Fall 1-31-2002

A review and assessment of novice learning tools for problem solving and program development

Christopher John Bladek
New Jersey Institute of Technology

ABSTRACT

A REVIEW AND ASSESSMENT OF NOVICE LEARNING TOOLS FOR PROBLEM SOLVING AND PROGRAM DEVELOPMENT

by
Christopher John Bladek

There is a great demand for the development of novice learning tools to supplement classroom instruction in the areas of problem solving and program development. Research in the area of pedagogy, the psychology of programming, human-computer interaction, and cognition have provided valuable input to the development of new methodologies, paradigms, programming languages, and novice learning tools to answer this demand.

Based on the cognitive needs of novices, it is possible to postulate a set of characteristics that should comprise the components an effective novice learning tool. This thesis will discover these characteristics and provide recommendations for the development of new learning tools. This will be accomplished with a review of the challenges that novices face, an in-depth discussion on modern learning tools and the challenges that they address, and the identification and discussion of the vital characteristics that constitute an effective learning tool based on these tools and personal ideas.

A REVIEW AND ASSESSMENT OF NOVICE LEARNING TOOLS FOR PROBLEM SOLVING AND PROGRAM DEVELOPMENT

by
Christopher John Bladek

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

January 2002

APPROVAL PAGE

A REVIEW AND ASSESSMENT OF NOVICE LEARNING TOOLS FOR PROBLEM SOLVING AND PROGRAM DEVELOPMENT

Christopher John Bladek

12/10/01

Dr. Fadi Deek, Thesis Advisor — Date
Associate Professor of Information Systems
Associate Dean of College of Computing Science
Director of Information Technology Program

12/10/01

Dr. James McHugh, Committee Member — Date
Professor and Acting Chairman of Computer Science Department

12/10/01

Dr. Qian Hong Liu, Committee Member — Date
Assistant Professor, Computer Science Department

BIOGRAPHICAL SKETCH

Author: Christopher John Bladek

Degree: Master of Science in Computer Science

Date: January 2002

Date of Birth:

Place of Birth:

Undergraduate and Graduate Education:

- Master of Science in Computer Science
  New Jersey Institute of Technology, Newark, NJ, 2002

- Bachelor of Science in Computer Engineering
  New Jersey Institute of Technology, Newark, NJ, 1997

Major: Computer Science

To my wife, Jacqueline

ACKNOWLEDGMENT

I would like to thank Dr. Fadi Deek for giving me the opportunity to develop my thesis under his supervision. His dedication as an advisor was exceptional. Dr. Deek provided me with great direction and support, and helped build my confidence. I would also like to thank Dr. James McHugh and Dr. Qian Hong Liu for participating in the assessment of my thesis as members of the review committee.

TABLE OF CONTENTS

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The process of learning computer science as a discipline is far removed cognitively from other disciplines because it involves a level of thought that has no real world counterpart. Computers are man-made machines that harness and process information. They are designed to think for us, yet we are the ones that have to program them to think. It is the manner in which information is represented and manipulated in computers that causes a great challenge to its understanding.

Upon enrolling in a computer science program, freshmen are usually unaware of the divide between what they have experienced in the world to this point and what they are about to experience on their new educational path. Most of them have some programming experience from secondary school education. This may provide them with a slight advantage in the programming aspect of computer science, but this guarantee is conditional on how involved the methodologies that they followed were.

Fortunately, most computer science curriculums provide courses for students that support introductory programming. These courses usually focus on the design of computers, their organization and architecture, logic and control, and data representation. The courses are intended to supplement the material presented in programming courses so that the cognitive connection can be made between computer software and hardware. Novices may struggle initially in their attempts to grasp the concepts involved in programming because the additional courses are often taken in parallel with the programming course, rather than before it.

Introductory programming languages abstract the underlying hardware and operating system by providing a high-level syntax, analogous to operations encountered in the real world, to control computer resources. However the extent of what can be considered analogous is usually limited to mathematical and relational operations. Concepts such as the declaration of variables, the association of data types with variables, the initialization of variables, looping constructs, recursion, and subprogram invocations are things that people do not take into account when going about making decisions in their daily routines. These concepts are intuitive to the thought process, but need to be explicitly represented in a program. This is where the difficulties experienced by novice programmers begin: they assume too little or too much about the reasoning ability of a computer and unsuccessfully attempt to rationalize with it.

It is for obvious reasons then that novice programmers need support to supplement classroom instruction. Research in the area of program development has been conducted for as long as computers have existed. Research in problem solving has been going on for centuries, but only in the past forty years has it been considered in the context of programming. The results of research have provided valuable input in improving the pedagogy of problem solving and program development, methodologies, paradigms, programming language design, and the development of novice learning tools.

The objective of this thesis is to postulate a set of characteristics that comprise the components an effective novice learning tool. The hypothetical learning tool is intended to be in the form a program development environment that integrates effective problem solving as a prerequisite to program development. To argue the need for such characteristics in the learning tool, an extensive background is warranted. The thesis will

first present a modern review of the challenges that novices are faced with. This is then followed by a state-of-the-art review of modern learning tools. These tools vary in their nature and the novice challenges that they address. The thesis will conclude with the identification and discussion of the vital characteristics that constitute an effective learning tool which should be considered in the development of new learning tools.

More specifically, Chapter 2 will identify in detail novice learning challenges such as problems attributed to pedagogy, psychology, programming languages and paradigms, and external influences. In the discussion on pedagogy, problem solving and program comprehension issues will be reviewed. An in-depth look at the syntactical, semantical, and pragmatic issues of programming languages will reveal exactly what the problem areas are and why they are inherently hard to overcome for novices. Each challenge that is identified will contain the reason why novices struggle in their understanding of the issue.

Chapter 3 is dedicated to identifying and discussing modern novice learning tools. These tools provide support for various aspects of problem solving and program development. Each tool is presented in a detailed approach, outlining the contribution that it makes to mending the issues it addresses. The chapter concludes with an extensive discussion on the Dual Common Model, the theoretical foundation of the learning tool SOLVEIT, which presents significant advantages over all other learning tools discussed.

Chapter 4 provides a detailed explanation of a practical session in SOLVEIT. For each of the six stages of SOLVEIT, the novice challenges that are addressed and how SOLVEIT can help overcome them will be discussed. Personal suggestions for the improvement and the evolution of SOLVEIT follow this discussion. For each stage of

SOLVEIT, recommendations for enhancements are provided. The chapter concludes with a discussion of the elements that constitute a comprehensive, easy-to-use, and most importantly, effective novice learning tool. Some of the elements that are identified are extracted from the tools discussed in this thesis. Others are based on personal ideas that may offer a significant contribution to the development of new learning tools. Conclusions for this thesis are presented in Chapter 5.

CHAPTER 2

DIFFICULTIES ENCOUNTERED BY NOVICES

Program development refers to the collective processes that one must execute in order to realize a working software solution from a given problem statement. By nature, novice programmers perceive and approach the program development task as solving a given problem by sitting at a computer and coding by trial and error. They often overlook the principles and methodologies that lay the foundation for successful software development. This behavior is very common and usually expected by instructors in first year programming courses because a vast majority of these students have not yet developed the critical knowledge and skills required to facilitate the proper development of programs.

Having said that, it is critical for computer science instructors to stress the teaching of these tasks, methods, and required skills in an introductory course so that a student will begin to develop them as early as possible. As a novice attempts to learn, use, and perfect the fundamentals of problem solving and program development, several road blocks will almost always be encountered.

Program development, in the context of this thesis, does not refer to the medial task of coding, but rather as the set of processes that follow the stage of problem solving. This chapter is devoted to identifying, categorizing, and providing a detailed explanation for a broad range of obstacles that a novice may encounter. Of these impediments, some are inherent by nature and are not easily avoidable, while others are induced by some human factor, whether it is a teaching method, or the design of a tool, language, or technique.

2.1 Pedagogical Roots

An ineffective pedagogy in an introductory programming course is one of the most vital factors that influence how a novice programmer will perform in higher-level courses. If instructors do not succeed immediately in teaching the required skills, many first year students will get discouraged as problems get harder to solve. It may eventually lead them away from the programming aspect of computer science, and perhaps from the computer science discipline altogether.

2.1.1 Curriculum and Textbooks

Not all universities require a course in computer organization as a prerequisite to an introductory programming course. In such a course, students learn about the internal workings of a computer such as how it operates, how a person can interact with it, memory organization, control flow, and input/output. Usually, the two courses are taken by the students in tandem. Consequently, during the first few weeks, novices will not really understand how or why their programs, as simple as they may seem, really work. They just assume that the computer understands the source code as written and performs whatever tasks it is supposed to. It is only after a few weeks, if that, when novices begin to make the connection between the material taught in the two classes. This delay in understanding can in the meantime cause students to rely on pre-existing external knowledge in drawing conclusions about the programs that they write, and consequently lay a mental crutch in their minds on which they may rely on in the future.

The pace at which an introductory programming class is taught is a significant contributor to a student's frustration. Textbooks, on which instructors typically base their

syllabus, play an important role in this. In many textbooks, a specific example will be repeated several times, usually whenever a new concept is being presented. This is done to demonstrate how the new concept can be applied to the previous example by building on it. It is also done to articulate how an alternative solution is possible for the previous example by applying the new concept. Students may stumble in their understanding of an example not necessarily because the new concept being present is difficult, but because they did not completely understand an earlier concept which is still being used in the new example [Liffick and Aiken, 1996]. This dependence among examples will frustrate a student, particularly in the case of when a new example builds on the previous one.

As is usually the case, an introductory programming course will be taught using only a subset of a programming language syntax. This approach, as intuitive as it may seem, is met with some resistance since "textbooks and other reference material rarely confine themselves to the selected subset" [McIver and Conway, 1996]. As will be seen later, compilers widen this knowledge gap because they certainly do not confine themselves to the selected subset.

2.1.2 Problem Solving

A major problem area attributed to pedagogy is the teaching and learning of problem solving skills. This area is often neglected or only briefly taught in a first year programming course. Part of this can be attributed to textbooks which often dive right into programming without providing support for problem solving as an initial step in program development. "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" [Pane and Myers, 1996].

When students are assigned to write programs, they frequently generate the source code without any organized thought process. They iteratively proceed by modifying their source code as required until the program generates the correct outputs. In their own experiences as programming instructors, [Suchan and Smith, 1997] claim that the majority of students begin to type their solutions seconds after the initial reading of a problem statement.

Students do not instinctively develop general problem solving skills by programming alone [Pane and Myers, 1996; Olson 1987]. This suggests that if problem solving skills are not explicitly taught, novices will never develop the proper skills and strategies that complement successful program development. Students that do not develop these critical skills will have problems writing programs in advanced courses.

Too often, problem solving methods are not interleaved within a programming language course. There are cases when a programming language can serve as a vehicle to facilitate a particular problem solving method. [Suchan and Smith, 1997] emphasize the use of the Ada 95 programming language to assist in teaching a specific problem solving method. They claim that the use of the language is very beneficial because the language constructs are very closely coupled to the steps of their Engineering Design Method. "The beauty of using Ada 95 as the implementation language is that it has features that map very well to the individual steps of the problem solving method" [Suchan and Smith, 1997].

Understanding the problem statement is the first and arguably most critical step in the problem solving process. If one cannot interpret the problem statement clearly and correctly, then one has not clearly identified the need for a solution, and should not proceed. The stages that follow this discovery stage have been historically debated over the years and as a result many variants have been synthesized. The remainder of this section will identify the three most common problem solving steps that have historically been the most troublesome for novices.

2.1.2.1 Problem Understanding and Analysis Stage. In this stage, the tasks of the problem solver include determining what the goal of the problem is, identifying the given values provided by the problem statement, and determining what the unknowns are. Once the problem is understood and the facts have been made clear, they are then organized and categorized. A frequent problem encountered by novices are escaping defects from this stage of problem solving. In fact, the earlier in the process a defect is introduced, the more potential it has to cause major problems, and the harder it will be to find. A defect at this stage may be the result of a misinterpreted statement or a vague requirement in which the instructor did not clearly convey all of the problem's parameters.

Novices lack a process to make educated assumptions, and in some cases, make the wrong assumptions. A defect of severe consequences may lead an inexperienced problem solver down a path which will lead to a subpar design and subsequently to a non-optimal or even incorrect solution. This leads to a novice's frustration, especially when it appears to them that their (eventual) implementation may be syntactically correct and semantically consistent with their design. Sooner or later, novices realize that

analysis defects always carry forward into their implementation [Suchan and Smith, 1997].

2.1.2.2 Solution Planning Stage. In this stage, the problem solver uses the information gathered and organized in the previous stage and begins to plan a solution. A common and effective approach to solution planning is the practice of subgoal decomposition. Restating the problem's goal in terms of subgoals, when performed correctly, can eventually lead to a simple solution implementation. In addition to subgoal decomposition, good problem solving practices advocate the need to identify alternative solutions.

As intuitive as it may seem, novices lack the ability to perform effective subgoal decomposition. Decomposing while coding is a factor that influences novice programmers' inability to perform effective subgoal decomposition [Bailie, 1991]. When novices lack an appropriate plan, they invent one using pre-programming knowledge [Pane and Myers 1996; Bonar and Soloway, 1989]. As [Bailie, 1991] indicates, one distinction between novice and expert programmers is the ability to plan using effective decomposition because "novice programmers do not possess the rich library of plans from which expert programmers frequently draw." Experts retrieve and assemble plans to solve problems. In contrast, novices solve problems in a linear fashion because they lack well-developed plans [Bailie, 1991; Nichols, 1981]. With experience, novices will eventually exhibit more plan-like behavior by decomposing problems "in a more orderly and balanced fashion" [Rosson, 1996].

It also appears very likely the case that novices lack the desire to execute the required steps of subgoal decomposition because they don't recognize the effectiveness

of such a practice. One contributing factor to this may be the simplicity of the programming assignments. Some students feel that it would be easier for them and take less time if they were to start writing source code without creating a plan for easy problems. This approach may be suitable for trivial problems, but this habit will eventually become hard to break. As a result of this negligence, novices may not develop this essential problem solving skill. It will eventually come back to haunt them when they enter advance computer science courses.

Students can learn effective planning skills through problems that necessitate their use, such as complex programs that require commonly used templates [Bailie, 1991; Lin, 1985]. However, this may seem too overwhelming to the beginner programmer. Another problem that may be encountered in this stage of problem solving is the impulse that novices may have to change something in their plan without a careful thought process. The quicker a solution can be found, they better they feel. As [Spohrer and Soloway, 1986] indicate, "novices may be so eager to optimize their plans that they do not adequately check if the optimization can really be carried out."

For novice programmers, there may also be a downside to the practice of planning alternative solutions. Novices rarely plan alternative solutions because their attention is focused on one solution path. More planning is required when alternative solutions exist possibly resulting in frequent source code changes [Pane and Myers, 1996; Gray, 1987].

2.1.2.3 Solution Design Stage. In the solution design stage, the problem solver takes the solution plan and begins to express it in the form of an algorithm. Subgoals are sequenced and modularized, the relationships amongst them are identified, logic and data structures are established, and algorithm development begins. Novices lack the knowledge to

perform these activities and need a process to guide them. [Suchan and Smith, 1997] observed two difficulties that novices encounter during this stage, and ironically enough they are exact opposites of each other. The first is that when creating an algorithm, the students try to include too much detail by almost writing source code. "This indicates that they created their algorithm at the keyboard or that they do not yet have the experience to use the correct level of abstraction" [Suchan and Smith, 1997]. The second problem is the lack of detail, often just a general plan for solving the problem summarized in a few lines of pseudocode. "The ideal algorithm is one that can be directly translated into code" [Suchan and Smith, 1997].

2.1.3 Program Comprehension

Program comprehension is inherently difficult for novices. [Ramalingam and Wiedenbeck, 1997] define program comprehension as the process that facilitates the understanding of an existing program, typically so that the program can be reused for a higher-level task. It is imperative that students learn program comprehension techniques at an early stage. Students must also develop the specific techniques to write their own programs so that can easily be comprehended by others. Unfortunately, the primary focus in many introductory programming courses is on developing programming solutions to problem statements. Very little attention is given to teaching comprehension skills for programs that have already been written, and more importantly how these programs can be possibly modified and/or applied to other similar problems. Few programming courses stress the teaching of defect discovery strategies which play a vital role in real world program maintenance.

Although not apparent at the time, the ability to comprehend a program coded by someone else will prove to be very valuable to a programmer in a real world software engineering environment. In such an environment, projects typically consist of several software engineers that design and write source code. The chances of one person writing all of the source code for a system or application are very slim. Programmers on a project are usually assigned the task of source code development for certain functionally cohesive areas. Understanding the source code written by another programmer is critical when all of the software components are combined to form a functioning system. The ability to comprehend existing programs is especially important nowadays since "programmers change jobs frequently and new people are constantly being added to projects." [Ramalingam and Wiedenbeck, 1997].

It has been suggested by researchers that a major difference between the program comprehension abilities of novice and expert programmers is the process of organizing program statements into useful groupings [Liffick and Aiken, 1996; McKeithen, Reitman, Reuter, and Hirtle, 1981]. The largest evident problem is in "making the leap from an understanding of individual program statements to the tasks being accomplished by *groups* of statements" [Liffick and Aiken, 1996]. Program functionality can be hard to understand because it is embedded in an assembly of coordinated program functions, perhaps being comprised of many program modules [Ramalingam and Wiedenbeck, 1997; Solloway and Ehrlich, 1984].

Many novices will read the program a line at a time and fail to associate the connective tissue between the program statements. They may view a computer program like they would a cookbook recipe with the expectation that all statements are executed in

the order they appear in a program, including those statements that are within procedures [Pane and Myers, 1996; Sleeman 1988]. Unless students have been trained to take a higher-level approach to program comprehension, valuable time could be spent getting nowhere.

A related area in program comprehension is the lack of training in pattern recognition. Novices are not effectively taught how to recognize patterns in source code. *Beacons* are common source code fragments that hint at the possible presence of high-level operations [Pane and Myers, 1996; Brooks 1983]. An example of a beacon is the swapping operation involved when two integers are sorted. This operation is accomplished with the use of a temporary storage locations.

Novices do not make good use of beacons because they have trouble recognizing patterns [Pane and Myers, 1996; Gellenbeck and Cook 1991; Wiedenbeck 1986]. Experts spend more time viewing important functional areas of the program than novices do [Pane and Myers, 1996; Crosby 1990]. "When experts are asked to comprehend or test programs, they use their general and domain-specific knowledge to develop hypotheses about program content, and then search for beacons that signify the presence of corresponding code structure" [Rosson, 1996].

2.2 Psychological Roots

A discussion on the psychology of programming warrants a book of its own. So much can be said about the thinking process of a novice programmer. It has been said that the task of developing programs may be more difficult for novices than it really should be

because "it requires solutions to be expressed in ways that are not familiar or natural for beginners" [Pane, Ratanamahatana, and Myers, 2001].

2.2.1 The Design of Programming Languages

The psychologies of problem solving and program development have been very extensively researched areas in computer science. One would tend to think that with all that has been said and written to improve these tasks, that there would be very few problems remaining. In the 1980's it was observed that research and funding for the development of new programming languages focused primarily on advances in technology, while practically abandoning psychological dimensions [Pane and Myers, 2000]. This still seems to be true today. The gap of knowledge between novice and expert programmers seems to be growing at an alarming rate. Studies and findings in the psychology of programming seem to have very little impact on the design of new programming languages [Pane and Myers, 2000].

In fact, there is very little separating the psychological aspects of the newer programming languages versus their predecessors. Some new languages make the situation even worse because programming language designers center their focus on usability in the real world, as opposed to making them easy to learn. Emerging languages are driven by innovation and advances in technology and are more powerful and trendier than their predecessors. They are "spawned from technical demands and innovations, and gain their critical mass from adoption by the technical community" [Pane and Myers, 2000].

2.2.2 Human-Computer Interaction (HCI)

Human-computer interaction is the study of humans interacting with computing systems. The study of human-computer interaction deals with how programmers or users relate their experience of using a computer to the real world, typically by associating it with previous, and sometimes unrelated, knowledge. Significant contributors to a negative experience are programming languages themselves. They have been designed without taking into account human-computer interaction issues [Pane, Ratanamahatana, and Myers, 2001; Newell and Card, 1985].

> *Programming language designers are (of necessity) highly intelligent experts in the field of programming, and are consequently far removed both temporally and cognitively from the difficulties experienced by the novice programmer. This gulf of experience and ability results in languages which are either too restrictive or too powerful (or sometimes, paradoxically, both) [McIver and Conway, 1996].*

As a result, novices have to adapt their thinking skills in ways that are unnatural to them. They are required to think about algorithms and data very differently than the way they visualize them in other contexts [Pane, Ratanamahatana, and Myers, 2001]. The variance between how programmers perceive a solution and the way it must be syntactically expressed affect beginners who are just learning to program as well as experienced programmers [Pane, Ratanamahatana, and Myers, 2001]. Simple higher-level operations may even seem unnatural to a novice. An example of this is a user-defined function in the C programming language that computes the average of a list of numbers. This function would require a few lines of source code to add all the numbers in the list and then perform the required division. If the student had used a spreadsheet application in the past to determine the average value of a column of numbers, the *average* operator would appear more natural than using a few lines of source code.

Novices find it difficult to conceptualize what is meant by a variable assignment, how they are stored in memory, and the lifetime of a variable. Some view a variable as an imaginary box that stores any type and amount of data and expect the computer to figure out the intent of a variable based on its name [Pane and Myers, 1996]. They have a difficult time understanding the concept of memory organization within a computer. As a result, frequent problems encountered are violation of boundary and range conditions, precision errors, and calculations that result in computational overflow and underflow. Students are surprised to find that the addition of two positive integers may possibly result in a negative sum.

Another problem that hampers novices is the initialization of variables. Some do not understand why a variable should be initialized prior to its use. Most of these misconceptions are predicated on mathematical experiences. In solving math problems on paper, one does not have to deal with the constraints imposed by a computer and subsequently will not have to deal with the issues mentioned above.

Program development environments and compilers also play a role in originating misconceptions. They are very few environments that animate data flow and control flow during the execution of a program in real time. Without the use of such a visual tool, the novice has a difficult time visualizing the execution of compiled source code residing in the main memory of a computer.

Novices feel that they can negotiate with a computer by using natural language syntax. They also expect a computer to interpret a program statement just as they would [Spohrer and Soloway, 1986]. These are both not possible because the computer does not understand and cannot infer what is intended [Pane, Ratanamahatana, and Myers, 2001;

Pea, 1986]. The use of a natural language as the programming language of choice may be detrimental to a novice. Such a language may blur the distinction of understanding how intelligent a computer is [Pane, Ratanamahatana, and Myers, 2001; Nardi, 1993].

