On the Design and Development of a UML-Based Visual Environment for Novice Programmers

Brian D. Moor and Fadi P. Deek
College of Computing Sciences,
New Jersey Institute of Technology, Newark, NJ, USA

Brian.Moor@njit.edu Fadi.Deek@njit.edu

Executive Summary

Few beginners find learning to program easy. There are many factors at work in this phenomenon with some being simply inherent in the subject itself, while others have more to do with deficiencies in learning methods and resources. As a result, many programming environments, software applications, and learning tools have been developed to address the difficulties faced by novice programmers. Of these tools, visual-based tools and the use of visualization have proven to be very effective in helping novices overcome several of these traditional difficulties. In this paper, we first examine the traditional difficulties that novice programmers encounter when take an introductory-level programming course are examined. It is important to gain an understanding of the scope of these difficulties first, as the rest of this paper considers how visual tools, visualization, and UML can be utilized to aid novice developers in these areas of difficulties. Next, we provide an analysis of several modern visual learning tools, including EROSI, AnimPascal, BlueJ, FLINT, BOOST, and SOLVEIT. In particular, we look at how these tools use visualization to help mitigate the difficulties novice programmers face. Each tool is also assessed based on its overall effectiveness of using visual aids and visualization to help the beginning programmer. We then turn our attention to the Unified Modeling Language (UML) and how it can be utilized to help the novice programmer in system design and modeling. The UML specification is carefully discussed, and aspects of the specification that hold the most potential for aiding novice programmers are identified. Finally, we focus on UML modeling and present the theoretical foundation for a new visual learning tool based on the UML standard. This proposed learning environment attempts to combine promising attributes of existing tools we previously examined, along with the potential benefits of UML-based modeling. The proposed tool would provide a superior learning environment for the novice programmer for several reasons. First, it is heavily based in the visual domain. Visual tools have continually proven to be extremely powerful in helping novices in learning abstract computer concepts. In addition, visualization helps novices construct a mental model of concepts, which is pivotal to further comprehension and understanding. Second, the proposed environment would allow for a constructivist learning approach, constraining the UML domain for novices, yet easily expanded for more complex projects as the student progresses. Finally, this tool would naturally aid in solution delivery and documentation of the learner's path to solution.

Keywords: programming, novice, difficulties, UML, visualization

Editor: Glenn Lowry

Background

Traditionally, novice programmers have encountered difficulties in learning to program in several different areas, including: the comprehension of fundamental computing concepts, the appropriate decomposition of the problem into easily managed sub-problems, design and implementation of a working solution, and the debugging of the resulting program. Research in the areas psychology of programming, human-computer interaction, cognition, and pedagogy have identified and classified a number of these problems that novices face when learning to program. Bladek and Deek (2005) note that novice difficulties can be adequately classified into the following categories: pedagogical roots, psychological roots, programming language paradigms, programming language intricacies, debugging skills, and external influences.

An ineffective pedagogy in learning programming is one of the most vital factors that will influence how a novice programmer will perform in subsequent experiences. Liffick and Aiken (1996) observe that beginning programmers may stumble in their understanding of a new concept not because it is difficult, but because it is depending on an earlier concept. Another contributor to an ineffective pedagogy can be attributed to the lack of adequate problem solving skills. Suchan and Smith (1997) note that as a result, novice programmers begin to write programs but generate code without any planning or organized thought process. Consequently, Pane and Myers (1996), as well as Olson, Catrambone, and Soloway (1987) collectively observe that general problem solving strategies should be explicitly acquired along with program development skills.

A great number of novice difficulties have been classified as having psychological roots. With computers, the manner in which information is represented and manipulated causes a challenge to its understanding. As a result, novices lack an adequate mental model of the machine's internals and how it operates. In addition, some of the programming concepts, such as recursion, are abstract in nature, having no real-world counterpart for the beginner to refer to.

Novices lack the ability to perform effective sub-goal decomposition. Baile (1991) notes that decomposing while coding is the primary factor that influences a programmer's ability to perform effective sub-goal decomposition. In addition, Pane and Myers (1996), as well as Bonar and Soloway (1985) both observe that when novices lack an appropriate plan, they tend to invent one using pre-programmed knowledge acquired through their real-world experiences. Deek, McHugh, and Hiltz (2000), however, conclude that these plans are often inappropriate because they do not take into account computer limitations.

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. The primary focus of beginning programming is typically on the development of programming solutions to problem statements. Very little attention is given to comprehension skills for programs that have already been written, and more importantly how these programs can possibly be modified and/or applied to other similar problems. Defect discovery strategies, which play a critical role in real-world program maintenance, are almost always ignored.

Along the lines of program comprehension is the difficulties novices face in the debugging of implemented programs. Very rarely does a program work perfectly upon the first execution attempt. Therefore, debugging proves to be an essential skill that a novice programmer must develop. Gugerty and Olson (1986) note that much of the skills for debugging must be learned through the experience of writing programs and since novices lack adequate program comprehension skills, errors are often inadvertently injected into programs while debugging. Consequently, Satratzemi, Dagdilelis, and Evageledis (2001) observe that, due to this lack of understanding, beginning programmers will repeatedly attempt to correct their errors with an understanding of the meaning of the error message produced by the compiler. For the novice, debugging frequently becomes an exercise of trail-and-error until their program works as expected.

Other novice programmer difficulties stem from the intricacies of programming languages and their associated paradigms. There are usually several programming language paradigms in practice at once. Understanding what a paradigm is and how it relates to the real world is frequently a problem for novices. Deciding on what paradigm is suited for the need of novices is typically an on-going debate. Regardless of which language paradigm is chosen, novices continue to struggle with issues involving language syntax; looping constructs; syntactical synonyms, homonyms, and elision; operator precedence; and the semantics and pragmatics of programming language constructs.

Because of the pervasive nature of the novice difficulties presented in this section, much research has been dedicated to the understanding of these difficulties and has led to the development of new methodologies, paradigms, programming languages, and novice learning tools to aid the beginning programmer. Visualization and visual novice tools have been identified as having great potential in helping the novice programmer overcome these difficulties. In the next section, we discuss modern visual learning tools, along with an analysis of how each tool attempts to resolve these difficulties and how effective they are in that task.

Analysis of Modern Visual Tools

Visual learners, as apposed to verbal or textual learners, retain information more efficiently through images than they do from verbal means. Barbe and Milone (1981) observe that as a result, perception of information via diagrams is extremely effective for this type of learner. The importance of visual tools and visualization usage to aid novice programmers is apparent. In this section, we will identify and analyze several modern visual-based learning tools based on how well they address the novice difficulties discussed in the previous section.

Classifications of Visual Learning Tools

For the purposes of this discussion, visual learning tools can be roughly classified into two categories: those that use visualization to display and animate constructs in programs that have already been written, and those that use visualization to aid novices in the program development process. These uses of visualization are mutually exclusive and are on completely different ends of the spectrum in terms of the ways in which they seek to aid the novice developer. Thus, we will discuss them in the context of these two distinct groupings.

Visual tools that utilize visualization to display and animate constructs in programs that have already been developed seek to aid novice developers in the areas of program comprehension, the understanding of abstract computer concepts, and in the debugging portion of the software development process. Tools that fall into this category include EROSI and AnimPascal.

On the other end of the spectrum, visual tools that utilize visualization to aid the novice programmer in actual development of programs seek to aid novices in the areas of abstraction, modeling, problem analysis, and problem solving. Tools that fall into this category include BlueJ, FLINT, BOOST, and SOLVEIT.

EROSI: Explicit Representer of Subprogram Invocations

The *Explicit Representer of Subprogram Invocations* (EROSI) system is an easy-to-use tool that attempts to aid novices via program visualization, particularly in the area of recursion. Recursion is an abstract concept that is very difficult for novices to understand and, as a result, tend to be avoided with the substitution of iterative algorithms. George (2000b) notes that EROSI helps novices obtain a “mental model to facilitate the comprehension and use of recursion as a problem solving technique”.

George (2000a) also notes that the basis for EROSI is the *dynamic-logical* model, in which each recursive invocation suspends the calling program. Control of program execution is transferred to a "new and unique" manifestation of the recursive program. Once subprogram execution is complete, control is transferred back to the calling program. Each invocation results in a new copy of the subprogram, which has its own unique set of variable and parameter values.

Bladek and Deek (2005) observe that at the heart of EROSI is the *Command Module*. The six components that support 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 the 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 in order 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.

