A Conceptual Model for Learning to Program in Introductory Programming Courses

Azad Ali
Indiana University of Pennsylvania, Indiana, Pa, USA

azad.ali@iup.edu

Abstract

The purpose of this paper is to develop a conceptual model for learning to program in entry level programming courses. The intended model is aimed at providing a framework to simplify learning to program at beginner or entry level programming courses. Learning to program is considered to be a difficult task to many students and it has been attributed to the continuous decline in enrollment in technology programs lately. The conceptual model being worked on in this paper is developed along another model widely used in program and systems development known as the Systems Development Life Cycle or SDLC. The SDLC model is used widely in academia to teach the planning of programs and systems. Hence, the SDLC model is used in this paper to guide in the creation of a new conceptual model that is aimed to provide a framework to simplify learning to program.

Keywords: Introduction to Programming, Programming Conceptual Model, difficulty learning programming, entry level programming, beginner programming courses.

Introduction

Teaching a first programming course has been the subject of numerous studies (Ali & Mensch, 2008; Carter & Jenkins,2002; Kelleher & Pausch,2005). Many confirm the notion that learning to program is considered to be a difficult task to the majority of students and has been a prime reason for students' drop out from computer courses (Anewalt, 2008; Porter & Calder, 2004). A study estimated that between 25 to 80 percent of students drop their first computer classes due to the difficulty they face in learning to program (Carter & Jenkins, 2002).

To computer educators, this issue of difficulty in first programming course has been subject of debate and there is a wide acknowledgement among computer academics that something needs be done to counter this issue. Many programs have taken active steps to offset these difficulty points (Marreno & Settle, 2005). Some changed the language that is typically taught in a first programming course, used a different textbook and/or took additional measures. But the problem of difficulty in learning a computer program persisted nevertheless.

To non-computer majors, this problem has caused additional steps. The basic service course in computer technology used to include programming. This basic service course has been replaced most often by a computer literacy course teaching productivity software. While there is an extended debate about whether to include programming topics

in a first technology service course, there is still a need to counter the difficulty issue with learning to program (Hartman, Nievergelt, & Reichert, 2001).

This paper adds to the discussion regarding steps to take to simplify the learning of programming languages in first programming courses. It takes previous studies a step further: It provides a conceptual model to simplify learning to program. This intended model of program learning is developed here along a widely known model used for planning systems and programs. The Systems Development Life Cycle (SDLC) is used in academia to teach program and system development planning. This paper uses the SDLC as a guide to develop a new model. The paper starts by describing the SDLC model and follows it by explaining the overall framework of this study and the steps followed to develop the intended model.

About the Systems Development Life Cycle

Programmers and systems developers often plan their programs and systems using a specified sequence of steps referred to as a Program Development (Schneider, 1999), Systems Development Cycle (Shelly, Cashman, & Vermatt, 2003) or Systems Development Life Cycle (Maraks, 2001). It is a phased approached aimed at solving computer problems, for examining an information systems and/or improving.

In academia, the SDLC is widely used in systems analysis courses as well as entry level programming courses in specific or any other programming courses in general (Vandever, 2008). In systems analysis course, the emphasis for covering the SDLC is on the larger picture of developing systems and the repetitive nature of the same tasks, thus used the word “cycle” in this kind of systems. In programming courses, the use of the SDLC is intended to give a glimpse of the steps that are followed to develop the program as well as give a conceptual understanding of the flow of the program prior to coding it.

Using the SDLC model to understand the program is more often beneficial to beginner programming students. The common notion that is often followed by beginner programming students is to jump into coding without having a conceptual understanding of the program or the problem they are trying to resolve. Following a planning model (such as the SDLC model) may help to shed some light on the concepts covered and the methodology followed to solve the problem in order to give some understanding of the program prior to coding it.

Analysts in the systems development field do not totally agree on the steps that need to be included in the model. However, the same analysts agree that the SDLC guides programmers through the steps of analysis, design, development, and implementation of a program or a system. Figure 1 shows the phases that are commonly agreed on regarding the steps in the SDLC model.