2.2.3 Metaphors

There are common misconceptions in the way that novice programmers perceive a computer. They have a hard time distinguishing between the "notational machine (the one they learn to control) and its relationship with the physical machine (the computer)" [Satratzemi, Dagdilelis, and Evageledis, 2001]. A metaphor is a familiar analogy for how the programming system works [Pane and Myers, 1996]. Metaphors are used to shorten the gap of how a user perceives a programming environment. "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" [Pane and Myers, 1996]. Historically, the von Neumann machine is traditionally used. However, this metaphor has no physical world counterpart [Pane and Myers, 1996]. The lack of a strong metaphor leaves the novice guessing about the model of the environment.

2.2.4 Knowledge Acquisition

Since problem solving and program development tasks are usually new concepts to novices, they are often difficult to absorb without pre-existing knowledge. Meaningful learning occurs when new material is associated with knowledge that already exists in memory [Mayer, 1981; Bransford, 1979]. [Mayer, 1981] identifies the human cognitive

system as being divided up into two types of memory: short term and long term. He hypothesizes the following three step process for meaningful learning to occur:

1. *Reception* - The learner must first pay attention to the incoming information.
2. *Availability* - The learner must process the appropriate prerequisite concepts in long term memory to absorb the new information.
3. *Activation* - The learner must use the prerequisite knowledge so that new material can be connected with it.

Since novices typically do not possess the required pre-existing knowledge about problem solving and program development, they are "forced to memorize" [Mayer 1981]. Given this deficiency, it is clear that the pace of the course can dictate how effectively a student learns. If a concept is taught too quickly or it is taught in an ambiguous manner, the student will not grasp the new concept, and as a result fall behind. This effect can quickly become a problem because the likelihood is that student will continue to fall further behind [Liffick and Aiken, 1996].

2.2.5 The Impact of Heuristics on Cognition

[Pane and Myers, 2000; Nielsen, 1994] discuss human-computer interaction principles, or heuristics, that apply to programming system design, such as consistency, simplicity, speaking the user's language, and error prevention. More specific guidelines such as closeness of mapping, viscosity, hidden dependencies, imposed guess-ahead, and visibility also provide an excellent foundation for assessing programming systems [Pane and Myers, 2000; Green and Petre, 1996]. Unfortunately, it is not easy to optimize all of these aspects at once, because of their interdependence [Pane and Myers, 2000]. By

improving a particular heuristic, performance of another heuristic may be reduced [Pane and Myers, 2000]. As an example, simplifying the syntax of a programming language to the flavor of a natural language may increase its viscosity, the level of effort required to make a small change in the program. Among these principles, [Pane and Mayer, 2000] identify three that seem to cause the most problems encountered in program development: visibility, closeness of mapping, and speaking the user's language.

Visibility is concerned with how much memory a programmer can processes and retain at a time. When attempting to read a program on a computer screen, on average 30 to 40 lines of source code may be visible at a time. This may not be enough for the programmer, especially if subprograms span more than 40 lines of source code. If the system does not make information available or readily accessible, the connections between program components may sometimes not be made. There are very few programming systems in existence that exhibit such a feature. As a result, a novice programmer will most likely struggle in this aspect.

Closeness of mapping determines the level of effort required to transform a mental plan into one that the computer can understand. The syntax of the programming language greatly affects this. Minimizing this difficulty of transformation can help the novice programmer [Pane and Myers, 2000]. If the language does not support an easy translation, programmers are forced to compose low-level primitives to achieve their high-level goals [Pane and Myers, 2000]. Psychologically, the composition of low-level primitives into high-level operations is difficult because they are unrelated to the task and it is hard to see what combination of them will produce the correct actions [Pane and Myers, 1996; Lewis 1987].

[Suchan and Smith, 1997] claim that the Ada 95 programming language exhibits this closeness of mapping. By using their problem solving methodology, once a mental plan has been developed, a structure chart has been drawn, an algorithm has been derived, pseudocode is then written in the form of source code comments. These comments can then very easily be translated into the Ada programming language. This however, is not the case with all languages. Some languages, such as C++, tend to have syntax that appears cryptic to the novice and translation of the mental plan into one compatible with the computer is not guaranteed to be an easy task.

Since novice programmers are just that, novices, they lack computer science domain knowledge. To make up for this deficiency, they tend to rely on their knowledge in other scientific realms. It is important that programming languages *speak the user's language*. When languages use words and symbols that are unfamiliar to users, or that have conflicting meanings in other domains, it is hard for novices to learn their proper use [Pane and Mayer, 2000].

2.3 Programming Language Paradigms

There are several programming language paradigms in practice today. Of these, the most common ones learned in a computer science curriculum are the imperative, object-oriented, and functional paradigms. Curriculums are frequently structured so that a novice will typically start out by taking an introductory course in an imperative language such as C or Pascal. Upon successful completion of the course, the student may then take a course in an object-oriented language such as C++ or Java. If the student feels up to the challenge, a functional programming language such as LISP or Prolog may be learned in

an advanced course on artificial intelligence. Due to the nature of this paper, the functional paradigm will not be discussed because it is not considered to be one that novices will encounter willingly.

Understanding what a paradigm is and how it relates to the real world is frequently a problem for novices. A vast majority of beginners don't even understand the concept of a paradigm, or know that they are actually using one. This can be attributed to the fact that not much is written in textbooks about this topic. This distinction between paradigms is usually made a bit more clear when the novice is exposed to a second paradigm. [McIver and Conway, 1996] contribute the following:

> *The underlying pedagogical difficulty is that students are not used to solving problems in a single pure paradigm. Much of the problem solving they do in the real world is procedural in nature (cooking a meal, totaling a restaurant bill, etc.), but other problems with which they are familiar are more amenable to constraint solving (dispute resolution, holiday planning, budgeting), a functional or pipeline approach (collaborative tasks, various types of component assembly), or even object-oriented methods (using an automated teller machine, learning physical skills).*

Some argue that it is important for a student to learn an imperative language first because it lays the cognitive foundation for the comprehension of other paradigms. It is also important to realize that when introducing a new paradigm to a novice, negative transfer from the prior paradigm may be retained as a consequence [Pane and Myers, 1996; Mendelsohn 1990, Siddiqi 1996, and Wiedenbeck, 1996]. A programmer that has previously learned an imperative language should have very little difficulty learning an object-oriented language with respect to the basics of programming such as variables and functions [Hagan and Markham, 2000]. This is especially true for the C and C++ languages because the former is a subset of the latter. However, in learning the advanced topics specific to the object-oriented paradigm, these ideas may contradict what the

student knew before especially if the language did not have a direct ancestor [Hagan and Markham, 2000].

Both of these consequences above are evident when students take their first course in C++. Their prior experience with C may influence their learning of the new paradigm. The programs that they write frequently do not take advantage of the benefits of the object-oriented features of the language. Many of them think that their programs are object-oriented based solely on the fact that instantiations of a *class* are made. Little do they realize that the *classes* that they have created are very similar to *structs* in C, the only difference being the default access privileges to the data members of the class. It is also difficulty for them to decide which logical units should be represented by objects and which as attributes of the objects [Pane and Myers, 1996; Détienne 1990].

As difficult as object-oriented programming can be to novices, some research has shown that it is more natural than the imperative approach [Ramalingam and Wiedenbeck, 1997; Borgida, Greenspan, and Mylopoulos, 1986; Rosson and Alpert, 1990]. One argument supporting this is that object-oriented design provides a better match to the way that designers conceptualize problems [Ramalingam and Wiedenbeck, 1997; Rosson and Alpert, 1990]. Object-oriented programming centers its focus on active objects, relationships, behaviors, and interactions whereas imperative programming languages emphasize decomposition into procedures that act on passive data structures [Ramalingam and Wiedenbeck, 1997; Rosson and Alpert, 1990]. However, in a study of error rates in program comprehension conducted by [Ramalingam and Wiedenbeck, 1997], it was determined that there were higher overall error rates in the comprehension of object-oriented programs than there were in imperative style programs.

2.4 Programming Languages

Nowadays, computer science curriculums favor the teaching of C++ in introductory programming courses. This language, and others like it, may not be a good choice for a novice's first exposure to programming because of how powerful they are. These languages are generally solution-oriented and intended for expert programmers. As an example, C++ provides high-level features such as information hiding, dynamic binding, inheritance, and polymorphism. In addition, it supports low-level features such as bit manipulation of raw pointers and memory allocation.

This freedom to program on opposites sides of the spectrum can confuse novices because they may not be able to distinguish between levels of abstraction. It has been observed by [McIver and Conway, 1996; Hofstadter, 1979] that novices have a hard time maintaining two or more conceptual perspectives at the same time. Powerful languages make writing complex programs easier but can be counter-productive for novice programmers [McIver, 2000]. In a paper on programming languages and their pitfalls, [McIver and Conway, 1996] identify the following reality:

> *Dichotomies of perspective, such as syntax vs semantics, static vs dynamic structure, process vs data, complicate the teaching of any programming language. The availability of low-level implementation-oriented constructs and high-level solution-oriented features in a single language only serves to increase substantially the already considerable cognitive demands placed on the student.*

One approach to improving the comprehension of language syntax and semantics is to design programming languages that appear more natural to a user. There are several natural languages in existence today, however they too are not flawless. Languages with natural-language-like syntaxes suffer from many usability problems [Pane,

Ratanamahatana, and Myers, 2001]. Where they make up for readability and simplicity, they lack in other heuristics such as viscosity.

It should be important to note that "striving for naturalness does not imply that the programming language should use natural language" [Pane, Ratanamahatana, and Myers, 2001]. The subsections that follow identify a sampling of programming language difficulties that novices often encounter. To include all of them would warrant a paper of its own, therefore, the list is shortened considerably.

2.4.1 Issues Involving Syntax

The complexity of a syntax will have a significant influence on how easily a novice will learn a programming language. In a study of novice problem solving paths conducted by [Satratzemi, Dagdilelis, and Evageledis, 2001], a conclusion was drawn that language syntax can be a negative factor in problem solving. A computer science department's choice of a first programming language is, therefore, crucial in first year student retention. The language must not be too complex, and at the same time, not too obscure.

Complex languages are well-known for the great number and variety of data types, as well as the operations that can be performed on them. "A wide range of features necessitates a commensurately complex syntax and often also entails a host of implicit operations and function calls, automatic conversions, type inferences, and resolutions of overloaded functions, variable and function scoping" [McIver and Conway, 1996].

On the opposite side of the spectrum are programming languages that have a very simple and primitive syntax. In such languages, syntax may be considered as cryptic to novices as complex syntaxes are. An example of this is the early LISP commands, CAR

and CDR. Another example comes from the Scheme programming language. In Scheme, there is only one data type, the list, and one operation, the evaluation of a list. "While this abstraction is very simple to explain, and not difficult for the beginner to grasp superficially, it does result in code that is difficult to read because of large numbers of nested parenthesis and the absence of other structuring punctuation" [McIver and Conway, 1996].

A syntax that provides operations that closely resemble other operations can frustrate a novice. In the C programming language, a common problem is the use of the assignment (=) and equality (==) operators. Similarly, in Pascal a case can be made for assignment (:=) and equality (=). In both languages, they are frequently inadvertently interchanged. The significant difference being that syntax prohibits one of them to be used where the other is legal [Pane and Myers, 1996]. In C, the following two statements would legal: *IF (a=0) b=1* and *IF (a==0) b=1*, though they may provide different results. The consequence of confusing these two operators is less damaging in Pascal because the use of the assignment operator is not permitted in a conditional statement.

Another area of confusion is the proper *timing* of syntax usage. A hot spot in this area is array indexing. In C, different rules exist for array declaration, initialization, and manipulation. If an array has been declared without being initialized, a different syntax applies for the insertion and deletion of the array elements. Initialization of arrays is done with curly braces (a[i] = {1, 2, 3}), whereas square brackets (a[0]=1; a[1]=2; a[2]=3;), are used for array indexing [Pane and Myers, 1996].

2.4.1.1 Syntax of Looping Constructs. Coding a loop can be a complicated task for a novice. Even though an algorithm may have been designed correctly on paper, the syntax of a loop may prevent the correct implementation of the loop. A few problem areas inherent to loops are the use of extra variables to count iterations [Pane, Ratanamahatana, and Myers, 2001], terminating conditions, and in some cases, when the iteration of a loop variable should be performed. In C++, a *FOR* loop requires a loop variable. Novices have a hard time understanding the need for it. A typical reaction from them is: "why can't you just specify that the loop is to be executed *n* times? Why does another variable have to be introduced?" The FOR loop also requires a terminating condition. There are several things that can cause confusion here. One of the major concerns is choosing the terminating condition to be $x<n$ or $x<=n$, where x is the loop variable and n is the number of iterations to be made.

Operations involving an array via a loop can lead to problems because the index of an array usually starts at 0 and goes to $n-1$, where n is the number of elements. By practice, loops usually start at value 1. Novices also may expect a loop to terminate abruptly as soon as the terminating condition is achieved, rather than waiting until the statements inside the loop have been executed a final time [Pane, Ratanamahatana, and Myers, 2001; du Boulay, 1989].

For novices, there is great confusion between the concept of the prefix and postfix increment operators for languages that support these features. The former increments the loop variable prior to execution of the loop, the latter increments the loop variable after the execution of the loop. Novices have a hard time deciding which to use, most likely because they are not trained to know which situation would warrant their use.

Recursion is also a very troublesome area. Beginners resort to using iteration as an alternative to recursion [Pane and Myers, 1996]. The underlying problem with recursion is that most novices cannot visualize separate and unique invocations of a function as well as flow control in a recursive function [George, 2000a].

2.4.1.2 Syntactical Synonyms, Homonyms, and Elision. A category of conundrums that frequently contributes to problems understanding programming language grammar is *syntactic synonyms*. This occurs when there exists more than one way to specify an entity. Synonyms frequently appear in introductory languages and are "analogous to certain sophisticated grammatical constructs in natural languages" [McIver and Conway, 1996]. It has been observed that novices have a hard time differentiating between two different instructions that accomplish the same operation [Pane and Myers, 1996; Eisenberg, 1987].

An example of a syntactic synonym in C++ is the dereference operation. When an array *A* of 10 integers has been declared (e.g. *int A[10]*) , the array variable name *A* represents a pointer to the memory location that contains the first integer of the array followed by the nine contiguous integers that were also allocated for the array. Assuming that the array elements have already been entered into the array, there are two ways to access the element in the first location using dereference operations, $j = A[0]$ and $j = *A$, where *j* is used to hold the value of the integer resulting from the dereference operation. These synonyms are minor irritants in learning a language, however they can also have a more serious and insidious effect by blurring the underlying programming concept in the student's mind, because that concept is no longer associated with a single clear syntax.

Conversely, yet equally confusing, are *syntactic homonyms*. These are constructs whose behavior is decided by the situational context in which they are used. These idiosyncrasies present "a more serious flaw in a language" [McIver and Conway, 1996]. Novices assume that because a construct works in a certain situation, the same behavior will be observed in all other situations. An example of this is pointer arithmetic. The use of the "++" operator on a pointer variable in C++ moves the location of the pointer to the next element in the data structure according to the number of bytes of the data structure element. It does not move it a fixed number of bytes for all situations.

Elision, the omission of a syntactic component from a grammatical construct, also contributes to the difficulty in learning a syntax [McIver and Conway, 1996]. When comprehending programs written by experienced programmers, novices will sometimes encounter these situations. Textbooks at times leave some component of an instruction out without justifying it with an explanation. In C++, novices are usually taught that functions which do not return a useable value, such as the *main* function, should still return an integer value, zero. However, experienced programmers may choose an alternative method by declaring the function to have a return type of *void*, in which case a *return* statement is not required at the end of the function body.

2.4.1.3 Syntax of Operator Precedence. Operator precedence varies from language to language. The error rate for novice programmers with respect to operator precedence is very high [Pane, Ratanamahatana, and Myers, 2001; Spohrer and Solloway, 1986]. Rather than following the operator precedence rules of the language, novices sometimes expect program statements to execute in their order of appearance [Pane and Myers, 1996]. Many students do not realize that operator precedence in a programming language

is not the same as it is in mathematics. Operators that are foreign to math, such as the C++ stream operators "<<" and ">>" are major contributors. Students usually learn that these errors are quite common and learn from their mistakes by adopting practices such as the use parentheses to ensure that the operations are carried out in the correct order.

2.4.2 Semantics and Pragmatics

Semantic knowledge refers to understanding the underlying mechanics of syntactical constructs. Pragmatic knowledge refers to understanding why a certain syntactic construct is used in lieu of another even though they may achieve the same end result. The distinction between the two is often blurred especially in a novice programming arena.

Semantical errors can hinders a novice programmer's progress because they are not clearly visible. In some cases they are not even remotely visible. Errors in programs are particularly difficult to find when they are of a semantic nature because novices may not fully understand the syntactical constructs that are being used. Compilers do not and can not find semantical errors because they do not know what the syntactical elements of a program are intended to achieve. This lack of support significantly magnifies this challenge.

Pragmatic knowledge is gained through experience and is something that novices are deficient in because of their lack of experience. Novices have very little, if any, rationalization skills regarding pragmatics. Often, they will choose a particular method of achieving a task simply because they understand the semantics of the chosen method better than the alternative method. At times, they do not even know that an alternative

exists or whether it would apply in the current situation. When alternatives exists, they often cannot make judgments as to which method is more applicable. An example of this is the use of recursion vs iteration when coding a loop. Since recursion can be very difficult for beginners to understand, they frequently resort to the use of iteration in loops. They may understand that that recursion in certain situations may outperform iteration, but choose not to use it because of its complex nature. Hesitation to use a syntactical construct such as recursion is harmful because some solutions can only be achieved through the use of recursion.

2.4.2.1 Semantics of Data Structures. Understanding the semantics of data structures can also pose a challenge to a novice. Complex data structures such as linked lists, stacks, and queues are particularly troublesome because there is more to the data structure than just the data type being stored in the list. The elements of a data structure may contain pointers to other elements, possibly several, depending on the intended usage of the data type. Interacting with these data structures is frequently a problem and pointers are commonly left dangling because they are not all accounted for when these data structures are manipulated.

Arrays, which are simpler to understand than linked lists, also cause problems. When dealing with arrays, novices associate the data structure as a series of boxes, each one capable of holding a data element. In a study conducted by [Satratzemi, Dagdilelis, and Evageledis, 2001] it was observed that students have a great tendency to confuse array positions with array elements. Students usually forget to consider boundary conditions, like arrays with no elements, or arrays with only one element [Satratzemi, Dagdilelis, and Evageledis, 2001]. Sometimes students inadvertently cross array

boundaries, especially when a loop is involved, and obtain unpredictable errors because memory locations that were not allocated to the array are being accessed. They may also not understand that the sorting of an array must be carried out explicitly by the programmer [Pane and Myers, 1996; Hoc 1990]

2.4.2.2 Semantics of Looping Constructs. To novices with very little programming experience, a loop is a foreign concept. They have a hard time correlating it with an operation in the real world. To a novice, a loop may seem like a complex way of specifying operations that can easily be expressed with aggregate set operations [Pane, Ratanamahatana, and Myers, 2001] such as a *sum* operator. Even the semantics of fundamentals such as the *while* loop can be startlingly difficult to comprehend [McIver and Conway, 1996].

The difficulties that novices frequently encounter when expressing their solutions with looping constructs are greatly due to the semantics of loops. This is evidenced by a inclination for them to avoid using loops by repeating instructions [Pane and Myers, 1996; Hoc, 1989; Onorato, 1986]. They lack the ability to generalize and have a hard time visualizing the behind-the-scenes actions of a looping construct [Pane and Myers, 1996; Kessler, 1989; Hoc, 1989; Onorato, 1986; Pirolli, 1985].

2.4.2.3 Semantics of Boolean Logic. The semantics of boolean expressions are also a great concern. The proper combinations of *AND* and *OR* in condition statements are the cause of many novice bugs. [Pane and Myers, 1996]. Creating the correct boolean expressions can be particularly difficult, especially if the student has not learned boolean algebra in an introductory computer organization course. The negation operation *NOT* is a major culprit because students fail to apply a simple boolean logic principle that the

AND and *OR* operators are interchanged when an expression is negated. Also clouding this issue is the syntactical aspect of certain programming languages. In boolean algebra, *ab* represents *a AND b*, and *a + b* represents *a OR b*. However, In C++, the syntax for the two is *a && b* and *a || b*, respectively. Furthermore, in C++, the operators *&* and | are confused with *&&* and || . The former pair specifies bit wise boolean operations, while the latter pair specifies boolean operations that are performed based on the logical state of the operands. Consequently, the ability to correctly formulate boolean expressions does not come naturally.

2.4.3 Abstraction and Modularization

Abstraction and modularization have been proven to be essential software development principles. These powerful concepts, as helpful as they can be to a programmer, unfortunately initiate comprehension problems for novices. They consider these concepts as high-level topics and tend to shy away from these chapters in a programming textbook.

Abstraction hides details by providing a simple "cover story" [Bucci, Long, and Weide, 2001]. A simple example of abstraction is the calling of a function to perform a specified operation. The details of the function do not need to be known within the part of the program that called the function. All that needs to be known is that given the input parameters to the function, the correct end result is produced. Novices do not always introduce functions into their programs, making them more volatile, hard to maintain, and difficult to comprehend.

Typically, beginners will write a program without using abstraction. It is not known to them that such bad habits can increase program complexity. This eventually

leads to program comprehension problems as the size of a program grows. Source code that is not abstracted when it makes sense to, can eventually lead to a program that is very difficult to understand and maintain. Simple concepts, such as writing a function to swap two integers are sometimes just coded by the novice into the main body of a program without the use of functions. This usually indicates that the student has not properly designed a solution plan.

Problem solving, which is frequently omitted by novices, is a prerequisite for abstraction. In problem analysis and solution planning, subgoal decomposition leads to the origin of abstraction. Each subgoal, when decomposed to exhaustion, will become an abstraction to the level from which it was derived. Upon complete and successful decomposition, each subgoal can eventually be coded into the desired programming language [Bailie, 1991; Miller and Miller, 1987]. If decomposition was performed correctly, the main portion of a program will not contain any implementation details of a subgoal (function, method, or procedure). Subsequently, each subgoal should not contain any implementation details of the subgoals that it is comprised of.

Modularization is the process of deploying an effective software engineering concept, source code reuse. This involves the creation and use of small programs that each usually contain of a library of related functions. Expert programming activity is facilitated through the adaptation of reusable source code towards the solution of a new problem [Bailie, 1991; Solloway, 1982]. New programs that need to be written can be done quicker because parts of it may have already been written. For a new program, all that needs to be done is to correctly access the pre-existing modules from within the new program.