Bladek and Deek (2005) continue to note that EROSI's four main menu choices of particular 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 in the learning of recursion: rather than providing random examples, a progressive approach is utilized.

As George (2000b) indicates, one of the benefits of EROSI is 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 all invocations. EROSI accomplishes this through the use of visualization and animation aided with the use of color and sound. The system indicates the execution of a recursive subprogram through *dynamic code visualization*, where the line of source code currently being executed is highlighted. EROSI also utilizes *dynamic algorithm visualization*, which is an animation feature that assists novices in the visualization of the mechanics of the algorithm being executed.

EROSI effectively uses visualization to aid the novice in the comprehension of recursive algorithms. This system helps the novice understand the flow of execution, program control, and data flow through multiple invocations of the same subprogram. Good and Brna (1996) note, however, that learning recursion has two aspects: the declarative aspects of learning "what it is", and the procedural aspects of "how to use it". Good and Brna continue to indicate that the difficulties of recursion are not eliminated once the concept of recursion is understood, as users will still encounter problems in applying it correctly and in the appropriate context.

EROSI effectively addresses the novice difficulty of comprehending abstract computer concepts, with a focus on the concept of recursion. Through its use of visualization, animation, and provided examples, is effective at aiding novices in the comprehension of recursive programs that have already been written. The system, however, lacks in the area of guiding users through how to create a recursive algorithms from scratch, and determining when it is appropriate to apply a recursive routine to a particular problem.

AnimPascal: Animated Pascal

As noted earlier, one of the primary difficulties novices face in learning how to program, is the ability to visualize and create a mental model of the actions involved in program execution. Most modern programming environments lack the ability to effectively animate the actions that the

program is performing. Of those that do provide such a feature, most are crude in nature and difficult to use, and thus prove to be not very helpful. In order to address this issue, Satratzemi et al. (2001) introduced a visual education environment designed to teach novice programmers how to develop, verify, debug, and execute programs. Based on the Pascal language, the two primary features of AnimPascal are: a program animator, and a mechanism that can be used to analyze the various recorded problem-solving paths.

AnimPascal provides the ability to create, edit, and compile standard Pascal programs. It has a graphical user interface that is simple and easy to use. AnimPascal's user interface consists of four distinct windows: *Source Window* is used for input and editing of source code, *Program Output Window* is used for capturing the output of the program, *Display Variable Window* is used for tracking changes to program variables and *Compiler Output Window* is used for displaying message output by the compiler.

At the heart of AnimPascal is its animation tool. As a user 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.

AnimPascal also has a history tracking mechanism, in which all compiled versions of a user's program are recorded. This feature is aimed toward programming instructors, as it is able to give them an insight into common programming misconceptions and frequently encountered programming mistakes. The instructors are then able to classify these misconceptions and mistakes, and change accordingly (i.e. more examples and better explanations) in order to prevent similar occurrences in future course offerings.

AnimPascal attempts to employ the use of visualization and animation to help novice programmers create a mental model of the execution of computer programs. Its "animation", however, is primarily textual based, with moving highlights in source code, which may not be abstract or visually object-oriented enough in nature to be an effective learning aid for most novice programmers. While AnimPascal's history tracking mechanism is interesting, it requires the instructor to take an active role and interest in the analysis of the data and determining the appropriate changes to their curriculum.

Similar to EROSI, AnimPascal focuses on aiding the user comprehend programs that have already been written. The system has no functionality to help the novice programmer effectively design and implement his/her own programs. Because of these considerations, AnimPascal has a very limited ability to help novice programmers overcome many of the difficulties discussed above.

BlueJ: The Interactive Java Environment

BlueJ is an integrated Java environment specifically designed for an introductory level. It was developed as part of a university research project focused on the teaching of object-orientation to novice programmers. Barnes and Kolling (2003) indicate that Special emphasis is placed of visualization and interaction techniques to create a highly interactive environment that encourages experimentation and exploration. BlueJ is based on the Blue system, which is an integrated learning environment, along with its own specialized language. BlueJ provides a Blue-like environment that has been extended specifically for use with the industry-standard Java language.

One of the fundamental characteristics of BlueJ is that not only can a user execute a complete application, but also has the ability to directly interact with single object of any class and executing their public methods. An execution of BlueJ is usually done by creating an instance of a class,

and then invoking one of the resulting object's methods. This facility can be very helpful in the development of an application, as users have the ability to test classes individually as soon as they have been written. This feature of BlueJ attempts to address the novice difficulty of program debugging. By having the ability to test as soon as classes have been written, novices are better able to track down exactly where any errors are occurring.

In the BlueJ's animated environment, the user can create classes and establish relationships between them by using functions available on a toolbar. Once an object has been created it is placed in the *Object Bench* and its public operations can be executed. During object and program execution, BlueJ provides the ability to inspect the internal state of the object. A traditional debugging environment, with the ability to set breakpoints and step sequentially through the code, is also provided. Inspection of variables during object execution is every easy, as they are automatically displayed in the debugger.

BlueJ represents a significant step forward in visual novice instruction tools. Through provided examples, basic animation of source code execution, and the ability to execute and test classes individually, BlueJ has an ability (though limited) to help beginners comprehend object-oriented programs that have already been written within its framework. BlueJ's primary benefit lies in its visualization of class objects and the associations between them. This system, unlike the others examined in this section up to this point, visually focuses on the object modeling rather than actual implementation of code. This encourages exploration and experimentation on the part of the user, which brings them into an active, rather than passive role in their computer programming education. McKeachie (1996) indicates that research has shown that in a more active environment, learners excel in comprehension, memory retention, and problem solving.

On the other hand, BlueJ probably has limited effectiveness as a novice-learning tool. First, BlueJ focuses on the object-oriented paradigm for modeling and design, as it relies on the class diagram as its basis structure. With concepts such as inheritance, abstractions, information hiding, and overloading of operators, the object-oriented paradigm may prove to be too complex to be effectively understood by beginners. Second, while BlueJ effectively utilizes visualization to display object classes and the associations between them, it lacks tools to aid the user in the proper design an implementation of object-oriented elements on their own.

FLINT: Flowchart Interpreter

Ziegler and Crews (1999) indicate that the Flowchart Interpreter (FLINT) system is an instructional programming environment designed specifically for the novice, and seeks to incorporate design, algorithms development, testing, and debugging, into one unified tool. Through its language independent nature, FLINT attempts to draw attention away from issues stemming from language-specific programming syntax. Its creators, Zeigler and Crews, claim that the use of their environment helps the user "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 predefined icons into a program construction area. When program construction is complete, the tool generates the appropriate source pseudo-code in the desired language. Ziegler and Crews (1999) indicate that research has shown that programmers that generate programs with such a tool are able to better comprehend the language syntax than if such a tool had not been utilized. It should be noted that pseudo-code cannot be directly compiled and executed, as it does not meet the requirements of programming language syntax.

The FLINT environment follows a structured approach to program design. It focuses around four stages of program design: 1) Development of a sound design, 2) Development of algorithms

based on the design, 3) Testing, and 4) Debugging. The first stage of the FLINT environment, as noted above, focuses on the development of a sound design. This is facilitated through a step-wise refinement process utilizing a graphical interface. The user constructs a program design in the form of a top-down structure chart consisting of rectangular boxes connected by lines representing a hierarchical representation of the program. The structure chart begins with one box at the top of the window that contains the description of the highest level of abstraction. Rectangular boxes that are children of this first box represent the first level of abstraction, or sub-problems. Each of these sub-problems nodes may contain any number of rectangles beneath it, as needed.

The second stage of FLINT assists the novice programmer in developing algorithms for each of the boxes in the structure chart. FLINT constantly monitors the state of the design throughout algorithm development by performing checks for connectivity between modules in the structure chart. To ensure consistency between the algorithm and the structure chart and to ensure that the user actually implements the design as specified, any algorithm that does not meet this requirement is rejected. Algorithms within FLINT are represented using structured flowcharts. The flowchart builder is designed to be simple to use. It uses conventional flowchart constructs that can be selected and moved using a point-and-click interface.

Testing is the third stage of FLINT. By executing their algorithm within FLINT, users are able to get immediate feedback. The instructor has the ability to customize FLINT's testing stage, so that users are required to perform a certain number of test cases per algorithm. FLINT prompts the user for sample input and the expected output for each test run. For each sample input/expected output pair, the user must confirm that the results provided by FLINT are consistent with the expected output.