Although Figure 1 shows six phases in the SDLC model, some of the phases can be combined into more general classifications. Thus, the steps of the original SDLC model can then be modified into a smaller model. In general, the phases of the SDLC model are combined here into three broad phases: planning phase, coding phase and testing and output phase. Figure 2 shows the modified version of the SDLC model.

Framework for This Study

This paper is developing a model that is intended to provide a framework to simplify learning to program. The model being worked here is going to be developed along the steps that are outlined in the modified SDLC model that is listed in Figure 2.

The intended model is going to list factors for addressing the difficulty points of learning to program from three different perspectives: first it addresses the factors that make learning to program

a difficult task. Second, it lists the steps that have been taken to simplify learning to program. These steps and factors are related to technological changes as well as methods in approaching the teaching of programming. Third, the model is going to list additional steps that are aimed to further simplify learning to program. These additional steps may be practiced in other courses, may require modifications in pedagogy or additional related technological changes.

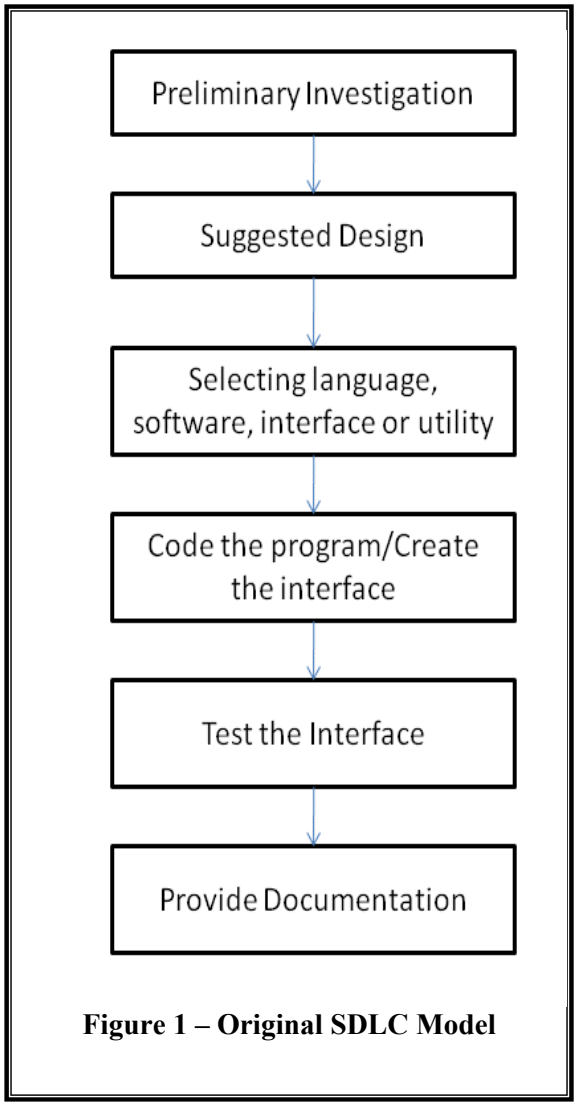

Figure 1 – Original SDLC Model

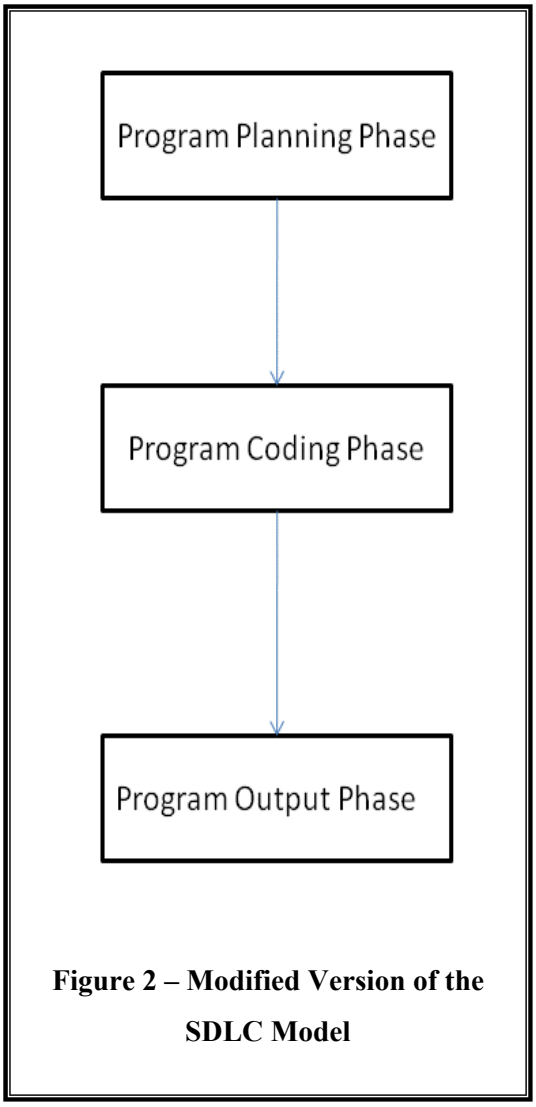

Figure 2 – Modified Version of the SDLC Model

The three steps that are outlined in the modified SDLC model in Figure 2 as well the three perspectives that are listed in the previous paragraph are combined into one model that this paper intends to develop. Table 1 shows the beginning of the model that this paper intends to complete.

Table 1 – The beginning of a model for learning to program

	DIFFICULTY REASONS	STEPS TAKEN TO SIMPLIFY	ADDITIONAL STEPS
Planning Phase			
Coding Phase			
Output Phase			

The next three sections of this paper are going to complete this model. Each section is aligned with one of the columns listed in Table 1. The first section describes the factors that make learning to program a difficult task. The next section explains about the changes that were made to address the difficulty points that are associated with program learning. The third section that follows lists additional steps that are suggested to further simplify learning to program. The last section combines information from all three sections presented and creates the intended model.

Learning to Program – Difficulty Points

This section explains the factors that make learning to program a difficult task. It discusses these difficulty points in three phases that are commonly used to develop a program: Planning phase, Coding phase and then output phase. The section first makes general review to the factors of programming difficulty and then delves into the specific difficulty points in each of the three phases outlined above.