Novices have a propensity to underutilize modularization. Part of this problem stems from the nature of programming assignments. Most programming instructors assign programs that have little or no relevance to earlier assignments. Students are not being trained to reuse source code that they have already written because of the assignment strategy used by an instructor.

Modular programming is frequently taught using a top-down strategy [Pane and Myers, 1996; Wirth 1983]. In this method, the program, first described at an abstract level, is subsequently refined into a modular hierarchy. Novices have a hard time with top-down strategies because their plans are based on step-by-step execution rather than actions that are performed by program components [Pane and Myers, 1996; Rogalski, 1990]. Novices may not have developed the skill of reading a program in a top-down manner. Instead they read the program like a book [Pane and Myers, 1996; Gellenbeck 1991; Jeffries 1982; Wiedenbeck 1986].

2.5 Debugging Skills

Very rarely does a program work perfectly upon the first execution attempt. Therefore, debugging proves to be an essential skill that a novice must develop. Debugging a program typically involves four aspects. First, the programmer must identify and correct all syntactical errors. Usually, the identification of these types of errors is a service provided by the compiler or interpreter. Once the program is syntactically correct, the student can then begin to test the functionality of the program through a previously identified test plan.

During testing, the student may encounter two types of errors. Run-time errors are those resulting from memory access violations, unsafe data structure usage, or any condition that causes the program to abort execution. Logical errors which do not affect the execution of a program are those that a student finds when the results of testing are evaluated. Once the sources of these errors are identified, in the source code and possibly in the design, the student makes the appropriate corrections and then re-evaluates the results. Finally, once the program seems to be functioning correctly, the student may choose to optimize certain areas of the source code in the interest of performance, readability, or other heuristics.

Though various debugging strategies are often taught in programming classes, "much of the skill of debugging is learned through the experience of writing programs" [Gugerty and Olson, 1986]. In a program comprehension experiment, [Gugerty and Olson, 1986] found that experts were able to identify more errors than novices, and in less time, mostly because they spent a significant amount of time comprehending the program and that they developed a more complete representation of the program. Since novices lack experience in program comprehension skills, it is evident that debugging is an activity that does not come easy for them. The authors concluded that novices also inadvertently injected errors of their own into the program while debugging. This is most likely due to drawing an incorrect conclusion about what the defect may be, whether it is or is not a defect, or attempting to correct a potential defect without examining all possible side effects of doing so. Another frequent cause of this is that "students repeatedly attempt to correct their errors without understanding the meaning of the error message produced by the compiler" [Satratzemi, Dagdilelis, and Evageledis, 2001].

2.6 External Influences

A major problem with novice programmers is their reliance on knowledge acquired in other domains. The fact that novices commonly guess at syntax and semantics, results in errors and confusion because programming language semantics may not compatible with semantics in other domains [Pane and Myers, 2000; Pane and Myers, 1996; Hoc, 1990]. The knowledge that they use interferes with the correct understanding of the language [Pane and Myers, 1996]. They frequently fall back on their natural language tendencies resulting in errors that arise from the incompatibility with the programming language constructs [Pane, Ratanamahatana, and Myers, 2001; Bonar and Soloway, 1989]. Among the influences are the numerous engineering disciplines, mathematics, and psychology. Since the nature of programming relies heavily on performing computations, mathematics is the most notorious culprit.

Languages often use keywords that are similar in appearance to those used in mathematics [Pane and Myers, 1996]. There are many inconsistencies between mathematical notations and their programming language counterparts such as the property of symmetry: $a = 2$ and $2 = a$ are only symmetric in the math domain [Pane and Myers, 1996]. Other inconsistencies between the two domains are [Pane and Myers, 1996; Hoc, 1990]:

- The lifetime of variables. Math treats the identity of two symbols as permanent for the duration of the problem, while it is transient in programming.
- Initialization of variables is not required in math.
- In math, the summation operator can have an arbitrary number of arguments. For some programming languages it is only a binary operator.

2.7 Other Contributions

In addition to the factors attributed to problem solving, program development, and programming languages, novices are faced with other challenges that may also cause problems in their learning experiences. This includes the use of various computer architectures, compilers, and program development environments. The subsections that follow will present an argument for each of these three factors.

2.7.1 Hardware Dependencies

There are numerous computer architectures in existence today. Most programming languages are designed for use on several different platforms. Portability issues increase the demands on novice programmers because they are "forced to contend simultaneously with the constraints of the underlying hardware" in addition to having to learn the syntax of a programming language [McIver and Conway, 1996].

Today, the most common operating systems in use are the *Unix* and *Microsoft Windows* suites of operating systems. C and C++ have been designed such that source code can be compiled and executed on both operating systems. However, the execution of the same program on both platforms will not necessarily provide the same results because of the different hardware and operating system architectures. An example of this inconsistency is the number of bytes that each platform uses to represent a data type. An integer on a Windows platform may be 64 bits long, whereas the same declaration on a Unix machine may only be 32 bits long. A novice who may not know such a fact would be astounded to find that the result of a mathematical operation produced the correct result on one platform, and a totally unexpected bizarre result on another.

> *There seems no convincing reason why the novice student, already struggling to master the syntax and semantics of various constructs, should also be forced to deal with the details of representational precision, varying machine word sizes, awkward memory models, or a profusion of conceptually equivalent but semantically distinct data types* [McIver and Conway, 1996].

With the competition in the real world, it is highly unlikely that all machine architectures and operating systems will ever exhibit the same behaviors. Someone is always trying to develop the newest, cutting-edge architecture that gives an edge over their competition. As a result, this problem is one that may never be solved.

2.7.2 Compilers

Compilers play a very important role in program development. They serve as an interface between the programmer and the computer, and as such, human-computer interaction issues come into play. A compiler's primary function is to translate programming language statements developed by the programmer into executable binary code and, if not entirely successful, to report error messages to the programmer. A compiler serves as a syntactical traffic cop by only allowing syntactically correct statements to be compiled. Those statements that have violated the syntax rules of the programming language are flagged by the compiler, usually resulting in log of error messages visible on the screen.

The complexity level of a compiler will greatly influence a novice programmer. If the compiler is too cryptic in providing error messages, as many popular ones are notoriously known for these days, a beginner will endure great difficulty in finding errors in their source code. Most introductory programming instructors advise their students to use compilers that are simple and easy to use. However, some students are able to obtain

access to popular commercial compilers such as Microsoft Visual C++, and as a result, have a difficult time in learning the how to use it. The focus is drawn from *how to program* to *how to use a complex compiler*. The use of production compilers should be avoided at a novice level because are not tailored for a student that is learning a language or how to program [Lewis and Mulley, 1998].

When an error is found by a compiler, an indication of what type of error occurred along with the line number of the offending statement source code is usually presented on the screen. This is often where a novice can encounter problems. There are generally three categories of errors that a compiler could produce: an informative error, a non-informative error, and an incorrect error. An informative error is one in which the correct line of source code is identified along with a supportive error message. A non-informative error is one in which the correct line of source code was identified, but the error message was less informative, somewhat cryptic, or beyond the scope of the programmer's knowledge. An incorrect error results when the compiler erroneously indicates that another part of the source code is the source of the problem.

The difficulties encountered in the use of compilers are closely intertwined with the material presented in a classroom textbook. Textbooks attempt to teach a subset of a language syntax. Usually the most useful and easiest to learn material is covered in an introductory course. Compilers, on the other hand do not limit themselves to such a subset [McIver and Conway, 1996]. The errors messages that they present may be beyond the scope and understanding of a novice student. As a result, these students have to deal with the full semantics of the language, even though they don't understand most of it.

A common omission that novices make in the use of compilers is the need to recompile source code upon modification. They are confused as to why the source code must be recompiled even for minor changes [Pane and Myers, 1996; du Boulay 1989]. In the case of programs that consist of multiple program modules in different files, some also feel that a source code modification in one module should result in a complete recompilation of all modules.

A compiler can be a very beneficial tool in a novice's learning experience. Once students learn how to navigate the functions of a compiler, they can learn to harness the information provided by the compiler and apply it to existing knowledge. As [Lewis and Mulley, 1998] state, a compiler can encourage "interactive learning during the program development process."

2.7.3 Program Development Environments

Today, there are many program development environments available to help the novice. Most of these environments are designed to overcome the many facets of problem solving and program development difficulties. It has been argued that many common features of program development environments are in fact not beneficial to novice students. As an example, many environments use different colors to highlight reserved keywords. This may be appealing to the eye, but may not necessarily serve a good purpose. Some researchers argue that only vital semantical information should be highlighted [Pane and Myers, 1996; Baecker, 1986].

On the other hand, such a color signaling feature can be very helpful. In C it is very easy to accidentally comment out a large block of source code since it is not a

syntactical error to have a missing close-comment delimiter [Pane and Myers, 1996]. Modern environments use color signaling to make this error visible [Pane and Myers, 1996] but these environments are not always available.

CHAPTER 3

ASSESSING MODERN DAY NOVICE LEARNING TOOLS

Research through the years has provided a solid foundation for the development of learning tools that stimulate learning in problem solving and program development. The term *learning tool* in the context of this thesis refers collectively to the software tools, models, and methodologies that enhance the processes in which novices learn. These tools vary in their use, presentation, and usefulness.

This chapter is dedicated to identifying and discussing modern tools that are in existence today. Some of these tools are designed to assist a novice in a certain stage of the problem solving or program development process. Others are quite comprehensive and may cover several stages. Although most of these tools are specific to a particular programming language or have been designed for a specific programming course at a university, the intent of this chapter is to identify the *nature* of the tools that are at a novice's disposal. Five categories of tools will be presented: tools that support problem solving, tools that simplify program comprehension, miscellaneous learning aides, program development environments with limited features, and a comprehensive program development environment.

3.1 Support for Learning Problem Solving

Perhaps the most important yet least supported aspect of the development of programs is problem solving. It is well understood through research that effective problem solving must preclude program development. Unfortunately, the development of problem solving skills eludes novices because they are rarely taught explicit problem solving strategies in

introductory programming courses. Also to their disadvantage, there are very few tools available that provide support for this critical stage of the software development.

In subsection 3.1.1, an effective problem solving methodology is presented which makes program development a simple task. By following the steps outlined by the authors, a student can easily code a software solution to a problem. In subsection 3.1.2, a unique problem solving tool aimed specifically at guiding a problem solving session is discussed. The tool is designed to train novices an effective approach to problem solving by supporting them through a structured process.

3.1.1 Programming Made Easy by a Problem Solving Methodology

Majors William K. Suchan and Todd L. Smith [Suchan and Smith, 1997] of the United States Military Academy have compiled strong evidence to support their success of a first semester programming course. The class introduces novices to programming with a strong emphasis on problem solving. To support the problem solving methodology, the instructors advocate the use of Ada 95 as the programming language of choice because of how naturally a software solution can be created once the problem solving steps have been executed. They claim that the use of Ada 95 also enhances a novice's understanding of each problem solving step that was encountered along the way through reflection.

It is important to note that the course is not intended to teach programming. Instead, programming with the use of Ada 95 is used merely to demonstrate the effectiveness of the problem solving methodology taught in the course and how simple programming can be once the pre-requisite steps have been executed. The problem

solving methodology used in the course is outlined in the Engineering Design Method that appears in an Ada 95 textbook authored by [Feldman and Koffman, 1996].

The foundation of the course is comprised of three basic building blocks: sequence, selection, and iteration, and is coupled with the problem solving methodology to provide students an easy approach to a working software solution. The six stages of the problem solving methodology are summarized sequentially as follows: problem statement understanding, problem analysis, solution design, test plan development, solution implementation, and test evaluation.

Problem statement understanding requires the student to read and comprehend the problem being presented. This requires interpreting exactly what the problem is, what the goal of the problem is, and whether or not that goal may be achieved given the information provided by the problem statement. Once the problem statement has been read and understood, the problem solver proceeds to the problem analysis stage for the discovery of components that will be required to derive a solution.

During the problem analysis stage the student identifies inputs, outputs, constraints, assumptions, constants, variables, and formulas. Although not discussed in their work, it is assumed that the extraction of all such information results in an organized collection of solution-critical components. The student will use this analysis later when developing the declaration section of the program. An effective analysis results in the correct identification of all variables as well as proper variable initialization.

Using the information gathered in the previous stage, the student then begins to develop a solution design. This initial steps of solution design involve the creation of a structure chart and an algorithm. Figure 1 contains a generic example of such a structure

chart. As is evident, the structure chart follows a commonly used strategy for subgoal decomposition: top-down design. The structure chart serves as a graphical representation of the solution design. In it, the student identifies subgoals with rectangles, and inputs and outputs to these subgoals with directional arrows entering and/or leaving the rectangles. The arrows indicate the directional use of parameter passing (in, out, or both) and "provide a tool whereby the students can visualize the information flowing out of one module and in to another, or in to a module and out of that same module."

The authors claim that the use of a structure chart greatly reduces parameter-passing errors during the implementation stage. During the solution implementation, the boxes in the structure chart will map to functions and procedures as applicable. In declaring the functions and procedures and their parameters, errors involving the direction of parameter passing will be reduced because the direction of each parameter will be derived directly from the structure chart.

Once the structure chart has been derived to the desired level of abstraction, an algorithm is then created with the aide of the structure chart. It is important to note that a structure chart uses no graphical means of representing sequence, selection, and iteration. These elements of a solution first appear in the algorithm. In designing an algorithm, the student must logically piece together the elements of the structure chart with the use of the three syntactical constructs as required. "An ideal algorithm is one that can be directly translated into code."

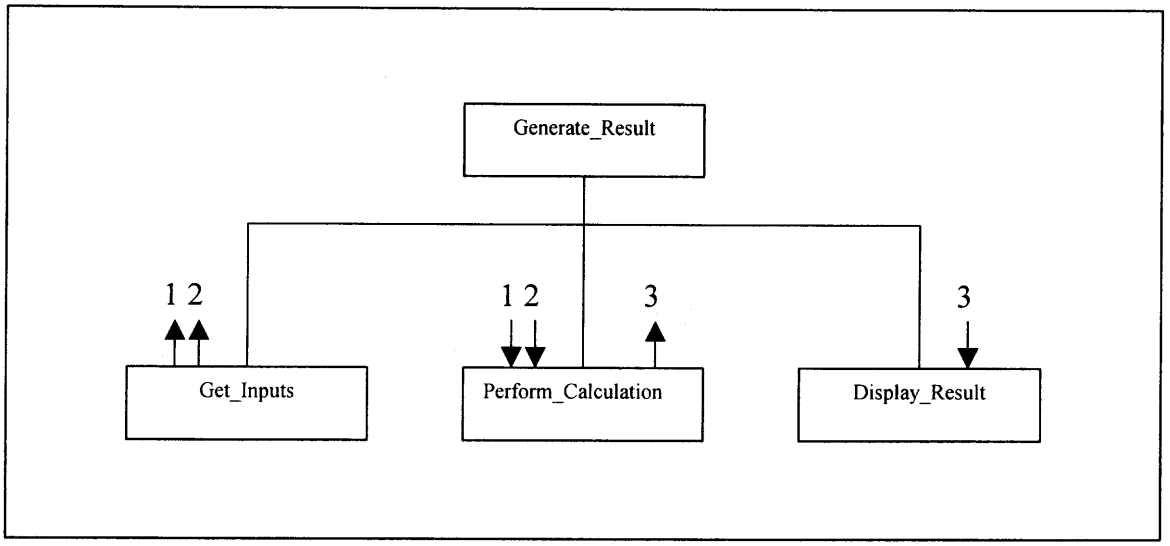

Figure 1 Generic Structure Chart

The authors do not indicate how the development of a test plan, the fourth step of the problem solving methodology, benefits from the use of Ada 95. It is therefore assumed that there is little, if any benefit. For completeness, it is noteworthy to mention their recommended approach to test plan development. The authors advocate the use of several test cases, each consisting of the inputs to be tested, the results that are expected, and its purpose. The test cases should cover a broad range of inputs, using a technique called "the *good*, the *bad*, and the *ugly*." The *good* inputs are valid inputs to verify program correctness. The *bad* inputs purposely exceed specified boundary conditions to see how a program handles such situations. The *ugly* inputs are used to verify that data of the wrong type is not accepted as valid input.

At this point, the student is ready to implement the solution. The student first enters source code comments and a program header, followed by a declaration section. The declaration section is derived directly from the analysis. The student simply enters all the necessary variable names, data types, ranges, and includes necessary initialization.

The algorithm is then entered in the body of the program in the form of pseudocode, line by line. Upon completion, the pseudocode is then easily translated into Ada 95 syntax. If the student had performed an effective analysis and designed a correct solution, the program should function properly at this point.

Finally, the student then evaluates the test cases defined in the prior stage. If the test plan is comprehensive and effective, the student may find errors that originated in the analysis and design stages. The similarities between the Ada 95 syntax and the output of the analysis and design stages allow for an easy process of finding errors. The authors admit that it may take several passes through the methodology to finalize a correct solution, but in their experiences, the correct source code is produced when the entire process is followed as it was meant to be.

3.1.2 GPCeditor: A Scaffold for Problem Solving

It is rare to find a program development environment that provides explicit support for problem solving. Developed at the Georgia Institute of Technology, the *GPCeditor* [Guzdial, Hohmann, Konneman, Walton, and Soloway, 1998] facilitates problem solving and program development in Pascal by providing a scaffold that guides novice problem solvers through a structured process. The intent of GPCeditor is to train novices an approach to properly developing programs until it becomes an integral part of their cognitive skills and the scaffold is no longer required. The GPCeditor isolates the programming aspect by focusing on goals and plans that achieve those goals. Forming the cognitive foundation for the GPCeditor are three logical steps: goal, plan, and code, hence the name **GPC**editor.

In the first step, a student identifies the goals in the problem statement. In the second step, plans for achieving those goals are defined. This can be done in one of two ways. Plans can be defined in terms of subgoals (as required) which will eventually themselves be defined as plans, or they can be defined through the utilization of pre-defined plans from a *Plan Library*. Sample plans provided by the authors are shown in Figure 2. Finally, these plans are transformed into source code and inserted into a program.

GET INPUT plan:

```
writeln (Prompt_Message);
readln (Input_Variable);
```

VALID DATA ENTRY plan:

```
repeat
write (Prompt_Message);
readln (Users_Choice);
if (Users_Choice > Floor Value) then
begin
Entry_Is_Valid := true;
end
else
begin
Entry_Is_Valid := false;
writeln (Error_Message);
end;
until (Entry_Is_Valid = true);
```

Figure 2 Sample Goal Plans

It is important to note that in the GPCeditor, source code is not entered directly by a student. The student assembles the source code contained in the plans to achieve the goals of the problem. The GPCeditor is unique in that it teaches the importance of problem solving while drawing the task of programming (and consequently issues of

syntax and semantics) completely away from the program developer. It is indeed a good tool for a novice just starting an introductory programming class.

On the GPCeditor screen, three menu selections facilitate the design of the solution: *Decompose*, *Compose,* and *Run*. The first selection involves problem analysis, goal discovery, and the mapping of plans to those goals. Under the *Decompose* menu, a submenu of three operations is displayed. These three operations exist for the purpose of manipulating goals, plans, and data objects. Plans cannot be created until goals are established, and as a result are disabled in the menu until they are applicable. To start, a student establishes a goal, names it for identification purposes, and provides a description of its process. Goal descriptions are subsequently displayed whenever the corresponding goal is selected in the *Goal-Plan List* view. Once goals are established, matching plans which contain the details of how the objectives of a goal are carried out, can be defined.

As described earlier, plan definition can be achieved in terms of subgoals or through the use of the Plan Library. The library is a repository of categorized information about plans. It includes the source code for each plan, what data is required to carry out the plan, and examples of the plan in use. Initially, the library consists of basic Pascal statements, functions, and some plan hierarchies.

As students create customized plans, they can be saved in the library for future reuse. This is one of the great benefits of GPCeditor: it encourages the learning of software reuse, a powerful and modern real-world technique. Once the goals and matching plans have been established, the student uses the *Data Objects Workspace* to specify the data that each plan requires. The workspace allows the student to match the

required plan data to existing data objects in the program as well as the creation of new data objects.

The next stage is the composition of the plans to create a solution. The operations to execute this step appear under the *Compose* menu item. The GPCeditor limits the placement of plans by only allowing them to appear at the same level with respect to other plans or nested within other plans. This eliminates flaky plan compositions such as interleaving where plans are distributed in multiple parts of the program.

Plan composition is performed in the *Composition View*. This view contains the source code that is entailed in each plan. It is important to note that all of the views in GPCeditor are integrated. The selection of a plan in the *Composition View* results in the display of the corresponding goal and plan in the *Decomposition View* and *Goal-Plan Lists*.

The third and final stage that the GPCeditor supports is testing. This is performed by selecting the *Run* option from the main screen. The GPCeditor features *Text* and *Graphics* execution views which allow the observation of data during program execution. It also supports tracing and stepping while highlighting the corresponding goals and plans currently being carried out.

3.2 Support for Learning Program Comprehension

As indicated in Chapter 2, novices lack program comprehension skills. Part of this is due to the fact these skills are not acquired because many programming courses focus solely on program construction. Two possible approaches in overcoming this difficulty are discussed in the subsections that follow.

One approach is the use of a tool that prohibits the construction of source code that is difficult to read. This tool, founded on a technique called literate programming, is the first of its kind for novice users. Until now, similar tools have only been designed for expert users. The other approach is to the use of a tool that provides visual support for program execution. The tool is specifically designed to teach recursion, a topic often avoided by novices.

3.2.1 A Foundation for Literate Programming Tools for Novices

Andy Cockburn and Neville Churcher [Cockburn and Churcher, 1997] present a prototype tool specifically adapted for novices that is founded on a programming technique called *literate programming*. Literate programming was first introduced by Donald Knuth in 1984. It is a strategy aided by a tool that is designed to prevent the construction of non-manageable, cryptic source code that is not easily interpreted by another reader. It is a method in which source code and documentation are generated to form a legible representation of the program that can easily be interpreted. The goal of a program comprehension tool is to provide a mechanism by which a reader can simplify the interpretation of the actions performed by program statements. The nature of literate programming lends itself to be identified as a program comprehension tool because the result of using the tool is a version of the program that is well-documented and easy to understand.

Historically, tools that facilitate literate programming have been designed with only the experienced programmer in mind. Cockburn and Churcher claim that those that could benefit the most from these tools, novices, cannot use them because of their

complexity. Modern tools are not user-friendly and the use of them requires significant syntactical understanding. "The promise of literate programming is therefore inaccessible to novice programmers." Cockburn and Churcher's adaptation is specifically designed with the novice in mind.