The fourth and final stage of FLINT is debugging. FLINT supports debugging by providing users with the capability of inserting breakpoints in their algorithms. Statements and variables are highlighted prior to execution, giving the novice a visual representation of the flow of control and data in the program. The pace of execution can also be controlled to aid novice comprehension by moving through the program at a pace they are comfortable with.

FLINT represents another significant leap forward over the tools discussed in this section up to this point. Through basic visualization and animation features, FLINT has a limited ability to help novice programmers debug and comprehend programs that have already been written. Perhaps the most significant advancement with FLINT over the other tools discussed is its emphasis on structured approach to design in problem solving and program development. With the use of structure charts, FLINT prohibits users from first getting a program to run and then abstracting a design from it. In addition, the user is only permitted to change the algorithm by first changing the design. These are all valuable skills for a novice programmer to develop and utilize as they progress in their computer science education.

While FLINT does an extremely good job in presenting users with a structured framework for problem solving and program development, it should be noted that at the time of the Ziegler and Crews (1999) publication, the FLINT system was still under development. Until the final system is released and tested with novice programmers, it is difficult to assess how truly effective it will be in aiding users with traditional novice difficulties. In addition, while FLINT provides a structured approach for program development from the structure chart on, the task of creating an effective structure chart outlining the decomposition of a program to solve the problem at hand is left predominantly to the user without any aids or guidance. As Rappin (1997) notes, some of the fundamental skills novices lack is the ability to create a decomposition of a problem domain into easily managed pieces, and the ability to recompose those pieces back into a coherent whole. Because FLINT lack support in these areas, the overall effectiveness of this toolset may be limited.

BOOST: Basic Object Oriented Support Tool

Basic Object Oriented Support Tool (BOOST) generates Smalltalk code and supports, but does not enforce the design activities indicated below. Rappin (1997) notes that the creator of BOOST not only allows users to move from one step to another at any time they wish, but encourages them to do so.

The main activities defined in the BOOST process are:

1) Brainstorming – creating a list of classes that may (or may not) be in the novice's design. These classes are called *candidate classes*.
2) Assigning Responsibility – BOOST allows the user to use Class, Responsibility, Collaborator (CRC) cards to list the responsibilities of each class in their design without regard to implementation details.
3) Design – In the design step, the user moves closer to implementation by using the responsibilities to inform the creation of attributes, services, and links.
4) Design Check – The design check feature of BOOST is used to test for some of the common errors in design.

When a user first enters BOOST, they are presented with the main design window. The user will first brainstorm a list of classes that may potentially be part of the design. These candidate classes are either rejected, or accepted to become part of the actual design.

Based on examination of the functionality provided by BOOST, it would probably not be very well suited to novice instruction. Much like BlueJ, BOOST focuses on the object-oriented paradigm for modeling and design, as it relies on class modeling as its basis structure. With concepts such as inheritance, abstractions, information hiding, and overloading of operators, the object-oriented paradigm may prove to be too complex to be effectively learned by a beginner. In addition, BOOST generates Smalltalk code, which would probably not be easily understood by novice programmers.

On the other hand, BOOST does offer some ideas that could be helpful in the instruction of novice programmers. Its brainstorming and activity responsibility stage would probably be easy for novices to use, as they could approach programming by modeling their solutions based on easily understood real-world counterparts. In addition, this stage would encourage users to explore a number of design possibilities. The latter two stages of design and design check would best be left to more advanced users examining object-oriented design principals.

Perhaps the best analysis of BOOST is best taken from a word in its name: *Basic*. BOOST provides very basic, unstructured design functionality that would be better suited toward a more advanced programmer. Overall, it does not provide enough structure and guidance for effective utilization by novice programmers.

SOLVEIT: Specification Oriented Language in Visual Environment for Instruction Translation

Deek along with McHugh (Deek, 1997; Deek & McHugh, 2002a, 2002b) propose the *Specification Oriented Language in Visual Environment for Instruction Translation* (SOLVEIT) system as an integrated environment designed for novice users to learn problem solving within the context of a software engineering framework. SOLVEIT is based on Deek's (1997) *Dual Common Model for Problem Solving and Program Development*. The dual common model is the foundation for SOLVEIT, and represents the integration of a cognitive model for problem solving with the tasks involved in program development. Based on this model, SOLVEIT encompasses six stages: problem formulation, solution planning, solution design, solution translation, solution testing, and so-

lution delivery. Through an integrated environment along with a collection of both general-purpose and stage-specific tools, SOLVEIT seeks to develop the user's problem solving and cognitive skills, while also developing the skills essential to successful program development.

With the SOLVEIT environment, it is the designer's belief that the user can perform the programming in a learning environment that allows them to utilize and develop necessary problem solving skills, such as information gathering, problem formulation and decomposition, and task and data flow organization with the encumbrance of at the outset of programming language dependent issues, such as syntax and intricacies of the development environment.

The first three stages of SOLVEIT are the problem solving stages, and the latter three are the program development stages. Using SOLVEIT, the user will usually progress through these stages sequentially, although the ability is provided jump both forward and backward between stages. The six primary stages of SOLVEIT, along with their associated sub-stages and tools are:

- *Problem Formulation* - users begin by describing the problem, extracting relevant facts from the problem description such as givens, unknowns, constraints, etc.
- *Solution Planning* - users at this stage begin to outline plans for solving the problem. Alternative approaches are considered and evaluated, and a candidate chosen. Goals are decomposed into further subgoals. Data models are drawn.
- *Solution Design* - structured charts are drawn up by the user using SOLVEIT's tools. Modules are created that match to subgoals defined in the previous stage. High-level algorithmic specifications of the modules are created.
- *Solution Translation* - algorithmic specifications produced by the user in the previous stage is used by the user as a basis for code translation to a specific programming language, whichever that may be.
- *Solution Testing* - tools are available to help the user in coming up with black-box and white-box test specifications to test his solution.
- *Solution Delivery* - SOLVEIT's maintains a log of all the user's activities during all stages. The system's document delivery management tool is used at this stage to organize and produce a complete documentation package of the entire user's work in this whole six-stage project.

SOLVEIT offers a highly visual environment to aid novice programmers in the various stages of problem solving and program development, while operating within a software engineering framework. It does not maintain an individualized user model like some other novice learning tools, but a system recorder does maintain a log on a user's progress throughout a problem solving sessions, which allows the replay of user/system interactions.

A primary advantage of SOLVEIT is that a novice needs very few, if any, requisite skills or knowledge in order to effectively use the system. This is due to the fact SOLVEIT focuses on generic problem solving and program development skills, as opposed to a specific programming language. The SOLVEIT environment is broken down into several discrete stages. The process of working through the first stage provides the user with the requisite skill set needed to continue through all the subsequent stages.

SOLVEIT presents the stimulus material to be learned implicitly through its design. Users begin the process of problem solving and program development in stage one, by first describing the problem to be solved. It should be noted that SOLVEIT does not directly tell the user *how* to solve the problem, but rather the user implicitly learns problem solving and program development skills while actually working with SOLVEIT and proceeding through its six stages of problem formulation, planning, designing, translating, testing, and delivery of the solution.

SOLVEIT provides general purpose and stage-specific tools to guide the user through its stages of problem solving and program development. These tools include the project notebook/graphics editor, the strategy discovery tool, the verbalization tool, and the goal decomposition tool. SOLVEIT's tools used to elicit performance from the user to demonstrate their understanding of the problem statement include the information elicitation tool, and the black-box/white-box testing tools. An example of SOLVEIT's information elicitation tool is shown in Figure 1; an example of its goal decomposition tool is shown in Figure 2.