Numerous studies have been conducted to identify the factors that contribute to the difficulty in learning to program. Baldwin and Kulijas (2001) for example noted about the difficulty that students face when learning to program "The majority of students, even those enrolled in computer science courses, find computer programming a difficult and complex cognitive task" and further explained that "learning programming demands complex cognitive skills such as planning, reasoning and problem-solving" (p. 1).

Other studies provided more comprehensive analysis of the factors that contribute to this difficulty. Dann, Cooper, and Pausch (2006) listed four factors that contribute to the difficulty in learning to program: Fragile mechanics of program creation, particularly syntax; the inability to see the result of computation as the program runs, the lack of motivation for programming and the difficulty of understanding compound logic and learning design techniques.

In a study conducted to suggest steps to simplify learning to program, Kelleher and Pausch (2005) compared a number of programming languages that are commonly used in beginner programming courses. The same study wrote the following about the difficulty of learning to program:

> "Learning to program can be very difficult for beginners of all ages. In addition to the challenges of learning to form structured solutions to problems and understanding how programs are executed, beginning programmers also have to learn a rigid syntax and rigid commands that may have seemingly arbitrary or perhaps confusing names. Tackling all of these challenges simultaneously can be overwhelming and often discouraging for beginning programmers" (p. 83).

The studies that were noted earlier point to one common fact and that is learning to program for a beginner student is considered to be a difficult task. However, the factors that contribute to these difficulties are not totally agreed upon. The remainder of this section explains further the difficulty factors of learning to program in three phases: Planning phase, Coding phase and Output phase.

Program Planning Phase

This is the phase of developing programs. The goal of this step is to develop an understanding of the concepts, the structure and the overall flow of the program. In other words, a conceptual understanding of the program is sought from this step to ease the stages that are followed at later phases. One of the major difficulties of this phase is the unfamiliar structure for computer programs. The structure that is followed in writing computer programs is dissimilar to other tasks taken by students from non-technology majors.

In computer programming, the term “structure” is repeated often and is practiced differently when writing programs. Actually, the word structure is considered the foundation in three different procedures in writing different programs. These three different procedures for controlling the flow of code are termed the three “programming control structures” which are: sequence control structure, selection control structure and iteration control structure. The level of “structure” is practiced differently in each of the three control structures that are mentioned above, thus an explanation of each control structure is warranted.

