol for assignment to avoid the confusion with equality [Baecker 1986]. As mentioned in "Beware of Misleading Appearances" on page 10, a typographical error or cognitive slip should result in an invalid program, so that the system will detect the error for the user [Green 1996].

In C, indexing of arrays is done with square brackets, but initialization is done with curly braces:

```
int a[] = {1,2,3};
```

In C, it is very easy to accidentally comment out a large block of code. It is not an error to have a missing close-comment delimiter. Instead, the comment continues to the next occurrence of the close-comment delimiter. Modern environments use color signalling to make this error visible.

Cross References:
4-4. "Beware of Misleading Appearances"
5-3. "Consistency with External Knowledge"
5-14. "Cognitive Issues"

References: [Baecker 1986, du Boulay 1989a, Eisenberg 1987, Fitter 1979, Gray 1987, Green 1996, Pane 1996]

4-6. Support Incremental Running and Testing with Immediate Feedback

Immediate feedback aids problem solving [Lewis 1987]. The ability to test partial solutions is an important feature for novices and experts alike. Running the program should be encouraged because it is a useful debugging strategy [Gugerty 1986b]. When novices adopt a practice of testing their code incrementally, they perform better [Goldenson 1991, Green 1996, Perkins 1986]. This has a bigger impact on performance than good error messages in a batch-compiled delayed-feedback environment [Davis 1993]. These observations can be interpreted as a call for lisp-like dynamic languages [Smith 1992], but in fact the environment can provide this feature in traditional compiled languages such as Pascal [Goldenson 1991].

The computational machine should reveal its internal workings, and should do it in terms of the language itself [du Boulay 1989b]. A powerful graphical debugger will be used by novices not only for debugging, but also as an aid to program comprehension even in the absence of bugs [Goldenson 1991]. However, novices must be trained to use these tools effectively [Miller 1994]. [Brusilovsky 1994a] points out that an important feature of mini-languages for novices is that they be capable of operating in *dialog* mode, where individual commands can be issued by the user and are executed immediately with visible feedback. [Clements 1995] lists this as a design principle for novice programming environments.

In traditional compiled languages, beginners are also confused by the need to recompile after making a change, and the need for their program to be complete before it can be run [du Boulay 1989a]. The Geo-Logo environment addresses this problem by updating the program's output as soon as code is changed [Clements 1995].

Context of Use:	Environment, programming paradigm
Justified by:	Empirical studies
Examples:	Spreadsheets provide immediate feedback, in contrast to other environments where the programmer must recompile, re-execute, and re-enter data in order to test a change. This is an important factor in the success of spreadsheets [Lewis 1987, Nardi 1993]. A powerful graphical debugger such as the one in the MacGnome environments shows live graphical views of the data structures which are continuously updated while the program is running. The debugger attempts to display the data in a semantically meaningful way, such as displaying records, linked lists, binary trees, and multidimensional arrays in the same way they are pictured in textbooks and in the classroom. Recursive procedure calls are also easier to understand because the call stack graphically represented in the debugger. These displays facilitate understanding of the data structures and the program that manipulates them [Miller 1994, Myers 1988].
Cross References:	5-12. "Choosing a Paradigm" 10-1. "Support for Testing and Debugging"

References:	[Brusilovsky 1994a, Clements 1995, Davis 1993, du Boulay 1989a, du Boulay 1989b, Goldenson 1991, Green 1996, Gugerty 1986b, Lewis 1987, Miller 1994, Myers 1988, Nardi 1993, Perkins 1986, Smith 1992]

5. Match Between System and the Real World

“The system should speak the user’s language, with words, phrases, and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order [Nielsen 1994].”

The research in this section investigates the expectations and behaviors that a novice brings to programming, such as natural language and knowledge of the real world. It discusses ways that the programming system can effectively exploit the expectations and support the behaviors, in order to maximize the net positive transfer to programming. Several points are made about common programming constructs that do not match well with the way users tend to express concepts in natural language. In addition, the task-dependent nature of the effectiveness of the programming system is examined. While this section covers consistency between the programming system and the outside world, “Consistency and Standards” on page 50 covers internal consistency. [Payne 1986] describes a formal method for assessing both of these forms of consistency.

5-1. Choose an Appropriate Metaphor

A metaphor is a familiar analogy for how the programming system works. When the metaphor is good, users can infer how the programming system works by referring to their existing knowledge and expectations about how the modeled system works; otherwise they might be required to learn a collection of rules that seem arbitrary. In order to maximize this transfer of knowledge, the metaphor should be based on, and conceptually close to, a concrete real-world system that is widely known by the user audience [Smith 1994]. This familiarity requirement is violated by Prolog, where misconceptions abound in users' models of searching, matching, and backtracking [Fung 1990, Fung 1987, Mendelsohn 1990]. An appropriate concrete model can have a strong positive effect on the usability of a programming language [Mayer 1989].

Context of Use:	Computational metaphor of the language
Justified by:	Empirical studies, expert opinion
Examples:	Traditional programming languages use a computational model of the von Neumann machine, which has no physical world counterpart. Learning this computational model is an important stumbling block for novices [du Boulay 1989a, du Boulay 1989b].

The spreadsheet is widely viewed as a successful instance of a programming language that is useful to non-programmers. An important advantage of the spreadsheet is that it is based on a metaphor that fits very well with the tasks of its audience: page-oriented numerical computation for financial or other purposes [Nardi 1993]. Use of spreadsheets requires mastery of only two concepts: cells as variables and functions as relations between variables. Typically users use fewer than ten functions, including basic arithmetic and rounding. It is not necessary to string together low-level primitives, allocate memory, name variables, include files, etc. It is a familiar, concrete, visible representation that allows users to feel as though they are working directly on the task [Lewis 1987]. Unfortunately, this metaphor does not seem to extend well to general-purpose computation.

Logo uses the metaphor of a turtle in a two-dimensional world. Even this concrete model has its difficulties. For example, when the turtle is facing south, the user's left is the turtle's right, and vice versa [Mendelsohn 1990]. [Resnick 1994] describes the extension of the Logo metaphor to parallel computing.

Boxer uses a two-dimensional spatial metaphor for the organization of programs, but is not constrained to a single-level hierarchy of cells as spreadsheets are. All computational objects are represented in terms of a hierarchy of boxes, which can contain data such as text or graphics, or can contain behaviors in the form of Logo-like programs. It uses the metaphor of a port for accessing the contents of a box from a distant place in space [diSessa 1989].

[Brusilovsky 1994a] surveys a collection of mini-languages for novices that are based on the metaphor of a robot, and claims that they are intrisically engaging and visually appealing. One such language that achieved widespread usage is Karel the Robot [Pattis 1995].

[Finzer 1993] describes “Programming by Rehearsal”, where the programming process consists of moving “performers” around on “stages” and teaching them how to interact by sending “cues” to one another.

Pursuit is a visual shell that uses a comic-strip metaphor to represent programs [Modugno 1996].

Exceptions: Several systems are cited by [Mendelsohn 1990] as moving from the abstract toward the concrete: ARK [Smith 1986a]; Boxer [diSessa 1989]; ThingLab [Borning 1985]. He concludes that in these concrete systems, semantic complexities are not banished, users can not see the plans, and the systems have poor visibility and viscosity.

Cross References: 5-2. “Consistency with Metaphor”
5-11. “Support Direct Manipulation and Definition by Example”

References: [Borning 1985, Brusilovsky 1994a, diSessa 1989, du Boulay 1989a, du Boulay 1989b, Finzer 1993, Fung 1990, Fung 1987, Lewis 1987, Mayer 1989, Mendelsohn 1990, Modugno 1996, Nardi 1993, Pattis 1995, Resnick 1994, Smith 1994, Smith 1986a]

5-2. Consistency with Metaphor

The language should be consistent with its metaphorical model. It should abide by any suggestions that can be derived from the metaphor. [Halasz 1982] lists three classic problems with metaphors: the target domain has features not in the source domain, the source domain has features not in the target domain, and some features exist in both domains but work very differently. Novices frequently encounter difficulty with the limits of the metaphor or analogy: mistakes arise out of attempting to extract too much from it.

Context of Use:	Metaphor and notation
Justified by:	Empirical analysis of frequent novice errors
Examples:	Novices expect the computer to understand the meaning of a variable based on its name. For example, students expect the read statement to select values based on the names of the variables. With `READLN(odd, even)` and the input “2 3”, they expect the computer to read 3 into `odd` and 2 into `even` [Putnam 1989, Sleeman 1988]. When a variable is described as a box or a slate, users assume that it can hold more than one value at a time like the real-world counterparts. For example, students will expect a read statement to accept multiple values into the variables being read into. In `READLN(odd, even)` with the input “2 3 5 10”, students expect the computer to read 3 and 5 into `odd` and 2 and 10 into `even`. When they encounter these multiple-value variables in a subsequent conditional statement, students variously think that: • only the first value is used, • the comparison cannot be made, or • the program implicitly loops until the values in the variables have been consumed [Putnam 1989, Sleeman 1988]. When a variable is described as a box, users sometimes think that a sequence such as `a=2` followed by `b=a` means that `a` no longer holds the value `2`. When computers are anthropomorphized, novices tend to expect computers to be non-rigid like humans, and thus are not precise in describing a task [du Boulay 1989a]. When a program is described as a recipe, novices expect that it is ok to leave out certain details, the way “remove egg from its shell” is left out of most recipes that use eggs [du Boulay 1989a].
Cross References:	5-1. “Choose an Appropriate Metaphor” 5-3. “Consistency with External Knowledge”
References:	[du Boulay 1989a, Halasz 1982, Putnam 1989, Sleeman 1988]

5-3. Consistency with External Knowledge

Users bring to the programming task external knowledge that might interfere with correct understanding of the language. Most beginner programming errors can be interpreted as incorrect transfers from other representation/processing systems to the computer device [Bonar 1989, Mayer 1987]. Where their knowledge is lacking, it is common for novices to guess incorrect language syntax or semantics; these errors are indicators of transfers from other knowledge domains that are not compatible with the programming language [Hoc 1990].

Languages often use keywords that are loaded with meaning from natural language, and notation that is like math. While the mnemonic value of keywords is useful, it is important to beware of what is called the *human interpreter problem*, where reading the program in a natural language manner leads to an interpretation that is inconsistent with the correct meaning of the program [Bonar 1988b, Spohrer 1989a, Spohrer 1986b, Taylor 1990]. Students attribute to the machine the reasoning power of an average human [Sleeman 1988]. A large portion of bugs arise from inconsistency within the programming language or inconsistency between the programming language and the user's outside knowledge of the world or natural language. [Pea 1986] discusses language-independent conceptual bugs which affect the ways in which novices perceive the computing domain.

Context of Use:	Notation, metaphor
Justified by:	Empirical studies, analysis of frequent novice errors, expert opinion

Examples: There is a litany of inconsistencies between mathematical knowledge and the use of similar notation in programming languages like Basic, Pascal and C. Some examples lie in the use of variables [du Boulay 1989a, Putnam 1989]:

• `a=2` and `2=a` are not symmetric in assignment as they are in math;
• math does not use the concept of previous and next values of a variable, such as in the assignment statement `a=a+1`;
• math treats the identity of two symbols as permanent for the duration of the problem, while it is transient in programming, such as in the assignment statement `a=b`;
• math offers no hint about the direction of assignment – whether the value of `b` goes into `a` or vice versa – in the assignment statement `a=b`;
• initialization of variables is a concept that is foreign to math or counting.

In math, the symbol "+" can be used as a summation operator with an arbitrary number of arguments. In many programming languages, "+" is only a binary operator [Hoc 1990]. Notably, spreadsheets do not restrict the addition to be a binary operator [Lewis 1987].

As mentioned in "Avoid Subtle Distinctions in Syntax" on page 11, Pascal does not permit a comma to be used as punctuation in a number [Pane 1996].

In natural language, `then` and `and` are often used in the sense of "what next", unlike their use in Pascal or C: "I went to the shop and then I bought a paper"; "wash your hands and set the table" [Bonar 1989, du Boulay 1989a].

In Pascal, `Repeat` is used before the item(s) being repeated, but in natural language the opposite is usually true [Bonar 1988b, du Boulay 1989a].

In Logo, `STOP` causes the flow of control to return from the current procedure to the caller, but kids misinterpret this to mean that execution is halted completely [Kurland 1989].

One example of how programming does not support the natural way of expressing a task is sorting. When users are asked to sort a series of real boxes, inserting a box pushes the other boxes to make room. When dealing with arrays, this operation must be carried out explicitly by the programmer [Hoc 1990].

[Spohrer 1986b] found that many novice bugs occur when the user generates code in an order that is different than the built-in operator precedence in the language. The user expects that the program will execute in the order of generation, rather than according to the operator precedence rules of the language. This suggests that all operator precedence should be explicit rather than implicit. For example, expressions could be automatically parenthesized to show their evaluation order.

Cross References: 4-5. "Avoid Subtle Distinctions in Syntax"
5-1. "Choose an Appropriate Metaphor"
5-2. "Consistency with Metaphor"
5-6. "Naturalness of the Programming Language"

References: [Bonar 1989, Bonar 1988b, du Boulay 1989a, Hoc 1990, Kurland 1989, Lewis 1987, Mayer 1987, Pane 1996, Pea 1986, Putnam 1989, Sleeman 1988, Spohrer 1989a, Spohrer 1986b, Taylor 1990]

5-4. Closeness of Mapping

[Hoc 1990] describes programming as adaption of a plan from a familiar strategy to one that is compatible with the computer. This adaption is a refinement process that can lead to the detection of incompatibilities between the real-world plan and the computer's capabilities, or to a program solution that is not optimal. Similarly, [Green 1996] describes programming as mapping of operations in the problem domain into corresponding operations in the program domain. Thus it should be helpful to have a closeness of mapping between the task domain and program entities. [Merrill 1993] proposes as a design principle for all learning environments that the translation process from the student's internal plans to the solution's external representation be minimized. The extent to which the language facilitates this is called the *expressiveness* of the language [Bell 1991].