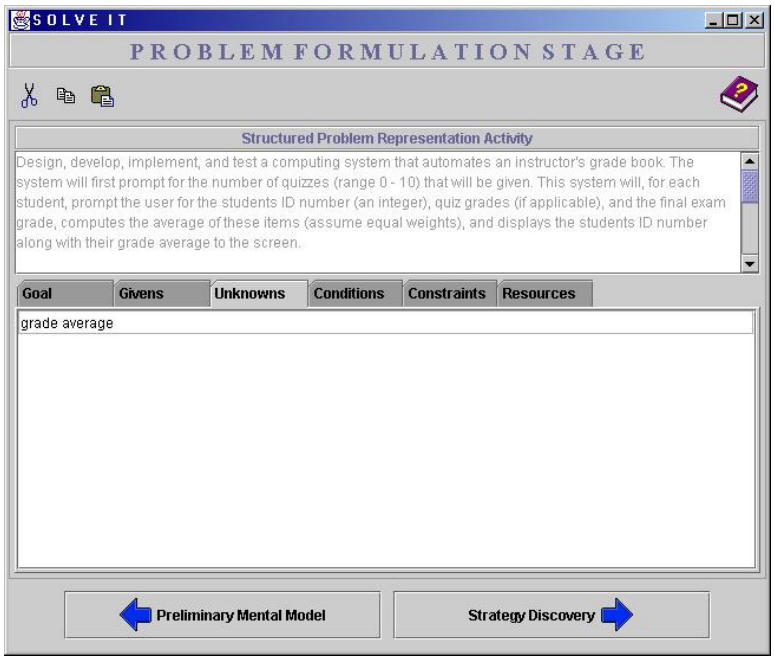

Figure 1: SOLVEIT's Information Elicitation Tool

Figure 2: SOLVEIT's Goal Decomposition Tool

SOLVEIT effectively provides feedback on a user's tasks during a particular stage by organizing the data submitted by the user in one stage and transforming it into relevant information used in subsequent stages. If a user encounters problems answering questions in one of these stages, this could be an indication that the user's answers in a previous section are incorrect and should be revisited.

Out of all the visual tools examined in this section, SOLVEIT appears to provide the most aid to a novice programmer in the areas of problem solving and program development. SOLVEIT offers a very structured approach to problem solving and program development in a visual software engineering framework. Much of SOLVEIT's claim to enhancing retention and transfer in its users lies in the fact that the system focuses on developing users' *skills* by following this structured approach, as opposed to tutorial systems which concentrate on teach users *instructional* content. It effectively uses a visual environment to gain the users attention and to obtain comprehension and understanding of the principals it is trying to teach.

SOLVEIT intentionally uses program-independent tools, since its focus is on software engineering principals, rather than program-specific syntactical issues. As a result, SOLVEIT can be used in conjunction with a variety of programming environments, such as Pascal or C. A by-product of this, however, is that SOLVEIT is unable to aid the user in the actual implementation and debugging of the actual program. It is also up to the user to go back and update SOLVEIT from the beginning when a logical error is found that requires changes to their design. In addition, although SOLVEIT does provide tools that allow users to get a fair grasp of their own understanding, it lacks any definitive support in the area of accessing user performance.

The Unified Modeling Language (UML)

Developing a model for large-scale software systems prior to implementation is as important as having a blueprint for a building before commencing its construction. For novice programmers, however, proper design and modeling prior to coding is often neglected. As the complexity of systems increases, so does the importance of a sound modeling technique. The Object Management Group (1997a, 1997b) proposes that in the implementation of a complex system, there are many factors that affect its overall success, but having a rigorous modeling language is essential.

What is UML?

> The Unified Modeling Language (UML) is a standardized language for specifying, visualizing, constructing, and documenting the intricacies of software systems, as well as for the modeling of business processes and other non-software systems. According to the Object Management Group (1997a, 1997b), UML represents a collection of engineering practices that have been proven successful in the modeling of large and complex software systems. UML uses mostly graphical notations to represent the design of software systems, and as a result, visually aids in the communication of project designs, the exploration of potential designs, and in the validation of the resulting design.

The primary goals in the design of UML, as indicated by the Object Management Group (1997a, 1997b) are:

1) Provide a ready-to-use, expressive visual modeling language to and in the development and sharing of meaningful models.
2) Provide extensibility and specialization mechanisms to extend core concepts.
3) Independence from any particular programming language or development methodology.
4) Provide a formal basis for the understanding of the modeling language.
5) Encourage growth of the Object Oriented methodology.

6) Support high-level development concepts, such as frameworks, collaborations, components, and patterns.
7) Incorporate documented best practices.

Businesses have sought techniques and methodologies to better manage the complexities of large-scale software systems. Additionally, the acceptance of World Wide Web (WWW) as a delivery medium, and the subsequent focus on the development of WWW-based systems, has made some facets of software engineering easier, but has made many others more difficult. The Object Management Group (1997a, 1997b) indicates that UML was designed to respond to these needs, and has been adopted by the industry for modeling object-oriented and component-based systems.

Object-oriented modeling languages began to appear in the mid-1970 to the late 1980s, as various methodologists experimented with different approaches to object-oriented analysis and design. The number of identified modeling languages increased from less than 10 to more than 50 during the period of 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language, thus fueling the "method wars." By the mid-1990s, according to the Object Management Group (1997a, 1997b), new iterations of these methods began to appear, and these methods began to incorporate each other's techniques, and a few clearly prominent methods emerged. The foundations of UML are nearly two decades old and encompass successful modeling attributes of its numerous predecessors.

Modeling with UML

The UML specification defines twelve types of modeling diagrams, divided into three categories: static application structures, dynamic behavior diagrams, and model management and organization diagrams. The UML modeling diagrams are placed into these categories as follows:

1) *Structural Diagrams* – Include the Class Diagram, Object Diagram, Component Diagram, and Deployment Diagram.
2) *Behavior Diagrams* – Include the Use Case Diagram, Sequence Diagram, Activity Diagram, Collaboration Diagram, and State Diagram.
3) *Model Management Diagrams* – Include Packages, Subsystems, and Models.

Each UML diagram is designed to enable system architects and developers view a software system from different perspectives and in various degrees of abstraction. Braun, Silvis, Shapiro, and Versteegh (2003) observe that UML diagrams commonly created and manipulated in visual modeling tools include:

- *Use Case Diagram* displays the relationship among actors.
- *Class Diagram* models class structures and contents using object-oriented components, such as classes, packages, and objects. It also displays the relationships of the object-oriented paradigm, such as encapsulation, inheritance, and associations.
- *Interaction Diagrams* include two types of diagrams:
  - *Sequence Diagrams* display the time sequencing of events between the objects in the model. It consists of time on the vertical axis, and different objects on the horizontal axis.
  - *Collaboration Diagrams* display interactions organized around objects and their links to one another. A number scheme is used to represent the sequencing of messages.
- *State Diagrams* display the sequences of states that an instantiated object can go through in its lifetime in response to external stimuli. The state diagram also shows the object's responses and actions.

- *Activity Diagrams* are a special kind of state diagram. In activity diagrams, most of the states are actions, and most of the transitions between these actions are triggered by completion of the actions in the source state. Activity diagrams resemble flowcharts and tend to focus on the internal processing within an object.
- *Physical Diagrams* also include two types of representations:
  - *Component Diagrams* display the high level packaging of the code itself. Also shown are the dependencies between components, including source code, binary code, and executable components.
  - *Deployment Diagrams* display the configuration of run-time environments, and the executable and code components that run on them. In the client-server world, for example, the DBMS would run on the database server, whereas the client application would run on another PC.

In the context of instructing novice programmers, many of these modeling constructs are advanced, as the beginner will not have had adequate real-world experience to comprehend what the diagrams actually represent. Introduction of advanced modeling tools at this stage would be counter-productive. In order to effectively utilize UML in the instruction of novices, a simplified subset of its modeling capabilities must be carefully chosen. Simpler constructs can be effectively utilized during this time period, whereas other more complex constructs are better left for introduction at a later time. In the next subsection, we will identify which UML diagrams are best suited to novice instruction, and in turn, define each one in detail and explain why they are a good fit.

Defining an Appropriate Subset of UML for Novice Instruction

The strength of UML is derived from its flexibility, robustness, and ability to represent, on varying degrees of abstraction, the intricacies of both computer and non-computer models. In a beginning environment, the introduction of the entire set of UML constructs would quickly overwhelm and frustrate the novice programmer. Thus, a proper subset of UML diagramming functionality that will best aid a novice programmer and leave other more complex constructs for introduction at a later time should be defined. Of the eight UML diagram types provided, the simplest and best suited to novice instruction include: Use Case Diagrams, State Diagrams, and Activity Diagrams. Class Diagrams, Sequence Diagrams, and Collaboration Diagrams all deal with object-oriented modeling and can be deferred. As Rappin (1997) notes:

> The skills involved in designing, evaluating, and building a valid OO model are similar to the modeling skills needed by other engineers in their design process. These skills include:
>
> - The ability to create a decomposition of a problem domain into more easily managed pieces.
> - The ability to recompose those pieces back into a coherent whole.
> - The ability to recognize the connections between the model and the original object.
> - The ability to evaluate the model for the purposes of predicting the behavior of the object.
> - The ability to test the validity of the model and change it as necessary.
>
> The lack of these skills is one of the primary obstacles faced in learning Object-Oriented methods.