The sequence control structure is where the commands are executed one line after another and is the simplest control structure to follow. Programmers can follow the code by reading one line after another. Although the commands may sound mysterious and the variables may not be clear, the lines can be read similar to reading a book. Familiarity with the syntax may help understand the program, but in either case it follows a pattern that is familiar to most people. Due to this familiarity, this control structure may be more understandable than the other two.

The selection control structure is included when the program executes certain blocks of code based on different conditions. The blocks of code may be within the same program file, or it can be written in a totally different program. During the execution of the program, branching out of sequence may lead to another location in the program and then the program may encounter another selection statement that branches out to another location. This kind of branching out may continue at several levels and it may not be clear at what point the program goes back to the original code that it branched out from. Two difficult points arises here from this kind of branching out. First, identifying the block of code that is executed as a result of this branching is difficult. Second, identifying the multiple level of block that are executed and then to return back to the original line of execution is tricky. Both of these difficulty points confuse novice programmers and lead to frustration as the program gets longer.

The iteration, or loop control structure, faces similar problems but at a different level. In this structure, programming code is executed specific number of times or until a specific condition is met to halt the execution of the loop. The difficulty here is similar to the selection statement in that it is not clear which statements are executed within the loop. Also, the idea of repetition is foreign to many users in thinking of a loop until a condition is met.

Initial teaching programs in BASIC programming language did not follow a particular structure. Programs were written in one block of code and they were not broken down into many distinct blocks. Beginners did not have to remember different module names, nor how to pass variables, parameters and other related names. This practice was simple enough to teach to students who are not familiar with programming. However, as programs became longer and as some of the tasks were repeated from one program to another, the need to structure the solution increased. This structured solution is achieved with two purposes in mind: first to break down the program into smaller modules to make them easy to read and follow (Schneider, 1999). The second purpose is to increase the usability of the code. In other words, a module written for one purpose need not be repeated in other programs over and over again. Instead, the module is written independently and other programs use the same module in their programs. However, this multiple level of branching out and calling other programs is confusing.

This call for structure is designed to make program development more efficient and aimed at standardizing the coding of programs and reusing existing code of the programs. However, this kind of structure is difficult for inexperienced programmers to understand. Later development in the programming industry introduced the use of Object Oriented Programming (OOP) methodology which stresses more of the usability and the structure issue in the program. The OOP methodology introduced many new concepts that needed to be understood along with learning the programming concepts. (Dann et al., 2006) noted that today’s beginning programmers have to learn

the original concepts of programming such as loop, selection, and iteration along with the new concepts of OOP such as classes, objects, encapsulation, inheritance and others. Thus, it adds additional steps in learning to program.

Program Coding Phase

During this phase, the student enters the program to the computer according to a specified set of rules and procedures termed as the "syntax" of the program.

Syntax is "the grammatical rule of the programming language" or so explained in typical programming courses. However, a closer look at the rules of syntax in programming languages reveals many differences from the grammatical rules of typical spoken languages. These differences have to do with the structure of commands, the stopping character, naming of variables, passing parameters and other related issues when entering lines of code in programs.

A computer program is written to execute a command or a series of commands (Porter & Calder, 2004). A program contains a series of instructions that use a set of variables to perform specific tasks. The program is usually typed in a text editor and saved on a storage media. It is then compiled to check for the correctness of the syntax. If there is an error in the program, the compiler displays an error message to tell the exact location and meaning of the error. The programmer accordingly fixes the errors, recompile and repeat the cycle again until all error are fixed and the program will be ready to be executed.

A common programming language used to be the BASIC programming language. The syntax in BASIC was simpler, thus fitting beginners who take it as their first programming language. The commands were separated by lines, in which one line represented one command. The lines were numbered sequentially, thus following the commands used to take sequential or logical order, similar to reading a book. The syntax used to be entered in a simple form that could be more understandable sometimes even to novice programmers.

BASIC was also simple enough to teach to beginning programmers. BASIC however was not able to handle large tasks that require writing longer programs. Thus the call increased to produce a language that is more understandable to the beginners. The call for a language that uses "English Like" statements started to increase so to overcome this problem. COBOL was the language used because of this feature. COBOL uses commands that looks like or resembles spoken English language. However, programs written in COBOL were longer and the programmer had to type all these commands which increased the chance for making syntactical errors.