Users have difficulty understanding low-level primitives and how to compose them to form high-level components of a plan [Hoc 1990, Nardi 1993]. This is one of the great cognitive barriers to programming [Lewis 1987]. From a psychological point of view, the composition of these primitives into high-level operations is difficult for the following reasons [Lewis 1987]:

- synthesis is inherently hard, because a large number of possible combinations must be explored;
- since primitives are unrelated to the task, they are difficult to understand;
- for the same reason it is hard to see what combination of primitives will produce the correct task-related behavior;
- synthesis must be carried out with little immediate feedback; when feedback becomes available it is informative only about the behavior of the big assembly rather than about the many little choices that had to be made in putting it together;
- since the fundamental maneuver is replacement of "what is wanted" by "what to do", information about intent is not expressed directly and must be maintained separately, mentally or physically; and,
- it requires plan merging: constructions must be formed that carry out multiple purposes simultaneously.

In addition, in traditional programming languages [Lewis 1987]:

- synthesis results in high viscosity because primitives and rules of combination are complex;
- commonly used primitives enforce the inner world / outer world distinction in which data manipulable by the system cannot be manipulated by the user;
- reliance on sequence as the fundamental mode of combination of primitives requires users to specify much irrelevant information; and,
- it takes great care and discipline to make sure the code is comprehensible.

Instead, users should be permitted to formulate the problem using the objects, relationships, and processes of the problem domain [Lewis 1987]. [Fitter 1979] cites a principle of *revelation* in good notational schemes: the notation perceptually mimics the solution structure.

Educators often address these problems by choosing to begin teaching with a small subset of the language, and to incrementally expand the subset until the entire language is finally exposed [Brusilovsky 1994b]. Sometimes, special programming environments are built around a starting subset of the full language. Another approach is to invent a special mini-language for teaching which is not a proper subset of any full language. Usually these provide only the minimal set of primitives that are necessary to embody the desired programming concepts, and make all operations visible as concrete actions on the screen. [Brusilovsky 1994a] surveys a collection of mini-languages that are used as an introduction to programming (e.g. Karel the Robot [Pattis 1995]).

A related requirement is to have built-in types and abstractions that are appropriate for the domain [Cordy 1992]. [Nardi 1993] points out that successful end-user systems are task-specific and empower the user. They lack the power of general purpose programming languages, but also lack the steep learning curve. This suggests the creation of a task-specific programming language for every task domain, which is incompatible with the desire to make a general-purpose language and environment. One approach to this dilemma is to provide high-level, task-specific, user-extensible libraries for the desired domains, and to try to eliminate the need for novices to learn a lot of low-level primitives [Green 1996, Green 1991, Guzdial 1992, Mendelsohn 1990].

Context of Use: Notation and environment

Justified by: Observation of individual users.

Examples: Here are two examples of the refinement that is required to adapt a familiar strategy or plan to a programming solution:
• Making room in an array for insertion. The programmer must satisfy pre-conditions that do not exist in the original situation.
• using “+” as a binary operator, with initialization of an accumulator variable, instead of simply requesting summation. The user must decompose elementary actions into even more elementary ones [Hoc 1990, Lewis 1987].

The spreadsheet has a close mapping to the domain of numerical tables in accounting, and the common operations that are performed in that domain.

There is a great distance in mapping the built-in operations of a language like Pascal to tasks such as manipulating strings.

[Green 1996] uses cognitive dimensions to identify the following examples of closeness of mapping:
• poor: adding a vector in C or Basic; looping in LabView; success/failure in Prograph.
• good: electronics instrumentation in LabView; persistent variables in Prograph; locality of program plans in Prograph and LabView.

[Nardi 1993] claims that Hypercard’s language, HyperTalk, is too much like a conventional programming language and not close enough to the end-user’s needs.

High-level languages are more expressive than assembly languages. For example, it takes fewer statements to implement a loop in a high-level language.

Exceptions:	Even with high-level task-specific components, there is evidence that novices have difficulty assembling them into programs [Spohrer 1989a].
Cross References:	4-1. “Use Signalling to Highlight Important Information” 4-6. “Support Incremental Running and Testing with Immediate Feedback” 5-1. “Choose an Appropriate Metaphor” 5-3. “Consistency with External Knowledge” 5-14. “Cognitive Issues” 6-2. “Viscosity” 9-1. “Principle of Conciseness”
References:	[Bell 1991, Brusilovsky 1994a, Brusilovsky 1994b, Cordy 1992, Fitter 1979, Green 1996, Green 1991, Guzdial 1992, Hoc 1990, Lewis 1987, Mendelsohn 1990, Merrill 1993, Nardi 1993, Pattis 1995, Spohrer 1989a]

5-5. Support for Planning

Some researchers describe expert programming as an opportunistic activity [Green 1990b, Green 1996]. However, [Ball 1995] claims that experts actually are using sophisticated strategies to schedule and prioritize their activities. [Soloway 1984] and [Rist 1995] describe programming as the composition of plans or schemas. Plan usage is pervasive among novice programmers, and when they lack an appropriate plan they use pre-programming knowledge to fabricate one which may be buggy [Bonar 1989]. An additional source of many novice bugs is difficulty with plan composition, where the programmer is unable to anticipate all of the interdependencies when combining programming plans [Spohrer 1986a, Spohrer 1989a, Spohrer 1989b, Spohrer 1986b]. This interleaving of plans should be minimized [Green 1987].

The programming environment should allow programmers to work directly in plan terms [Mendelsohn 1990, Parker 1987]. Many aspects of program planning are difficult for novices: they do not know how to choose key components, they are stumped by a blank screen, and they need a process to guide their programming. An environment that assists in this planning process yielded improvements in novice program generation [Guzdial 1992]. Universe [Parker 1987], TEd [Ormerod 1996], and Bridge [Bonar 1987, Bonar 1990, Bonar 1988a] are similar systems. Bridge provides an intermediate representation for plans that avoids dispersing them: icons that fit together like jigsaw puzzles, with smaller icons for values and constants. [Corbett 1995] describes a tutoring environment where students are required to explicitly state subgoals, which most environments leave implicit. An empirical study showed that students reached mastery more quickly in this environment. Similarly, the GIL Lisp tutoring environment encourages users to make explicit explanations and predictions of ordinarily implicit behaviors and states, allows access to internal states that would otherwise be invisible, and has an on-screen representation of the structure of partial solutions to help students track their solution process [Merrill 1993, Merrill 1992, Reiser 1992]. These features lead to superior performance in both textual and diagrammatic programming environments [Merrill 1994].

Developing a suite of idioms, or plans, for solving small-scale goals are one of the important difficulties for novice programmers [du Boulay 1989a]. Selection of an appropriate plan from the suite is difficult [Scholtz 1993]. These skills should be taught explicitly [Perkins 1989], and required parts of a plan should be prompted by the syntax (e.g. initialization) [Green 1987].

The Prolog community introduced the concept of programming *techniques*, which are a small set of meta-plans that are language-dependent but domain-independent [Brna 1991]. [Bowles 1994] and [Ormerod 1996] describe systems that use techniques as a framework to support program editing, analyzing novice errors, in tracing and debugging, and in teaching programming skills.

Context of Use:	Environment
Justified by:	Empirical studies, expert opinion
Cross References:	5-3. “Consistency with External Knowledge” 5-4. “Closeness of Mapping”

References: [Ball 1995, Bonar 1987, Bonar 1990, Bonar 1989, Bonar 1988a, Bowles 1994, Brna 1991, Corbett 1995, du Boulay 1989a, Green 1990b, Green 1987, Green 1996, Guzdial 1992, Mendelsohn 1990, Merrill 1993, Merrill 1994, Merrill 1992, Ormerod 1996, Parker 1987, Perkins 1989, Reiser 1992, Rist 1995, Scholtz 1993, Soloway 1984, Spohrer 1986a, Spohrer 1989a, Spohrer 1989b, Spohrer 1986b]

5-6. Naturalness of the Programming Language

[Ledgard 1980] observed that natural language is better than a notational editing language for text editing, but [Curtis 1988] found that a textual pseudocode and graphical flowcharts were both better than natural language in program comprehension. When novices get stuck in their programming task, they rely on natural-language plans that they acquired before exposure to programming [Bonar 1989]. When the natural language plan is not compatible with the programming language, a bug will result. Several studies have found that when non-programmers are asked to write step-by-step informal natural language procedures, many different people use the same phrases to indicate looping structures and other standard programming tasks [Biermann 1983, Bonar 1986, Miller 1981]. Furthermore, students confuse phrases in natural language with the English keywords of a language like Pascal, and thus write their code as if it has the semantics of natural language [Bonar 1988b].

[Miller 1981] finds that nonprogrammers omit many actions from natural language problem solutions, thus relying on a human-like interpreter to fill in the missing details. For example, they use fewer control structures in written instructions than are required in actual program solutions. However, [Galotti 1985] was able to elicit nonprogrammers to use more control instructions by describing the instructee as a naive alien. One possible explanation is that novices use rules of cooperative conversation (see [Grice 1975]), expecting the computer to possess a modicum of common sense, and thus don't state the obvious. However, even when control structures are present, [Galotti 1985] found that instructions about iteration, or about what to do if a test condition is not met, are often vague or unspecified.

[Nardi 1993] points out serious problems with attempts to model programming languages after natural languages: the computer and programmer do not have the shared context that is present in human-human conversation, and it is not obvious where the limits of the computer's understanding are. Indeed there will be such limits as long as artificial intelligence has not been achieved.

However, novices are capable of learning and using programming languages that are not based on natural language, as long as they are task-specific and high-level [Nardi 1993]. Even these should support the ways that people naturally express problem solutions [Hoc 1983, Miller 1981]. When given a choice of programming methods to accomplish a task (e.g. sorting), novices tend to use the method they would use by hand, even if it is more complicated than another method [Hoc 1990, Nyuyen-Xuan 1987].

Context of Use:	Notation
Justified by:	Empirical studies and observations of individual users
Examples:	Many novice bugs are caused by a confusion of the correct choice of `AND` or `OR` in combining boolean tests. An example of this problem is when the programmer is checking for valid input from a menu of choices. Often novices will code `if (ch <> 'a') OR (ch <> 'b') OR (ch <> 'c')`, when the correct logical connector is `AND`. This problem appears to result from users' lack of an operational understanding of DeMorgan's Laws of logical identity, which describe the way the negation interacts with AND and OR

in propositions. Since the english language treats this situation informally ("if the choice is not a, b, or c"), the confusion is not unexpected [Spohrer 1986a, Spohrer 1986b].

One way to avoid the above problem in Pascal is to use a set, where `if (ch <> 'a') AND (ch <> 'b') AND (ch <> 'c')` becomes `if NOT (ch IN ['a', 'b', 'c']).` But there are two things to note about this syntax. First, the most natural way to express this in English ("if the **character** is **not in** the set...") leads novices to misplace the `NOT`, coding the expression illegally as `if (ch NOT IN ['a', 'b', 'c']).` Second, even when the `NOT` is positioned correctly in the expression, parenthesis are required for it to be evaluated correctly; such issues of operator precedence rarely surface in natural language.

Another confusion is that programming languages often use `OR` for inclusive-or, while natural languages use the word for exclusive-or.

Cross References: 5-3. "Consistency with External Knowledge"
5-4. "Closeness of Mapping"
5-5. "Support for Planning"

References: [Biermann 1983, Bonar 1986, Bonar 1989, Bonar 1988b, Curtis 1988, Galotti 1985, Grice 1975, Hoc 1983, Hoc 1990, Ledgard 1980, Miller 1981, Nardi 1993, Nyuyen-Xuan 1987, Spohrer 1986a, Spohrer 1986b]

5-7. Visual vs. Textual

[Myers 1990] presents a taxonomy of visual programming languages. There is a widespread tendency to expect visual languages to be superior to text for novice programming. [Green 1991] calls this graphical superlativism, and cites the following claims in favor of visual languages over textual languages: two-dimensional visual perception is more natural and efficient than reading text; it is easier to get an overview of program structure in visual systems; it is easier to read a visual program because purely syntactic devices are reduced; the number of variable names is reduced in visual programs; in visual systems, relationships between components are expressed by lines rather than symbols, making it easier to follow the routes; iconic representation of components may be easier to discriminate and recognize than textual names and symbolically-expressed relationships; extra information is conveyed by the spatial layout of the visual program (secondary notation). [Blackwell 1996] is a comprehensive survey of these claims from the visual programming literature. If the claims are true, the benefits may be particularly strong for novices.In a comparison of text-based and visual rapid-prototyping tools on a simple programming task, novice performance was closer to that of experts with the visual tools [Hasan 1996]. In Pursuit, a visual language based on a comic-strip metaphor was shown to be more effective than an equivalent textual language for novice generation of shell script programs [Modugno 1996].

Graphical superlativism was supported in [Cunniff 1987], where novices were able to recognize certain simple structures and to hand-execute short program segments more quickly and more accurately in a graphical language than in an equivalent textual language. However, there is a considerable amount of research indicating that graphical superlativism does not hold in larger more complex programs or for “deprogramming” tasks, where the novice must derive high-level goals and plans from the program text in order to fully understand and extend the program.

Diagrammatic notations are good only for certain purposes [Gilmore 1984]. [Green 1991] claims that formalisms based on control flow are linear with exceptions, so they are easily represented in a textual language; while formalisms based on data flow may be more appropriately represented in a visual language. However, [Curtis 1988] found that a flowchart representation was superior to textual pseudocode when the task involved tracing flow of control, but not for discerning high-level relationships. If there is any advantage of flowcharts over text it is at the detailed level, rather than at the overview level [Green 1992]. This may be because flowcharts are poor for modularity [Green 1990a]. There is little advantage in using flowcharts for supplementary documentation, although they are useful when they display knowledge that is difficult to extract from the program text [Shneiderman 1986, Shneiderman 1977]. [Atwood 1978] found that a textual program design language was better than a flowchart. Overall, graphical programs take longer to understand than textual ones [Green 1992, Green 1991]. [Moher 1993] compared program comprehension in graphical vs. textual representations and found that graphics were no better than text and sometimes considerably worse.

Flowcharts are of little help in debugging. They help to trace execution flow and localize the area where the bug is located, but are insufficient to identify the actual bug [Brooke 1980a, Brooke 1980b].

Visual languages are not more natural than text [Nardi 1993]. Most visual languages have high viscosity – they require a lot of effort in layout rearrangement when making changes; and they impose an extra burden on the user to guess ahead so that they format the program nicely and avoid future rearrangement [Green 1996]. Another problem with visual languages is their inefficient use of screen space [Nardi 1993].