As it is unlikely for a beginner to possess these skills, the object-oriented paradigm and its associated models are best left for more advanced experiences. It should be noted, however, that UML is primarily designed as a modeling tool for the object-oriented paradigm. By eliminating UML

diagrams based on the object-oriented paradigm and focusing on imperative models, we are not defeating the basic nature of UML itself, but instead providing a structured approach to the introduction of UML.

As noted above, Activity Diagrams focus on the internal processing within objects, which is inherently imperative in nature. State Diagrams focus on an object's response to external stimuli, which can be viewed as sub-procedure calls in an imperative language. Thus, UML can be introduced simplistically in the form of these diagramming constructs, and later when object-oriented principals and their associated UML diagrams are introduced, the learner will be able to augment their modeling knowledge and experience. Activity and State Diagrams simply become "wrapped" in the Class and associated diagramming constructs. In this structured approach, learners will already be familiarized with the concepts of modeling, and simply expand their knowledge to the object-oriented domain. Similarly, Component Diagrams and Deployment Diagrams both deal with the advanced topics of large system component packaging, distribution, and the running of executable modules. These topics can also be deferred.

The Potential Benefits of UML in Novice Instruction

The adoption of UML in early instruction of programming may also resolve several of the difficulties encountered by novice programmers. Perhaps most importantly, the usage of UML places an emphasis on system modeling early on in a learner's experience. With an emphasis on UML modeling, a programmer is forced to model solutions to problems and adopt an organized approach to the design before coding while implicitly documenting a solution in a visual manner. In addition, learners will have a visual representation of abstract concepts while allowing for a gradual approach to modeling. Simple UML constructs can be introduced early on, and can be built upon as advanced modeling constructs are introduced. In the next section, we will propose a new visual learning tool that has its foundations based in UML. It will attempt to incorporate some of the benefits of UML and also bridge the two classifications of visual tools noted in the earlier review, namely providing both a tool that can aid the novice with designing visual solutions and aid with the understanding of an implemented program with visual depictions at runtime.

A New Visual Learning Tool with Foundations in UML

Previously, the potential benefits of employing UML in novice programmer instruction were identified and discussed. In addition, modern visual learning tools were critiqued and classified into two categories: those that use visualization to aid novice comprehension with programs that have already been written, and those that use visualization to help the novice actually design and implement programs. It was noted, however, that modern visual learning tools fall into either of these mutually exclusive categories; no tool was identified as having facilities to aid the novice programmer in both. In this section, we discuss the high level attributes of a new visual tool that attempts to incorporate the usage of UML and spans both identified categories of modern visual tools to aid the novice programmer. The intent is to provide foundations for a new class of visual tools that with the functionality and aid to the novice programmer.

An Emphasis on Modeling

The learning tool and associated instruction emphasizes modeling early on. As Rappin (1997) notes, the lack of modeling skills is one of the primary obstacles beginners face in learning the Object Oriented methodology. Strong emphasis on the development of effective modeling skills and the adoption of UML to aid in the learning of these skills is a significant step toward the effectiveness of learning programming.

A set of visual modeling constructs

The visual learning tool will rely exclusively on the UML Activity Diagram, which easily allows for the incorporation of more advanced functionality. The tool shall have a visual point click/drag-and-drop environment. The toolbox will contain the UML visual constructs as shown in Figure 3.

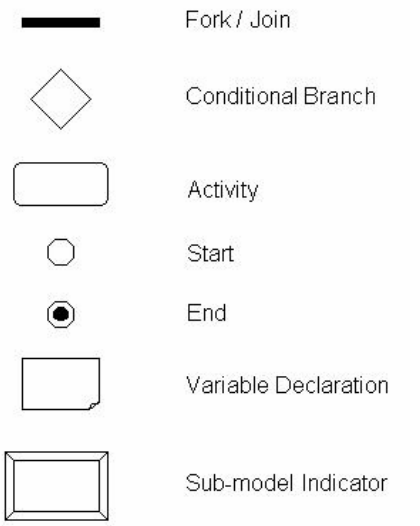

Figure 3: Learner's Toolbox of Available UML Constructs

The user will have the ability to create these constructs in a visual workspace. After clicking on the construct, the user will have the availability to type a descriptive name for it. By clicking and dragging from the edge of one construct to another, the user will be able to create directional arrow links between the constructs, thus indicating the relationship and activity flow between them.

Instruction and intervention

It should be noted that this tool is not intended to act as a modeling and program development tutor. The intervention of an expert, typically the instructor in a traditional classroom, is an important part of this learning environment. The tool is designed to supplement novice instruction of modeling and design concepts. An expert, such as the instructor, can evaluate user's work at predefined intervals over the course of a learning session. As modeling concepts are introduced, Rappin (1997) notes that the skills that need to be refined include the ability to create an appropriate decomposition of the problem into easily managed pieces, the ability to recompose these pieces into the whole, and the ability to recognize connections between the model and the original problem statement. The tool can be used independently to reinforce and apply design concepts as they are being introduced.

The first task of a programming project focuses on the problem understanding and analysis stage of problem solving. In this stage, the tasks of the problem solver include the determination of what the goal of the problem is, identifying the givens provided by the problem statement, and determining what the unknowns are. A frequent problem encountered by novices is the introduction of defects in this stage of problem solving. As Deek (1997) points out, the earlier in the problem solving process that a defect is introduced the more potential it has to cause major problems and the harder it will be to find. Suchan and Smith (1997) also note that a defect of severe consequences may lead an inexperienced problem solver to a sub par 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, but incorrect due to a logical design error. Additionally, Suchan and Smith note that sooner or later, novices realize that analysis defects always carry forward into their implementation.

The deliverable is the Use Case Diagrams. These diagrams can be used to ensure that the learner has fully comprehended the problem statement and that all of the requirements have been identified and understood. The importance of these diagrams lies in the fact that learner's comprehension of the problem statement can be verified. As indicated earlier, an error introduced at this stage will magnify itself later in the program development process. The second deliverable is the UML Activity Diagram, which can be used to validate the learner's modeled solution. The level

of abstraction allowed in the UML Activity Diagram needs to be set based on the evaluation of the learner's comprehension of the modeling paradigm and ability to independently use programming-level constructs. This will be discussed later in this section.

Coding Behind the Model

Much efforts has been devoted to determining the appropriate programming language for novice instruction, the proper paradigm to use in early instruction, and whether the entire language constructs or a limited set of instructions should be available to the learner. Some, such as McIver (2000), argue that powerful languages make writing complex programs easier, but can be counter-productive for novice programmers while others, such as Cooper, Dann, and Pausch (2000), recommend sidestepping the programming language altogether. This work remains outside of this debate and the approach advocated by the design of the tool being discussed acknowledges the importance of programming but places importance on modeling and design without downplaying the role of the programming language.

The learner can access the coding screen by double-clicking on an activity or conditional construct, and enter the code behind it. Syntactical programming constructs can be introduced gradually, dependent on the level of modeling abstraction. Less abstraction will require less coding behind each element, while more abstraction will allow the learner greater control over the implementation of their code. The coding window is graphical, with a toolbox that will allow for the insertion of basic programming constructs, such as looping constructs, and variables. To simplify the coding process, as much information as possible is obtained from the model itself, thus presenting the learner with a template to work with.