The use of the newer languages in the curriculum, such as Java and C++, added a new dimension to the complexity of the syntax. These languages attempted to minimize coding. They used a lot of abbreviated codes to make it easier to write programs. However, these languages used characters that are easy to miss while typing. For example, the program used characters such as a semi colon to indicate the end of the command and the use of curly brackets to indicate beginning and end of block of codes. These characters can be confusing and easily mixed with regular brackets. Also, users sometimes do not know when to use regular brackets or curly brackets because both types are used at various stages within the program. These issues get complicated when the program contains multiple and nested block of codes. These blocks of code must each have their own opening and closing brackets. When these blocks of code are nested at multiple levels, beginning programmers have trouble understanding which bracket belongs to what block.

The error messages that are generated by the compiler are not always helpful. Sometime, these error messages are designed for advance programmers and the wording of the messages may not help beginning programmers understand their meanings. In other instances, the error messages may point to a particular line of code while the actual error is at a previous or a totally different

section within the program. In these cases, the beginning programmer keeps looking at the line where the error message is pointing and can't identify the error which can become frustrating.

Program Output Phase

This phase consists of generating the output from the program, compares the actual output with the intended outcome from the program and fixes any discrepancies between the two. The discrepancies between the intended and the actual output are what are termed as "logical errors" in programming. A logical error may occur as a result of wrong calculation routine, incorrect placement of certain commands or missing certain logical steps in achieving the output. The problem with the output deals not only with the difficulty of finding the logical error, but also with the amount of time that is needed to display the correct output or find such errors. In other words, the difficulty is specific to the time spent to display and correct the output (time/output ration).

A common first program that is used during an entry level programming courses displays a message that prints "Hello World" to the audience. Additional typical "first" programming examples may include writing a program to convert Fahrenheit to Centigrade or converting miles driven to kilometers.

Writing the programs to produce the examples that are mentioned above may follow different steps when using one programming language versus another. However, it is safe to say that producing simple outputs like the ones described above take a number of steps and a certain amount of time. To the average students that do not know a lot about programming, putting this kind of effort to produce a simple output may not be justified and may not be time efficient. After all, the same students can repeat similar statements and produce the same calculations multiple times with less effort. Students may question the feasibility of spending this much time to produce simple outputs that are generated from writing programs. Added to all of that, the process of finding and correcting the logical errors may require a certain level of understanding the logic and structure of the program which takes back to first issue of the unfamiliar programming structure to beginner level programming students.

Simplifying Learning to Program – Steps Taken

A number of studies have been conducted that acknowledges the difficulty with learning a new programming language and suggested steps to simplify this learning process. These suggestions range from simple (such as changing the programming language) to more detailed suggestions that deal with the conceptual model and the paradigm of teaching the programming languages.

Herbert (2007) noted that in order to make it easier to learn programming, three factors must be maintained: minimize the syntax, provide visual feedbacks and shorten the creative cycle of conceptualization, and improve the implementation and results. In a study that explained about the conceptual model and the learner's understanding of the programming language, Baldwin and Kulijas (2001) noted the following:

> "It has been argued that conceptual models can serve to enhance learners' conceptual understanding of programming. The methods used to enhance the development of accurate mental models include: designing the interface so that users can interact actively with it; using metaphors and analogies to explain concepts; and using spatial relationships so that users can develop capabilities for mental simulations" (p.1).

Dann, Cooper, and Pausch (2006) noted three topics that students should learn in programming courses: algorithm thinking and expression, abstraction, and appreciation of elegance. Adams

(2008) explained that in order to solve the problems associated with introductory programming courses, faculty must include examples that are engaging and capture the imagination of today's student.

Herbert (2007) clarified that the best way to teach programming ideas is to expose it to the students gently, and then to gradually add more and more detail until one day they realize they've learned quite a bit. This kind of approach to learning programming is referred to as the "spiral approach". The process can be long and sometimes tedious, thus programming educators may need to motivate students along the way to keep them interested. The remainder of this section explains about the steps that have been taken to simplify learning to program within each of the three phases of the model being developed in this paper.