[Nardi 1993] points out that spreadsheets are based on a textual language, and yet are very successful. However, spreadsheet programmers make use of visual imagery in planning manipulations, implying that mental images of program layout are an important resource [Saariluoma 1994].

Context of Use:	Notation, environment
Justified by:	Empirical studies, observations of individual users, expert opinion
Examples:	KidSim is a programming environment that attempts to be completely visual [Smith 1994]. It allows the user to construct "simulations" of agents in a two-dimensional grid. This was motivated by results in an earlier system named Playground [Fenton 1989], where children had great difficulty with a scripting language, even though a structure editor was provided to assist with syntactic correctness. However, in empirical tests of KidSim, the authors found that text was helpful in some situations. For example, adding the text "and if" at the beginning of each conditional expression made rules easier to understand [Cypher 1995]. While KidSim's mostly-visual approach appears to be productive in this limited domain, it is not at all clear that a useful general purpose programming environment could be completely visual. In visual languages, the graphics must be well-designed and recognizable [Green 1991]. Flow of data is very difficult to perceive in standard textual languages, but data flow languages make this information easily accessible [Green 1990a].
Cross References:	5-8. "Effectiveness of Notation is Task Dependent" 6-1. "Avoid Requiring Premature Commitment" 6-2. "Viscosity" 6-3. "Support Secondary Notation"
References:	[Atwood 1978, Blackwell 1996, Brooke 1980a, Brooke 1980b, Cunniff 1987, Curtis 1988, Cypher 1995, Fenton 1989, Gilmore 1984, Green 1990a, Green 1992, Green 1996, Green 1991, Hasan 1996, Modugno 1996, Moher 1993, Myers 1990, Nardi 1993, Saariluoma 1994, Shneiderman 1986, Shneiderman 1977, Smith 1994]

5-8. Effectiveness of Notation is Task Dependent

[Green 1992] describes the *match-mismatch* phenomenon, where different notations are better depending on the task. Performance is best when the structure of information sought matches the structure of the notation, and mismatch leads to poor performance. There are several examples of this in “Visual vs. Textual” on page 27. However, this effect may be diminished by other factors such as prior experience and the programmer’s dominant mental representation of the program [Good 1996]. While these analyses are based on program understanding, rather than program generation, there is a substantial amount of parsing and understanding during the coding process. For example, [Green 1987] proposes a model of programming where, due to working memory limitations, the programmer forgets some parts of the program that are already written, and is forced to *parse* them in order to recover their details. That research found that the parsing problem is more severe in Basic and Prolog than in Pascal. The task-dependent effectiveness of notation suggests that a programming environment that supports multiple (e.g. visual and textual) representations of the program might be a fruitful endeavor.

Context of Use:	Notation, environment
Justified by:	Empirical studies
Examples:	Traditional structured languages with nested conditionals support forward analysis of *sequence* information, e.g. “given these inputs, what is the result”. Declarative languages support backward analysis of *circumstantial* information, e.g. “given this output, what must the inputs have been” [Green 1992]. Features that facilitate the parsing task may work contrary to the code generation task. For example, the features that make Pascal easier to parse into plan structures may be the very features that inhibit linear generation of code. In many other notations it seems easier to develop code than to recover its meaning [Green 1990b].
Examples:	5-4. “Closeness of Mapping” 5-5. “Support for Planning” 5-7. “Visual vs. Textual” 8-1. “Minimize Working Memory Load”
References:	[Good 1996, Green 1990b, Green 1987, Green 1992]

5-9. Control Structures

One area difference between spreadsheets and many other programming languages is control structures. The lack of control structures in spreadsheets is an advantage [Nardi 1993]. [Lewis 1987] also points this out, and claims that spreadsheets are the model of the future because they allow the learner to suppress the *inner world* of programming, the world of variable declarations, loops, and I/O. Relationships among variables can be set up declaratively, and the system will maintain consistency. [Wandke 1988] cites a dramatic increase in cognitive effort when using control structures, leading to a reluctance to define macros for repetitive tasks even if it would dramatically reduce the number of keystrokes required to perform a task.

[Rogalski 1990] found that:

- high-level control structures are more difficult to express than the “goto” or “jump” style of control, but the latter is more difficult for managing complex control flow correctly;
- control structures that use positive alternatives present fewer difficulties than negative ones (e.g. `repeat until X` is easier to understand than `while not X`, especially if `X` is a compound expression);
- difficulty increases with depth of nesting; and
- students with a better background in math learn new control structures faster.

[Sime 1977a] and [Sime 1977b] confirm that high-level control structures help the novice to manage flow of control, and also found that a structure editor assisted novices in generating correct nested control structures.

Many novice errors with control structures can actually be attributed to misconception of variables. Describing a variable as a name or an address is the first step toward fixing this, although a more complex model is required when variables occur in iterative or recursive programs in imperative languages: the variable is no longer an address with a value, but needs to be seen as a function of execution, or a sequence of values [Samurçay 1989].

Indeed, most introductory programming textbooks focus a great deal of attention on the use and understanding of control structures, suggesting that details about how control structures work is an area of great difficulty for novices. However, in a study of high-frequency bugs, [Spohrer 1986a, Spohrer 1989a, Spohrer 1986b] found that only about one-third of bugs arise from novice misunderstandings of control structures. [Arblaster 1979] found that any type of structure is better than no structure at all, and that hierarchical structuring is not better than other types of structuring.

Context of Use:	Notation
Justified by:	Empirical studies, observations of individual users
Examples:	With IF statements, students make the following errors [Putnam 1989, Sleeman 1988]: • expect the program to halt with an error if the condition on the IF statement is false and there is no ELSE clause; • expect both the THEN and ELSE clauses to be executed;

• expect the THEN clause to execute whether or not the condition is true;
• treat a statement after an ELSE-less IF statement as though it is the ELSE clause.

Exceptions:	Prolog is an attempt at suppressing the inner world of programming, and is notoriously difficult for novices.
Cross References:	5-3. “Consistency with External Knowledge” 5-4. “Closeness of Mapping” 5-6. “Naturalness of the Programming Language” 5-10. “Loop and Recursion Control Structures”
References:	[Arblaster 1979, Lewis 1987, Nardi 1993, Putnam 1989, Rogalski 1990, Samurçay 1989, Sime 1977a, Sime 1977b, Sleeman 1988, Spohrer 1986a, Spohrer 1989a, Spohrer 1986b, Wandke 1988]

5-10. Loop and Recursion Control Structures

A common area of difficulty for novices is looping. Part of this can be attributed to an inability to generalize, which is evidenced by a tendency for novices to make a list of repeated instructions instead of coding a loop [Hoc 1989, Onorato 1986], or to an inability to develop an adequate mental model of the looping structure [Kessler 1989, Pirolli 1985]. Sometimes the bugs in novices' mental models are subtle and difficult to detect [Kahney 1989]. However, a large part of the difficulty of loops may be overcome by designing the looping control structure(s) carefully.

Pascal provides a `while` loop, where the looping condition is checked at the top of the loop (top-exit); and a `repeat` loop, where the looping condition is checked at the bottom (bottom-exit). Other possibilities are: a loop that can exit from a check in the middle of the loop (middle-exit); or a loop that exits from anywhere as soon as the condition fails (daemon-exit). In describing a plan, novices use a bottom-exit strategy when it seems easier, but then revert to a middle-exit strategy for all other situations [Wu 1991]. [Rogalski 1990] reinforces this with the finding that the top-exit strategy is more difficult than the bottom-exit strategy, hypothesizing that novices have difficulty representing and expressing a condition about an object that they have not yet operated on. Pascal does not provide a middle-exit loop control structure, so novices are forced to adapt their middle-exit plan to a while or repeat loop when they write the code; causing performance to suffer. [Soloway 1989] found that providing a middle-exit control structure would increase accuracy and would not interfere with program readability.

Construction and expression of the loop invariant is an important component of an iterative plan. But, in spontaneous verbal plans novices tend to base their models of loops on representing a succession of actions, rather than on representing the invariant relationships among variables. Even when asked explicitly, novices have difficulty specifying a loop invariant. Also, novices tend to use different names at each step of the iteration to label the same functional variable, and they do not spontaneously elaborate an exit condition [Rogalski 1990].

Beginners tend to use an iterative model for recursion. This model is compatible with tail-recursion, but fails in the more general case [Kurland 1989, Rogalski 1990]. For this reason, [Rogalski 1990] recommends that recursion be taught before iteration.

However, in a more detailed scrutiny of novice models of recursion, [Kahney 1989] found that while a large number of novices (> 50%) appear to have a iterative model, in fact most of them actually have no consistent model at all. [Kessler 1989] analyzed transfer between iteration and recursion, and found positive transfer from iteration to recursion, but no transfer from recursion to iteration. They conclude by recommending that iteration be taught before recursion.

Context of Use:	Notation
Justified by:	Empirical studies, observations of individual users
Examples:	Many students expect a while loop to terminate as soon as its condition fails (daemon-exit) rather than waiting until the condition is tested at the "top of the loop" [Bonar 1989, Sleeman 1988].

Students often make the following errors related to loops [Putnam 1989, Sleeman 1988]:
• interpret a statement that is adjacent to (after) a loop as though it is contained within it;
• execute only the last statement inside a loop multiple times;
• attribute looping behavior to a begin-end block;
• believe that a variable holds more than one value and thus treat a conditional statement as a loop;
• believe that the for-loop control variable does not have a value inside the loop, or that it is acceptable to change its value inside the loop; and,
• interpreted the range of values on the for-loop control variable as a constraint on the values of a different variable inside the loop.

Cross References:	5-3. “Consistency with External Knowledge” 5-4. “Closeness of Mapping” 5-5. “Support for Planning” 5-6. “Naturalness of the Programming Language” 5-8. “Effectiveness of Notation is Task Dependent” 5-9. “Control Structures”
References:	[Bonar 1989, Hoc 1989, Kahney 1989, Kessler 1989, Kurland 1989, Onorato 1986, Pirolli 1985, Putnam 1989, Rogalski 1990, Sleeman 1988, Soloway 1989, Wu 1991]

5-11. Support Direct Manipulation and Definition by Example

Some languages, such as cT [Sherwood 1988], and Turing [Cordy 1992] permit the user to interactively define the objects that will be manipulated by the program, and then to embed them directly in the program. In textual languages, this saves a lot of effort because writing a program to define these objects would be tedious [Lewis 1987]. Hypercard and Visual Basic invert the process, by having the programmer sketch the graphics and then attach programs to the graphics [Green 1990a]. However, when these methods are used, there is often a serious problem with the distinction between *use* and *mention* of the object (see examples below) [Smith 1992]. This distinction should be avoided [Lewis 1987, Smith 1992]. One way to achieve this is by modeling the system after the physical world, with the following implications: "a) [the system] must have object-oriented semantics, so that objects can directly present their own state and behavior, b) it must be dynamic, allowing incremental changes from the interface, c) it must be visual, so that all capabilities of the language are present in the interface, and d) it must avoid enforcing any kind of [distinction between use and mention of the object] [Smith 1992]."

Context of Use:	Notation, environment and metaphor
Justified by:	Expert opinion
Examples:	The distinction between use and mention can be seen by considering a button in a direct-manipulation interface. Pressing the button is a *use* – it causes the button to perform its action. However, when moving or resizing the button, clicking on it should not perform the action – this is *mention*. Often handles are provided for mention tasks, but this provides only one level – you can use the handles but you can not mention them [Smith 1992]. In BASIC, the distinction between use and mention comes up in `PRINT "Q:"; Q`, where first the name of the variable is printed, then its value. Students misinterpret `"Q:"` as [Putnam 1989]: • a comment that is not executed; • the same as the unquoted use of the variable; or, • referring to the very first value that was ever stored in the variable.
Cross References:	5-12. "Choosing a Paradigm"
References:	[Cordy 1992, Green 1990a, Lewis 1987, Putnam 1989, Sherwood 1988, Smith 1992]

5-12. Choosing a Paradigm

Most of the research in this report studies the classical imperative paradigm of computing where the user is in control of a single thread of execution. There are many other programming paradigms, including object-oriented, event-based, functional, programming by demonstration, graphical rewrite rules, autonomous agents, data flow, production system or rule-based programming, logic programming, parallel programming, etc. Some of these paradigms have achieved widespread use in research and professional software development communities. In other cases, only experimental systems have been developed to test a paradigm or a mixture of several paradigms. From a usability point of view, there is much room for investigation of this area, to determine the strengths and weaknesses of the various paradigms, and how the best features of multiple paradigms might be mixed into an effective novice programming system. Also, when introducing a new paradigm to people with some programming experience, there is a risk of negative transfer from the prior paradigm [Mendelsohn 1990, Siddiqi 1996, Wiedenbeck 1996].

Object oriented programming is widely advocated as a paradigm for quickly building programs from reusable components. However, this idea, when carried too far, has been found to have a detrimental impact on performance. Object oriented programming may have benefits for up to three levels of class hierarchy, but deeper hierarchies have been found to be difficult to work with [Daly 1996]. A cognitive phenomena called *conceptual entropy* may be the root cause of this problem [Dvorak 1994]. Object-oriented design is not necessarily a “natural” design method. Programmers have difficulty deciding which logical entities should be represented as objects and which as attributes of the objects [Détienne 1990]. Perhaps careful construction of the programming environment could assist users with these problems and limit the use of object-oriented programming to situations where it will be helpful.

Multiple inheritance may have additional advantages over single-inheritance object-oriented programming, but surveys of experienced programmers reveal mixed opinions. Some argue that it produces a more complex design, is more difficult to test, is more difficult to reuse, and is easy to abuse; while others argue that it produces a more appropriate design, and facilitates reuse and maintenance. There is little doubt that it adds complexity. An additional concern is that multiple inheritance is often implemented where it is inappropriate, resulting in object-oriented software that is more complex than is necessary. This leads to a recommendation to use multiple inheritance only where there is a strong case for using it [Daly 1995a, Daly 1995b].