The authors present their argument for the need for such a tool based on factual evidence: although novices may be taught problem decomposition techniques, many do not adhere to them and tend to write their programs in an unorganized manner. Subsequently, they may add comments to their source code after they have written their program, something they should not do. These students fail to realize that documentation and comments are intended to support the coding *process* in addition to supporting a reader's comprehension of the program.

Generally speaking, the use of a literate programming tool transforms a *literate program*, consisting of *chunks of code* and supporting documentation, into one that is easier to comprehend. The chunks represent cognitive units in the program and do not necessarily correlate to syntactic constructs of the programming language. The chunks can be *tangled* to produce source code that is ready to be compiled. They can also be *woven* into a documentation-oriented version of the program that provides "extensive cross-referencing and indexing of program elements." The resulting woven program is easier to view because it is typeset. It contains all of the chunks from the literate program as well as cross-reference information depicting the relationship among the chunks.

Modern tools require that a programmer learn a complex syntax for identifying the *chunking structure* and *cross-referencing* within a program. Cockburn and Churcher attempt to make this part easier so that novices can use it. Supporting their cognitive

foundation for the novice tool, their primary focus centers on the elements of the user interface.

Among the features they advocate for such a tool is the "explicit support" for documentation. They argue that a user interface should present an element for displaying documentation through the use of a text or graphics editor, rather than the traditional insertion of source code comments during a program's evolution. They also encourage an element that provides a highly visual structured representation of a program. Their example of such a visualization of a program resembles a modern day file manager used to view the contents of a computer's file system structure. The authors suggest a "syntactic-correctness" approach for the user interface to prevent syntactical errors. The novice should be able to point-and-click on chunks rather than making typographical errors in typing the names of chunks.

Cockburn and Churcher's prototype provides three interfaces for facilitating the process of literate programming: the *structural view*, the *program editor*, and the *documentation editor*. The lefthand window contains the structural view. It is similar in appearance and function to a file manager as mentioned earlier. The program editor is in the center window. This window contains the contents of the literate program. On the right is the documentation window. This window is used to provide the supporting document of the chunks that are in the program editor window. All three interfaces are mutually exclusive in that a modification in one view results in a related change to the other two. This takes away the responsibility of making sure that contents of each window is consistent with the other two.

In the structural view, nodes represent the chunks of the literate program. Browsing the program structure is accomplished by expanding and collapsing each node by selecting it. This feature allows the reader to view different levels of abstraction in the program thus enhancing comprehension.

Supporting the program editor interface is a hyper-text editor and text-viewer for the source code of the literate program. The hypertext allows the reader to expand chunks to show implementation details and contract them just so that the chunk name is visible. Chunk names are identified with an underlined blue text when contracted, and a shaded struck-through red text when expanded for details. When a chunk is expanded, its appropriate representations appear in the structural view and the documentation editor. The documentation editor allows text and graphics to document the actions performed by a chunk. It also provides hypertext-like feature similar to the program editor. Documentation related to a chunk can be contracted and expanded as desired.

In the creation of a literate program, a programmer can enter source code in one of two ways: directly or using a top-down strategy. If a programmer chooses to enter source code directly, there is no requirement that the program be written in a literate program format. Once the source code is complete, the programmer invokes a feature that chunks portions of the source code. The programmer selects the source code that is desired to be chunked and selects the *Make Chunk* option from a menu to obtain the desired result. The tool performs all the necessary details in creating the actual chunk. The automatic generation of the chunks through the use of a menu selection is really what makes this a valuable tool for novices. It spares them from the syntactical details of chunking.

Alternatively, the programmer may choose a top-down strategy using step-wise refinement. In this case, empty chunks are created in the structural view using a desired hierarchy. Each chunk can be refined into lower levels by adding empty chunks below it as desired. The only requirement to this approach is that the contents of each chunk must be written in a literate program style.

3.2.2 EROSI : Program Visualization Made Easier

[George, 2000b] introduces *EROSI*, the *Explicit Representer Of Subprogram Invocations*. EROSI is an easy to use tool that aides program visualization, particularly in the understanding of recursion. It helps novice programmers obtain a "mental model to facilitate the comprehension and use of recursion as a problem solving technique." Recursion is a very difficult concept for beginners learning to program. As indicated in the previous chapter, novices tend to avoid recursion by attempting to substitute iteration in favor of it at all costs. The author's intent is to attract students to use this powerful technique so that such avoidances are a thing of the past.

George defines a recursive algorithm as "a process capable of calling new instantiations of itself with control passing forward to successive instantiations (*active* flow of control) and back from terminated ones (*passive* flow of control)." The basis for EROSI is the *dynamic-logical model*. In this model, each recursive invocation suspends the calling program. Control of execution is transferred to a "new and unique manifestation" of the recursive subprogram. Once subprogram execution has completed, control is transferred back to the program that made the call. Each invocation results in a new copy of the subprogram, consisting of its own unique variable and parameter values.

Among the benefits of EROSI, the author claims that the use of the tool enhances the comprehension of sequential program and subprogram execution, the suspension of a calling program resulting from a subprogram invocation, the transition of control flow to the new copy of the subprogram and its resulting actions, the return path to the suspended calling program, and data flow through the life spans of all invocations.

Visualization and animation aided by the use of color and sound is what gives EROSI its appeal. It delineates the execution of a recursive subprogram through a technique called *dynamic code visualization*, in which the current line of source code being executed is highlighted. It also facilitates *dynamic algorithm visualization*, an animation feature that assists the novice in visualizing the mechanics of an algorithm being deployed.

At the heart of EROSI is the *Command Module*. The role of this module is to coordinate the activities of the system's constituent components as well as for providing the user with a menu system. The six components surrounding the Command Module are the *Graphic and Utility Units, Text Files, Simulation Modules, User Input, Visual Output,* and the *Event Logger.*

The *Graphic and Utility Units* handle graphical displays. *Text Files* comprise the menu from which instructions may be invoked or information be provided. *Simulation Modules* simulate sample program execution. *User Input* is generated by the student to invoke a simulation. Program simulation and applicable program output is presented in the *Visual Output* component. The *Event Logger* is a utility that records miscellaneous metrics about the use of the tool and observations about the user.

The main menu allows the user to customize the behavior of EROSI and provides a user guide. The four menu choices of real interest are: *Subprograms Without Parameters, Subprograms With Parameters, Complex Calls,* and *Recursion.* Each of these selections when invoked, presents the user with submenus that contain programs that can be simulated with custom user input. The choices are ordered in increasing difficulty and program complexity. This provides the novice with an easier transition to learning recursion: rather than providing random examples, a progressive approach is utilized.

The distinction between the *Subprograms Without Parameters and Subprograms With Parameters* options is obvious. Both are used to introduce the novice to the *dynamic-logical* model of execution. The *Complex Calls* option covers more advanced topics such as nested subprogram invocations. It is not recommended for a user to invoke this option without completely understanding the lessons presented in the earlier sessions. The fourth option, *Recursion*, simulates recursive programs as its name implies. These simulations are rendered "as an extension of the *dynamic-logical* subprogram execution model as visualized in previous simulations."

A simulation in EROSI begins with the appearance of two windows. On the left side of the screen is the window containing the source code. On the right side, a smaller window is used to display program outputs resulting from the execution of the program text. The user simply follows the instructions on the bottom of the screen to begin and quit the simulation. During the simulation, the system may prompt the user for an input to a variable. This also appears at the bottom of the screen.

The example programs provided by EROSI are specially adapted for novices and are simple in nature and appearance. When an input is required, the user can see where in the program the input is required because the system highlights the area that requires the input in a flashing yellow manner. Once the user data has been entered, the area that was highlighted gets filled in with the user input.

During simulation, the color of program text changes from white to red as each line is executed. An audible indication is given as well. When a subprogram call is about to be made, an arrow appears to the right of the statement. The arrow moves away from the calling program while marking a trail as it proceeds to the subprogram. After a certain distance, the arrow stops its progression and a new small brown window appears containing the program statements and variable values of the subprogram. The trail made by the arrow allows the student to visualize control flow. Successive subprogram calls result in the same trail-marking process. After subprogram execution is complete, a new arrow travels in the reverse direction indicating the flow of control is being returned to the calling program.

3.3 Miscellaneous Learning Aides

Since the birth of the Internet and technologies like hypertext, several learning aides have been developed that take advantage of these innovations. Novices that are learning program development concepts now have intelligent tools to support them. Two such learning aides are discussed in the subsections that follow. The first is a electronic textbook that teaches a student how to program in the Scheme programming language. Among other features, it provides hyperlinks that allow a student to quickly locate that

meaning of a word or concept. This eliminates the need of searching through a physical textbook. The second learning aide is an adaptive tutoring system that teaches the LISP programming language. A user initiates a session by using a browser via the WWW (World Wide Web). Based on the student's performance the system makes logical deductions to provide a learning experience specifically adapted for each student.

3.3.1 A State-of-the-Art Multimedia Textbook

Legacy intelligent tutoring systems such as the *LISP tutor* [Grillmeyer, 1999; Anderson 1987] and *GIL*, (Graphical Instruction in LISP) [Grillmeyer, 1999; Reiser, Ranney, Lovett, and Kimberg, 1989], are designed to teach a student the LISP programming language and draw their attention to learning the creation of functions. [Grillmeyer, 1999] chooses a significantly different approach to teaching. Rather than designing an intelligent tutor system to teach Scheme (a dialect of LISP), he introduces an interactive tool that facilitates the learning process more effectively. His tool, called *Exploring Computer Science with Scheme,* is a multimedia textbook.

Research has shown that hypertext-based learning promotes cognition [Grillmeyer, 1999; Spiro and Jehng, 1991] better than the conventional method of reading a textbook. The most prominent features of Grillmeyer's tool are the ability to navigate via hyperlinks, place personal bookmarks, and enter personal notes. It also provides an animation feature that allows novices to visualize how certain lists and functions work.

Designing such a tool is a very arduous task. An appealing interface does not imply that the designer has succeeded in his ultimate goal of increasing the cognition of novice programmers. Hypermedia entails hypertext, audio, still images, video, and

animation. It is important that hypermedia be used carefully so that novices do not get discouraged. A poorly designed system would provide no benefit, supporting the argument that it would much easier to read a textbook the old-fashioned way instead.

Grillmeyer pays a great deal of attention to preventing hypermedia pitfalls. Among the considerations for his system was the desire to prevent novices from getting lost in a series of hyperlinks. A novice can easily get lost navigating in hypertext because the concept of where one is with respect to the entire text is lacking. In addition, confusion can set in when attempting to return to prior locations in the text.

In designing his multimedia textbook, the author compromised between a tool that contains an abundance of links to other pages, and one that contains no links and uses just a scroll bar for navigation. A backtracking feature is included so that a reader can return to the previous page, the chapter overview, and personal bookmarks that a reader may have placed somewhere along the line. The organization of the book is by chapters, each loaded independently. Scrolling can be performed on any page that is presented on the screen. Jumping to the previous or next page is accomplished by using the *previous* and *next* buttons, respectively. Additional features include three special types of hyperlinks. The first allows the reader to view a chapter layout. The second allows a learner to define a location to jump to by entering a bookmark identifier. The third provides the novice with the ability to look up definitions and examples by using words or phrase searches.

A useful utility of this system is the *notebook* tool. Readers of the text can enter any text into the notebook as they desire. *Cut* and *paste* is also supported in cases where it would be quicker than typing out the entire concept or example being presented. The

notebook also supports a book marking feature. This allows the students to navigate the contents of their personal notebooks as they grow dynamically.

3.3.2 ELM-ART II: An Adaptive Web-Based Tutoring System

The *ELM* (Episodic Learner Model) family of intelligent tutoring systems was developed at the University of Trier in Germany. In chronological order of their development, these systems are called *ELM-PE* (Programming Environment), *ELM-ART* (Adaptive Remote Tutor), and *ELM-ART II*, the successor to ELM-ART. The intent of these systems is to provide support for novices learning the LISP programming language. The primary focus of these systems is to record an individual's learning history in a database of episodes which are in turn used to form an *episodic learner model* that adaptively guides learning [www.psychologie.uni-trier.de:8000/projects/ELM/elm.html]. In their paper on WWW-based tutoring systems [Weber and Specht, 1997] discuss ELM-ART II, the newest member of the family.

To understand the benefits of ELM-ART II with respect to its ancestors and other hypermedia tools, a discussion of the evolution of ELM is first warranted. The first ELM system to be developed was ELM-PE. This on-site tool provides support for example-based programming, solution analysis, testing, and debugging. In ELM-PE, students hone their programming skills of new LISP concepts by working on exercises. Unfortunately, ELM-PE was not portable or remotely accessible because of its platform dependence and enormous program size. With the desire to overcome these obstacles, ELM-ART, a WWW-based version of the system was created.

ELM-ART was founded on a client-server paradigm. Text files formerly used in ELM-PE, reside on a server in the form of HTML files. Users are required to install the appropriate client software (a web browser) to access the lessons contained on the server. The text pages are grouped by sections and subsections and are associated with the appropriate concepts to be learned. A *conceptual network* is built up by relating the concepts to each other on a prerequisite/outcome basis.

ELM-ART monitors user inputs to the system and uses them to create an individual learner model. When a user visits a page, the corresponding concept in the conceptual network is marked as being visited. When viewing sections, subsections, and overview screens, a colored icon precedes each selectable item. The color of the icon is one of three: red, yellow, and green, analogous to a traffic light scheme, and represents the status of that item with respect to the individual learner model. An indication of red discourages the learning of a particular item, usually because prerequisite material has not been covered yet. Green indicates that the section is ready to be learned. Yellow indicates that the material may be suitable for the student to learn at this point, but is not recommended by ELM-ART. For each concept in the conceptual network, ELM-ART provides live examples and performs intelligent diagnoses of a user's solutions. ELM-ART also allows the evaluation of function calls in an evaluator window. To interact with ELM-ART, a user codes a solution and sends it to the server.

It became apparent after a while that ELM-ART had its shortcomings. In particular, it was realized that a textbook cannot simply just be converted into pages of hypertext and serve as an effective electronic learning tool. In a physical textbook, any page can be viewed just by turning to that page. A computer screen however does not

provide this capability. The user must sequentially pass each and every page to get to a desired page. Furthermore, a newer adaptive solution had to be designed; one that would react more appropriately to user interaction. These and several other design innovations led to the development of ELM-ART II.

The architecture of the conceptual network in ELM-ART II differs from its predecessor. The organization of *units* follows a hierarchy: lessons, sections, subsections, and terminal pages. Units are entities that contains *static slots* and *dynamic slots*. Static slots are used to store administrative information related to the concept. This includes the prerequisites for the concept, its relationship to other concepts, and the qualifications that are attained once that concept has been learned. Dynamic slots include the lesson text to be presented to the user.

The units presented on the screen are annotated by utilizing a quasi-traffic light scheme similar to its predecessor. The red, yellow, and green colored indicators represent the same status as in the original ELM-ART. A new color, orange, takes on different meanings. For a terminal page, an indication of orange means that the system has deduced that the contents of the page will be familiar to the user due to the successful completion of a previous concept that may entail the current concept. For non-terminal pages, it indicates that only portions under the hierarchy of that page have been visited.

Positioning the cursor over a colored circle will result in the display of the corresponding meaning of the color in the status line at the bottom of the browser's window. When a slot has been visited, an indication is made in the user's personal individual learner model. The visitations in each interaction with ELM-ART II are remembered for each user for subsequent sessions.

Units for terminal pages contain *test slots* in which example problems are given, or alternatively new concepts are introduced. There are four types of example problems: yes/no, forced-choice, multiple choice, and free-form. Each test unit contains criteria for evaluating how well the student understood the concept. A certain percentage of correct answers is required for successful completion. Examples are presented continuously until the exit criteria is met.

Feedback is provided with each incorrect answer with an accompanying explanation. When the user successfully meets the exit criteria, the qualifications for that concept are considered to have been fulfilled. The achievement of a qualification is in turned reflected (by induction) in other related units and/or more advanced units where the qualification is a prerequisite to the new unit.

ELM-ART II supports individual curriculum sequencing which prevents users from getting lost in hyperspace. This is achieved through the use of *next* and *back* buttons present in the navigation bar. The usage of *next* prompts the system to use an algorithm to select the next appropriate concept to learn, given the current state of the individual learner model.

The intelligence behind ELM-ART II can deduce a user's problem solving behavior and choose example problems that would most benefit the user. A user can also ask the system to diagnose a particular solution if the user cannot identify the error. The response is not an immediate answer. The system guides the student through a discovery process by giving hints that progressively reveal the answer.

3.4 Program Development Environments

Program development environments are the most common form of programming tool. In the subsections that follow, three modern environments designed for novices are discussed. The first environment provides support for problem solving, a feature not commonly present in such tools. It facilitates design, algorithm development, testing and debugging while eliminating issues involved with programming syntax. The second environment is designed for a simple programming language that was specifically designed to minimize the number of constructs and operations. This effectively is what happens in a introductory programming course; a subset of the language is taught. The third environment provides a program animation feature that helps a novice visualize what is happening behind the scenes of a program being executed.

3.4.1 FLINT: An Instructional Programming Environment

Of all the program development environments in existence today, a majority of them are not designed for novice users. Uta Ziegler and Thad Crews [Ziegler and Crews, 1999] introduce an instructional programming environment (under development at the time of their publication) called *FLINT (Flowchart Interpreter)*. It is a tool designed specifically for the novice, which incorporates design, algorithm development, testing, and debugging. It is known for its programming language independence and its ability to draw attention away from issues involving programming language syntax.

Ziegler and Crews argue that expert-oriented environments are not recommended for novice instruction because "only the complete syntax of a full-blown programming language is supported." Furthermore, they rightfully state that "problem solving and

design are not supported." The authors claim that the use of their environment helps the student to "develop a view of programming in which design and testing are integral parts of program development."

FLINT focuses on two types of programming: *iconic programming* and *pseudo programming*. An iconic programming tool facilitates the creation of a program by allowing the user to insert pre-defined icons into a program construction area. When program construction is complete, the tool generates the appropriate source code in the desired language. Research has shown that students that generate programs with such a tool are able to better comprehend the language syntax than if they had not used the tool [Ziegler and Crews, 1999; Calloni and Bagert, 1997].

Pseudo programming is primarily a method of teaching abstraction and data structure usage. The *pseudo language* cannot be compiled and executed because it does not meet the requirements of a programming language syntax. It is merely a vehicle by which instructors can introduce proper program development techniques to novices. Its purpose is to prevent novices from programming haphazardly without any organized thought process.

The first stage of FLINT deals with the development of a sound design. FLINT facilitates this through the process of step-wise refinement using its graphical interface. The student constructs a program design in the form of a top-down structure chart which is visible in a dialog window. The structure chart consists of rectangular boxes connected by lines that represents a hierarchical view of a program. It begins with one box at the top of the window that contains the description of the highest level of abstraction.

Rectangular boxes beneath it represent the first level of abstraction (subproblems). Each subproblem may contain more rectangles beneath it as required.

The presence of the boxes is managed through a menu with four choices: *add box*, *move box*, *delete box*, and *copy box*. When adding a box, one of the following four identifiers is associated with it: *get information*, *output information*, *calculate*, and *other*. When creating a box, the first line of text in the box will contain a description of the type of work to be performed. This includes information transfer (input/output) and calculations. The next line in the box is bolded and represents the name of the step. This name will later be used as the name of the algorithm for that box. After these first two lines, the remainder of the box is filled in with what actions must be done (i.e. for a *get information* box it may say "get numerator and denominator from user," for a *calculate* box it may say "perform division," and for an *output information* box it may say "print quotient to screen"). This text is not analyzed by FLINT, but may be used for a instructor's analysis later on. With the use of structure charts, FLINT prohibits students from first getting a program to run and then abstracting a design from it.

The second stage of FLINT assists in developing algorithms for each of the boxes in the structure chart. Algorithm construction cannot begin until the design has been completed. Once the student is ready to proceed, each box from the structure chart can be selected for algorithm construction by double-clicking on it. Throughout algorithm development, FLINT constantly monitors the state of the design by performing checks for connective tissue. As an example, to ensure consistency FLINT makes sure that a box at any level contains at least one reference to each of the boxes underneath it. Algorithms that do not meet this criteria are rejected. This and other preventive measures ensure that

the student implements the design as specified. Only a change in design will allow a change in the implementation.

Algorithms are represented in FLINT using structured flowcharts. In constructing an algorithm, FLINT only allows complete program constructs to be added, deleted, moved, or copied. As a result, "the constructed flowchart will always be structured and syntactically valid." The flowchart builder is simple to use. It uses conventional flowcharting shapes which can be selected using the point-and-click interface.

During algorithm development, the student declares all variables that will be used along with a short description of each. Only variables that have been declared can be selected in an algorithm. Similar to the construction of the structure chart, a menu is used to select what action is to be performed on the flowchart. This includes adding, deleting, moving, and copying statements. A submenu exists for the addition of a statement. It consists of the following statement types: input, output, assignment, selection, repetition, and subprogram. Developing an algorithm in such a way "allows teachers to show students how to construct algorithms incrementally."

Testing is the third stage in FLINT. Using FLINT a student can get immediate feedback by executing an algorithm. An instructor can customize FLINT so that a student is required to perform a certain number of test cases per algorithms. FLINT prompts the student for the input and the expected output of each test run. For each test run, the student must then confirm that the results provided by FLINT are consistent with the expected output.

The fourth and final feature supported by FLINT is debugging. To facilitate debugging when errors are found, FLINT provides the capability to insert breakpoints.

Statements and variables are highlighted prior to execution and use, giving the novice a visual representation of the control and data flow of the program. The pace of the execution can also be controlled to simplify comprehension of these flows. As touched on earlier, a change in the algorithm will only be permitted through a change in the design.

3.4.2 X-Compiler: A Simple Language and its Programming Environment

In almost all introductory programming courses, only a chosen subset of a programming language will be taught. In some cases, a new programming language is designed to totally entail a subset of another language. [Evangeledis, Dagdilelis, Satratzemi, and Efopoulos, 2001] present such a language and a customized programming environment that accompanies it. The language, called *X*, is a subset of Pascal. It is supplemented by *X-Compiler,* a programming environment implemented on a Microsoft Windows platform.

In order to design an effective programming language, the authors based their design of *X* on a set of heuristics that they believed were the most influential to a novice programmer. One concern was simplifying the syntax. This was accomplished by eliminating data types and the need for variable declarations. The entire syntax of *X* is surprisingly small and very primitive. In fact, it only supports three relational operators: greater than (>), equal to (=), and not equal to (<>). The assembly code counterpart also consists of a reduced set of assembly instructions. The pseudo-assembly code runs on a virtual machine with two registers.