Program Planning Phase

The use of planning tools has been introduced to simplify learning program structure. These programming tools are widely used in introductory level courses. They range from visual planning tools such as flowchart and hierarchy chart to text-based planning tools such pseudo code and structured walk through.

The advantages of using the visual planning tools (such as flowcharts and hierarchy charts) are that they are "visual" thus it is easier to monitor them and understand the flow. On the other hand, the main disadvantage of these visual tools is that they use different symbols to represent different operations. These symbols tend to be confusing sometimes and may contribute to further frustrating the students instead of helping them. The text-based tools (such as pseudo code and structured walk-through) have the advantages of being text based, so they will be easier to understand. However, these tools tend to use abbreviations that and different level of structure which may confuse the students further.

The structure of the program has been addressed in various programming languages. Programming code editors have improved significantly and serve as an aid to help the programmer with the structure. Some editors require the programmer to indent certain elements of the program to indicate belonging to a particular block or segment of the program. Other editors do the indentation for the programmer. An example of this indentation would be coding an "if" statement in the program. Once the programmer codes the first line of the statement, the editor automatically indents the code under it to indicate a subordinate. Other editors complete the statement for the programmer such as the case in writing HTML programs. When the programmer codes the opening tag for HTML, the editor automatically completes the code for the closing tag once the user types the back slash character.

Program Coding Phase

The programming industry in general has been trying to solve the syntax issues of programming languages for some time. Starting with the early days of teaching programming, error messages displayed from compilers were generally vague and did not help in identifying the meaning of the error or the location in the program where the error took place. As programming increased, the error messages became more descriptive and meaningful. Often, the programming editors classify the error code to different categories and color code each differently to make it easier to understand the error message. Other occasions when the editors make mass correction for the programmer, such instance when changing a variable name in one place and the editor will make the changes in all other places where the variable is listed.

As programming turned into full gear with the Graphical User Interface (GUI) objects, more help was provided to programmers. Manuals and online help provided examples of how to code. Description of error messages became more elaborate. Helpful hints were given as the programmer

types the program. An example of this is the use of "intellicense" where the users type something and the editor gives suggestions to complete the commands.

The advancement of the code editors were intended to minimize typing and to help locate the error message as they occur. Despite these advancements, however the scope of programming and the tasks that it accomplishes necessitated additional changes that complicate the code further. Therefore, some of the problems with coding were helped by the technological advances but other complications surfaced along the way.

Program Output Phase

For many students, programming is considered "boring" and therefore they shy away from taking programming courses. Thus, the logical solution may be to have them write programs that are interesting to fade away this notion of "boring" programs. This use of interesting application is more crucial at the output phase because the students want to see the result of their work at this stage. If the result is interesting, it may encourage them to work on additional programs. On the other hand, if the output is too simple, students in entry level programming courses may question the feasibility of spending that amount of time to generate a simple output.

Engaging the user in the output seems to be the key for solving this issue of "boring" outputs. Getting the user to interact with the output of a program helps with the development of the program at two fronts. First it provides a feedback to the user, and second it helps makes programming interesting (Porter & Calder, 2004).

Technological advances and the use of GUI objects in programming courses helped with this type of output when writing programs. By using GUI objects, programmers can use buttons, shapes and other tools that make a program more interesting than just plain "text" output. In these cases, simple first programming courses can assign programs to display output that is interesting rather than the old "hello world" type of output that used to be typical in first programming example.

Guibert, Girard, and Guitet (2004) stressed on the positive experience of using programming by example (PbE) where programmers design methods to provide continuous feedback during program execution. Providing such continues feedback engages the student in the program as well as provides feedback so the student is aware of what is taken place during program execution.

The use of debugger to trace program execution proved to be helpful in locating logical error in the program. The debugging tool can also be helpful in understanding the logic of the program further. It traces the execution of the program line by line so that different variables may be examined during the debugging mode. By debugging and finding out the program execution, debuggers may become a helpful tool in making the program output more interesting.

Simplifying Learning to Program – Additional Steps

This section explains about additional suggestions that are aimed to simplify learning to program at introductory or beginner level programming courses. Some of these steps have been implemented in some programs. However, some require the use of additional languages or the implementation of different tools in order to make them work.