Viewing procedures as “object-like entities” offers semantic power and syntactic elegance, but novice programmers view them with few “object-like” properties [Eisenberg 1987]. The authors suggest ways to improve instruction and the environment to overcome this. Note that this object-oriented view of procedures in a functional language is different than pure objects in object-oriented programming languages. [Rist 1996] describes the procedures in a functional language as encapsulations of goals with their plans; and points out that this encapsulation is orthogonal to the encapsulation of data with operations in an object-oriented language. Further, he states that goals and plans are not well captured in an object-oriented language.

Context of Use:	Environment
Justified by:	Empirical studies, informal observation of users, expert opinion.

Examples: KidSim [Cypher 1995, Smith 1995] uses graphical rewrite rules and programming by demonstration to provide an end-user programming system for symbolic simulations of agents in a two-dimensional grid. In KidSim, user testing revealed that arbitrarily deep hierarchies caused difficulties for children, so a one-level simple inheritance scheme was adopted [Smith 1994].

AgentSheets is a paradigm that consists of a large number of autonomous communicating agents organized in a grid [Repenning 1993]. This spatial metaphor supports the problem solving process, which includes creating and changing external representations of the problem as well as exploring problem spaces [Repenning 1994].

ToonTalk uses video-game animation in a city populated by robots as the means of creating and viewing programs [Kahn 1996]. It uses programming by example, but instead of automatic induction or learning, requires the user to introduce generality by removing details from the example.

LiveWorld is a programming environment based on rule-like agents that are responsive to their environment [Travers 1994]. It uses a novel object system that makes computational objects, such as behavioral rules, concrete and accessible like graphical objects.

ShopTalk enhances direct manipulation with natural language text, in order to overcome some of the limitations of direct manipulation in specifying objects and actions [Cohen 1989].

Cross References: 5-1. “Choose an Appropriate Metaphor”
5-3. “Consistency with External Knowledge”
5-5. “Support for Planning”
5-7. “Visual vs. Textual”
5-11. “Support Direct Manipulation and Definition by Example”
8-1. “Minimize Working Memory Load”

References: [Cohen 1989, Cypher 1995, Daly 1996, Daly 1995a, Daly 1995b, Détienne 1990, Dvorak 1994, Eisenberg 1987, Kahn 1996, Mendelsohn 1990, Repenning 1993, Repenning 1994, Rist 1996, Siddiqi 1996, Smith 1995, Smith 1994, Travers 1994, Wiedenbeck 1996]

5-13. Modularity and Abstraction

Abstraction of functionality into modules is a powerful programming concept. It can promote information hiding, reduce the amount of code that must be understood in detail, and provide a suite of primitives that can be composed to implement new functionality. When programmers understand code at an abstract level they are more likely to reuse that code in other appropriate places, and that reuse is more likely to be by invoking the code (making a procedure call), which is a more efficient form of reuse than making a copy of the code in the new context [Hoadley 1996]. But novices are not ready to use the abstraction tools which are emphasized by modern languages [Mendelsohn 1990].

Modular programming is often taught through a discipline known as top-down design, where the program is first described at an abstract, high level, then refined into a modular hierarchy [Wirth 1983]. However, hierarchically designed programs are not always easy to develop and comprehend [Curtis 1989, Perkins 1989]. Top-down strategies are difficult for novices because their spontaneous strategies or plans are based on concrete mental execution; action-oriented rather than object-oriented [Rogalski 1990]. And, taking modularity and abstraction to the extreme can interfere with locality and visibility (see "Locality and Hidden Dependencies" on page 8) [Green 1996]. One problem with comprehension of modular programs is that novices do not yet have the expert strategy of reading a program in a top-down, order-of-execution manner – instead they read the program like a book [Gellenbeck 1991b, Jeffries 1982, Wiedenbeck 1986b]. Novices focus on the very literal and concrete, rather than the abstract, hierarchical, general view used by experts [Onorato 1986]. An environment that helps the novice to read and understand the program in a modular fashion and to identify meaningful sections may alleviate this problem. Novices using such an environment is described make very effective use of modularity [Miller 1994].

A modular program can be modified faster than an equivalent non-modular program when at least one of the following conditions hold [Korson 1986]:

- modularity has been used to promote information hiding, which localizes changes;
- existing modules provide a suite of useful generic functions that can be composed to implement new functionality; or,
- the modification requires an extensive understanding and modification of the existing code.

However, modularity did not help in other cases, such as adding a new feature to a program.

Another kind of abstract thinking that is difficult for novices is writing a general solution to a problem rather than a solution that is specific to the situation (e.g. a program that sorts a list). This requires students to make a shift from value processing to variable processing; and to elaborate some of the control decisions that are not consciously made in solving a specific problem [Hoc 1990]. Also, a lack of abstraction is evident in the tendency of novices to code loops as sequential actions, *unrolled* [Hoc 1989]. Examples and analogies play an important role in learning and understanding, and explanations help learners to generalize the examples [Lewis 1987].

Context of Use:	Environment, notation, and instruction
Justified by:	Empirical studies, observations of users, expert opinion.

Examples:	The programming environment can support modularity and reduce its negative attributes by providing multiple views, such as call graph, outline, and class hierarchy views [Miller 1994, Roberts 1988]. Spreadsheets do not support modularity and abstraction very well. Abstraction is presented at a fixed level and hierarchical representations are not supported [Lewis 1987]. KidSim uses programming by demonstration to address the concrete vs. general abstraction problem. It allows users to create an abstract rule by demonstrating what that rule should do in a specific (concrete) situation. The system then automatically generalizes the specific rule into a more abstract one [Cypher 1995].
Cross References:	4-3. “Locality and Hidden Dependencies” 5-3. “Consistency with External Knowledge” 5-5. “Support for Planning” 5-6. “Naturalness of the Programming Language”
References:	[Curtis 1989, Cypher 1995, Gellenbeck 1991b, Green 1996, Hoadley 1996, Hoc 1989, Hoc 1990, Jeffries 1982, Korson 1986, Lewis 1987, Mendelsohn 1990, Miller 1994, Onorato 1986, Perkins 1989, Roberts 1988, Rogalski 1990, Wiedenbeck 1986b, Wirth 1983]

5-14. Cognitive Issues

There are several findings about cognitive aspects of programming which should be kept in mind.

Context of Use:	Environment, notation and instruction
Justified by:	Empirical studies, observations of individual users, expert opinion.

Examples:

Meta-cognitive strategies are used to select an operation when several different ones could be applied. They play an important role in effective programming. Users may benefit if the system either suggests a particular strategy or plan, or minimizes the number of situations where an operation must be selected from among multiple choices [Bell 1991]. However, the latter suggestion may conflict, for example, with a desire to support multiple looping strategies. As mentioned above (see “Avoid Subtle Distinctions in Syntax” on page 11), more planning is required when there are many different legal solutions to a problem [Gray 1987]. When more planning is required, it is more likely to cause backtracking, where the programmer is taken away from the progressive activity of coding. The programming environment could use an expert system to guide the novice in this decision making [Návrat 1993, Návrat 1996].

Experts are more likely than novices to develop complex high-level representations of the program. This happens a top-down fashion when looking at large units and fitting large pieces together, and in a bottom-up fashion when identifying chunks of code and deducing how they fit into the goal hierarchy [Boehm-Davis 1996]. Anything that the environment can do to assist novices in forming high-level representations may be helpful.

As mentioned above (see “Closeness of Mapping” on page 20), an abundance of low-level primitives is one of the great cognitive barriers to programming [Lewis 1987].

Another great difficulty of programming is that it is much more precision-intensive than other subjects [Perkins 1988].

Students do not spontaneously develop general problem-solving skills from programming experiences (transfer effect) [Olson 1987]. There is only weak evidence that programming is a medium that creates new ways of dealing with existing knowledge, and no evidence that through programming practice, children develop cognitive skills that are identifiable and transferable to other situations such as: analogical and temporal reasoning, mathematical operations, planning of action, error correction, and development of logical and spatial operations [Mendelsohn 1990]. Many studies of children using LOGO confirm this absence of transfer [Dalby 1985, Nickerson 1985, Pea 1984, Perkins 1985]. However, when a skill is explicitly taught in the framework of programming with an emphasis on transfer,

transfer to other domains (e.g. debugging skills, expository writing, spatial cognition skills, planning) can occur [Carver 1988, Carver 1987, Goldenson 1996, Lee 1993, Lehrer 1988, Mayer 1987]. In addition to acquiring the programming skill, the student must also recognize the relevance of the acquired knowledge to the new domain [Fay 1988]. Successful demonstrations of transfer result from effective teacher mediation rather than simply exposure to programming [Clements 1993, Mayer 1988].

Users who are averse to risk are more successful in structure editor based programming environments, where syntax errors are not possible [Neal 1987].

Acquiring the syntax of a language is difficult for children. The cognitive demands of getting the syntax right thus interfere with the task of getting the semantics right. This phenomenon has been observed in other domains such as writing. Systems that attempt to relieve the syntactic burden on the student, such as a flexible language syntax or a structure editor, should permit the student to devote more resources to the semantics of the programming task [Fay 1988].

Although computer programming is often characterized as a set of non-interacting subtasks (e.g. specification, design, planning, coding, testing, debugging, documenting, etc.), in practice there are substantial interactions among them. “This is a fundamental feature of programming [that arises] from the cognitive characteristics of the subtasks [and] the high uncertainty in programming environments... [Pennington 1990].”

Knowledge and problem-solving strategies work together in programming. Fragile knowledge (partial, hard to access, or misused) can be compensated by effective strategies. Exploratory use of the language and other elementary problem solving strategies should be explicitly taught [Perkins 1986]. In a course that helped students to form a clear mental model of the computer, provided them with heuristics to helm the conceptualize and organize the elements of the programming language, and equipped them with problem solving tools and strategies, students performed better that a control group that did not use these techniques [Perkins 1988].

Cross References:
4-5. “Avoid Subtle Distinctions in Syntax”
4-6. “Support Incremental Running and Testing with Immediate Feedback”
5-4. “Closeness of Mapping”
5-5. “Support for Planning”
5-9. “Control Structures”
5-10. “Loop and Recursion Control Structures”
5-13. “Modularity and Abstraction”
8-1. “Minimize Working Memory Load”

References: [Bell 1991, Boehm-Davis 1996, Carver 1988, Carver 1987, Clements 1993, Dalby 1985, Fay 1988, Goldenson 1996, Gray 1987, Lee 1993, Lehrer 1988, Lewis 1987, Mayer 1988, Mayer 1987, Mendelsohn 1990, Návrat 1993, Návrat 1996, Neal 1987, Nickerson 1985, Olson 1987, Pea 1984, Pennington 1990, Perkins 1985, Perkins 1986, Perkins 1988]

5-15. Instructional Design

Here are some educational observations that can be used to guide the design of the environment.

Context of Use: Instruction

Justified by: Empirical studies, observations of individual users, expert opinion

References: [Davis 1993] presents a model of learning to program where students develop a system of rules (some incorrect) about programming, and apply them either consistently or intermittently. An effective intervention may be to encourage students to reflect on their rules and relate them to feedback received from the computer. The environment could be built to support this reflective phase of knowledge acquisition.

One of the areas of difficulty for novices is orientation to programming: finding out what programming is for, what problems can be tackled, and what the advantages are. Another is coming to terms with the duality of the computer as the manager of program creation vs. the machine executing the program. The latter problem is exacerbated by early examples that print text to the screen, where the output of the program looks almost exactly like the program itself [du Boulay 1989a]. Perhaps better examples could be chosen.

[du Boulay 1989a] also points out that another crucial concept that is confusing for novices is the computer's rigidity, which can be confused by the way the computer is described (see "Consistency with Metaphor" on page 17 for more details).

Teachers should [Hoc 1990]:
• design situations that require learners to solve programs in a general way, rather than specific to a particular situation;
• be aware of the kinds of real-world solutions that students may attempt to transfer to the programming task, so that they may be reinforced or suppressed as appropriate; and,
• provide direct, immediate feedback about the way that the learner adapts the real-world plan to the programming situation.

[Papert 1980] uses computing as a medium, a support for elaborating environments in which the child constructs his/her own knowledge. Discovery and exploration are emphasized over a structured curriculum. [Heller 1986] investigated the difference between a structured Logo curriculum and a experiential Logo learning environment, and found that the structured environment is better for obtaining thorough knowledge in a limited time, while the experiential environment is better for a broadly framed "growth experience". [Littlefield 1988] confirmed that a structured approach was better than an unstructured approach for language mastery, and also found that a mediated-teaching approach produced mastery levels equivalent to the structured approached. However, the mediated approach resulted in superior performance on near-transfer tasks than the structure approach, which in turn was superior to the unstructured approach.

Cross References: 4-6. “Support Incremental Running and Testing with Immediate Feedback”
5-2. “Consistency with Metaphor”
5-3. “Consistency with External Knowledge”
5-5. “Support for Planning”
5-14. “Cognitive Issues”

References: [Davis 1993, du Boulay 1989a, Heller 1986, Hoc 1990, Littlefield 1988, Papert 1980]

6. User Control and Freedom

“Users often choose system functions by mistake and will need a clearly marked “emergency exit” to leave the unwanted state without having to go through an extended dialog. Support undo and redo [Nielsen 1994].”

The research in this section extends the above interpretation of “user control and freedom” to include issues of flexibility in the programming system. Giving control and freedom to users will make it easier for them to accomplish the programming task in their own ways.

6-1. Avoid Requiring Premature Commitment

A modern view of programming, where programs are developed in an exploratory, opportunistic, incremental fashion, requires that the programming system allow programmers to postpone decisions until they are ready for them. The system should avoid situations where correct generation of a piece of code requires subsequent pieces to be known. For example, tidy layout in some graphical programming languages requires the user to anticipate the space requirements of parts of the program that are not yet written [Green 1990b, Green 1987, Green 1996].