X-Compiler contains many interesting features. Among these, it provides a behind-the-scenes look at the execution of a program, starting with source code

compilation, the correspondence of assembly code to source code during execution, and the ability to view register values during execution. Another feature that separates it from other environments is the simplicity of error reporting. X-Compiler provides detailed and easy to understand error messages that facilitate easy debugging. This is critically important for novices because error messages that cannot be understood serve no purpose to a struggling student.

X-Compiler provides the programmer with editing, debugging, and program execution capabilities. Five windows are presented: (1) source code, (2) assembly code, (3) registers and temporary variables, (4) user variables, and (5) output. Two modes of operation are supported: novice and advanced. In the novice mode, only windows 1 and 5 are active. A great feature is the help system which is invoked when any keyword, operation, or delimiter is double-clicked.

Errors found at compile time appear in a pop-up window. The window contains two drop-down lists, one for detected errors and one for compiler warnings. To gain an understand of the error or warning, the student selects the desired element with a mouse-click. When a selection is made, the error message is explained on the screen and the offending source code line is highlighted.

X-Compiler also features a great debugger. Students can step through the execution of a program and observe data and control flow. For each line of source code currently being executed, the associated assembly code statements are highlighted. In addition, system registers, temporary and user variables are updated dynamically in the step-wise program execution process. When the program calls for an input variable, a

dialog box prompts the user for the required input. Any output generated by the program will be displayed in the output window.

3.4.3 AnimPascal: Programming Aided by Animation

One of the greatest learning curves that novice programmers must overcome is the ability to visualize the actions of program execution. Most programming environments lack the capability to "paint a picture" of what actions the program is performing. Of the few environments that do provide such a feature, most are not easy to use and are sometimes not helpful because of their crude nature. [Satratzemi, Dagdilelis, and Evagelidis, 2001] introduce *AnimPascal*, an educational programming environment designed to teach novices how to develop, verify, debug, and execute programs. The two most prominent features of AnimPascal are a program animator, and a mechanism used by instructors to analyze the recorded problem solving paths taken by novices.

AnimPascal provides the ability to edit and compile standard Pascal programs. The graphical user interface is simple and easy to use. It consists of four windows: (1) source area, (2) program output, (3) display variables, and (4) compiler output. Compiler errors detected by AnimPascal are highlighted in red in the compiler output window. Green highlight is used for compiler warnings, hints, or notes regarding the program. The student may click on any of these messages and the cursor in the source code will move to the appropriate line of source code for which the issue was raised.

AnimPascal records all compiled versions of a student program with a history tracking mechanism. This feature can prove to be very useful for programming instructors. It gives them an insight to common programming misconceptions and

frequently encountered mistakes. These observations can then be fed back into the curriculum to prevent future occurrences.

At the heart of AnimPascal is an educational animation tool. As a student steps through a program, the current source code line being executed is highlighted. During the step-wise execution process, variables are updated in the *display variables* window, output appears in the *program output* window whenever output producing statements are encountered, and windows prompting a user for input appear whenever a statement requiring a user input is executed.

3.5 An Integrated Program Development Environment: SOLVEIT

A programming environment that integrates problem solving and program development in a well-defined guided process is not easy to find. As was evident in this chapter, none of the learning tools that were identified provide support for learning a complete problem solving and program development methodology. A tool that guides a novice from the initial problem description stage all the way through the software delivery stage is a system that a novice would greatly benefit from.

Developed at the New Jersey Institute of Technology, *SOLVEIT* (Specification Oriented Language in Visual Environment for Instruction Translation) is such an environment. SOLVEIT is designed specifically for novices and fully supports problem solving and program development. It provides support for problem solving, solution planning, solution design, testing, and delivery.

In the subsections that follow, the architecture of the *Dual Common Model*, the theoretical foundation for SOLVEIT [Deek, 1997; Deek and McHugh 2002; Deek,

McHugh, and Turoff, submitted for publication] will be discussed. For each of the six stages of the Dual Common Model, the required input, the resulting output, and the necessary knowledge to accomplish each activity at each stage will be identified. The first part of Chapter 4 will be devoted to user interaction with SOLVEIT and how it can help minimize the challenges that novice programmers face.

3.5.1 High-Level View of the Dual Common Model

The Dual Common Model is comprised of two main systems: The *Problem Solving and Program Development* system and the *Cognitive System.* The former identifies the activities that are performed at each stage of the process. The latter contains all of the necessary cognitive skills and knowledge required to successfully perform the activities in the problem solving methodology. The fusion of these two systems forms the heart of the Dual Common Model.

3.5.1.1 Problem Solving Methodology. Research in problem solving methodologies is nothing new. Over the years newer approaches have been introduced but the nature of problem solving has not changed much. In fact, it has convincingly converged to a steady state. Using this reality as their basis, the authors have synthesized a common model for problem solving based on several methodologies. The intent of the common model is to encapsulate historically proven approaches to problem solving while at the same time create a framework for problem solving which will be tailored for program development. In essence, the common model consists of problem definition and understanding, solution planning, solution design and implementation, and verification and presentation.

3.5.1.2 Program Development Tasks. In supporting the tasks of the problem solving methodology, program development tasks serve to implement programs once the required problem solving steps have been accomplished. It is important to note the distinction between the two stages as well as the importance of problem solving as a prerequisite to program development. The former results in a conceptual solution, whereas the latter is a realization of the former in the shape of a working software solution. The program development tasks used in the Dual Common Model include program composition, program comprehension and reuse, debugging of programs, test and verification of programs, program maintenance, and documentation.

3.5.1.3 Cognitive System. A cognitive model lays the foundation for the *Cognitive System*. This model focuses on the skills and knowledge required for the tasks involved in problem solving, rather than the processes themselves. The model is comprised of three elements: *cognitive processes*, *cognitive structure*, and *cognitive results*, introduced by Bloom, Sternberg, and Gagne, respectively.

Bloom's framework for cognitive processes consists of three low-level and three high-level processes for problem solving. The three low-level processes are *knowledge*, *comprehension*, and *application*. *Knowledge* is the ability to correlate a new problem with a previous problem solving experience. *Comprehension* refers to understanding the problem currently being presented, demonstrated in part by the ability to describe the goal of the problem. *Application* involves utilizing the remembered knowledge and acquired comprehension to the solve the problem being presented. The three high-level processes are *analysis*, *synthesis*, and *evaluation*. Analysis includes subproblem decomposition, solution by analogy, and other problem solving strategies. Synthesis

involves integrating component parts and establishing relationships among components to create a practical solution to the problem. Evaluation is the process of verifying that the problem statement requirements were met.

Sternberg uses three components to form a theoretical foundation for a cognitive structure of human thinking. The first component, *knowledge acquisition*, parallels the work of [Mayer, 1981; Bransford, 1979] in which meaningful learning occurs when new material is associated with pre-existing knowledge. The second component, *performance*, includes activities such as goal decomposition and task coordination that assist in designing and implementing a problem solving plan. The last component, *metacognition*, controls a programmer's thinking process.

Gagne advocates that good learning environments should produce beneficial cognitive results. Among these learning outcomes, he identifies the following as being the most critical: *verbal information*, *intellectual skills*, *cognitive strategies*, and *attitudes*. The ability to correctly restate the description of a problem, or identify the goal of a problem in one's own words demonstrates the fact that *verbal information* has been acquired. The ability to further identify the entities of a problem and how the goal of the problem can be achieved through various operations on these entities in a solution plan demonstrates the possession of *intellectual skills*. The ability to utilize knowledge given the presented facts to design a problem solution indicates the ownership of a suite of *cognitive strategies*. *Attitudes* are personal preferences that a student will develop as a function of the interaction with a learning environment. Although not directly linked to problem solving, this learning outcome is crucial because interactions that result in negative attitudes hinder the learning process.

3.5.2 Stage 1: Problem Formulation Stage

Given the problem statement as an input, the goal of this initial stage of the Dual Common Model is to produce a knowledge base of information that will be used as an input for subsequent stages. Table 1 contains the input and output parameters as well as the required knowledge and skills required for the Problem Formulation stage.

Input:	- Problem Statement
Output:	- Knowledge Base
Required Knowledge & Skills:	- Domain Knowledge - Problem Modeling Skills - Communication Skills

Table 1 Parameters and Requirements for the Problem Formulation Stage

The Problem Formulation stage is broken down into three functional activities: the construction of a *preliminary problem description,* the conception of a *preliminary mental model,* and the creation of a *structured representation of the problem.* The outcome of each activity stimulates the execution of the subsequent activity. The evolved result of all three activities will be an organized knowledge base containing all information relevant to the problem which includes the goals, givens, unknowns, conditions, and constraints.

A variety of methods can be used for constructing a preliminary problem description, the purpose of which is to make sure the problem solver has a firm grasp and understanding of the problem to be solved. This activity is important because it is intended to ensure that information in the problem is not overlooked or misinterpreted. Having completed the thorough analysis, the problem solver then begins to visualize a preliminary mental model. The is effectively achieved through verbalization and the use

of inquiry questions. By using these techniques, the problem solver can identify what is known about the problem and what needs to be found out. A structured representation of the problem is then created by extracting the information from the problem statement. The problem solver organizes and categorizes the essential facts into a knowledge base, ignoring any unnecessary details.

Bloom's low-level cognitive process of knowledge is used in problem formulation. The problem solver must gather all relevant information, organize the information, and combine it with prior domain knowledge to achieve the goal of problem comprehension. Sternberg's knowledge acquisition component of human thinking supports this process through the use of knowledge recollection. An important learning outcome of this stage is the acquisition of problem comprehension evidenced by verbal information.

3.5.3 Stage 2: Solution Planning Stage

The goal of this stage is to transform the knowledge base created in the previous stage into a solution plan. This stage includes the use of strategies such as subgoal decomposition and the identification of solution alternatives. Table 2 contains the input and output parameters as well as the required knowledge and skills required for the Solution Planning stage.

Input:	- Knowledge Base
Output:	- Solution Plan
Required Knowledge & Skills:	- Strategy Discovery Skills - Goal Decomposition Skills - Data Modeling Knowledge

Table 2 Parameters and Requirements for the Solution Planning Stage

The Solution Planning stage is broken down into three functional activities: *strategy discovery*, *goal decomposition*, and *data modeling*. Strategy discovery requires a problem solver to identify possible approaches to solving the problem and determining which alternative is appropriate. This is usually determined by using pre-existing knowledge as well as by gaining an understanding of the problem's requirements. Goal decomposition involves the refinement of the ultimate goal into subgoals that are easier to solve. The process of subgoal decomposition is repeated as required until the appropriate level of abstraction is achieved. The givens and unknowns contained in the knowledge base are then used to design preliminary data structures through the process of data modeling.

Two of Bloom's cognitive processes are used in solution planning. The low-level process of knowledge is demonstrated by the ability to use the necessary knowledge to perform the decomposition of goals into smaller subgoals. The high-level process of analysis involves organizing the results of problem decomposition into a coherent hierarchy of subgoals. Sternberg's performance component of human thinking facilitates these processes by directing solution planning and subgoal decomposition. An important learning outcome of this stage is the acquisition of intellectual skills. These skills are learned in the process of applying knowledge to the creation of a solution plan.

3.5.4 Stage 3: Solution Design Stage

The goal of this stage is to transform the solution plan from the previous stage into a solution design. Two cognitive activities used in the Solution Design stage are the organization of solution components and the creation of an algorithm. Table 3 contains

the input and output parameters as well as the required knowledge and skills required for solution design.

Input:	- Solution Plan
Output:	- Solution Design
Required Knowledge & Skills:	- Organizational Skills - Refinement Skills - Function/Data Specification Knowledge - Logic Specification Knowledge

Table 3 Parameters and Requirements for the Solution Design Stage

The Solution Design stage simultaneously marks the end of the problem solving process and the beginning of the program development process. It is broken down into three functional activities: *organization and refinement*, *function/data specification*, and *logic specification*. The first activity involves the refinement of the preliminary solution plan from the previous stage into solution components and the assignment of a preliminary function statement to each. This refinement process is repeated until each subgoal can be considered a well defined task.

The iterative process of refinement will usually require the rearrangement and resequencing (organization) of subgoals as well as their relationships to other subgoals. A structure chart is used to depict the hierarchical structure of the subgoal (module) decomposition. Once decomposition is complete, the function of each module has to be specified as well as the relationships (data flows) to other modules. The function of a module is merely a statement of the goal of the module. Data flows are derived from the previous two stages. A product of this activity is a data dictionary which contains the names of all data flows, corresponding source and destination module names, and data types.

Logic Specification is the next step. This involves the generation of pseudocode in algorithms that give definition to the modules. The pesudocode includes the identification of data and control structures as well as operators. These algorithms can eventually be translated into the programming language of choice.

Synthesis, one of Bloom's high-level cognitive processes, is used in solution design. Synthesis entails the strategies used in component integration, component rearrangement, and establishing relationships among components. Sternberg's performance component of human thinking facilitates these processes by guiding decomposition tasks and directing component organization and resequencing. An important learning outcome of this stage is the acquisition of cognitive strategies, tactics that assist the generation of a solution through the transformation of knowledge and information.

3.5.5 Stage 4: Solution Translation Stage

The goal of this stage is to translate the solution design from the previous stage to a coded solution in a specific programming language. It is the problem solver's first encounter with language syntax and semantics. Solution translation proceeds by establishing a sequential order for module translation and then performing the required translations from pseudocode to a programming language syntax. Table 4 contains the input and output parameters as well as the required knowledge and skills required for solution translation.

Input:	- Solution Design
Output:	- Coded Solution
Required Knowledge & Skills:	- Domain Knowledge - Problem Knowledge - Strategic Skills - Organizational Skills - Syntax Skills - Semantic Skills - Pragmatic Skills

Table 4 Parameters and Requirements for the Solution Translation Stage

The Solution Translation stage is broken down into three functional activities: *implementation*, *integration*, and *error diagnosis*. Following the order of the structure chart, the implementation activity results in a set of programming instructions that can be compiled and executed. The algorithms designed in the prior stage get transformed into individual instructions utilizing the proper programming language operators. The data structures that were also previously identified are transformed into type definitions, variable declarations, and variable initializations. Source code comments are added to help support the documentation of the program. Integration may apply if existing source code is to be reused, either partly or in its entirety, in the current problem being solved.

During the process of source code compilation and execution, errors may occur. This is where error diagnosis, typically called debugging, comes into play. Errors can be classified into three categories: syntactical errors, run-time errors, and logical errors. Syntactical errors are usually caught by a compiler and are relatively easy to fix because the compiler has identified the location of the errors. Run-time errors are not caught by the compiler. These errors occur during the execution of a program, typically when an operation violates some aspect of the operating system. A logical error is the most

difficult to find, especially for a novice, because the source of it may originate from any stage.

The Solution Translation stage utilizes two of Bloom's cognitive processes. The high-level process of synthesis is used when creating the sequential order of modules for the translation from pseudocode into source code. The low-level process of application is demonstrated by the use of syntactical and semantical knowledge when transforming pseudocode into source code. Sternberg's knowledge acquisition and performance components of human thinking support these processes. Knowledge acquisition is used when integrating pre-existing source code into the solution during the integration activity in this stage. The ability to recognize that a previously solved problem can be applied in the new situation demonstrates this characteristic. The performance component guides the tasks of module organization, translation, integration, and error analysis. An important learning outcome of this stage is the acquisition of intellectual skills. Once this stage is complete, the student has acquired the knowledge to perform effective error diagnosis.

3.5.6 Stage 5: Solution Testing Stage

The goal of this stage is to verify that the coded solution has met the requirements of the problem statement. The result of this process is a verified solution. This is achieved through the use of test cases that verify the functionality of the program. Table 5 contains the input and output parameters as well as the required knowledge and skills required for solution testing.

Input:	- Coded Solution
Output:	- Verified Solution - Test Results
Required Knowledge & Skills:	- Domain Knowledge - Problem Knowledge - Metacognitive Skills - Strategic Skills - Syntax Skills - Semantic Skills - Pragmatics Skills

Table 5 Parameters and Requirements for the Solution Testing Stage

The Solution Testing stage is broken down into three functional activities: *critical analysis, evaluation, and revision.* During critical analysis the problem solver must go back to the original problem statement and create test cases based on the goals and requirements specification of the problem. Test cases must be designed so that each module is independently tested and then the program as a whole is tested. A vast range of input data should be used to verify program correctness.

The intent of this activity is to uncover errors that were not discovered during planning, design, and implementation. When errors are found, they should be immediately corrected. When errors reveal a defect in solution design, the activity of revision begins. The problem solver must retrace the solution path to identify the location of the error and when appropriate devise an alternative solution. Depending on which stage of the Dual Common Model an error originated from, the corrective action must propagate forward through the remaining steps.

Once the program has passed the necessary testing criteria, the problem solver undergoes a reflection period by evaluating the problem solving path that was followed. This activity is very important, as it will most definitely have an effect on a future

problem solving task. The problem solver analyzes the pros and cons of the chosen strategy and formulates an opinion based on how effective (or ineffective) the experience was.

Two of Bloom's high-level cognitive processes are used in solution testing. A post-solution design analysis is made to determine the necessary parameters for test case design. This requires a re-examination of what the goals, requirements, and input ranges are, as well as what the expected outputs should be. An evaluation takes place when the generated test case outputs are verified against what is expected. Sternberg's *metacognitive* component of human thinking is used in this stage.

It should be important to note that the problem solver uses metacognition not only during the evaluation activity, but throughout the problem solving process. Metacognition is what helps a problem solver determine if the strategies used to design the solution were effective, and if not, suggestions to improve the process for future use are introduced. An important learning outcome of this stage is the acquisition of a positive attitude that one can perform a self-evaluation of the problem solving experience.

3.5.7 Stage 6: Solution Delivery Stage

The goal of this stage is to transform the verified solution and test results into documented form. Documentation in a program is a very important stage because it plays a pivotal role in program maintenance and source code reuse. Programs that have been properly documented are much easier to modify and have a higher probability of being reused as a component in a larger program than those that are poorly documented.

Table 6 contains the input and output parameters as well as the required knowledge and skills required for solution delivery.

Input:	Verified Solution Test Results
Output:	Documented Solution Documented Test Results
Required Knowledge & Skills:	Organization Skills Communication Skills

Table 6 Parameters and Requirements for the Solution Delivery Stage

The Solution Delivery stage is broken down into three functional activities: *documentation, presentation,* and *dissemination.* For novices, the process of documentation typically involves the generation of source code comments and explanations to support the source code. In some circumstances, depending on the nature of the assignment, software engineering approaches to documentation may be required.

A partial list of such documentation would include a user manual, a Version Description Document (VDD) that contains the modification history of the program, and a Software Test Description (STD) that contains the details of the each test case. Depending on the extent of testing that was performed, the software test document may contain a verification cross-reference matrix that contains the mappings of the problem requirements to the appropriate test case names.

The presentation of the solution follows the completion of documentation. This usually involves the generation of a report. The report should contain evidence that the program has verified all problem goals and requirements. It also should contain the strategies and approaches that were used during problem solving, solution design, and solution implementation stages, as well an evaluation of how effective the methods used

in these stages were. Dissemination, when required is the final activity. It involves the distribution of the program and documentation to interested parties.

In solution delivery, Bloom's high-level cognitive process of synthesis is used to bring the source code, test results, and documentation into one complete deliverable product. Sternberg's performance component of human thinking supports the synthesis process. An important learning outcome of this stage is verbal information which is acquired through the solution presentation process, especially when oral communication is involved.

CHAPTER 4

CHARACTERISTICS OF AN EFFECTIVE LEARNING TOOL

This chapter identifies and addresses the elements of a learning tool that would most likely benefit novice programmers. Based on the needs of novices, and the shortcomings of the tools listed in Chapter 3, recommendations are provided for the development of future work. These recommendations are based on a review of the literature, the novice challenges identified in Chapter 2, and the partial support provided by the tools discussed in Chapter 3.

Of the tools identified in Chapter 3, SOLVEIT is by far the most comprehensive in its coverage of addressing novice challenges. The first section of this chapter will illustrate a typical session with SOLVEIT. For each of the six stages of SOLVEIT, novice difficulties will be identified and the manner in which SOLVEIT assists in overcoming them will be discussed. The second section of this chapter will identify areas in which SOLVEIT can improve or evolve. The suggestions are based on a review of the literature and personal ideas. The intent of these recommendations is to hopefully stimulate the evolution of SOLVEIT to a new level. Due to the primary focus on problem solving in SOLVEIT, not all of the challenges identified in Chapter 2 are addressed by it. The third and final section of this chapter will identify the elements of a learning tool that should be considered in the development of future learning tools which may follow in the footsteps of SOLVEIT.

4.1 An Overview of SOLVEIT

The first step in a new SOLVEIT session is to run the executable file. The executable file can be run by invoking the appropriate desktop icon. Once the introductory screen appears, a new project is created with the *File* menu option. Alternatively, an existing project can be opened for modification. After a new project name has been entered by the user, two windows appear: the main window and a utility window.

The utility window contains a graphical guide to using SOLVEIT. It serves as a road map for the six stages of the SOLVEIT. The presentation style is that of a graphical file system viewer such as Windows Explorer. At the highest level of the hierarchy is SOLVEIT. The six stages of SOLVEIT follow beneath it in the hierarchy and are indented to the right with respect to the top level. Each of the activities associated with the six stages are present underneath the respective stage and are similarly intended further to the right to show that they are beneath that stage in the hierarchy.

The SOLVEIT guide also serves as a browser for quick navigation. When an activity in the guide is selected with a mouse click, the appropriate action is taken in the main window to allow the user to interact with SOLVEIT with the selected activity. This browsing feature allows a user to review and make modifications in an earlier stage if a defect is found at a later stage, or perhaps upon reflection the user had decided to make a change in design.

The main window in SOLVEIT serves as the interface between the six stages of SOLVEIT and the user. It is similar to the scaffolding mechanism that was employed in the GPCeditor in Section 3.1.2 in that it explicitly guides a novice user through the required steps of problem solving and program development. The initial screen in this

window contains the status of the project. For each of the six stages, the associated activities use a check mark identification system to show which of them have already been completed. Each activity is initially performed in sequence, however nothing prevents the skipping of an activity. From the initial screen in this window, the user may enter a particular stage by selecting the appropriate icon. When a stage is selected, the main window is resized, repainted, and the appropriate information is presented for that particular stage. The window at that point contains information relative to the first activity in that stage. At the bottom of the window are two buttons which are used to navigate through all of the stages and activities in SOLVEIT. It is similar to traversing the structure of the utility window in a sequential fashion.