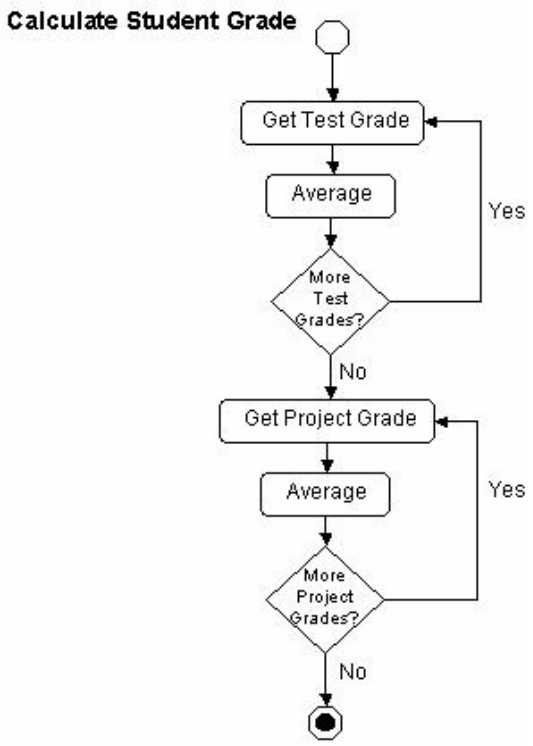

Figure 4: Calculation of a Grade with Little Abstraction

Varying Degrees of Abstraction

As Bucci, Long, and Weide (2000) observe, abstraction hides details by providing a simple "cover story". Early in the learning process, novice programmers should be expected to model even the most simplistic aspects of the solution, as this will give them gain a fundamental understanding of how to properly design and code these elements. Consider the UML activity diagram depicting how to calculate a student's grade in a course, as shown in Figure 4.

The diagram in Figure 4 is very explicit in how to calculate a grade in a course. It is explicit in its methods, even in the depiction of the looping constructs used in averaging a test and project scores. This level of detail leaves very little flexibility to the programmer in the coding. Most of the code behind each activity or conditional construct will only be a line or two. This level of abstraction, used initially with a novice programmer, can effectively and correctly aid in learning basic programming concepts. As novice programmers advance, however, they can be given more control over

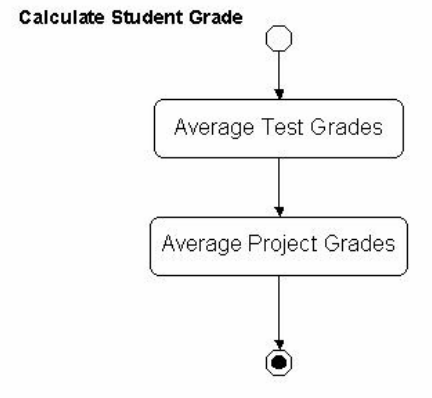

Figure 5: Calculation of a Grade with Greater Abstraction

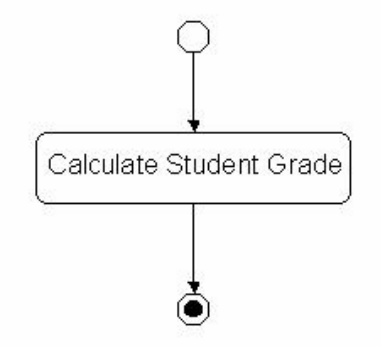

Figure 6: Calculation of a Grade with Full Abstraction

the coding of their projects. Consider the following model for the calculation of a grade in a course in Figure 5.

The diagram in Figure 5 provides the same functionality as the diagram in Figure 4, only with a greater level of abstraction. The looping constructs are no longer shown. The programmer will need to code these loops behind each activity element in the diagram independently. Finally, for the advanced novice programmer, the calculation of the student's grade can be fully abstracted. Consider the diagram shown in Figure 6.

In Figure 6, the details of the calculation of the grade are left entirely to the programmer. Thus, the programmer will have full control over how this process is implemented. The code behind this activity will simply have a blank screen. It is left up to the programmer to implement the loops and know that both test scores and project scores must be averaged. This activity diagram will normally not be in a diagram on its own, as shown above. Instead, it will be part of a larger activity diagram. Although the details of how a grade is calculated are not shown, the importance of its depiction in the model is the recognition on the part of the novice developer that this activity must be performed.

Visualization of Program Flow, State, and Execution

Serious problem that plagues many programmers is that learning to write, test, and debug programs requires an understanding of why and how the program (computer) solves the problem. As Cooper et al. (2000) observe, many programmers are unable to visualize the steps of the execution of the program, and as a result, are unable to figure out what went wrong when things do not work. Additionally, Cooper et al. note that the area in which a novice's program comprehension problem normally lies is in the understanding of program state:

> In imperative languages, a trace of the program with memory snapshots can be used in an effort to assist learners in figuring out what is going on. However, using traces may actually add to some learners' confusion! We believe the source of confusion in figuring out what went wrong, in all but the most trivial code, is an inadequate understanding of the program's state.

Visualization is one approach to assisting the learner in comprehending what task that each piece of a program can be expected to perform, and how the pieces work together to perform the overall task of solving the problem at hand. Along similar lines, animation of program execution can be used to help the learner in this task of "putting the pieces together".

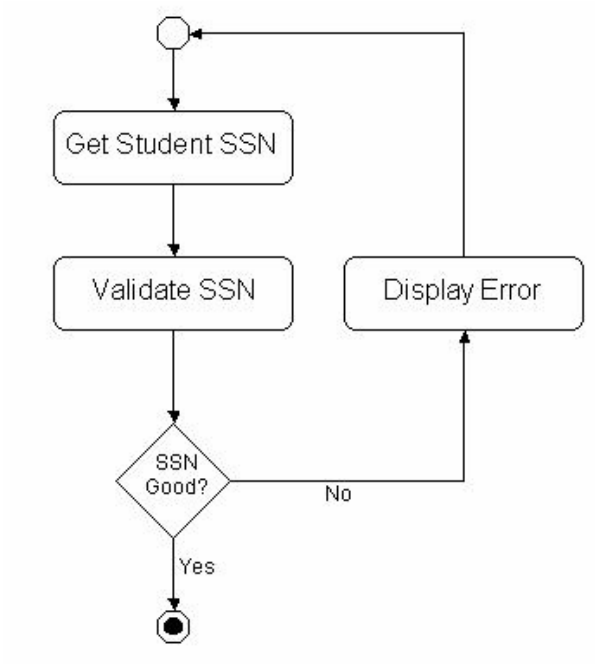

Figure 7: Activity Model for the Validation of a SSN

Naps (1996), as well as Stasko, Dominque, Brown, and Price (1998), indicate that the use of animation is not a new idea. Among the numerous research efforts arguing for visualization, Shu (1988) presents a particularly strong case by considering programming to require both parts of the brain, and focuses on the need to involve the artistic half, which can be satisfied by the involvement of pictures in the process. Many of the systems created for visual programming, however, are fairly complex and difficult for beginners to work with. In addition, they tend to rely on an underlying programming language, which the learners needed to master.

With a UML-based learning tool, the animation of the programmer's design is relatively straightforward. The UML activity diagram is a visual construct by its underlying nature, but the model can also be expressed in textual format and can be easily translated and understood by the computer. The animation of the model can be performed as it executes, thus enabling the novice to better understand the model and aid in the debugging process by helping the learner understand where potential logical problems are located. Consider a simple system that gets a Social Security Number (SSN) and validates its format. An activity diagram modeling this system may look like that shown in Figure 7.

After the model is implemented and the activity and conditional constructs have been coded, it can be executed interactively. An execution of this model may look like that shown in Figure 8.

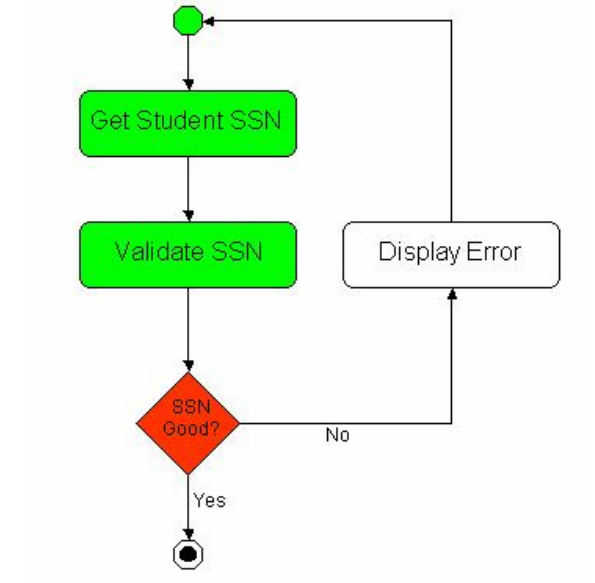

Figure 8: Model Execution for the Validation of a SSN

In the model shown in Figure 8, the green boxes indicate activities or conditions that have already been successfully completed, while the red box shows the activity or condition that is currently executing. A yellow coloring would show program execution errors. In this diagram, it can be see that the "Get SSN" and "Validate SSN" activities have already been completed. The execution of the model is currently on the "SSN Good?" conditional construct. As an activity diagram may be invoked as a sub-model from another diagram, the system must have the ability to organize various invocations of the model. This is

accomplished through the activity table, shown in Table 1:

Table 1: Example of an Activity Table During Execution of a Model

Instance ID	Calling Model	Input	Return
1	abc	n/a	n/a
2	xyz	n/a	n/a
3	xyz	n/a	n/a

In Table 1, it can be seen that this model has been invoked three times, once from model "abc", and twice from model "xyz", with no input or return parameters. Clicking on the row for instances 1, 2, or 3 will display the model and its current state, as shown in Figure 8.