Context of Use:	Environment, notation
Justified by:	Informal observation of users, expert opinion
Examples:	Early in the code generation task, structure editors are good at reducing the problem of premature commitment, because they allow the programmer to leave holes where details can be filled in later [Green 1996]. However, later in the programming task, when modification is more prevalent than generation of new code, structure editors can make it difficult to back out of an earlier choice, thus exhibiting premature commitment [Green 1990b]. Structure editors that permit editing the program textually as well as structurally can avoid the latter problem (e.g. electric-C mode in Emacs, or the MacGnome structure editors [Miller 1994]). [Green 1996] uses cognitive dimensions to identify the following examples of premature commitment: • Text based languages require premature commitment at the structural level – they encourage development of the program in a linear order by adding text at the growing tip, rather than leaving holes to be filled in later. • Visual languages are less demanding about development order, but as mentioned above they require guess-ahead at the layout level – without careful lookahead, layout will become messy and is difficult to correct.
Cross References:	4-6. “Support Incremental Running and Testing with Immediate Feedback” 5-5. “Support for Planning” 5-7. “Visual vs. Textual”
References:	[Green 1990b, Green 1987, Green 1996, Miller 1994]

6-2. Viscosity

Viscosity is a measure of how much effort is required to make a small change to the program [Green 1996]. The final text of a program rarely corresponds to the order it was generated; therefore revision is intrinsic in programming [Davies 1996]. [Fitter 1979] cites a principle of *revisability* in good notational schemes: a system should make it easy to revise existing code in a program. Independent of other factors, it is desirable to minimize viscosity. As mentioned above (see "Avoid Requiring Premature Commitment" on page 45), a modern view of programming, where programs are developed in an exploratory, opportunistic, incremental fashion, requires that the programming system allow easy additions or changes to existing code [Green 1990b, Green 1996].

As mentioned above (see "Locality and Hidden Dependencies" on page 8), [Lewis 1987] proposes replacing *programming by synthesis* with *programming by modification*, where a library of examples is provided from which the programmer chooses an appropriate one for a starting point, identifies needed modifications, then modifies it to suit the current need. This emphasis on modification of existing code elevates viscosity as an important factor. However, [Nardi 1993] claims that programming by modification is not any more natural than other approaches to programming.

Context of Use:	Notation
Justified by:	Empirical studies, observation of individual users
Examples:	Visual programming languages have a high viscosity. [Green 1996] measured an order of magnitude increase in time to make a small change in the LabView visual programming language than in a textual language. In order for the programmer to preserve readability, LabView required many "enabling" steps before the goal could be addressed. Adding a line of code to a Basic program may require many lines to be renumbered, resulting in high viscosity. Sometimes a small change to a program has a domino effect, requiring many additional non-local changes. In object-oriented programming there is less of this cascading of changes, resulting in reduced viscosity [Green 1990b]. Spreadsheets have low viscosity: changes to a part are not constrained by other parts [Lewis 1987]. However, they are difficult to redesign when attempting to adapt another user's spreadsheet to one's own task.
Cross References:	4-3. "Locality and Hidden Dependencies" 4-6. "Support Incremental Running and Testing with Immediate Feedback" 5-7. "Visual vs. Textual" 5-12. "Choosing a Paradigm" 6-1. "Avoid Requiring Premature Commitment" 6-3. "Support Secondary Notation"

References:	[Davies 1996, Fitter 1979, Green 1990b, Green 1987, Green 1996, Lewis 1987, Nardi 1993]

6-3. Support Secondary Notation

Secondary notation is information that is embedded in the program text that is not part of the syntactic structure that is meaningful to the system [Petre 1992]. There is a huge amount of important information that is not actually part of the program [Berlin 1993, Green 1995]. Experts use comments, white space, and typography to carry semantic domain knowledge about the program, and at least some of these benefit novices too [Gilmore 1986, Payne 1984, Riecken 1991]. The most common kind of secondary notation in textual languages is spatial layout through indentation and alignment [Gellenbeck 1991a]. However, the effect of spatial layout is less important than a good choice of notation [Curtis 1988]. Spreadsheet users make use of visual imagery in planning manipulations, implying that mental images of layout is important [Saariluoma 1994].

Indentation is the principle means of spatial representation in Pascal and other textual languages [Cunniff 1987]. When used consistently, indentation has been shown to improve comprehension [Cunniff 1989, Kesler 1984, Miara 1983, Vessey 1984]. Note that when indentation is used correctly and consistently, it is redundant with curly braces in C [Baecker 1986]. However, indentation can interfere with locality by breaking up semantic units in favor of syntactic units [Shneiderman 1986]. Color can supplement indentation in assisting the user to understand control flow [Van Laar 1989].

Textual languages allow a substantial amount of secondary notation, while visual languages obscure attempts to use grouping as a secondary notation [Green 1996]. As mentioned above (see “Use Signalling to Highlight Important Information” on page 6), secondary notation should be used to improve access to information that is needed but obscured [Green 1990b]. The value of secondary notation implies that the environment must facilitate it, by allowing the user flexibility on these details, and perhaps by explicitly supporting the recommendations in [Gellenbeck 1991a]: modules should be preceded by 1-3 lines of preview statements; and module names should be short, mnemonic, derived from the preview statement, and begin with a verb.

Context of Use:	Notation and environment
Justified by:	Empirical studies, expert opinion
Examples:	Comments, white space, and typography are examples of secondary notation that should be supported in the programming system. Meaningful variable names aid comprehension [Gellenbeck 1991b]. In a study using graphical programs, novices were not able to extract the information in secondary notation that would have assisted their comprehension [Green 1991]. Training is required in order for them to exploit secondary notation. Python uses indentation for lexical scoping, eliminating the redundant use of braces or begin-blocks [Watters 1995].

Cross References:	4-1. “Use Signalling to Highlight Important Information” 4-3. “Locality and Hidden Dependencies” 5-5. “Support for Planning” 5-7. “Visual vs. Textual” 5-9. “Control Structures” 5-10. “Loop and Recursion Control Structures” 5-15. “Instructional Design” 6-1. “Avoid Requiring Premature Commitment” May conflict with objectives of 4-4. “Beware of Misleading Appearances”
References:	[Baecker 1986, Berlin 1993, Cunniff 1987, Cunniff 1989, Curtis 1988, Gellenbeck 1991a, Gellenbeck 1991b, Gilmore 1986, Green 1990b, Green 1995, Green 1996, Green 1991, Kesler 1984, Miara 1983, Payne 1984, Petre 1992, Riecken 1991, Saariluoma 1994, Shneiderman 1986, Van Laar 1989, Vessey 1984, Watters 1995]

7. Consistency and Standards

“Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions [Nielsen 1994].”

The research in this section investigates internal consistency of the programming language. Consistency with the outside world is discussed in the section “Match Between System and the Real World” beginning on page 14. As mentioned there, [Payne 1986] describes a formal method for assessing both kinds of consistency.

7-1. Consistency in Notation

The language should be self-consistent, and its rules should be uniform [du Boulay 1989b]. It should abide by any suggestions that can be derived from other places in the language, so that learners can infer one part of the language from another part [Green 1996]. It should minimize exceptions so that generalization of rules results in correct notation [du Boulay 1989a]. Conditionals with extra cues help both novices and experts [Sime 1977c]. As mentioned above (see "Avoid Subtle Distinctions in Syntax" on page 11), novices get confused when there are two different syntaxes to accomplish the same effect [Eisenberg 1987].

The meanings of keywords should be context independent. Novices are less likely than experts to organize language keywords in a meaningful way. Instead, they tend to focus on surface features [McKeithen 1981]. This argues against gratuitous re-use of the same keyword for different concepts.

Context of Use:	Notation
Justified by:	Empirical studies, analysis of frequent novice errors, expert opinion
Examples:	Syntactic consistency can be enforced by having all control structure take the form `X ... end X` [Cordy 1992].

The following are some examples of violations of consistency in notation:

In the Pascal language, all procedures end with semi-colon except the very last one which ends with a period, all statements end with a semicolon unless preceding an `else` or `until` statement, and lists of parameters are separated by semicolons in the declaration of a procedure and by commas in the call to the procedure.

In C, there are three different kinds of braces used in various situations: `{}`, `()`, and `[]`. They are not interchangeable, and sometimes they are mixed in an inconsistent way (see "Avoid Subtle Distinctions in Syntax" on page 11).

Pascal is not syntactically consistent in its control structures. Most of them terminate with `end`, but the `repeat` statement does not.

Many novice bugs arise out of inconsistency in the way the language treats different data types [Spohrer 1986b]. For example, white space is treated differently when reading into a numeric variable than it is when reading into a character variable [Spohrer 1986a].

The keyword `static` in C++ has many different meanings depending on context.

Exceptions:	Even when a language embraces consistency, students may not appreciate it. For example, in LISP the rules for evaluation are consistent, yet students adopt a series of special-case rules for certain control structures such as `cond`.
Cross References:	4-4. “Beware of Misleading Appearances” 4-5. “Avoid Subtle Distinctions in Syntax” 5-2. “Consistency with Metaphor” 5-3. “Consistency with External Knowledge”
References:	[Cordy 1992, du Boulay 1989a, du Boulay 1989b, Eisenberg 1987, Green 1996, McKeithen 1981, Sime 1977c, Spohrer 1986a, Spohrer 1986b]

8. Recognition Rather Than Recall

“Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate [Nielsen 1994].”

This section addresses the memory demands of programming.

8-1. Minimize Working Memory Load

Working memory limitations account for a large part of the inferior performance of novice programmers [Anderson 1985]. While experts are good at utilizing external memory to relieve their memory load, novices rely extensively on working memory. This indicates that for novices it is important that the environment minimize memory load, since they do not have the fallback of externalization [Davies 1993, Davies 1996]. TEd addresses this problem by maintaining a visual record of all edits [Ormerod 1996]. It is claimed that this supports display-based problem solving because it acts as an external repository of important programming knowledge, and it cues the programmer to focus on unsatisfied goals.

Context of Use:	General
Justified by:	Empirical studies
Examples:	Neither visual programming languages nor textual languages are good at supporting “display-based” problem solving, where the system’s display is used as a memory aid, reducing demands on working memory [Green 1996].
Cross References:	5-7. “Visual vs. Textual” 6-1. “Avoid Requiring Premature Commitment”
References:	[Anderson 1985, Davies 1993, Davies 1996, Green 1996, Ormerod 1996]

9. Aesthetic and Minimalist Design

“Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility [Nielsen 1994].”

The research in this section investigates conciseness.

9-1. Principle of Conciseness

[Cordy 1992] describes a principle of conciseness, which argues against redundant symbols such as program preambles, punctuation, and explicit declaration of variables and their types. Elimination of punctuation addresses a problem with the misplaced use of syntax which is described above (see “Avoid Subtle Distinctions in Syntax” on page 11). But if, for example, punctuation is replaced with line breaks as a statement separator, the problem of wrapping long lines must be handled sensibly.

Another aspect of conciseness is to allow optional information, with intelligent defaults [Cordy 1992]. Programming system should be flexible in allowing the user to elide undesired details and just fill in the obvious details, to stop requiring exact truth and instead allow an executable “cognitive approximation” to the solution [Lewis 1987].

However, conciseness can be subverted by a desire for elegance or parsimony of primitives. This can lead to absurdities. For example, early versions of Prolog did subtraction by inverse addition [Green 1990a]. [Mendelsohn 1990] cautions not to take *economy* and *elegance* as virtues in their own right: these are misplaced in designing languages for novices. “In looking for ever more abstracted ways to express behavior, modern languages have excised most clues to goal and purpose that are essential to novice understanding [Bonar 1990].” Novice programmers are more verbose than expert programmers in describing tasks to computers or to humans [Onorato 1986]. Thus experts have the edge in conciseness and preciseness.

Context of Use: Notation

Justified by: Empirical studies, expert opinion

Examples: The type of a variable could be inferred from its initialization, eliminating the necessity to declare its type.

Pascal has a large program preamble that only contributes one meaningful piece of information, the name of the program, “`p`”:

```
Program p (input,output);
```

APL takes conciseness to the extreme, at the expense of an excessive number of cryptic primitives. But, as mentioned above (see “Closeness of Mapping” on page 20), an abundance of primitives is problematic [Lewis 1987].

HyperTalk is a compromise, because optional syntax allows the same expression to be expressed concisely or verbosely, depending on the programmer’s preference. However, as mentioned above (see “Avoid Subtle Distinctions in Syntax” on page 11), more planning is required when there are many different legal solutions to a problem [Gray 1987].

Exceptions: Conciseness should be balanced with the findings of [Sime 1977c] that more verbose control structures help both beginners and novices to manage flow of control.

Cross References:	“Use Signalling to Highlight Important Information” on page 6 4-5. “Avoid Subtle Distinctions in Syntax” 5-3. “Consistency with External Knowledge” 5-4. “Closeness of Mapping”
References:	[Bonar 1990, Cordy 1992, Gray 1987, Green 1990a, Lewis 1987, Mendelsohn 1990, Onorato 1986, Sime 1977c]

10. Help Users Recognize, Diagnose, and Recover from Errors

“Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution [Nielsen 1994].”

The research in this section investigates some common kinds of bugs and ways that the system can help the user identify and fix them.

10-1. Support for Testing and Debugging

Testing and debugging are areas of difficulty for novices [du Boulay 1989a]. As mentioned above (see “Support Incremental Running and Testing with Immediate Feedback” on page 12), [du Boulay 1989b] claims that the computational machine should reveal its internal workings in terms of the language itself. This can be interpreted as a call for a source-level debugger and tracer with data visualization. [Mendelsohn 1990], and [Eisenstadt 1989] describe such systems for Prolog, and [Miller 1994] describes one for Pascal.

Research by [Gugerty 1986a, Gugerty 1986b, Kessler 1986, Nanja 1987] observed that novices tend to add new bugs to the program while debugging. This suggests that the environment should provide a checkpointing feature, or a selective undo feature, to help the user recover from these errors.

A survey of experienced programmers found that the most common root cause of bugs was memory getting clobbered or used up [Eisenstadt 1993]. Anything the environment can do to prevent or detect these problems would be helpful, especially since this kind of bug is more difficult to detect because often there is chasm – a distance in time and code proximity – between the cause and the effect of the bug. That survey also found that using a debugger and instrumenting the code with print statements were much more common than code-reading in expert debugging, suggesting that a good debugger is essential.

However, novices who are able to program do not automatically gain debugging skills in the process of learning to program, suggesting that it would be useful for debugging strategies to be taught or for the environment to be proactive in suggesting strategies.

Context of Use:	Environment
Justified by:	Empirical studies, observations of individual users, expert opinion
Examples:	Spreadsheets do not support debugging very well. The underlying formulas are not revealed unless explicitly made visible, so only one cell can be examined at a time; and variables are usually labelled with their cell location rather than a meaningful or mnemonic name.