4.1.1 Challenges in Problem Formulation

The first stage of SOLVEIT is the Problem Formulation stage. As indicated in the Dual Common Model in Section 3.5.2, the resulting product of this stage is a knowledge base of information about the given problem which represents the foundation for the eventual solution. The creation of a knowledge base is the result of a careful and detailed analysis of the problem statement by the problem solver.

Unfortunately for most beginners this analysis process ends after a brief glance at the problem statement. Novices tend to rush through the analysis so that they can jump right into the design and source code. For simple programs this may not be so detrimental, but as problem complexity increases it will become apparent to a novice how important it will be to complete the required steps of problem formulation. The reason that novices typically proceed in such a manner is because they are rarely taught the

importance of performing problem statement analysis. Most introductory programming courses focus on design and source code. Textbooks support the classroom focus by stressing language syntax and semantics.

Fortunately, SOLVEIT can guide a novice problem solver through the activities involved in problem formulation so that these skills can be absorbed mentally for use in advanced program development environments. In advanced environments, such activities are assumed to be the responsibility of the problem solver, not the programming environment.

The first activity in the Problem Formulation stage is the construction of a Preliminary Problem Description. This is a relatively straight-forward activity and does not necessarily require too much work. Upon selecting the Problem Formulation stage from the project status view, the application presents a problem description editor for the entry of the Preliminary Problem Description. The problem solver is required to enter a paraphrased version of the problem statement according to a personal interpretation of it. The purpose of the editor is so that the descriptive wording provided by the user can be stored in a reference database for use in future activities in the SOLVEIT session.

Once the user's entry is saved in the reference database, the user proceeds to the next activity, the creation of a Preliminary Mental Model. The purpose of this activity is for novices to examine the validity and correctness of their interpretation of the problem description. The entry into this activity automatically invokes SOLVEIT's *verbalization tool* which presents a series of questions to the user to stimulate the reflection process. The main window consists of a split screen: the upper half contains the problem description entered in the previous activity, and the lower half provides an area for a user

to answer the inquiries presented by the verbalization tool. A small divider between the split-screens serves as the location in which the verbalization tool presents its interrogations.

The types of questions presented by the verbalization tool are those that a problem solver is required to answer in order to fully understand a problem statement. Most novices do not bother asking these types of questions to themselves. Of those that do, some only consider a subset of the questions; typically the bare essentials to create a lackadaisical program such as the goal and the givens. As a result, they do not even know what the purpose of the program that they eventually write is, whether or not it was designed properly, and whether or not it conforms to the requirements of the problem statement. In a sense, SOLVEIT's verbalization tool is training a novice problem solver how to initially approach a new problem. It is hoped that the novice will ascertain this question-asking process for use in the future without the assistance of SOLVEIT.

Currently, the four questions that are asked by the verbalization tool are: (1) What is the goal? (2) What are the givens? (3) What are the unknowns? (4) What are the conditions? Answering these questions should be a relatively trivial task since the problem description is present in the upper half of the split screen. Of course, this premise is based on the assumption that the user's personal interpretation of the problem statement is correct. If any of the questions cannot be answered, or the answer is not clear or immediately apparent, the novice must re-examine the original problem statement and correct the preliminary problem description appropriately. The verbalization tool therefore serves as a check to make sure that the problem description is correct and has the required information.

Once all four questions have been answered, they are stored in the reference database for future use. At this point the novice should have a well-defined preliminary problem description and a clear preliminary mental model of the problem statement. The user is then ready to proceed to the third and final activity in problem formulation: the creation of a structured problem representation.

For this final activity, a split screen is once again provided. The upper half consists of the user's preliminary problem description and the lower half represents an electronic filing cabinet with folder tabs for six problem statement elements: goals, givens, unknowns, conditions, constraints, and resources. This filing cabinet will be used to store data for the six elements and serves as the *information elicitation tool.* The objective here is to extract, categorize, and organize important information from the problem description. The goal is to produce a knowledge base of facts that will be used in subsequent stages of SOLVEIT.

Unlike the verbalization tool, the information elicitation tool has no direct interaction with the user. Users are on their own to mine the necessary information from the problem description and enter it under the appropriate element's folder. To accomplish this, the user highlights the text in the problem description and adds it to one of the six categories. The mental extraction and organization of such information from a problem description is rarely done by novices. As an example, information such as conditions and constraints are sometimes not considered by a novice until a first version of the source code is developed. The importance of the initially omitted information is then recognized and subsequently incorporated into the source code to meet the requirements of the problem statement. This approach is often dangerous and can lead to

unpredictable results. SOLVEIT stimulates the information elicitation process in hopes that the activity becomes a natural habit that novices will acquire.

4.1.2 Challenges in Solution Planning

The second stage of SOLVEIT is the Solution Planning stage. As indicated in the Dual Common Model in Section 3.5.3, the resulting product of this stage is a solution plan that results from the transformation of the knowledge base that was created in the Problem Formulation stage. In the Solution Planning stage, an initial plan is chosen from a set of alternative plans. By integrating the information gathered from problem formulation, this plan will be refined and eventually represented by a series of subgoals and a preliminary data model; both of which are required for design.

It is a proven fact that effective solution planning eludes novice problem solvers. This is most likely attributed to the fact that they do not know where to begin. Due to their lack of domain knowledge, solution planning for a novice may only consist of the planning of an algorithm, often focusing on program control and neglecting the considerations for necessary data structures. Fortunately, the Solution Planning stage of SOLVEIT provides a novice with the support and direction required to accomplish the objectives in planning a viable solution.

The first activity in the Solution Planning stage is Strategy Discovery. This is a particularly troublesome area for novices evidenced by their tendency to strictly focus on only one path in determining a solution plan. Alternative solutions are seldom considered, most likely because they have a difficult time mentally visualizing two or more approaches to achieving the desired goal at the same time. Rather than being confused by

the available alternatives, they tend to adhere to a blind path to the solution and revise it as required, even though it may end up not being the optimal approach. SOLVEIT helps novices overcome this difficulty by providing a work area for the generation of alternative strategies.

Upon selecting the Solution Planning stage from the project status view, or proceeding directly from the Problem Formulation stage, the application presents a user area for keeping track of possible solution approaches. The *strategy discovery tool* employed in this activity provides the user with a simpler way to identify possible approaches, evaluate them, and weigh them against other approaches. For this activity to be productive, the problem solver must perform the necessary brainstorming to find possible approaches to the problem, even if there is an obvious approach. For each strategy that is *discovered*, the user is free to enter any necessary text to describe the pros, the cons, and any miscellaneous information that needs to be considered should that approach be chosen. Once the discovery process has concluded, the alternatives have been weighed, and an optimal strategy is chosen, the user then creates a brief outline containing the steps that will be followed to achieve the selected strategy.

Upon completing the brief outline, the user proceeds to the second activity of this stage: goal decomposition. Strategies like goal decomposition are rarely used by novices and is evidenced by the programs that they write. The source code is often hard to manage and difficult to comprehend. Programs like this also have unnecessary amounts of source code because novices do not take advantage of looping constructs and subprogram abstraction.

For those novices that are experienced enough to recognize the benefits of subgoal decomposition, lack of training in performing effective decomposition results in outcomes that provide very little benefit. It is therefore obvious that novices need a tool to guide them through the process of goal decomposition. SOLVEIT facilitates this process with the *goal decomposition tool.* This tool provides support for a high-level decomposition of the goal identified in the Problem Formulation stage.

When the goal decomposition activity is started, the window is split into two screens. The upper half contains a work area for the goal description, and the lower half contains a work area for subgoal descriptions. To start, the previously entered goal is retrieved by SOLVEIT from the reference database and inserted in the upper work area. The user identifies the goal with a short name in the *Goal Name* dialog box. To decompose the goal, the *Decompose Goal* icon to the right of the window is selected.

Once a goal is selected for decomposition, control shifts to the lower work area, leaving the upper work area inactive and non-modifiable. At this early stage of decomposition, only subgoals that are considered to be one abstraction level lower with respect to the goal should be created. This by design is not to overwhelm the novice by having them perform exhaustive subgoal decomposition and lose track of the focus of solving the problem. The idea is to create a high-level representation while at the same time build a novice's confidence level.

The subgoal creation and identification process is initiated by selecting the *New* icon. Once created, a subgoal name can be entered in a dialog box similar to the one for the goal name in the upper half of the window. A subgoal description can then be entered in the lower work area. Browsing through all of the subgoals that are created by the user

is accomplished by using the *Next Subgoal* and *Previous Subgoal* icons. Once all subgoals have been identified and reviewed, the control returns to the upper window by selecting *Exit subgoals*.

At this point, the user is ready to encounter the third and final activity of this stage, Data Modeling. This activity integrates the givens and unknowns identified in the Structured Problem Representation activity with the subgoals just identified. It is important to determine which data each of the subgoals will interact with. This mapping process will decrease the chance of parameter initialization and parameter passing errors, two common bugs in novice programs.

To facilitate this process, an electronic filing cabinet with folder tabs for givens, unknowns, and additional intermediate data is presented. Each of the three categories when selected, has columns for identifying the name of the given, unknown or intermediate data, a description of it, its origin in the case of a given, and its destination in the case of an unknown. A final column serves to indicate which subgoals the item maps to. The end result is a preliminary data model that will help facilitate the solution design in the next stage of SOLVEIT.

4.1.3 Challenges in Solution Design

The third stage of SOLVEIT is the Solution Design stage. As indicated in the Dual Common Model in Section 3.5.4, the resulting product of this stage is a solution design that is a product of the transformation of the solution plan that was created in the previous stage. This is perhaps the most challenging stage for a novice because it is this stage that requires the most domain knowledge.

Solution design, as defined by SOLVEIT, requires the skill of subgoal decomposition through progressive refinement, the ability to specify definitions for modules, and the knowledge to perform logic specification. None of these three cognitive activities are naturally acquired and because of this novices need something to guide them. Three activities are involved in this stage. The subgoals identified in the solution plan are decomposed further and refined into modules. The resulting modules are then given definitions (the goal of a module) and direction of data flow between modules is determined. The modules are then formally defined in logic using pseudocode.

As was indicated in the Solution Planning stage, novices need support to perform effective subgoal decomposition. To further decompose the first-level subgoals identified in the previous stage, a more robust tool is required than the goal decomposition tool that was used to identify those first-level subgoals. The new tool must provide visual support so that a novice can keep track of all of the different subgoals as well as hierarchical the level of the subgoals. This is important because as the subgoal hierarchy grows, complexity and the problem solver's memory demands increase.

The *module organization/refinement tool* takes care of this need in the first activity of solution design, the Organization and Refinement activity. Upon invoking this activity, a hierarchical chart is displayed. The chart contains rectangles which represent the goal and subgoals, and directional arrows that indicate the direction of decomposition in a downward fashion. Initially the goal and subgoals presented are those that were identified during goal decomposition in the Solution Planning stage. Inside each rectangle, the goal and subgoal names that were assigned by the problem solver are indicated. Above each rectangle, a numerical system is used to identify and keep track of

the subgoal hierarchy. At the top of the hierarchical tree, the goal is at level 0. Subgoal hierarchy is identified by *x.y*, where *x* is the level in the hierarchy, and *y* is a number that distinguishes between subgoals that share a common ancestor.

As an example, the first first-level subgoal in the hierarchy would be identified as *Subgoal 1.1*. If that first-level subgoal is further decomposed into three new subgoals, then those new subgoals would be identified as *Subgoal 2.1*, *Subgoal 2.2*, and *Subgoal 2.3*. If all three of these second-level subgoals are to be further decomposed into two additional subgoals each, then each path would result in a *Subgoal 3.1* and *Subgoal 3.2*.

The module organization/refinement tool makes the decomposition process an easy task for novice users. It eliminates the nuisances encountered when performing this activity on paper. The user simply selects a rectangle and right clicks on the mouse to get two options: *organization* and *refinement*. If organization is selected, additional options are presented to allow for the rearrangement of the subgoal in the hierarchy or the renaming of the subgoal. If refinement is selected, additional options are presented to decompose a subgoal or to delete a subgoal.

Rearranging a subgoal in the hierarchy prompts the user to enter the name of the new ancestor for the subgoal. The user may move a subgoal vertically or horizontally in the hierarchy. The act of decomposing a goal prompts the user to enter a subgoal name. Once the name is entered, a new rectangle appears beneath the one that was selected for decomposition. It contains the name of the subgoal just entered, and a numerical value indicating its level in the hierarchy.

Once subgoal decomposition has been performed and a refined subgoal hierarchy is in place, the next activity that a user is required to perform is to provide definitions for

the modules (once a subgoal has been formally defined it is referred to as a module, signifying the fact that it will become a logical entity in the implemented solution) and identify the direction of data flow between modules. Again, this is an area which novices lack experience in. The Function/Data Specification activity provides a tool called the *function/data specification tool* to guide this process. This process will help a novice reduce parameter passing errors and missing data flows in their solution. SOLVEIT supports the process with an easy to use graphical interface that simplifies the module/data flow association task.

Upon starting this activity, multiple views are presented. On the far left is a file manager-like representation of the subgoal hierarchy created in the previous activity. The first thing that the user has to do is select a subgoal. Towards the top of the screen an area appears for the *Function Statement*. It contains the description of the subgoal as was entered in the previous activity. Beneath the function statement is a work area to perform the required data association with the modules. This data is automatically presented to the user for association; it consists of the givens, unknowns, and intermediate data that was identified in the Solution Planning stage. The user selects each data element associated with a module, and identifies whether it is an input or an output to the module. The user specifies the origin of the data and whether or not the data is of simple or composite type. This act of associating modules with data flows results in a data dictionary, an essential software engineering element. The resulting data dictionary gets stored in the reference database.

The final activity of this stage is the specification of a high-level algorithm. This is accomplished by using the *module logic specification editor* in the Logic Specification

activity. The generation on an algorithm for each subgoal requires at the very least semantic knowledge. Since algorithm details are language independent, syntax issues are delayed until solution implementation. As a result, this activity does not place strong cognitive demands on the user as the previous activities did. However, given the resulting output of the previous two activities, it is not always a straight-forward process to design the logic.

Fortunately, SOLVEIT provides an *algorithmic constructs toolbox* which aides users in their design of algorithms. The user selects a module for logic specification from the left window pane and performs the required construction using the available choices from the toolbox. The toolbox includes data types, control structures, and mathematical, relational, and logical operators. Once all of the algorithmic logic is entered, the third and final problem solving stage of SOLVEIT is complete. This also marks the end of problem solving activities and signifies the start of program development.

4.1.4 Challenges in Solution Translation

Of the difficulties that novices encounter in program development, the ones most commonly addressed in the literature seem to be those related to syntactical issues. A review of Chapters 2 and 3 supports this fact. With this in mind, the authors of SOLVEIT approached the design of their tool with the intention of not addressing syntactical issues.

The essence and the heart of SOLVEIT is its support for problem solving. In fact, it is the premise for SOLVEIT, that effective problem solving precludes the creation of a properly functioning program, that the authors chose not to provide support for source code translation. Furthermore, programming environments are abundant these days and

imposing a new integrated programming environment in SOLVEIT that the user may not be comfortable with may cause that user to lose interest in the tool.

The only support for the Solution Translation stage of SOLVEIT is the creation of a text file that contains the algorithms defined in the Solution Design stage. This text file can be opened in the text editor of the programming environment of choice. All the user has to do is translate the algorithmic logic into the desired programming language syntax. At this point, due to the guided processes followed in the first three stages of SOLVEIT, the algorithms should contain all the required logic and data, and translation into source code should be trivial.

4.1.5 Challenges in Solution Testing

After source code has been generated and successfully compiled during solution translation, it is then time to proceed to the fifth stage of SOLVEIT, the Solution Testing stage. This stage is frequently omitted by novices. One reason is that the distinction between testing and debugging may not be clear to them. The few novices that possess the desire to perform solution testing lack the strategies to perform the task effectively. There are several reasons for this general deficiency. A test plan is something that takes time to do, and instructors are generally not interested in reading and grading such plans. The focus in programming assignments is to generate a working program and for this reason, novices do not pay much attention to testing.

Also contributing to this is the growing trend of introductory programming course offerings in secondary schools. More and more high schools are making programming courses available to encourage advanced college placement. Programming languages are

chosen because of their popularity in the real world, not for their merits. Students at the secondary school level do not have the capability to comprehend the software process so instructors focus strictly on the generation of simple programs without teaching the importance of testing, among other software engineering principles. As a result, these students enter college courses without experience in testing.

Testing, as distinguished from debugging, is the verification that the goals of the problem were met. In the Solution Testing stage of SOLVEIT, novices are guided through a two-step process to help develop adequate test cases. Two tools are used to facilitate the testing process: the *black-box testing tool* and the *white-box testing tool*, each used in the respective sequence of the two-step process. The former is used for testing at the module-level (specifications/goals) while the latter is used to test the internal implementation of modules.

Upon entering the Solution Testing stage, the Blackbox Testing activity begins. The screen once again contains the module hierarchy in the left window pane. The top window pane contains the three outputs from the Problem Formulation stage: the preliminary problem description, the answers to the inquiries during the preliminary mental model activity, and the structured problem representation. These three outputs reappear in this stage because they contain the vital information that will be required to design the tests. The intent here is for the user to reflect back at the Problem Formulation stage to stimulate test case development.

The user identifies testable requirements by reading the information in the top window pane and deciding which module(s) the problem statement phrase affects and should be tested for verification. When a decision is made that a particular module is

affected by a problem statement phrase, the user selects that module in the left window pane. Beneath the top window pane is a work area for the development of tests. When a module is selected for test case development, the work area will reflect the fact that test cases created in the work area will apply to the selected module. The user enters the data to be tested for the module in the input window, and the expected result in the output window.

This process is repeated for as many sets of test data that are required to verify the requirements in the problem statement. It is good practice to cover a wide range of test values. This typically includes boundary conditions, a random sampling of data within the valid ranges, and data that if not properly addressed in the program could create an unstable state. This test case information is compiled and will be seen in a generated output file once the user is done identifying all tests. This output file can then be printed and used as a check off sheet when validating the tests.

The next testing activity is white-box testing. In this activity the left side of the window contains the program source code. The user is prompted to read through the source code and identify and mark the control structures and control variables in the program. Once this is done, the user provides test data for each variable, as required, to cover all possible control paths for various test inputs. The test case identification method is similar to that used in black-box testing. The test case information is compiled and will be seen in a generated output file once the user is done identifying all tests. This output file can then be printed and used as a check off sheet when validating the tests.

The student at this point returns to the programming environment that was used during solution translation and conducts the tests that were just identified. Using the

desired testing method, the test data identified in the test cases is entered when prompted and the results are evaluated. Errors that are discovered will require an inspection of the source code or backtracking through SOLVEIT.

4.1.6 Challenges in Solution Delivery

Not to be overlooked, the activity of preparing a report for an assignment submission is something that is frequently not properly done. Traditionally, students are only required to hand in a hardcopy of the source code for a program and a sampling of program outputs based on user inputs. Nowadays, with the amount of information available to novices on the internet, it is easier than ever for novices to find source code to programming assignments, thus defeating the learning process. Instructors assign programs for a student's learning benefit, rather than as a classroom requirement, and tend not to grade the submitted assignments. This trend of not submitting documentation to solutions will obviously lead students away from learning a key software engineering process.

Preparing a status report for a given problem is something that novices most likely do not know how to do, further supported by the fact that pedagogy of programming does not address this. Fortunately, through the process of solving a problem with SOLVEIT, a summary report has been automatically generated by the application. The report contains the results produced at each stage of SOLVEIT. Among other things, it includes the subgoal/module hierarchy tree as well as the test cases that were generated in the previous activity. This report can be printed or saved to directly to a file. It is expected that the novice will read the report and reflect on the processes that were

followed. This reflection process is important because students need to identify and understand the cause of their errors and realize the need to implement preventative strategies in the future.

4.2 Recommendations for the Improvement and Evolution of SOLVEIT

It is fair at this point to recognize the fact that SOLVEIT is a far more comprehensive learning tool for novices than the ones discussed in Chapter 3. The Dual Common Model (the theoretical foundation for SOLVEIT discussed in Section 3.5), and the novice challenges that are overcome through the use of SOLVEIT (argued in Section 4.1) clearly provide enough evidence that this tool gives novices a great advantage. However, no matter how comprehensive and useful anything in this world is, there is always room for improvement and evolution.

With this is mind, the ideas presented in this section address various ways that SOLVEIT can be improved in certain areas and can evolve in others. The personal views presented are intended to initiate feasibility studies for future enhancements to SOLVEIT. The suggestions are based on current software engineering techniques that the tool does not directly address or only touches upon. It is assumed that most of these principles have been specifically excluded from SOLVEIT because they may be overwhelming to a novice programmer. However, it may be possible to incorporate some of the missing pieces of the software engineering methodology at an introductory level. The intent of including these principles is not an attempt to fully entail a complete methodology in a novice-oriented flavor, but to further enhance the learning experience by providing additional support that will help novices reduce defects. It is believed that these

recommendations can help SOLVEIT become even yet more robust, while at the same time provide a more beneficial learning experience to the novice user.

4.2.1 Problem Formulation Stage

Recalling from Section 4.1.5 (the Solution Testing stage of SOLVEIT) the development of black-box test cases is supported by the three outputs of the Problem Formulation stage. These three items: the preliminary problem description, the answers to the inquiries during the preliminary mental model activity, and the structured problem representation, are presented in a window above the work area for test case generation. Given these three inputs, and the visual presentation of the test case work area, it does not appear that it would be clear to a novice what the correct approach to designing such test cases would be. This is because the information from problem formulation is not structured in a helpful way. The wording in the preliminary problem description is in a paragraph format, the answers to the inquiries appear to be only designed to help formulate a thorough preliminary problem description, and the structured problem representation is only a mining process that extracts the components of a problem description into an organized and categorized knowledge base. These are all useful outputs for solution planning, however given this information in the Solution Testing stage, there is no evidence that the development of test cases would be a straight-forward process for novices.

What this initial stage of SOLVEIT lacks is an activity in which concrete goal-oriented requirements are identified and documented. These requirements would be used to guide the novice in the test case development process. The development of

requirements is a very important stage in the software engineering process and should be included at some level to complement the test case development activities in SOLVEIT. The generation of goal-oriented requirements would be comparable to system level requirements in the software engineering domain. That is, the internal implementations of the solution to the problem at this stage are not yet known and do not need to be known in order to generate these requirements.

A new SOLVEIT process could guide the user through the requirements generation stage similar to the elicitation that takes place in generating the structured representation of the problem. The user should understand that each statement of the problem description can map to (usually) more than one goal-oriented requirement. Furthermore, each statement in the problem description may be a compound statement that can be broken down into numerous logical components.