The ability for learners to execute a model that they have designed, implemented, and coded is very important; it allows the visualization of how the computer executes the model. Visualization of program and model execution can also aid novice programmers in the debugging process.

Visualization in Model Debugging

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. Programming errors that must be debugged normally fall into three main categories: syntactical errors, semantic errors, and logical errors. The compiler or interpreter usually catches syntactical errors. A visual UML model, however, can aid in the identification of semantic and logical errors.

Semantic errors, such as memory access violations, unsafe data structure usage, or any condition that causes program execution to abort, can be flagged in the model by changing the activity box that the error occurred in to another color, such as yellow. This will help the novice narrow down where the error occurred. Logical errors, on the other hand, can be easily identified by looking for abnormalities in the model state progression during the execution phase. The programmer can then alter the model to fix the errors and easily re-execute the model to see if the problem has been fixed.

Satrazemi et al. (2001) observe that during the debugging process, programmers repeatedly attempt to correct their mistakes without understanding the error or the meaning of the message produced by the compiler. Visualization of the execution of a UML model can be a powerful in aiding the programmer to understand why a problem occurred. Attempts to fix the error will have the potential to be more focused and thought out, rather than random guesses.

Comprehension of Abstract Concepts

Novice programmers have traditionally had many problems in the understanding and comprehension of abstract computer concepts, particularly those that do not have a well-defined counterpart in the real world. In order to gain skills and master abstract concepts, learners need the ability to generate mental models. As Deek and Espinosa (2005) note, merely saturating a learner with information will not achieve learning, but instead learners will simply continue to memorize and regurgitate information and not gain any fundamental skills in the process. This tool can help facilitate the learner's transformation of abstract concepts into understood mental models through its fundamental use of visualization.

Novice comprehension of abstract computer concepts can be aided in one of two ways. First, the concept can be modeled, and the associated code written behind it. The learner can then execute the model, and through the observation of the progression of the model state, obtain a good men-

tal model of how the abstract concept actually works. Second, the learner can be provided the model in the tool and asked to write the code behind it. While this may require more work on the part of the learner, the level of comprehension should be greater than with the former. One of the areas that a UML-based novice tool can be utilized to aid the novice programmer is in the area of recursion.

Example of recursion as an abstract concept

Recursion has traditionally been a very troublesome area for novices. Pane and Myers (1996) observe that beginners will normally resort to using iteration as an alternative to recursion. 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, 2000b). This is the primary area in which the usage of a graphical UML-based tool can help. Consider the recursive procedure for calculating the factorial of a number. This can be represented in a UML activity diagram as shown in Figure 9.

In the simple UML model shown in Figure 9 for the calculation of recursively, there is only conditional construct and two activities. The area of importance is in the activity on the left. As it can be seen, it contains an invocation of a sub-model, which happens to be the Factorial sub-model itself. Watching this diagram change state as it executes will not be enough to facilitate learner comprehension of recursion, but does become effective when considered with the model's activity table, an example of which is shown in Table 2:

Table 2: Example of an Activity Table During Execution of the Factorial Model

Instance ID	Calling Model	Input	Return
1	n/a	3	6
2	Factorial	2	2
3	Factorial	1	1
4	Factorial	0	1

With this activity table, the learner can watch model state changes among multiple invocations of the model, along with the input and return values from each one. This interactive, graphical dem-

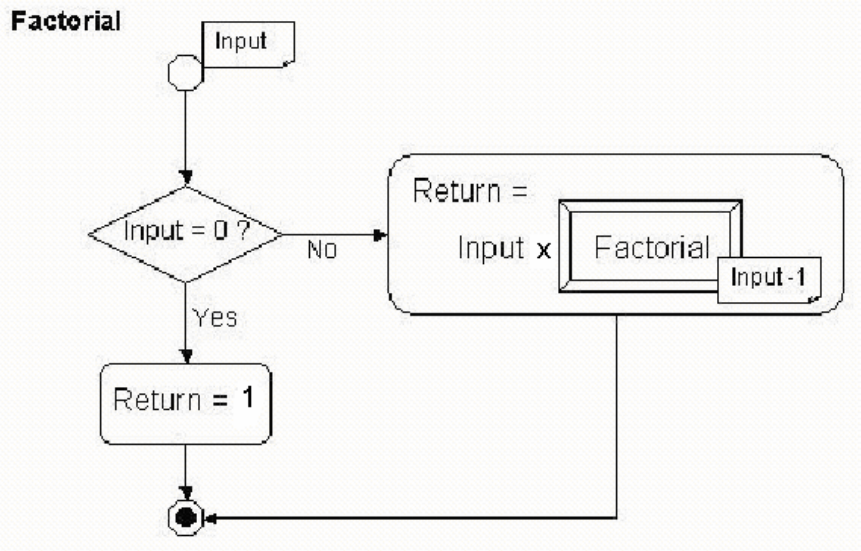

Figure 9: Activity Diagram of Recursive Factorial Model

onstration of simple recursion should give the learner a firm grasps on this abstract concept. Representations of recursion should not be confined to merely and illustrative role. As Good and Brna (1996) indicate, benefit may be derived if novices are involved in, at the very least, the interpretation of representations, and more likely, in their construction. Therefore, a structured approach should be taken when teaching recursion by giving the learner a completed model to watch execute; also by giving the learner the model, but having to write the code behind it; and finally, by giving the learner the task of developing the model and writing the code behind it.

The usage of visualization in this learning tool is not enough, however, to fully teach the concept of recursion. As Good and Brna (1996) point out, there are two fundamental problems areas in the learning of recursion: the declarative aspect of learning "what is", and the procedural aspect of knowing "when to use". It should be noted that usage of a visual UML-based tool such as the one discussed here can help with the declarative aspect of learning recursion, but only fully comprehension, experience, and application of the principal can adequately teach a programmer the procedural aspects of recursion.

A Constructivist Learning Theory Approach

Bruner (1966) observed that learning is an active process in which learners construct ideas or concepts based on the current/past knowledge. It is within this theoretical framework that the concept of this proposed UML-based visual tool to aid novice programmers was developed.

To begin with is the emphasis on modeling. While the entire UML specification would be too complex to introduce to a novice programmer, we defined a subset of the UML toolset that a novice could easily learn and apply to simple programming projects. As Rappin (1997) indicates, this modeling experience will easily allow to add UML modeling constructs, such as Class, Sequence, and Collaboration diagrams, to enhance the modeling skills of learners in a gradual approach. Further, we incorporated the quantification of how much abstraction the novice learner is allowed in their models. Instruction begins with little abstraction, so as to introduce the learner to basic programming constructs. As the learner progress, the level of abstraction is increased, leaving these details to the learner. The learner's models will then focus on more high-level activities. Third, we identified the use of an imperative language, such as Pascal or C, for the coding that needs to take place behind the constructs of the UML model.

Conclusions

In this paper, we first summarized the traditional difficulties faced by novice programmers. Next, we analyzed modern visual learning tools designed to mitigate some of the identified difficulties though the use of visualization and animation. The Unified Modeling Language (UML) was then discussed within the context of how it may be utilized in a new tool to effectively aid the novice programmer. Finally, we discussed the theoretical foundations of a new UML-based tool that combines modeling, software engineering principals, and visualization, into a unified environment to help novice programmers.

Visual tools have proven to be extremely powerful in helping novices in learning abstract computer concepts, such as recursion. Visualization also helps novices construct a mental model of concepts, which is pivotal to further comprehension and understanding. The tool discussed aids the novice in visualizing a model of their proposed solutions and also helps the novice visualize how the model behaves as the computer executes it by watching its state change during execution. The tool also emphasizes modeling and software engineering principals from the very beginning.

In addition to these visualization benefits, this tool would aid in solution delivery and documentation of the learner's path to solution. As the solution would be based on the constructed UML model, it would be mostly self-documenting. Being able to execute the UML model would also

enable the tool to handle test plans, so that the learner can compare the results of the solution to a set of expected outcomes.