Many novice bugs are caused by boundary or fence-post problems, such as:
• off-by-one bugs;
• not permitting the value zero where it should be permitted, or permitting it where it should not be permitted;
• the decision whether a boundary value should be handled as a special case or by one of the conditions that it divides;
• confusion between the number of values in a range and the highest value in a range:
• drawing incorrect parallels between constructs that have different boundary values, such as hours ranging from 1..12, but minutes ranging from 0..59 rather than 1..60.
To the extent that the environment can help to clarify these confusions,

novice productivity should be enhanced [Spohrer 1986a, Spohrer 1986b].

A major source of bugs is failure to guard against illegal or missing data [Cunniff 1989].

Students sometimes interpret the assignment statement `A:=B` as a swap operation, a print statement, a no-op, or a boolean comparison operation [Sleeman 1988].

Cross References:	4-3. “Locality and Hidden Dependencies” 4-6. “Support Incremental Running and Testing with Immediate Feedback” 5-15. “Instructional Design”
References:	[Cunniff 1989, du Boulay 1989a, du Boulay 1989b, Eisenstadt 1993, Eisenstadt 1989, Gugerty 1986a, Gugerty 1986b, Kessler 1986, Mendelsohn 1990, Miller 1994, Nanja 1987, Sleeman 1988, Spohrer 1986a, Spohrer 1986b]

11. Help and Documentation

“Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user’s task, list concrete steps to be carried out, and not be too large [Nielsen 1994].”

The research in this section investigates a kind of documentation called guiding knowledge.

11-1. Provide Guiding Knowledge

Guiding knowledge is a brief document that describes everything a naive user needs to know about the system [Bell 1994]. It describes the metaphor, explains general concepts of the system and its use, and contains advice about how to go about solving problems. Examples and analogies play an important role in understanding [Lewis 1987]. Users will have difficulty if there are hidden assumptions, if there is any necessary information missing from the guiding knowledge, if the guiding knowledge is not consistent with itself or the system's metaphorical model, or if the guiding knowledge does not convince the user that the recommended plans will work. However, it is desirable for the guiding knowledge to be brief, because it is the initial hurdle to using the system. This argues for a good metaphor, because it will be easily explained and consistent, so that the guiding knowledge can be brief.

Context of Use: Documentation and metaphor

Justified by: Based on the *programming walkthrough* method for assessing the programming language designs [Bell 1994], which in turn was based on the *cognitive walkthrough* method for evaluating user interfaces [Lewis 1990, Polson 1992].

Examples: The Macintosh introduced the mouse, menus, icons, and windows in a direct-manipulation graphical user interface. Despite the fact that most new users were learning a radically new system, the guiding knowledge was remarkably small: a thin manual and a short online guided tour.

When the guiding knowledge suggests a plan for solving a particular problem, but does not convince the user that this plan will work, users apparently believe that they do not understand the guiding knowledge, and begin searching for alternative plans that are "better". This may be related to meta-cognitive strategies (see "Cognitive Issues" on page 39).

Examples: 5-1. "Choose an Appropriate Metaphor"
5-5. "Support for Planning"
5-14. "Cognitive Issues"

References: [Bell 1994, Lewis 1987, Lewis 1990, Polson 1992]

12. Conclusions

This report attempts to organize the existing research about novice programmers in a way that will facilitate its use in guiding the design of new programming systems. The authors welcome comments and additions to this material.

There are a number of questions that are not addressed by the research we were able to find. For example:

- How does verbosity affect novice programming effectiveness? Are languages like HyperTalk, with optional extra words that enhance natural-language readability, more effective than terse languages, and what new problems do they exhibit?
- What are the relative strengths and weaknesses of the various paradigms of programming, such as the event-driven model?
- To what extent can careful design of the programming environment solve many of the problems that have been identified in this report?

We hope that future research will address these questions.

In addition to the results summarized here, there are general Human-Computer Interaction (HCI) principles that apply to all computer systems, including programming systems (e.g., [Macaulay 1995, Nielsen 1994, Tognazzini 1992]). Designers of programming systems should consider these general HCI principles as well as the issues that are directly related to programming.

13. Acknowledgments

The authors acknowledge Diana Bental, Alan Blackwell, Albert Corbett, Igor Curcio, David Gilmore, Rachelle Heller, Mark Ireland, Bonnie John, Pavol Návrat, Tom Ormerod, and Robert Rist for their comments and suggestions.

14. Bibliography

[Anderson 1985]	Anderson, J.R. and R. Jeffries (1985). "Novice LISP Errors: Undetected Losses of Information from Working Memory." Human-Computer Interaction 1: 107-131.
[Arblaster 1979]	Arblaster, A.T., M.E. Sime and T.R.G. Green (1979). "Jumping to Some Purpose." The Computer Journal 22: 105-109.
[Atwood 1978]	Atwood, M.E. and H.R. Ramsay (1978). Cognitive Structures in the Comprehension and Memory of Computer Programs: An Investigation of Computer Debugging. Alexandria, VA, U.S. Army Research Institute.
[Baecker 1986]	Baecker, R. (1986). Design Principles for the Enhanced Presentation of Computer Program Source Text. Proceedings of CHI'86 Conference on Human Factors in Computing Systems. M. Mantei and P. Orbeton. Boston, ACM: 51-58.
[Baecker 1990]	Baecker, R.M. and A. Marcus (1990). Human Factors and Typography for More Readable Programs. Reading, MA, Addison-Wesley Publishing Co. (ACM Press).
[Ball 1995]	Ball, L.J. and T.C. Ormerod (1995). "Structured and Opportunistic Processing in Design: A Critical Discussion." International Journal of Human-Computer Studies 43: 131-151.
[Bell 1994]	Bell, B., W. Citrin, C. Lewis, J. Rieman, R. Weaver, N. Wilde and B. Zorn (1994). "Using the Programming Walkthrough to Aid in Programming Language Design." Software–Practice and Experience 24(1): 1-25.
[Bell 1991]	Bell, B., J. Rieman and C. Lewis (1991). Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough. Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems. S. P. Robertson, G. M. Olson and J. S. Olson. New Orleans, ACM Press: 7-12.
[Berlin 1993]	Berlin, L.M. (1993). Beyond Program Understanding: A Look at Programming Expertise in Industry. Empirical Studies of Programmers: Fifth Workshop. C. R. Cook, J. C. Scholtz and J. C. Spohrer. Palo Alto, CA, Ablex Publishing Corporation: 6-25.
[Biermann 1983]	Biermann, A.W., B.W. Ballard and A.H. Sigmon (1983). "An Experimental Study of Natural Language Programming." International Journal of Man-Machine Studies 18: 71-87.

[Blackwell 1996] Blackwell, A.F. (1996). Metacognitive Theories of Visual Programming: What Do We Think We Are Doing? Proceedings of the VL'96 IEEE Workshop on Visual Languages. Boulder, CO: in press.

[Boehm-Davis 1996] Boehm-Davis, D.A., J.E. Fox and B.H. Philips (1996). Techniques for Exploring Program Comprehension. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 3-37.

[Bonar 1986] Bonar, J. (1986). Mental Models of Programming Loops. Pittsburgh, Learning Research and Development Center, University of Pittsburgh.

[Bonar 1987] Bonar, J., R. Cunningham, P. Beatty and P. Riggs (1987). Bridge: Intelligent Tutoring with Intermediate Representations. Pittsburgh, University of Pittsburgh.

[Bonar 1990] Bonar, J. and B.W. Liffick (1990). A Visual Programming Language for Novices. Principles of Visual Systems. S.-K. Chang. Englewood, CA, Prentice-Hall.

[Bonar 1989] Bonar, J. and E. Soloway (1989). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 325-353.

[Bonar 1988a] Bonar, J.G. and R. Cunningham (1988a). Bridge: An Intelligent Tutor for Thinking About Programming. Artificial Intelligence and Human Learning: Intelligent Computer-Aided Instruction. J. Self. London, Chapman and Hall: 432.

[Bonar 1988b] Bonar, J.G. and R. Cunningham (1988b). Bridge: Tutoring the Programming Process. Intelligent Tutoring Systems: Lessons Learned. J. Psotka, L. D. Massey and S. A. Mutter. Hillsdale, NJ, Lawrence Erlbaum Associates: 409-434.

[Borning 1985] Borning, A. (1985). “A Prototype Electronic Encyclopedia.” EACM Transactions on Office Information Systems 3: 63-88.

[Bowles 1994] Bowles, A., D. Robertson, W. Vasconcelos, M. Vargas-Vera and D. Bental (1994). “Applying Prolog Programming Techniques.” International Journal of Human-Computer Studies 41: 329-350.

[Brna 1991] Brna, P., A. Bundy, T. Dodd, M. Eisenstadt, C.K. Looi, H. Pain, D. Robertson, B. Smith and M. van Someren (1991). “Prolog Programming Techniques.” Instructional Science 20: 111-133.

[Brooke 1980a] Brooke, J.B. and K.D. Duncan (1980a). “Experimental Studies of Flow-chart Use at Different Stages of Program debugging.” Ergonomics 23: 1057-1091.

[Brooke 1980b] Brooke, J.B. and K.D. Duncan (1980b). “An Experimental Study of Flow-charts as an Aid to Identification of Procedural Faults.” Ergonomics 23: 387-399.

[Brooks 1983] Brooks, R. (1983). “Towards a Theory of the Comprehension of Computer Programs.” International Journal of Man-Machine Studies 18: 543-554.

[Brusilovsky 1994a] Brusilovsky, P., E. Calabrese, J. Hvorecky, A. Kouchnirenko and P. Miller (1994a). Mini-languages: A Way to Learn Programming Principles.

[Brusilovsky 1994b] Brusilovsky, P., A. Kouchnirenko, P. Miller and I. Tomek (1994b). Teaching Programming to Novices: A Review of Approaches and Tools. Educational Multimedia and Hypermedia: Proceedings of ED-MEDIA 94. T. Ottmann and I. Tomek. Vancouver, BC Canada, Association for the Advancement of Computing in Education: 103-110.

[Carver 1988] Carver, S.M. (1988). Learning and Transfer of Debugging Skills: Applying Task Analysis to Curriculum Design and Assessment. Teaching and Learning Computer Programming: Multiple Research Perspectives. R. E. Mayer. Hillsdale, NJ, Lawrence Erlbaum Associates: 259-297.

[Carver 1987] Carver, S.M. and S.C. Risinger (1987). Improving Children's Debugging Skills. Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Shepard and E. Soloway. Norwood, NJ, Ablex: 147-171.

[Clements 1993] Clements, D.H. and J.S. Meredith (1993). “Research on Logo: Effects and Efficacy.” Journal of Computing in Childhood Education 4(4): 263-290.

[Clements 1995] Clements, D.H. and J. Sarama (1995). “Design of a Logo Environment for Elementary Geometry.” Journal of Mathematical Behavior 14: 381-398.

[Cohen 1989] Cohen, P.R., M. Dalrymple, D.B. Moran, F.C.N. Pereira, J.W. Sullivan, J. Robert A. Gargan, J.L. Schlossberg and S.W. Tyler (1989). Synergistic Use of Direct Manipulation and Natural Language. Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems: 227-233.

[Corbett 1995] Corbett, A.T. and J.R. Anderson (1995). Knowledge Decomposition and Subgoal Reification in the ACT Programming Tutor. Artificial Intelligence in Education, 1995: Proceedings of the 7th World Conference on Artificial Intelligence in Education. J. Greer. Charlottesville, VA, AACE: 469-476.

[Cordy 1992] Cordy, J.R. (1992). Hints on the Design of User Interface Language Features – Lessons from the Design of Turing. Languages for Developing User Interfaces. B. A. Myers. Boston, Jones and Bartlett Publishers: 329-340.

[Crosby 1990] Crosby, M.E. and J. Stelovsky (1990). “How Do We Read Algorithms? A Case Study.” Computer 23(1): 24-35.

[Cunniff 1987] Cunniff, N. and R.P. Taylor (1987). Graphical Versus Textual Representation: An Empirical Study of Novices' Program Comprehension. Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Sheppard and E. Soloway. Norwood, NJ, Ablex. 114-131.

[Cunniff 1989] Cunniff, N., R.P. Taylor and J.B. Black (1989). Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A Comparison of FPL and Pascal. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 419-429.

[Curtis 1989] Curtis, B. (1989). Five Paradigms in the Psychology of Programming. Handbook of Human-Computer Interaction. M. Helander. North-Holland, Elsevier.

[Curtis 1988] Curtis, B., S. Sheppard, E. Kruesi-Bailey, J. Bailey and D. Boehm-Davis (1988). “Experimental Evaluation of Software Documentation Formats.” Journal of Systems and Software 9: 1-41.

[Cypher 1995] Cypher, A. and D.C. Smith (1995). KidSim: End User Programming of Simulations. Proceedings of CHI'95 Conference on Human Factors in Computing Systems. Denver, ACM.

[Dalby 1985] Dalby, J. and M.C. Linn (1985). “The Demands and Requirements of Computer Programming: A Literature Review.” Journal of Educational Computing Research 1: 253-274.

[Daly 1996] Daly, J., A. Brooks, J. Miller, M. Roper and M. Wood (1996). Evaluating the Effect of Inheritance on the Maintainability of Object-Oriented Software. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 39-57.

[Daly 1995a] Daly, J., J. Miller, A. Brooks, M. Roper and M. Wood (1995a). Issues on the Object-Oriented Paradigm: A Questionnaire Survey. Glasgow, University of Strathclyde Department of Computer Science**:** 44.

[Daly 1995b] Daly, J., M. Wood, A. Brooks, J. Miller and M. Roper (1995b). Structured Interviews on the Object-Oriented Paradigm. Glasgow, University of Strathclyde Department of Computer Science**:** 34.

[Davies 1993] Davies, S.P. (1993). Externalising Information During Coding Activities: Effects of Expertise, Environment and Task. Empirical Studies of Programmers: Fifth Workshop. C. R. Cook, J. C. Scholtz and J. C. Spohrer. Palo Alto, CA, Ablex Publishing Corporation: 42-61.

[Davies 1996] Davies, S.P. (1996). Display-Based Problelm Solving Stragegies in Computer Programming. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 59-76.