As is commonly done in software engineering, testable requirements should include the word *shall* so that the novice later recalls what the subgoal *shall* do when a certain test case input is entered. As an example, if a phrase from the problem statements reads "The range of valid values for the number of test grades must be between 0 and 10," then the following three derived requirements would certainly be possible: "If the number of test grades entered is less than 0, then the program shall generate error message #1." "If the number of test grades entered is greater than 10, then the program shall generate error message #2." "Otherwise, the program shall accept the number of test grades."

It is important that each derived requirement be mapped to a subgoal. During test case validation, this will assure that each subgoal has been verified by the requirements

that are derived from it. Of course, the validation process should not proceed until all requirements have been verified against the problem statement. That is to say that all requirements must be carefully inspected against the goals and subgoals of the problem statement before test case development. It is believed that a complete set of derived requirement statements will help the novice develop comprehensive, structured, and cohesive test cases that verify the goals contained in the problem statement. It is also believed that the novice will start to get a clearer mental picture of the parameters of the problem statement which will eventually aid in solution planning and design.

4.2.2 Solution Planning

The most obvious deficiency in the Solution Planning Stage in SOLVEIT is the lack of support during the Strategy Discovery activity. Users are on their own to discover possible solution paths. They are presented a blank screen as a scratchpad for working out all of the alternatives and the hope is that they will end up choosing the best one. Given that the users of SOLVEIT are primarily novices, it is clear that this stage may pose a difficult challenge because they lack the ability to plan and need a process to guide them. Also, they will most likely be intimidated by the blank screen and not know how to proceed.

The development of novice support in this activity may turn out to have the most potential for the success of SOLVEIT. This premise is based on the fact that there are many alternative paths to a solution based simply on the various programming paradigms in existence today. Two of the most popular approaches, procedural programming and object-oriented programming should be supported by SOLVEIT because they are

commonly used in introductory programming courses. It appears that at the time SOLVEIT only supports the procedural approach.

To assist in the difficult task of strategy discovery, SOLVEIT could provide *solution strategy templates* for all supported paradigms in a *template library*. Templates would be chosen from the library and would be used to identify potential solution strategies based on the nature of the problem statement. An elicitation tool could be used to probe the user for answers regarding the goals and nature of the problem. The templates that most closely matched the answered questions would appear on the screen. The user would then be able to browse through the list of presented templates, weigh the merits of each, and choose the most appropriate one. The user would then proceed by entering a project plan for the selected strategy using the template as a guide.

4.2.3 Solution Design

The Solution Design stage of SOLVEIT appears to be the most well-developed stage of all six. Of the three activities that comprise this stage, the one that would appear to give novices the most problems is the generation of algorithms in the Logic Specification activity. During this activity, the user is presented with the hierarchical tree of modules and is required to generate algorithms for each in a pre-determined order. The activity is supported with a toolbox that contains commonly used data types, control structures and operators.

At this point, accuracy is of the utmost importance because this is the final stage of problem solving. The Solution Translation stage which follows this stage relies on the accuracy of the algorithms developed in this stage because these algorithms are

transformed directly into programming language syntax. Given that the control flow and data flow in an algorithm are no easier to visualize than in its source code counterpart, a visual tool is therefore needed to ensure the accuracy of the algorithms as they are developed.

It is believed that a new support tool should be introduced into SOLVEIT to provide this visual support. During algorithm construction, the tool would automatically generate a traditional flowchart, corresponding to the logic in the algorithm, in real time. As the user enters the logic for an algorithm, the respective flowchart element is added to the designated work area. This will help novices ensure that the algorithms that they are building for each module are in accordance with the definition of what the module is intended to achieve. It is believed that such visual support will help novices construct algorithms with fewer errors because they will be able to visualize the flow of control and data in their algorithms. Fewer defects in algorithm will translate to fewer defects in program source code.

4.2.4 Solution Translation

As explained in Section 4.1.4, SOLVEIT does not address solution translation for good reasons. The only support for this stage is a text file generated by SOLVEIT containing the algorithm designed in the previous stage. This text file is used as a guide in the translation of the algorithm to source code. It is assumed at this point that the algorithm is correct because the problem solving methodology was followed to completion. However there is no guarantee that the solution translation follows a direct mapping to programming language syntax. This closeness of mapping was discussed in Section 2.5.5.

Semantical errors introduced during translation may lead novices to incorrectly believe that the errors may have been introduced in a prior stage, especially if the error is syntactically correct. Supporting this is the fact that compilers are not designed to find these types of errors. They do not serve as verification tools to make sure the source code is consistent with the algorithm. It is obvious that many errors may be introduced during solution translation, most of which do not have roots in the problem solving stage. All of these reasons support the fact that novices need a tool to help facilitate translation from algorithm to source code.

In the interest of diverting syntactic issues away from SOLVEIT and the Dual Common Model, while at the same time giving novices some support during this stage, a suggestion can be made to help novices during solution translation. To introduce this suggestion, the following needs to be understood. With SOLVEIT solution translation takes place in the programming environment of choice and is required to be done in the programming language supported by that environment. Most environments in existence serve only as a *drawing table* rather than a tool to help novices develop source code. Of course, these environments provide other valuable features such as color signaling to identify keywords and *auto complete* which eliminates the need to remember the exact spelling of a syntactical construct, but none of these features support the direct translation of algorithm statements into source code. Users are presented with a blank screen are expected to enter their source code without any process to guide them. Fortunately, the generation of an algorithm in the previous stage resulted in a text file than can be opened within the programming environment editor. This file serves as a simple guide for performing translation and immediately takes away the intimidation factor of a blank

screen. The text file generated by SOLVEIT is definitely a great stepping stone in assisting novices in solution translation, however this area could use some improvement.

The suggestion here would be for the text file to be somewhat more informative. The text file would be more helpful and result in far fewer semantical errors during translation if it contained supportive source code comments generated by SOLVEIT. These comments would be interleaved with each logical entity and/or statement of the algorithm and would identify common pitfalls that occur when a particular entity is translated. It is believed that adding such a feature would make the novice more aware of potential errors that could be introduced. These programming language independent comments could be presented in the form of a checklist.

After performing an initial translation, the novice would then be required to mentally answer the set of questions in the checklist. These questions are intended to make the novice reflect on the source code and make sure that common errors were not overlooked. Modifications are made by the user as required and the checklists are deleted from the file when the user is confident that the translation has followed the guidelines.

4.2.5 Solution Testing

As indicated in Section 4.2.1, the generation of test cases could be made easier if the presentation of concrete requirements supplemented this stage. However, based on these requirements, it is still not clear that novices would be able to generate thorough and comprehensive test case data. Some requirements may be worded ambiguously, while others may be open to interpretation. It is believed that an elicitation tool could support test case development to ensure that a high quality standard is met. For each requirement

that was derived in problem formulation, two set of questions could be asked about each requirement.

The first set of questioning would lead the user down a path to confirm that the requirement is not vague, misleading, or inadequate. This inquiry process could end up in one of two possible outcomes. The first outcome would be that the student realizes at some point during the questioning that the requirement is not concrete and requires a clarification or needs to be further broken down. In this case, the question session is aborted, the appropriate modifications are made, and the process is repeated. The second outcome would be that the requirement is adequate and is ready have test case data associated with it since all questions have been answered affirmatively.

The second set of questioning would only begin once it was determined that all of the requirements were sound and correct. This new line of questioning would lead the user down an adaptive path of questioning to determine the nature of the requirement. The purpose of this is to determine what *type* of requirement is being presented. Examples of some requirements types are: performance requirements, range testing requirements, and boundary condition requirements.

Once the system has induced a determination, a checklist of commonly used test case suggestions based on the requirement type will be generated and provided to the user as a guide for test case development. It is believed that the two-step process of verifying the authenticity of the requirements and providing suggestions for test cases based on the nature of requirements will assist the novice user in generating test cases that will reveal any errors that may have been unknowingly introduced in the process.

4.2.6 Solution Delivery

As indicated in Section 4.1.6, SOLVEIT does most of the work in generating the documentation for the submittal of a programming assignment. Since very few, if any modern tools do this, it is really not fair to say that this area needs improvement. The only recommendation that can be made to this stage would be to incorporate the suggested improvements from the previous five sections above. This of course would include a new area for the list of goal-oriented requirements. To complement this list, a requirements cross-reference matrix could be generated which would contain the list of requirements and which modules they map to. With this added information, the instructor grading the assignment can see if the student fully understood the problem statement before planning the solution and test cases.

4.3 Recommendations for Future Work

Each of the learning tools discussed in this thesis provides a respectable contribution towards overcoming novice challenges. Some tools by design are more comprehensive than others, but the underlying motive is the same: to give the novice a mental edge in some aspect of problem solving and/or program development activities. This final section of this chapter is devoted to identifying the elements of a learning tool that appear to be most effective and should form the foundation of a high quality learning tool. The elements are grouped into three categories: problem solving, program development, and human-computer interaction. These recommendations and other personal ideas can serve as a guideline in the development of effective learning tools for novice programmers.

4.3.1 Problem Solving Elements

Problem solving is an area of computer science that is frequently discussed in the literature but has received little attention from developers of software tools. Most modern tools focus on program development and provide little support for problem solving. This obviously is a trend that needs to be reversed. SOLVEIT has taken a significant leap in the development of such tools. With all that has been said about SOLVEIT in this thesis, it is clear that this tool can serve as a model for future learning tools in problem solving and program development.

One of the most important characteristics of a good problem solving tool should be its ability to guide novices in the process of planning because this aspect seems to be the most troublesome to them. It is less difficult for novices to read and understand a problem statement and determine the problem parameters than it is to initiate solution planning. As indicated in Section 2.1.2.2, novices do not possess a mental library of plans, and to compensate they tend to invent them using pre-programming knowledge. For this reason alone, it is obvious that support for solution planning should be a necessary and integral component of a learning tool.

In SOLVEIT, solution planning is achieved through a sequence of three activities: strategy discovery, goal decomposition, and data modeling. These inputs are then fed into the Solution Design stage which eventually help the user develop an algorithm. Algorithms are created using the inputs from earlier stages and should be a straight-forward process provided that all logical entities were correctly identified in the earlier stages and their associated activities.

In the GPCeditor discussed in Section 3.1.2, a solution plan is achieved in a slightly different way. Plans for goals/subgoals are either defined in terms of additional subgoals or by selecting a pre-defined plan that matches the objective of a particular goal or subgoal from a system library. The determination is based on how generalized the goal/subgoal is. The subgoal decomposition and refinement processes are repeated until each subgoal maps to a plan that can be extracted from the library. These plans are in the form of pseudocode, and when concatenated appropriately, form the algorithm for the solution.

Although both tools have slightly different approaches for solution planning, the end result of a logically sound algorithm, designed and created with a fully guided system support capability, is achieved. The algorithm is guaranteed to have a far lesser chance of containing defects than it would if users were on their own developing algorithms. It is now with greater confidence that the novice programmer can perform the required source code translation.

Another aspect of a problem solving tool that should be included as a design consideration is the support for deriving explicit goal-oriented requirements from the problem statement. As indicated in Section 4.2.1, this activity needs to be performed during the problem formulation stage because it is at this point that the user is trying to gain a thorough understanding of the problem statement. In the process of creating goal-oriented requirements based on the problem statement, the user is confirming his understanding of each phrase, as well as setting up the potential to generate well-defined test during test case development. Test cases based on requirements are guaranteed to

cover a fuller range of potential defects than those based on assumptions and the wording of the problem statement.

Visual aide support is also something that should continue to evolve as new technologies to facilitate this type of help are invented. It is a true statement that most novice programmers are visual learners. When presented with visual evidence to support the learning of a process or an explanation, students learn quicker and are more apt to retain the lesson learned in long-term memory. For this reason, elaborate graphical user interfaces are a welcome idea in the development of novice problem solving tools.

The presence of goal and subgoal hierarchies in a graphical tree format and other visual aides such as color signaling free the user of any additional mental capacity that would be required otherwise. They allow the user to focus on the activity on hand and eliminate the need to memorize the order in which subgoals are located in the hierarchy. Hypertext-sensitive work areas also provide a great navigational benefit. Such mechanisms allow the user to quickly locate the data associated with a subgoal, requirement, or data element. The flowchart recommendations discussed in 4.2.3 would provide a novice with a mechanism to visualize the flow of control in their algorithm. This flowchart could come in handy during the activity of debugging. To the extent that there are fewer things to memorize, novices will perform better with the aid of visual support.

4.3.2 Program Development Elements

As the theme of this research warrants, program development activities should not begin until an exhaustive and systematic problem solving session has been completed. In the

previous section, suggestions for the problem solving elements of a novice learning tool were presented. To complement these suggestions, recommendations for program development issues need to be addressed. This section will cover those recommendations and why they should be considered for inclusion.

The completion of the problem solving stage, in the context of this thesis, is signified by the successful generation of an algorithm. This algorithm is in turn used to generate the source code. This translation process marks the starting point of program development. It also marks the point in time that novices face their greatest challenges; those that deal with syntax and semantics. A mechanism is needed to give novice programmers a slight advantage when dealing with these issues.

As suggested in Section 4.2.4, the automatic generation of a checklist containing syntactic and semantic guidelines, interleaved within the algorithm, could help overcome some of the most commonly introduced errors in solution translation. Each of these checklists would be entered automatically into the algorithm based on the nature of the logic or control construct to which it applies. It is believed that this generation of checklists could greatly benefit the source code translation process. The use of the checklists would result in the reduction of commonly introduced errors such as array indexing, memory access violations, boundary conditions, exceptions, and system-oriented limitations.

Another issue of program development that should be considered is program readability. Traditionally, source code in a program is accompanied by supportive comments. Recalling the discussion on the technique of literate programming in Section 3.2.1, novices sometimes defeat the purpose of source code comments by adding them

after the fact rather than using them as a guide for translation into source code. In SOLVEIT, these comments are indirectly created by the user simply by following the process. The pseudocode contained in the algorithm that is used for translation serves as the source code comments once the translation is complete.

Regardless of how the comments for source code are generated, they may still not contain enough information to represent the source code that they are associated with. Furthermore, most modern tools do not provide a device to deter the writing of non-manageable source code. Program readability issues are not to be considered lightly because the lack of readability may render a program worthless for future use. If a program cannot be easily modified or the contents of it be understood, then the possibility of source code reuse, an extremely valuable software technique, has been virtually eliminated. To address this issue, a tool based on the literate programming technique should be employed upon generation of source code and comments. The resulting literate program would be easier to understand than its counterpart, and would have a greater potential for reuse.

As in the problem solving recommendations, visual aides could greatly benefit novices in source code translation. The use of color signaling to identify keywords is commonly employed in today's visual learning environments. As discussed in Section 2.7.3, this concept may not be as helpful as it is intended to be. A better approach would be to highlight semantically related information such as the elements of a looping construct.

A handful of the tools discussed in Chapter 3 contained support for the visualization of program execution at some level. It is highly recommended to continue

this trend in future learning tools. In an effort to evolve this concept, a suggestion can be made to integrate hypertext into the visualization of program execution. In this approach, each line of source code in a program is linked to the elements that it is derived from, utilizing a reverse engineering approach. As an example, the following scenario would be possible. When a user is tracing program execution, the current line of source code that is about to be executed is traditionally highlighted. If during debugging or testing, the user identifies a potential defect, the suspect line of source code can be traced back to its origins simply by clicking on it. This feature would greatly reduce the time involved in each iteration of design, code, and testing.

4.3.3 Human-Computer Interaction Elements

In Section 2.2.2, a discussion of how unnatural the task of programming is to novices was presented. A leading contributor to this was the fact that programming language designers do not take into account human-computer interaction issues when designing new languages. Instead, innovations and competition prevail, resulting in a widening learning gap. What is needed is a way to narrow this gap so that novices do not get intimated and frustrated.

Short of incorporating such issues directly into programming languages, it is very difficult to provide support for overcoming these cognitive difficulties. One approach that was employed in Section 3.4.2 was the concept of subsetting a language. In this approach, a new, smaller language encapsulates only the basic instructions of a more comprehensive language. This minimizes the library of syntax available to a novice user, preventing them from straying down the dark corridors of a complicated language syntax.

This approach seems practical and is what introductory programming courses aim to do. As indicated in Section 2.1.1, textbooks can defeat this practice because they are rarely designed for a specific course and may cause a deviation from the focus. This can be considered a non-issue if an instructor explicitly instructs the students to omit certain sections. What is more important is the program development tool and the compiler that resides in it. Given that compilers are designed to accept the complete syntax of a programming language, errors that are reported to the user may have very little use if their level of complexity is beyond the knowledge of the user. The approach in Section 3.4.2 included a customized compiler, specially designed for the subsetted language. Such a compiler would be required to effectively teach a subset of a programming language and should be considered as an option when designing a language-dependent program development environment.

In Section 2.5.5, one of the heuristics of a programming language that was discussed was its *closeness of mapping* and its effect on transforming a plan into a program. If translation is not easy novices will struggle. Programming languages complicate this issue when they contain several instructions that are similar in appearance, or when there is more than one instruction to achieve a desired result. A suggestion can be made to simplify this mapping. The recommendation here is to have a feature that allows the user to highlight a line in an algorithm and have the system provide suggestions for performing the translation to language syntax. Each suggestion would contain an explanation for each possible choice and an example of its use. This would allow the user to select the one that is really intended by reducing the guess work.

It is believed that given the plan (the algorithm), this feature could simplify the transformation into a program.

CHAPTER 5

CONCLUSIONS

The objective of this thesis was to reveal the characteristics that comprise the components of an effective novice learning tool. Based on these findings, recommendations for future work in the development of novice learning tools were provided. Additional suggestions were made based on the shortcomings of the tools as well as personal contributions. To qualify the necessity of these characteristics in the development of a new tool, convincing arguments for each were supplied. A summary of this investigation follows.

To achieve the goal of this thesis, evidence was first needed to support the need for each characteristic. The evidence was pieced together from three sources. First, an in-depth discussion on the challenges that novices face was presented. This identified the needs of novices. This discussion was then followed by an extensive review of several novice learning tools. This began the process of associating particular needs with the characteristics of a learning tool, clearly identifying how essential a particular characteristic of the learning tool is for a particular need.

In the process of reviewing the learning tools, it became apparent that although each tool made some contribution to the identification of characteristics sought after, some aspects of software engineering were not addressed. These omissions helped form a portion of the personal suggestions for the characteristics of an effective learning tool. Other suggestions for the characteristics were based on genuine ideas. The combination of the proven characteristics identified in the learning tools with the personal recommendations in this thesis can serve as a foundation for the development of future learning tools.

In Chapter 2, an extensive review of the challenges that novices encounter was presented. Several dimensions of problem solving and program development were discussed. The chapter opened with a discussion on the contribution of classroom pedagogy to novice difficulties. Perhaps the most important of all the arguments presented for pedagogy was that novices are rarely taught explicit problem solving skills in introductory programming courses. The focus in these classes is primarily on program development. Accomplices to this problem are the textbooks that are used for such courses. The textbooks typically dive into the aspect of programming and neglect the mental development of problem solving strategies.

It was also indicated in the chapter that most textbooks introduce their own learning obstacles based on the manner in which they are written. The examples presented in such textbooks frequently build on examples that appeared earlier in the lesson. If a student had not entirely grasped a simpler concept in the prior example, a trend of ignorance would begin to develop. In this case, the student would most likely not even attempt to understand the more advanced problem based on the fact that the prior example was not understood.

The inherent nature of problem solving difficulties was the next item of discussion. The general observation here was that novices do not know how to plan, nor can they naturally and effectively conceive a process to guide them in planning without explicit instruction. The weakest areas identified were the cognitive activities required for problem statement analysis, goal decomposition, and algorithm specification. Novices need classroom instruction on how to develop these skills. Taking the easy route by

coding a solution to a problem without taking the intermediary steps will only get a novice so far. As program complexity grows, so does the need for a plan.

The next issue under investigation in the chapter was novice performance in program comprehension. It was identified that novices have a very difficult time understanding source code that was not authored by them, or ironically in some cases was, but in a chaotic, non-legible manner. They have a hard time mentally putting the proverbial "pieces of the puzzle together." They can manage to read a program in a *statement by statement* approach, but fail to see the connective tissue between groups of statements, and subsequently struggle when trying to understand the logical function provided by the program. The culprit here again is the lack of training in the classroom on effective program interpretation skills. Techniques and strategies need to be strongly addressed by instructors.

Another contributing factor is the fact that instructors do not motivate their students effectively to reuse source code from earlier assignments. This is essential to programming because the ability to comprehend some else's source code is vital in the field of software engineering. Source code may need to be maintained or reused as a component to a more complicated program. On software projects, techniques are used to make source code more generic by following conventions and standards, as well as providing supportive source code comments. In classrooms, such techniques are rarely covered in detail.

A discussion on the psychology of programming followed. As indicated, the psychology of programming has had only a minor impact on the development of programming languages and program development environments. This was argued with

evidence that programming languages fail to take human-computer interaction issues into consideration. Novices have to adapt their way of thinking in ways that they normally would not in a non-programming domain. They have a hard time understanding how data is represented and manipulated by a machine. One thing novices fail to understand is that there is no direct mapping from the mathematical domain to the programming domain. A major contributor to the hardships of learning to program is the universal truth about the process of knowledge acquisition. Since novices cannot relate incoming information to prior experiences, they are forced to memorize rather than learn.

The chapter continued with discussions on programming paradigms and programming language issues. One point of view presented was the novices sometimes do not even recognize what a paradigm is until they are introduced to more than one. Difficulties encountered with language syntax issues included the syntax of looping constructs, the use of syntactic synonyms, homonyms, and elision, and operator precedence in expressions. Examples of each category were provided to support the arguments.

The discussion then transitioned to two other aspects of programming languages, semantics and pragmatics. Most novices are initially unaware of semantical errors that may exist in their program because compilers are not designed to catch such errors. Another problem attributed to semantics is the initialization and manipulation of data structures such as arrays and pointers. Novices frequently do not index arrays correctly and leave dangling pointers in list manipulations. Pragmatic knowledge at the first-year level is virtually non-existent and takes time to develop in a novice. This can only be achieved through experience. If alternative approaches are not taught correctly, students

may rely on the approach they feel more comfortable with even if that approach may not be the most efficient one. The subsection ended with a discussion on abstraction and modularization, which are closely tied into the goal decomposition activity in problem solving.

The chapter was then summed up with a discussion on debugging strategies, external influences, computer hardware issues, and compilers. A frequent observation in the debugging activity is that students correct errors that are identified by compilers usually without knowledge as to why an error had occurred. In this sense, they may actually introduce errors into the program. Computer hardware plays a minor role but is worth mentioning. Various machine architectures may cause the same program to operate differently on two different computers which will confuse a novice. The biggest problem with compilers is that they may produce error messages that are beyond the knowledge of the novice. The error messages may be too complex and may mislead a novice down the wrong path.