The discussions on the UML-based visual modeling tool presented in this paper focused only a conceptual design. Work is needed to further define the toolset, and to specify how the UML model will be linked to the code behind it. Work is also needed to define the constraints and parameters of the framework. Although much work is still need to utilizing a functioning example of the proposed UML-based tool, novice programmers would strongly benefit from its emphasis on design and modeling, as well as its visualization capabilities.

Acknowledgements

The authors wish to acknowledge the efforts of Yashvind Bhasin, NJIT PhD student, in editing the manuscript.

References

Bailie, F. (1991, March). Improving the modularization ability of novice programmers. *The Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education*, *23*(1), 277-282.

Barbe, W. B. & Milone, M. N. (1981, February). What we know about modality strengths. *Educational Leadership*, *38*, 378-380.

Barnes, D. & Kolling, M. (2003). *Objects first with Java: A practical introduction to using BlueJ*. Harlow, England: Prentice Hall/Pearson Education.

Bladek, C. & Deek, F. P. (2005). Understanding novice programmers difficulties as a requirement to specifying effective learning environments. In R. Nata (Ed.), *New directions in higher education*. Nova Science Publishing.

Bonar, J. & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions. *Human-Computer Interaction, 1*(2), 133-161.

Braun, D., Silvis, J., Shapiro, A., & Versteegh, J. (2003). *UML tutorial*. Kennesaw State University Object Oriented Analysis and Design Team. Retrieved April 2003 from: http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/index.htm

Bruner, J. (1966). *Toward a theory of education.* Cambridge, MA: Harvard University Press.

Bucci, P., Long, T. J., & Weide, B. W. (2001, February). Do we really teach abstraction? *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education*, *33*(1), 26-30.

Cooper, S., Dann, W., & Pausch, R. (2000). ALICE: A 3-D tool for introductory programming concepts. *Proceedings of 5th Annual CCSC Northeastern Conference*.

Deek, F. P. (1997). *An integrated environment for problem solving and problem development*. Unpublished PhD Dissertation, New Jersey Institute of Technology, Newark, New Jersey.

Deek, F. P. & Espinosa, I. (2005). An evolving approach to learning problem solving and program development: The distributed learning model. *International Journal on E-Learning*, *4*(4), 409-426.

Deek, F. P. & McHugh, J., (2002a). An empirical evaluation of specification oriented language in visual environment for instruction translation (SOLVEIT): A problem-solving and program development environment. *Journal of Interactive Learning Research*, *13(*4), 339-373.

Deek, F. P. & McHugh, J., (2002b). SOLVEIT: An experimental environment for problem solving and program development. *Journal of Applied Systems Studies, 2*(2), 376-396.

Deek, F. P., McHugh, J., & Hiltz, S. R. (2000). Methodology and technology for learning programming. *Journal of Systems and Information Technology*, *4*(1), 25-37.

George, C. (2000a, March). EROSI – Visualizing recursion and discovering new errors. *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education*, *32*(1), 305-309.

George, C. (2000b, April). Experiences with novices: The importance of graphical representations in supporting mental models. *12th Annual Workshop of the Psychology of Programming Interest Group*, 33-44.

Good, J. & Brna, P. (1996). Novice difficulties with recursion: Do graphical representations hold the solution? *Proceedings of the European Conference on Artificial Intelligence in Education*, Lisbon, Portugal, September 30 - October 2, pp. 364-371.

Gugerty, L. & Olson, G. (1986, April). Debugging by skilled and novice programmers. *Proceedings ACM SIGCHI on Human Factors in Computing Systems*, *17*(4), 171-174.

Liffick, B. & Aiken, R. (1996, June). A novice programmer's support environment. *ACM SIGCSE Bulletin, Proceedings of the Conference on Integrating Technology into Computer Science Education*, *28*(SI), 49-51.

McIver, L. (2000, April). The effect of programming language error rates of novice programmers. *12th Annual Workshop of the Psychology of Programming Interest Group*, pp. 181-192.

McKeachie, W. J. (1996). *Teaching tips: A guidebook for the beginning college teacher* (8th edition). Lexington, DC; Heath, MA: Heath and Company.

Naps, T. L. (1996, June). An overview of visualization: Its use and design. *Proceedings of the Conference on Integrating Technology into Computer Science Education*. Barcelona, Spain, pp. 192-200.

Object Management Group (1997a). *Introduction to OMG's Unified Modeling Language (UML).* Retrieved April 2003 from: http://www.omg.org/gettingstarted/what_is_uml.htm

Object Management Group (1997b). *What is OMG-UML and Why Is It Important?* (UML Primer). Retrieved April 2003 from: http://www.omg.org/news/pr97/umlprimer.html

Olson, G. M., Catrambone, R., & Soloway, E. (1987). Programming and algebra word problems: A failure to transfer. In G. M. Olson, S. Shepard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop* (pp. 1-13). Norwood, NJ: Ablex Publishing.

Pane, J. & Myers, B. (August 1996). Usability issues in the design of novice programming systems. *School of Computer Science Technical Report CMU-CS-96-132*, Carnegie Mellon University, Pittsburgh, PA.

Ramalingam, V. & Wiedenbeck, S. (1997). An empirical study of novice program comprehension in the imperative and object-oriented styles. *Proceedings of the Empirical Studies of Programmers Workshop*, ACM, pp 124-139.

Rappin, N., (1997). On the design of a modeling tool for students of object oriented programming. *Empirical Studies of Programmers Workshop*.

Satratzemi, M., Dagdilelis, V., & Evageledis, G. (2001, June). 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.

Shu, N. C. (1988). *Visual programming*. New York: Van Nostrand Reinhold.

Stasko, J. T., Dominque, J., Brown, M., & Price, B. (1998). *Software visualization - Programming as a multimedia experience*. Cambridge, MA: MIT Press.

Suchan, W. & Smith, T. (1997, November). 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*.

Ziegler, U. & Crews, T. (1999, March). An integrated program development tool for teaching and learning how to program. *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education*.

Biographies

Brian D. Moor is currently a Special Lecturer of Information Technology at New Jersey Institute of Technology. He earned a Bachelor of Science in Computer Science from NJIT in 1996, followed by a Master of Science in Computer Science from NJIT in 2003. Brian plans on pursuing a PhD in Information Systems, also from NJIT, with a specialization in the psychology of programming and the development of new learning tools and instructional techniques for novice programmers.

Brian also has extensive industry experience in the areas of object oriented design and programming, relational databases, graphical user interfaces, enterprise application integration, and has mastered numerous programming languages. He has successfully completed major project implementations for leading companies such as Nabisco, Kraft Foods, Philip Morris, Computer Sciences Corporation, and the United States Army.

Fadi P. Deek received his B.S. Computer Science, 1985; M.S. Computer Science, 1986; and Ph.D. Computer and Information Science, 1997 all from New Jersey Institute of Technology (NJIT). He is Dean of the College of Science and Liberal Arts, and Professor of Information Systems and Mathematical Sciences at NJIT where he began his teaching career as a Teaching Assistant in 1985. He is also a member of the Graduate Faculty - Rutgers University PhD Program in Management. Dr. Deek maintains an active funded-research program. His research interests include Learning/Collaborative Systems, Software Engineering, Programming Environments, and Computer Science Education. Dr. Deek has published over 100 papers in journals and conference proceedings and he has given over 40 invited and professional presentations. He is also the author of eight book chapters and the co-author (with J. McHugh) of the book *Computer-Supported Collaboration with Applications to Software Development* (Kluwer Academic Publishers, 2003, 265 pages) and the co-author (with J. McHugh and O. Eljabiri) of the book *Strategic Software Engineering – An Interdisciplinary Approach* (Taylor & Francis Group - Auerbach Publications, 2005, 360 pages). Dr. Deek has received numerous teaching, research and service awards: The NJIT Student Senate Faculty of the Year Award, given to him in 1992 and 1993; the NJIT Honors Program Outstanding Teacher Award in 1992; the NJIT Excellence in Teaching Award in 1990 and 1999; the NJIT Master Teacher Designation in 2001 and the NJIT Robert W. Van Houten Award for Teaching Excellence in 2002. He has also been awarded the NJIT Overseers Public and Institute Service Award in 1997 and the IBM Faculty Award in 2002.