[Davis 1993] Davis, E.A., M.C. Linn, L.M. Mann and M.J. Clancy (1993). Mind Your Ps and Qs: Using Parentheses and Quotes in LISP. Empirical Studies of Programmers: Fifth Workshop. C. R. Cook, J. C. Scholtz and J. C. Spohrer. Palo Alto, CA, Ablex Publishing Corporation: 62-85.

[Détienne 1990] Détienne, F. (1990). Difficulties in Designing with an Object-Oriented Programming Language: An Empirical Study. Proceedings of INTERACT '90 Conference on Computer-Human Factors. Cambridge, England: 971-976.

[diSessa 1989] diSessa, A.A. and H. Abelson (1989). Boxer: A Reconstructible Computational Medium. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 467-481.

[du Boulay 1989a] du Boulay, B. (1989a). Some Difficulties of Learning to Program. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 283-299.

[du Boulay 1989b] du Boulay, B., T. O'Shea and J. Monk (1989b). The Glass Box Inside the Black Box: Presenting Computing Concepts to Novices. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 431-446.

[Dvorak 1994] Dvorak, J. (1994). “Conceptual Entropy and its Effect on Class Hierarchies.” IEEE Computer 27(6): 59-63.

[Eisenberg 1987] Eisenberg, M., M. Resnick and F. Turbak (1987). Understanding Procedures as Objects. Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Shepard and E. Soloway. Norwood, NJ, Ablex: 14-32.

[Eisenstadt 1993] Eisenstadt, M. (1993). Tales of Debugging from the Front Lines. Empirical Studies of Programmers: Fifth Workshop. C. R. Cook, J. C. Scholtz and J. C. Spohrer. Palo Alto, CA, Ablex Publishing Corporation: 86-112.

[Eisenstadt 1989] Eisenstadt, M. and M. Brayshaw (1989). An Integrated Textbook, Video, and Software Environment for Novice and Expert Prolog Programmers. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 447-466.

[Fay 1988] Fay, A.L. and R.E. Mayer (1988). Learning LOGO: A Cognitive Analysis. Teaching and Learning Computer Programming: Multiple Research Perspectives. R. E. Mayer. Hillsdale, NJ, Lawrence Erlbaum Asociates: 55-74.

[Fenton 1989] Fenton, J. and K. Beck (1989). An Object-Oriented Simulation System with Agent Rules for Children of All Ages. Proceedings of OOPSLA'89. New York, ACM: 123-137.

[Finzer 1993] Finzer, W.F. and L. Gould (1993). Rehearsal World: Programming by Rehearsal. Watch What I Do: Programming by Demonstration. A. Cypher, MIT Press.

[Fitter 1979] Fitter, M.J. and T.R.G. Green (1979). "When Do Diagrams Make Good Computer Languages?" International Journal of Man-Machine Studies 11: 235-261.

[Fung 1990] Fung, P., M. Brayshaw, B. du Boulay and M. Elsom-Cook (1990). "Towards a Taxonomy of Novices' Misconceptions of the Prolog Interpreter." Instructional Science 19: 311-336.

[Fung 1987] Fung, P., B. du Boulay and M. Elsom-Cook (1987). An Initial Taxonomy of Novices' Misconceptions of the Prolog Interpreter, Institute for Educational Technology, The Open University.

[Galotti 1985] Galotti, K.M. and W.F. Ganong, III (1985). "What Non-Programmers Know About Programming: Natural Language Procedure Specification." International Journal of Man-Machine Studies 22: 1-10.

[Gellenbeck 1991a] Gellenbeck, E.M. and C.R. Cook (1991a). Does Signalling Help Professional Programmers Read and Understand Computer Programs? Empirical Studies of Programming: Fourth Workshop. J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson. New Brunswick, NJ, Ablex Publishing Corporation: 82-98.

[Gellenbeck 1991b] Gellenbeck, E.M. and C.R. Cook (1991b). An Investigation of Procedure and Variable Names as Beacons During Program Comprehension. Empirical Studies of Programming: Fourth Workshop. J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson. New Brunswick, NJ, Ablex Publishing Corporation: 65-81.

[Gilmore 1986] Gilmore, D.J. (1986). Structural Visibility and Program Comprehension. People and Computers: Designing for Usability. M. D. Harrison and A. F. Monk. Cambridge, Cambridge University Press.

[Gilmore 1988] Gilmore, D.J. and T.R.G. Green (1988). “Programming Plans and Programming Expertise.” Quarterly Journal of Experimental Psychology 40a: 423-442.

[Gilmore 1984] Gilmore, D.J. and H.T. Smith (1984). “An Investigation of the Utility of Flowhcarts During Computer Program Debugging.” International Journal of Man-Machine Studies 20: 331-372.

[Goldenson 1996] Goldenson, D.R. (1996). Why Teach Computer Programming? Some Evidence about Generalization and Transfer. Proceedings of NECC'96 National Educcational Computing Conference.

[Goldenson 1991] Goldenson, D.R. and B.J. Wang (1991). Use of Structure Editing Tools by Novice Programmers. Empirical Studies of Programming: Fourth Workshop. J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson. New Brunswick, NJ, Ablex Publishing Corporation: 99-120.

[Good 1996] Good, J. (1996). The 'Right' Tool for the Task: An Investigation of External Representations, Program Abstractions and Task Requirements. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 77-98.

[Gray 1987] Gray, W. and J.R. Anderson (1987). Change-Episodes in Coding: When and How Do Programmers Change Their Code. Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Sheppard and E. Soloway. Norwood, NJ, Ablex: 185-197.

[Green 1990a] Green, T.R.G. (1990a). The Nature of Programming. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 21-44.

[Green 1990b] Green, T.R.G. (1990b). Programming Languages as Information Structures. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 118-137.

[Green 1987] Green, T.R.G., R. Bellamy, K.E. and J.M. Parker (1987). Parsing and Gnisrap: A Model of Device Use. Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Sheppard and E. Soloway. Norwood, NJ, Ablex.

[Green 1995] Green, T.R.G. and R. Navarro (1995). Programming Plans, Imagery, and Visual Programming. Proceedings of INTERACT-95. K. Nordby, D. J. Gilmore and S. Arnesen. London, Chapman and Hall.

[Green 1992] Green, T.R.G. and M. Petre (1992). When Visual Programs are Harder to Read than Textual Programs. Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). G. C. van der Veer, M. J. Tauber, S. Bagnarola and M. Antavolits. Rome, CUD.

[Green 1996] Green, T.R.G. and M. Petre (1996). “Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework.” Journal of Visual Languages and Computing 7(2): 131-174.

[Green 1991] Green, T.R.G., M. Petre and R.K.E. Bellamy (1991). Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture. Empirical Studies of Programming: Fourth Workshop. J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson. New Brunswick, NJ, Ablex Publishing Corporation: 121-146.

[Grice 1975] Grice, H.P. (1975). Logic and Conversation. Syntax and Semantics III: Speech Acts. P. Cole and J. Morgan. New York, Academic Press.

[Gugerty 1986a] Gugerty, L. and G.M. Olson (1986a). Comprehension Differences in Debugging by Skilled and Novice Programmers. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 13-27.

[Gugerty 1986b] Gugerty, L. and G.M. Olson (1986b). Debugging by Skilled and Novice Programmers. Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems: 171-174.

[Guzdial 1992] Guzdial, M., P. Weingrad, R. Boyle and E. Soloway (1992). Design Support Environments for End Users. Languages for Developing User Interfaces. B. A. Myers. Boston, Jones and Bartlett Publishers: 57-78.

[Halasz 1982] Halasz, F. and T.P. Moran (1982). Analogy Considered Harmful. Proceedings of Human Factors in Computer Systems: 383-386.

[Hasan 1996] Hasan, H., C. Jones and E. Gould (1996). Prototyping Tools for Expert and Novice Application Development. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 99-107.

[Heller 1986]	Heller, R.S. (1986). Different Logo Teaching Styles: Do They Really Matter. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 117-127.
[Hoadley 1996]	Hoadley, C.M., M.C. Linn, L.M. Mann and M.J. Clancy (1996). When, Why and How Do Novice Programmers Reuse Code? Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 109-129.
[Hoc 1983]	Hoc, J.-M. (1983). Analysis of Beginner's Problem-solving Strategies in Programming. The Psychology of Computer Use. T. R. G. Green, S. J. Payne and G. van der Veer. London, Academic Press: 143-158.
[Hoc 1989]	Hoc, J.-M. (1989). Do We Really Have Conditional Statements in Our Brains? Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 179-90.
[Hoc 1990]	Hoc, J.-M. and A. Nguyen-Xuan (1990). Language Semantics, Mental Models and Analogy. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 139-156.
[Jeffries 1982]	Jeffries, R.A. (1982). Comparison of Debugging Behavior of Novice and Expert Programmers. Pittsburgh, PA, Department of Psychology, Carnegie Mellon University.
[Kahn 1996]	Kahn, K. (1996). “Drawings on Napikins, Video-Game Animation, and Other Ways to Program Computers.” Communications of the ACM 39(8): 49-59.
[Kahney 1989]	Kahney, H. (1989). What Do Novice Programmers Know About Recursion. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 209-228.
[Kesler 1984]	Kesler, T.E., R.B. Uram, F. Magareh-Abed, A. Fritzsche, C. Amport and H.E. Dunsmore (1984). “The Effect of Indentation on Program Comprehension.” International Journal of Man-Machine Studies 21: 415-428.
[Kessler 1986]	Kessler, C.M. and J.R. Anderson (1986). A Model of Novice Debugging in LISP. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 198-212.
[Kessler 1989]	Kessler, C.M. and J.R. Anderson (1989). Learning Flow of Control: Recursive and Iterative Procedures. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 229-260.

[Korson 1986] Korson, T.D. and V.K. Vaishnavi (1986). An Empirical Study of the Effects of Modularity on Program Modifiability. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 168-186.

[Kurland 1989] Kurland, D.M. and R.D. Pea (1989). Children's Mental Models of Recursive Logo Programs. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 315-323.

[Ledgard 1980] Ledgard, H.F., J. Whiteside, A. Singer and W. Seymour (1980). “The Natural Language of Interactive Systems.” Communications of the ACM 23(10): 556-563.

[Lee 1993] Lee, A.Y. and N. Pennington (1993). Learning Computer Programming: A Route to General Reasoning Skills. Empirical Studies of Programmers: Fifth Workshop. C. R. Cook, J. C. Scholtz and J. C. Spohrer. Palo Alto, CA, Ablex Publishing Corporation: 113-1136.

[Lehrer 1988] Lehrer, R., T. Guckenberg and L. Sancilio (1988). Influences of LOGO on Children's Intellectual Development. Teaching and Learning Computer Programming: Multiple Research Perspectives. R. E. Mayer. Hillsdale, NJ, Lawrence Erlbaum Asociates: 75-110.

[Lewis 1987] Lewis, C. and G.M. Olson (1987). Can Principles of Cognition Lower the Barriers to Programming? Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Sheppard and E. Soloway. Norwood, NJ, Ablex: 248-263.

[Lewis 1990] Lewis, C., P. Polson, C. Wharton and J. Rieman (1990). Testing a Walkthrough Methodology for Theory-Based Design of Walk-Up-and-Use Interfaces. Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems: 235-242.

[Littlefield 1988] Littlefield, J., V.R. Delclos, S. Lever, K.N. Clayton, J.D. Bransford and J.J. Franks (1988). Learning LOGO: Method of Teaching, Transfer of General Skills, and Attitudes Toward School and Computers. Teaching and Learning Computer Programming: Multiple Research Perspectives. R. E. Mayer. Hillsdale, NJ, Lawrence Erlbaum Asociates: 111-135.

[Macaulay 1995] Macaulay, L. (1995). Human-Computer Interaction for Software Designers, Thompson Computer Press.

[Mayer 1988] Mayer, R.E. (1988). Introduction to Research on Teaching and Learning Computer Programming. Teaching and Learning Computer Programming: Multiple Research Perspectives. R. E. Mayer. Hillsdale, NJ, Lawrence Erlbaum Asociates: 1-12.

[Mayer 1989] Mayer, R.E. (1989). The Psychology of How Novices Learn Computer Programming. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 129-159.

[Mayer 1987] Mayer, R.E. and A.L. Fay (1987). “A Chain of Cognitive Changes with Learning to Program in LOGO.” Journal of Educational Psychology 79: 269-279.

[McKeithen 1981] McKeithen, K.B. (1981). “Knowledge Organization and Skill Differences in Computer Programmers.” Cognitive Psychology 13: 307-325.

[Mendelsohn 1990] Mendelsohn, P., T.R.G. Green and P. Brna (1990). Programming Languages in Education: The Search for an Easy Start. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 175-200.

[Merrill 1993] Merrill, D.C. and B.J. Reiser (1993). Scaffolding the Acquisition of Complex Skills with Reasoning-Congruent Learning Environments. Proceedings of the Workshop in Graphical Representations, Reasoning, and Communication from the World Conference on Artificial Intelligence in Education (AI-ED '93). Edinburgh, Scotland, The University of Edinburgh: 9-15.

[Merrill 1994] Merrill, D.C. and B.J. Reiser (1994). Scaffolding Effective Problem Solving Strategies in Interactive Learning Environments. Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society. Atlanta, GA, Lawrence Erlbaum Associates.

[Merrill 1992] Merrill, D.C., B.J. Reiser, R. Beekelaar and A. Hamid (1992). Making Processes Visible: Scaffolding Learning with Reasoning-Congruent Representations. Proceedings of the Intelligent Tutoring System Conference. C. Frasson, G. Gauthier and G. I. McCalla. New York, Springer-Verlag: 103-110.

[Miara 1983] Miara, R.J., J.A. Musselman, J.A. Navarro and B. Schneiderman (1983). “Program Indentation and Comprehensibility.” Communications of the ACM 26: 861-867.

[Miller 1981] Miller, L.A. (1981). “Natural Language Programming: Styles, Strategies, and Constrasts.” IBM Systems Journal 20(2): 184-215.

[Miller 1994] Miller, P., J. Pane, G. Meter and S. Vorthmann (1994). “Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University.” Interactive Learning Environments 4(2): 140-158.

[Modugno 1996] Modugno, F., A.T. Corbett and B.A. Myers (1996). Evaluating Program Representation in a Visual Shell. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 131-146.