Chapter 3 provided an in-depth look at some of the novice learning tools in use today. Each of these tools addressed at least one novice challenge identified in Chapter 2. The foundation and functionality of these tools, as well as the aspects of problem solving and program development that they are intended to address, were discussed in detail. The chapter began with a discussion on problem solving tools. A specific problem solving methodology was discussed in detail to show how adherence to it would result in a simplified process to generate an Ada 95 program. This was then followed by a review of the GPCeditor. The intent of this tool is for the student to learn how to devise plans. The

tool forces the user to define goals in terms of subgoals or pre-defined plans from a library. Subgoals are further defined in terms of additional subgoals or plans.

The chapter continued with a discussion on program comprehension tools. A technique usually used in advanced systems, but adopted for novice programmers was discussed. This technique, literate programming, prevents the generation of source code that would be difficult for a reader other than the author to interpret. Through the use of this program novices will learn how to generate programs that are well-documented and easy to understand. A review of the EROSI tool followed. This tool provides visual support for the comprehension of subprogram invocations. It was specifically developed to aide in the understanding of recursion, a particularly troublesome area for novices.

Some miscellaneous learning aides were then introduced. The first was a multimedia textbook to teach a programming language. It represents a new trend in learning, electronic textbooks, which provide a navigational capability similar to a web browser. Research has shown that hypertext learning promotes cognition more effectively than the traditional method. ELM-ART II was then discussed. This web-based tool is a adaptive learning tool that guides users in learning LISP. ELM-ART II is an intelligent system because it can deduce a problem solver's behavior and choose a suitable set of example problems that will more likely benefit the user.

Three program development environments were introduced in the next section. FLINT draws syntactical issues away through the use of iconic programming. Users are required to create a solution design before they can create any algorithms. Algorithm specification is facilitated through the use of flowcharts. FLINT constantly monitors all dependencies during solution design and algorithm specification to ensure consistency.

The next tool discussed, X-Compiler, simplifies the programming process by reducing the number of valid programming instructions. This reduction in the number of instructions will force the user to focus on primitive operations. The compiler is designed specifically to not distract the user with complex error messages which is frequently a problem with other compilers. The last tool in this category was AnimPascal.
Among other features, this environment allows the visualization or source code execution which contributes to program comprehension skill acquisition.

The chapter concluded with an in-depth discussion of the Dual Common Model, the cognitive foundation for SOLVEIT. This discussion identified the problem solving methodology, program development tasks, and the cognitive requirements for each activity in SOLVEIT. For each stage of the problem solving and program development process, the activities, their inputs and outputs, and the required knowledge and skills were identified. This introduction paved the way for the discussion of the use of SOLVEIT in the Chapter 4.

Chapter 4 began with an extensive look at how SOLVEIT can help novices conquer the most difficult challenges that they are faced with. With the use of SOLVEIT, students learn a variety of skills for problem solving and program development activities. The first topic discussed was how SOLVEIT helps students develop the critical thinking skills required for the analysis of a problem statement. A question asking tool provoked the thought process to help establish a clear mental model of the problem to be solved. Complete problem understanding was evidenced by the ability to formulate the problem statement in one's own words. A process then guided the user in identifying the parameters of the problem. This included the goal, the givens, and the unknowns.

The focus then shifted to the tasks of strategy discovery and goal decomposition. In strategy discovery, users identify alternative solutions and weigh them against each other to discover the optimal choice. The goal decomposition process in SOLVEIT is staggered into two levels by design so that novices do not get overwhelmed from the start. An initial first level subgoal decomposition was encountered in solution planning followed by a more elaborate strategy in solution design. The data modeling activity followed the first-level goal decomposition. In this activity, the givens and unknowns of the problem statement were associated with the first-level goals just identified.

The last stage of the problem solving process involved designing the solution. In this stage, subgoals were decomposed as required, modules were defined, and the logic specification process began. A particularly troublesome area for novices is the generation of algorithms. Fortunately with the support provided by SOLVEIT, this task was immediately simplified with the presentation of the problem's parameters previously identified as well as the module names. This process was aided with an algorithm specification toolbox.

Program development activities were the next items on the agenda. As the SOLVEIT session progressed, it was indicated that solution translation activities were specifically not addressed because SOLVEIT was designed to draw attention away from syntactical issues. However, solution testing and delivery, two items frequently not considered by novices, were supported by SOLVEIT. For solution testing, SOLVEIT provides support for black box and white box testing, two important software engineering techniques used in the real world. An automated process helps novices develop the test

data to verify the correctness of their programs. The solution delivery stage produced a text file that summarized the results of each activity in the SOLVEIT session.

In the second part of the chapter, recommendations for the improvement and evolution of SOLVEIT were provided. These findings were based on personal experiences and ideas. Among these suggestions, it is believed that the most beneficial one is the activity of generating goal-oriented requirement statements. It was hypothesized that creating a simple requirements specification would not only help in gaining a more thorough understanding of the problem statement, but would create a domino effect that would continue all the way through solution verification. Concrete requirements would greatly assist in the development of comprehensive test cases. These test cases would then result in a software solution that has fully satisfied the goals identified in the problem statement.

Among the other suggestions were the use of solution strategy templates to identify possible alternative approaches to a solution, a real time flowchart that is created during algorithm specification to aid cognition through visualization, the generation of solution translation guidelines interleaved in an algorithm to avoid common errors, a requirement verification tool that justifies whether or not a requirement statement is concrete enough to design a test for, and a verification cross-reference matrix document to report the mapping of requirement statements to module names.

The last section of the chapter introduced recommendations for future work in development of learning tools. This included problem solving, program development, and human-computer interaction elements. As discussed in the chapter on novice difficulties, and evidenced in many of the tools discussed in this thesis, support for planning appears

to be the most critical characteristic in the problem solving aspect. Novices simply do not know how to plan, nor are they naturally trained to do so. By providing a guided process, each of the learning tools in their own way attempt to make a novice learn the required skills and absorb them into their strategy arsenal.

It was also reiterated that the explicit development of requirement statements derived from the problem statement would serve as a valuable component of a learning tool. As mentioned earlier, this would promote the understanding of the problem at hand as well as guarantee the development of better programs. The recommendations for problem solving concluded with recommendations for the enhanced development of more involved visual aides. Some of the learning tools discussed provided such support. This trend should continue to evolve using such new technologies like the use of hypertext and multimedia.

Recommendations for program development were presented next. This included the automatic generation of solution translation guidelines as mentioned in the recommendations for SOLVEIT and the use of literate programming as suggested by the prototype learning tool in Chapter 3 to enhance program readability. As with problem solving components, visual support for the coding process and program execution were recommended for continued use with the interest of introducing new technologies. An interesting idea presented was the use of hypertext to provide the ability for a user to trace the origin of a line of source code. This would help in the debugging process because the origin of a semantical defect may not be immediately clear, especially when source code is the only thing on the screen.

The final section of Chapter 4 was devoted to identifying the characteristics of a learning tool related to human-computer interaction issues. To make program development seem more natural to a novice programmer, the concept of subsetting a language seems like an intuitive approach, provided that compilers and other support equipment do not deviate from the subset. A final recommendation was made to provide support for the solution translation process. This would be facilitated by highlighting a line of an algorithm and having the tool provide recommendations for the associated syntactical statement.

In closing, the arguments in the preceding chapters as well as in the conclusions above have provided a firm case for the recommendations of future work. Through the process of identifying novice challenges, their needs, the current technology to support overcoming their challenges, and the suggestions provided for future work, the evidence clearly supports the arguments. The characteristics of an effective novice learning tool have been identified and justified. The benefits of SOLVEIT clearly demonstrate how comprehensive and robust the learning tool is. Developers of novice learning tools are strongly encouraged to follow in the footsteps of SOLVEIT with the added recommendations presented in this thesis.

REFERENCES

F. Bailie, "Improving the modularization ability of novice programmers," *ACM SIGCSE Bulletin, Papers of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education,* vol. 23, issue 1, pp. 277-282, Mar. 1991.

P. Bucci, T.J. Long, and B.W. Weide, "Do we really teach abstraction?" *ACM SIGCSE Bulletin, Papers of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education,* vol. 33, issue 1, pp. 26-30, Feb. 2001.

A. Cockburn and N. Churcher, "Towards Literate Tools for Novice Programmers," *Proceedings of the Second Australasian Conference on Computer Science Education*, pp. 107-116, Jul. 1997.

F. P. Deek, "An Integrated Environment for Problem Solving and Program Development," *Unpublished Ph.D. Dissertation, New Jersey Insitutue of Technology,* 1997.

F. P. Deek and J. McHugh, "SOLVEIT: An Experimental Environment for Problem Solving and Program Development," *Journal of Applied Systems Studies, Special Issue on Distributed Multimedia Systems with Applications,* in press, 2002.

F. P. Deek, J. McHugh, and M. Turoff, "Problem Solving and Cognitive Foundations for Program Development: An Integrated Model," submitted for publication to *Cognitive Science: A Multidisciplinary Journal*, 2001.

G. Evangeledis, V. Dagdilelis, M. Satratzemi, and V. Efopoulos, "X-Compiler: Yet Another Integrated Novice Programming Environment," *Proceedings of IEEE International Conference on Advanced Learning Technologies*, pp. 166-169, Aug. 2001.

M. Feldman and E. Koffman, *Ada 95 Problem Solving and Program Design*, 2nd edition, Addison-Wesley, 1996.

C. George, "EROSI - Visualizing Recursion and Discovering New Errors," *ACM SIGCSE Bulletin, Papers of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* vol. 32, issue 1, pp. 305-309, Mar. 2000a.

C. George, "Experiences with Novices: The Importance of Graphical Representations in Supporting Mental Models," *12th Annual Workshop of the Psychology of Programming Interest Group,* pp. 33-44, Apr. 2000b.

O. Grillmeyer, "An Interactive Multimedia Textbook for Introductory Computer Science," *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, pp. 286-290, Mar. 1999.

L. Gugerty and G. Olson, "Debugging by Skilled and Novice Programmers," *ACM SIGCHI Bulletin, Conference Proceedings on Human Factors in Computing Systems*, vol. 17, issue 4, pp. 171-174, Apr. 1986.

M. Guzdial, L. Hohmann, M. Konneman, C. Walton, and E. Soloway, "Supporting Programming and Learning-to-Program with an Integrated CAD and Scaffolding Workbench," *Interactive Learning Environments*, vol. 6, No. 1-2, pp. 143-179, 1998

D. Hagan and S. Markham, "Does It Help to Have Some Programming Experience Before Beginning a Computing Degree Program," *ACM SIGCSE Bulletin, 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, vol.32, issue 3, pp. 25-28, Jul. 2000.

S. Lewis and G. Mulley, "A Comparison Between Novice and Experienced Compiler Users in a Learning Environment," *ACM SIGCSE Bulletin, Proceeding of the 6th Annual Conference on the Teaching of Computing / 3rd Annual Conference on Integrating Technology into Computer Science Education on Changing the Delivery of Computer Science Education*, vol. 30, issue 3, pp.157-161, Aug. 1998.

B. Liffick and R. Aiken, "A novice programmer's support environment," *ACM SIGCSE Bulletin, Proceedings of the Conference on Integrating Technology into Computer Science Education*, vol. 28, issue SI, pp. 49-51, Jun. 1996.

R. Mayer, "The Psychology of How Novices Learn Computer Programming," *ACM Computing Surveys (CSUR)*, vol. 13, issue 1, Jan. 1981.

L. McIver and D. Conway, "Seven Deadly Sins of Programming Language Design," *Proceedings, Software Engineering: Education and Practice 1993 (SE:E&P'96)*, pp. 309-316, 1996.

L. McIver, "The Effect of Programming Language Error Rates of Novice Programmers," *12th Annual Workshop of the Psychology of Programming Interest Group,* pp. 181-192, Apr. 2000.

J. Pane and B. Myers, "Usability Issues in the Design of Novice Programming Systems," *Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-96-132, Pittsburgh, PA*, 85 pages, Aug. 1996.

J. Pane and B. Myers, "The Influence of the Psychology of Programming on a Language Design: Project Status Report," *12th Annual Workshop of the Psychology of Programming Interest Group,* pp. 193-208, Apr. 2000.

J. Pane, C. Ratanamahatana, and B. Myers, "Studying the language and structure in non-programmer's solutions to programming problems," *International Journal of Human-Computer* Studies, vol. 54, no.2, pp. 237-264, Feb. 2001.

V. Ramalingam and S. Wiedenbeck, "An Emperical Study of Novice Program Comprehension in the Imperative and Object-Oriented Styles," *ACM, Emperical Studies of Programmers*, pp. 124-139, 1997.

M. Rosson, "Human Factors in Programming and Software Development," *ACM Computing Surveys*, vol. 28, no.1, Mar. 1996.

M. Satratzemi, V. Dagdilelis, and G. Evageledis, "A system for program visualization and problem-solving path assessment of novice programmers," *Annual Joint Conference Integrating Technology into Computer Science Education, Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, pp. 137-140, Jun. 2001.

J. Spohrer and E. Soloway, "Novice Mistakes: Are the Folk Wisdoms Correct?" *Communications of the ACM*, vol. 29, issue 7, Jul. 1986.

W. Suchan and T. Smith, "Using Ada 95 as a Tool to Teach Problem Solving to Non-CS Majors," *Annual International Conference on Ada, Proceedings of the Conference on TRI-Ada '97*, Nov. 1997.

G. Weber and M. Specht, "User Modeling and Adaptive Navigation Support in WWW-based Tutoring Systems," In A. Jameson, C. Paris, C. Tasso (Eds.) *User Modeling: Proceedings of the Sixth International Conference UM97.* Vienna, New York, Springer Wien New York, 1997.

U. Ziegler and T. Crews, "An Integrated Program Development Tool for Teaching and Learning How To Program," *Technical Symposium on Computer Science Education, The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education,* Mar. 1999.

GLOSSARY

Abstraction
A way of allowing a program to specify that some process is to be done (by calling a subprogram) without providing details of how it is to be done in the calling program.

Algorithm
In programming terms, a formula for solving a problem for a computer program. An algorithm needs to be translated into a programming language syntax before it can be compiled and executed as a program.

Assembly Code
The microprocessor-level code equivalent of program source code. This code contains instructions for manipulating data at the hardware register level. A source code instruction usually maps to several sequential assembly code instructions.

Assumption
In terms of problem statement understanding, an educated guess made for some aspect of the problem that may not be explicitly presented.

Beacon
A cognitive element involved in the task of program comprehension. A beacon is a mental grouping of source code statements that collectively represent a high-level operation.

Cognition
The mental process of knowing through awareness, perception, reasoning, and judgment.

Cognitive Process
In terms of the Dual Common Model, a process that is required when performing a specific task during problem solving. The six cognitive processes used to solve problems in the Dual Common Model are: analysis, synthesis, evaluation, knowledge, comprehension, and application.

Cognitive Result
In terms of the Dual Common Model, the learning outcomes acquired by a student through the use of a learning tool. Four cognitive results achieved through the use of the Dual Common Model are: verbal information, intellectual skills, cognitive strategies, and attitudes.

Cognitive Structure
In terms of the Dual Common Model, the hypothesized human systems of thinking that support the execution of cognitive processes. The three cognitive structures identified in the Dual Common Model are: knowledge acquisition, performance, and metacognition.

Constraint
A requirement explicitly called out in a problem statement that restricts or confines an element of the problem statement or the manner in which the solution must be achieved.

Compiler
A software component, usually integrated into a program development environment, that translates the source code of an entire program into machine language that can then be executed.

Debugging
The task of performing syntactical and semantical error analysis within a program.

Defect
A general term used to identify an error that has been introduced during any of the activities of problem solving and program development.

Elicitation
In terms of problem formulation, the process of extracting, organizing, and categorizing the elements of a problem statement.

Elision
The intentional omission of a component of a programming instruction, the result being the same if it were not omitted.

Goal Decomposition
An approach to solution planning in which the goal of a problem is broken down into subgoals.

Hardware Dependency
A constraint imposed by the architecture of a particular computer that affects the execution behavior of a program. The most common constraint is the size (in bits) of a primitive data type. A program compiled and executed on two different machine architectures may behave differently in some situations.

Heuristics
Usability issues of a learning tool that relate to human-computer interaction issues.

Human-Computer Interaction
The study of how humans interact with computers, and how to design effective computer systems for humans use.

Hypermedia
An information retrieval system that enables a user to gain access to various mediums of information such as text, audio, video, still images, and animations for a particular subject.

Iconic Programming
An introductory visual programming technique in which icons that perform specified actions are selected from a palette and inserted into a program work area. This technique is used in FLINT.

Integration
In terms of programming, the process of logically piecing together reusable software components to form a program.

Interpreter
Similar in purpose to a compiler, a software component that translates a source code statement into machine language and executes it before proceeding to the next sequential instruction.

Knowledge Acquisition
The process of learning new material and storing it in long-term memory by associating it with pre-existing knowledge.

Literate Programming
A programming technique that improves program readability by combining source code with formatted text to produce a well-documented version of the program.

Logic Specification
With respect to the Dual Common Model, the activity of constructing algorithmic logic for each module that was defined during solution planning and solution design.

Looping Construct
A logical entity of a program that provides a mechanism to repeat the execution of source code instructions in order to achieve a desired result.

Metaphor
In terms of computers and programming, a familiar analogy for how a programming system works.

Methodology
A systematic process of achieving a desired outcome by following a pre-defined set of tasks or procedures.

Modularization
The technique of breaking up large, sequential programs into smaller segments, called modules, in the interest of improving program readability and maintainability.

Novice Learning Tool
In the context of this thesis, any mechanism that facilitates the learning of some aspect of the activities involved in problem solving and program development.

Operator Precedence
A programming language dependent characteristic that determines the order of execution of mathematical (or other allowable) operators that appear in a single line of source code.

Operating System
The core software component of a computer that is designed to control the hardware in order to allow users and applications to make use of it.

Paradigm
An archetype for modeling and solving problems.

Pedagogy
The methods and strategies used by instructors in teaching new material to their students.

Postfix Operator
With respect to looping constructs in certain programming languages, an operator that increments or decrements its operand after executing the statements in the loop.

Pragmatic Knowledge
In terms of problem solving and program development, the ability to recognize alternative approaches to a solution.

Pragmatics
A problem solving and program development issue concerned with identifying alternative solutions.

Prefix Operator
With respect to looping constructs in the certain programming languages, an operator that first increments or decrements its operand before executing the statements in the loop.

Preliminary Problem Description
With respect to the Dual Common Model, a self-authored interpretation of a problem statement.

Preliminary Mental Model
With respect to the Dual Common Model, a concrete mental representation of the problem statement conceived by asking inquiry questions about the problem statement.

Problem Formulation
The first of six stages of the Dual Common Model. The goal of this stage is to produce a knowledge base of information based on the problem statement that will be used as an input to subsequent stages.

Problem Solving
Refers to the activities performed in the process of designing an algorithmic solution to a given problem statement. With respect to the Dual Common Model, this covers three of the six stages: Problem Formulation, Solution Planning, and Solution Design. The completion of an algorithm signifies the end of problem solving and the start of program development.

Program Comprehension
The activity of reading a program to identify how it works and what services it provides.

Program Development
Refers to the activities performed in the process of generating, testing, and delivering a program based on the algorithm developed during problem solving. With respect to the Dual Common Model, this covers the final three stages: Solution Translation, Solution Testing, and Solution Delivery.

Program Development Environment
A software application that facilitates the construction, compilation, and error analysis of computer programs, usually for a particular programming language.

Pseudo Programming
A programming technique that facilitates the learning of abstraction and data structure usage. This technique is used to develop algorithms in FLINT.

Recursion
In terms of programming, a method of achieving a loop by having a function call a new instantiation of itself.

Refinement
In terms of goal decomposition, the process of breaking down a goal into smaller subgoals until the desired level of abstraction has been achieved.

Relational Operation
A comparison of two operands for equality.

Scaffold
With respect to learning, a temporary mental platform used to assist the process of learning a particular methodology or activity.

Semantic Knowledge
In terms of program development, refers to understanding the underlying mechanics of syntactical constructs.

Semantics
The underlying mechanics of syntactical constructs.

Software Test Description
In Software Engineering, a deliverable document that contains the entire suite of test cases that will be used to validate a software product in the presence of the customer.

Software Engineering
A systematic approach to the analysis, design, implementation and maintenance of software. It includes methodologies for defect prevention and process improvement.

Solution Delivery
The last of six stages of the Dual Common Model. The purpose of this stage is to incorporate the verified solution and results from the Solution Testing stage into a documented form.

Solution Design
The third of six stages of the Dual Common Model. The goal of this stage is to transform the solution plan (created in the Solution Planning stage) into a solution design. The end result of this stage is an algorithm that is ready to be translated into a programming language syntax.

Solution Planning
The second of six stages of the Dual Common Model. The goal of this stage is to transform the knowledge base created in the Problem Formulation stage into a solution plan.

Solution Testing
The fifth of six stages of the Dual Common Model. The goal of this stage is to verify that the goals of the problem were met.

Solution Translation
The fourth of six stages of the Dual Common Model. The goal of this stage is to translate the algorithm created in the Solution Design stage into the source code of a particular programming language.

Source Code
Refers to the syntactical statements (instructions) in a computer program that are entered by a programmer.

Subgoal
A logical component of a higher-level goal.

Subgoal Decomposition
An approach to solution planning in which subgoals are continually broken down into more subgoals until the desired level of decomposition has been achieved.

Subprogram Invocation
The action of calling a subprogram from within a program (or subprogram) to perform a specified action. In the case of recursion, a subprogram calls itself.

Syntactical Homonym
A syntactical construct whose behavior is decided by the situational context in which it is used.

Syntactical Synonym
A syntactical construct that can achieve the same result as another (different) syntactical construct.

Syntax
The rules that govern the formation of valid source code statements in a programming language.

Template
An entity having a preset format, used as a starting point for a particular application so that the format does not have to be recreated each time it is used.

Validation
The process of ensuring that the requirements of a problem have been properly implemented in a program. This is achieved through the process of solution testing.

Variable
An abstraction of a computer memory location identified by a user-provided name.

Verbalization
A method of drawing out information from the mind of a problem solver through the process of asking questions and recording answers.

Verification
The process of ensuring that requirement statements are consistent with the goals of the problem statement and making certain that they are all properly covered in the test case definitions.

Verification Cross Reference Matrix
In the Software Engineering domain, a deliverable document that contains a mapping of test case names to requirement numbers.

Version Description Document
In the Software Engineering domain, a deliverable document that is used to identify all modifications to a software product with respect to an earlier version.

Viscosity
The level of effort required to make a small change in a program