[Moher 1993] Moher, T.G., D.C. Mak, B. Blumenthal and L.M. Leventhal (1993). Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. Empirical Studies of Programmers: Fifth Workshop. C. R. Cook, J. C. Scholtz and J. C. Spohrer. Palo Alto, CA, Ablex Publishing Corporation: 137-161.

[Myers 1990] Myers, B.A. (1990). “Taxonomies of Visual Programming and Program Visualization.” Journal of Visual Languages and Computing 1(1): 97-123.

[Myers 1988] Myers, B.A., R. Chandhok and A. Sareen (1988). Automatic Data Visualizations for Novice Pascal Programmers. Proceedings of the IEEE 1988 Workshop on Visual Languages. Pittsburgh, PA: 192-198.

[Nanja 1987] Nanja, M. and C.R. Cook (1987). An Analysis of the On-Line Debugging Process. Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Shepard and E. Soloway. Norwood, NJ, Ablex: 172-184.

[Nardi 1993] Nardi, B.A. (1993). A Small Matter of Programming: Perspectives on End User Computing. Cambridge, MA, The MIT Press.

[Návrat 1993] Návrat, P. and V. Rozinajová (1993). “Making Programming Knowledge Explicit.” Computers in Education 21(4): 281-299.

[Návrat 1996] Návrat, P. and V. Rozinajová (1996). “Knowledge Based Programming: An Experiment in Selecting a Data Type.” Arab Gulf J. Science. Res. 14(1): 79-100.

[Neal 1987] Neal, L.R. (1987). User Modelling for Syntax-Directed Editors. Human-Computer Interaction - INTERACT '87. H. J. Bullinger and B. Shackel. New York, Elesevier.

[Nickerson 1985] Nickerson, R.S., D.N. Perkins and E.E. Smith (1985). The Teaching of Thinking. Hillsdale, NJ, Lawrence Erlbaum Associates.

[Nielsen 1994] Nielsen, J. (1994). Heuristic Evaluation. Usability Inspection Methods. J. Nielsen and R. L. Mack. New York, John Wiley & Sons: 25-62.

[Nyuyen-Xuan 1987] Nyuyen-Xuan, A. and J.-M. Hock (1987). “Learning to Use a Command Device.” European Bulletin of Cognitive Psychology 7: 5-31.

[Olson 1987] Olson, G.M., R. Catrambone and E. Soloway (1987). Programming and Algebra Word Problems: A Failure to Transfer. Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Shepard and E. Soloway. Norwood, NJ, Ablex: 1-13.

[Onorato 1986] Onorato, L.A. and R.W. Schvaneveldt (1986). Programmer/Nonprogrammer Differences in Specifying Procedures to People and Computers. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 128-137.

[Ormerod 1996] Ormerod, T.C. and L.J. Ball (1996). An Empirical Evaluation of TEd, A Techniques Editor for Prolog Programming. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 147-161.

[Pane 1996] Pane, J.F., A.T. Corbett and B.E. John (1996). Assessing Dynamics in Computer-Based Instruction. Proceedings of ACM CHI'96 Conference on Human Factors in Computing Systems. Vancouver: 197-204.

[Papert 1980] Papert, S. (1980). Mindstorms: Children, Computers, and Powerful Ideas. New York, Basic Books.

[Parker 1987] Parker, J. and B. Hendley (1987). The Universe Program Development Environment. Proceedings of INTERACT'87.

[Pattis 1995] Pattis, R.E., J. Roberts and M. Stehlik (1995). Karel the Robot: A Gentle Introduction to the Art of Programming. New York, Wiley.

[Payne 1986] Payne, S.J. and T.R.G. Green (1986). “Task-Action Grammars: A Model of the Mental Representation of Task Languages.” Human-Computer Interaction 2(2): 93-133.

[Payne 1984] Payne, S.J., M.E. Sime and T.R.G. Green (1984). “Perceptual Structure Cueing in a Simple Command Language.” International Journal of Man-Machine Studies 21: 19-29.

[Pea 1986] Pea, R. (1986). “Language-Independent Conceptual "Bugs" in Novice Programming.” Journal of Educational Computing Research 2(1).

[Pea 1984] Pea, R.D. and D.M. Kurland (1984). “On the Cognitive Effects of Learning Computer Programming.” New Ideas in Psychology 2: 137-168.

[Pennington 1990] Pennington, N. and B. Grabowski (1990). The Tasks of Programming. The Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 45-62.

[Perkins 1985] Perkins, D.N. (1985). “The Fingertip Effect: How Information-Processing Technology Shapes Thinking.” Educational Researcher 14: 11-17.

[Perkins 1989] Perkins, D.N., C. Hancock, R. Hobbs, F. Martin and R. Simmons (1989). Conditions of Learning in Novice Programmers. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 261-279.

[Perkins 1986] Perkins, D.N. and F. Martin (1986). Fragile Knowledge and Neglected Strategies in Novice Programmers. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 213-229.

[Perkins 1988] Perkins, D.N., S. Schwartz and R. Simmons (1988). Instructional Strategies for the Problems of Novice Programmers. Teaching and Learning Computer Programming: Multiple Research Perspectives. R. E. Mayer. Hillsdale, NJ, Lawrence Erlbaum Asociates: 153-178.

[Petre 1992] Petre, M. and T.R.G. Green (1992). “Requirements of Graphical Notations for Professional Users: Electronics CAD Systems as a Case Study.” La Travail Humain 55(1): 47-70.

[Pirolli 1985] Pirolli, P. and J.R. Anderson (1985). “The Role of Learning from Examples in the Acquisition of Recursive Programming Skills.” Canadian Journal of Psychology 39: 40-272.

[Polson 1992] Polson, P.G., C. Lewis, J. Rieman and C. Wharton (1992). “Cognitive Walkthroughs: A Method for Theory-Based Evaluation of User Interfaces.” International Journal of Man-Machine Studies 36(5): 741-773.

[Putnam 1989] Putnam, R.T., D. Sleeman, J.A. Baxter and L.K. Kuspa (1989). A Summary of Misconceptions of High School Basic Programmers. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 301-314.

[Reiser 1992] Reiser, B.J., D.Y. Kimberg, M.C. Lovett and M. Ranny (1992). Knowledge Representation and Explanation in GIL, An Intelligent Tutor for Programming. Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complimentary Approaches. J. H. Larkin and R. W. Chabay. Hillsdale, NJ, Lawrence Erlbaum Associates: 111-149.

[Repenning 1993] Repenning, A. (1993). Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments. Dept. of Computer Science. Boulder, University of Colorado at Boulder**:** 171.

[Repenning 1994] Repenning, A. and T. Sumner (1994). Programming as Problem Solving: A Participatory Theater Approach. Workshop on Advanced Visual Interfaces. Bari, Italy: 182-191.

[Resnick 1994] Resnick, M. (1994). Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds. Boston, The MIT Press.

[Riecken 1991] Riecken, R.D. (1991). What Do Expert Programmers Communicate by Means of Descriptive Commenting? Empirical Studies of Programming: Fourth Workshop. J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson. New Brunswick, NJ, Ablex Publishing Corporation: 177-195.

[Rist 1995] Rist, R.S. (1995). “Program Structure and Design.” Cognitive Science 19: 507-562.

[Rist 1996] Rist, R.S. (1996). System Structure and Design. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 163-194.

[Roberts 1988] Roberts, J., J. Pane, M. Stehlik and J. Carrasquel (1988). The Design View: A Design Oriented, High-Level Visual Programming Environment. Proceedings of the 1988 IEEE Workshop on Visual Languages. Pittsburgh, PA: 213-220.

[Rogalski 1990] Rogalski, J. and R. Samurçay (1990). Acquisition of Programming Knowledge and Skills. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 157-174.

[Saariluoma 1994] Saariluoma, P. and J. Sanjaniemi (1994). “Transforming Verbal Descriptions into Mathematical Formulas in Spreadsheet Calculation.” International Journal of Human-Computer Studies 41(6): 915-948.

[Samurçay 1989] Samurçay, R. (1989). The Concept of Variable in Programming: Its Meaning and Use in Problem-Solving by Novice Programmers. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 161-178.

[Scholtz 1993] Scholtz, J. and S. Wiedenbeck (1993). An Analysis of Novice Programmers Learning a Second Language. Empirical Studies of Programmers: Fifth Workshop. C. R. Cook, J. C. Scholtz and J. C. Spohrer. Palo Alto, CA, Ablex Publishing Corporation: 187-205.

[Sherwood 1988] Sherwood, B.A. (1988). The cT Language. Champaigne, IL, Stipes Publishing Company.

[Shneiderman 1986] Shneiderman, B. (1986). Empirical Studies of Programmers: The Territory, Paths and Destinations. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 1-12.

[Shneiderman 1977] Shneiderman, B., R.E. Mayer, D. McKay and P. Heller (1977). “Experimental Investigations of the Utility of Detailed Flowcharts in Programming.” Communication of the ACM 20: 373-381.

[Siddiqi 1996] Siddiqi, J., R. Osborn, C. Roast and B. Khazaei (1996). The Pitfalls of Changing Programming Paradigms. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 219-231.

[Sime 1977a] Sime, M.E., A.T. Arblaster and T.R.G. Green (1977a). “Reducing Programming Errors in Nested Conditionals by Prescribing a Writing Procedure.” International Journal of Man-Machine Studies 9: 119-1226.

[Sime 1977b] Sime, M.E., T.R.G. Green and D.J. Guest (1977b). “Psychological Evaluation of Two Conditional Constructions Used in Computer Languages.” International Journal of Man-Machine Studies 5: 105-113.

[Sime 1977c] Sime, M.E., T.R.G. Green and D.J. Guest (1977c). “Scope Marking in Computer Conditionals: A Psychological Evaluation.” International Journal of Man-Machine Studies 9: 107-118.

[Sleeman 1988] Sleeman, D., R.T. Putnam, J. Baxter and L. Kuspa (1988). An Introductory Pascal Class: A Case Study of Students' Errors. Teaching and Learning Computer Programming: Multiple Research Perspectives. R. E. Mayer. Hillsdale, NJ, Lawrence Erlbaum Asociates: 237-257.

[Smith 1995] Smith, D.C. and A. Cypher (1995). KidSim: Child Constructible Simulations. Proceedings of the Imagina '95 Conference. Monte-Carlo: 87-99.

[Smith 1994] Smith, D.C., A. Cypher and J. Spohrer (1994). “KidSim: Programming Agents Without a Programming Language.” Communications of the ACM 37(7): 54-67.

[Smith 1986a] Smith, R. (1986a). The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages. Dallas, IEEE.

[Smith 1992] Smith, R.B., D. Ungar and B.-W. Chang (1992). The Use-Mention Perspective on Programming for the Interface. Languages for Developing User Interfaces. B. A. Myers. Boston, Jones and Bartlett Publishers: 79-89.

[Smith 1986b]	Smith, S.L. and J.N. Mosier (1986b). Guidelines for Designing User Interface Software. Bedford, MA, MITRE**:** 478.
[Soloway 1989]	Soloway, E., J. Bonar and K. Ehrlich (1989). Cognitive Strategies and Looping Constructs: An Empirical Study. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 191-207.
[Soloway 1984]	Soloway, E. and K. Ehrlich (1984). “Empirical Studies of Programming Knowledge.” IEEE Transactions on Software Engineering SE-10: 595-609.
[Soloway 1988]	Soloway, E., J. Pinto, S. Letovsky, D. Littman and R. Lampert (1988). “Designing Documentation to Compensate for Delocalized Plans.” Communications of the ACM 31(11): 1259-1267.
[Spohrer 1986a]	Spohrer, J.C. and E. Soloway (1986a). Alternatives to Construct-Based Programming Misconceptions. Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems: 183-191.
[Spohrer 1989a]	Spohrer, J.C. and E. Soloway (1989a). Novice Mistakes: Are the Folk Wisdoms Correct? Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 401-416.
[Spohrer 1989b]	Spohrer, J.C., E. Soloway and E. Pope (1989b). A Goal/Plan Analysis of Buggy Pascal Programs. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 355-399.
[Spohrer 1986b]	Spohrer, J.G. and E. Soloway (1986b). Analyzing the High Frequency Bugs in Novice Programs. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 230-251.
[Taylor 1990]	Taylor, J. (1990). “Analysing Novices Analysing Prolog: What Stories Do Novices Tell Themselves About Prolog?” Instructional Science 19: 283-309.
[Tognazzini 1992]	Tognazzini, B. (1992). Tog on Interface. Reading, MA, Addison-Wesley Publishing Co.
[Travers 1994]	Travers, M. (1994). Recursive Interfaces for Reactive Objects. Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems. Boston.
[Van Laar 1989]	Van Laar, D. (1989). Evaluating a Colour Coding Programming Support Tool. People and Computers V. A. Sutcliffe and L. Macaulay. Cambridge, Cambridge University Press.

[Vessey 1984] Vessey, I. and R. Weber (1984). "Conditional Statements and Program Coding: An Experimental Evaluation." International Journal of Man-Machine Studies 31: 47-60.

[Wandke 1988] Wandke, H. (1988). User-Defined Macros in HCI: When Are They Applied? Berlin, Sektion Psychologie der Humboldt-Universität zu Berlin.

[Watters 1995] Watters, A.R. (1995). Tutorial Article No. 005: The What, Why, Who, and Where of Python. UnixWorld Online.

[Wiedenbeck 1986a] Wiedenbeck, S. (1986a). "Beacons in Computer Program Comprehension." International Journal of Man-Machine Studies 25: 697-709.

[Wiedenbeck 1986b] Wiedenbeck, S. (1986b). Processes in Computer Program Comprehension. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 48-57.

[Wiedenbeck 1989] Wiedenbeck, S. (1989). The Initial Stage of Program Comprehension, University of Nebraska.

[Wiedenbeck 1996] Wiedenbeck, S. and J. Scholtz (1996). Adaptation of Programming Plans in Transfer Between Programming Languages: A Developmental Approach. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 233-253.

[Wirth 1983] Wirth, N. (1983). "Program Development by Stepwise Refinement." Communications of the ACM 26(1): 70-74.

[Wu 1991] Wu, Q. and J.R. Anderson (1991). Strategy Selection and Change in Pascal Programming. Empirical Studies of Programming: Fourth Workshop. J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson. New Brunswick, NJ, Ablex Publishing Corporation: 227-238.