The previous section of this paper illustrated some steps that have been taken to simplify learning to program. These steps have been applied when teaching general purpose programming languages such as C++, Java and Visual Basic. However, a new breed of languages has been introduced to simplify the introduction of a first programming course. This new breed of languages has been termed as "beginner level programming languages" (Kelleher & Pausch, 2005). Example of these newer languages includes Alice, Kara, Scratch and others.

The common characteristics of these beginner level programming languages is that they are "special purpose programming languages" as they are aimed mainly at simplifying the learning of programming languages. Hence they are being used at introductory level courses in programming (termed CS0). These languages have many features that may be incorporated into regular beginner courses in programming. Computer teachers that used these newer languages have different suggestions for implementing them in introductory programming courses. The remainder of this section explains these suggestions within each of the three phases of the model that is developed in this paper. It illustrates various features that are presented as they apply to the phase being discussed.

Program Planning Phase

Dann et al (2006) noted the benefits that can be gained from using storyboarding in program planning for beginning programmers. A previous section of this paper noted that the planning tools help the students conceptualize the solution before getting into the coding phase. Story boarding has the distinct advantages that they are commonly used in the industry, they are publicly known, and thus it is easier to understand their work due to their wider use.

In a study conducted by Hartman, Nievergelt, and Reichert (2001) suggested the use of "finite state machines" about teaching beginner programming and noted further "programming practiced as an educational exercise, free from utilitarian concern is best learned in a toy environment, designed to illustrate selected concepts in the simplest setting" (p. 1). The study suggested introducing the concepts of programming to beginner students in a game environment where the gaming give them limited actions to learn simple control routines such as if/else structure. The purpose of this finite state and gaming is to stimulate the learning of concepts through gaming and interesting applications.

The use of objects that resembles life instances may help with the conceptual understanding of the characteristics mentioned in such courses. In many cases, the beginner level programming languages introduce objects that are close to real-life objects and drag it to show the concepts of the newer concepts of Object Oriented Programming. For example, a person can be considered as an object, the person has properties (width, height), methods (run, walk) and functions can be created for the objects. So a program that displays the manipulations of different objects of the person (width, height) may be more understandable than static programs that display text manipulation and simple calculations.

Program Coding Phase

Additional studies suggested means of communicating instructions that did not include a lot of tying thus aimed at minimizing syntax. Kelleher and Pausch (2005) suggested simplifying the syntax so that beginners can more easily learn or find alternate ways to communicate their instructions to the computers. Baldwin and Kuljis (2001) suggested the use of visual systems in creating a program and further noted about this issue:

> "Syntax problems occur as a result of programmers incorrectly typing commands into the program. An effective way to deal this problem is to eliminate typing the syntax. However commands have to be coded into the program in order for the program to be executed and produce the intended output" (p. 1).

A number of studies have suggested using visual objects that produce commands and lines of code. These visual objects can be buttons or images that can be dragged and dropped into the lines of code. Once dragged, the objects provide the programmer with different options. For example, if programmers want to execute moving an object from one location to another, this can do so by dragging the object from the location where it is placed move and drop it in code. Once

dropped in the correct location the program then displays a menu asking for the next option to select. In other words, this kind of coding eliminates the possibility of syntax errors by using pre-defined visual objects that can be moved and placed in the lines of code.

Program Output Phase

Baldwin and Kuljis (2001) stressed on the benefits of using visual metaphors in program output because the metaphors provide more meaningful clues than textual output. The use of metaphors in program output is often more helpful to understanding the process of the program than simple text output. Furthermore, seeing objects that resemble characters in metaphors may justify more the time spent on developing the program than displaying simple text output. In metaphors and visual object output, the output is real and many can relate to their development or stories. Versus text output may be simple and have no direct relation to real life.

Laakso et al (2008) stressed the use of role variables to enhance Novice's debugging work and understanding programming logic. They stressed the use of abstract debugging skills as an essential part of programming skill. The same authors introduced a new tool called Ville that combines visual debugging features with the support for roles variables. The study that were conducted by Laakso et al tested using Ville on different group of students to measure the effectiveness of using visual debugging tools on beginner programmers. They concluded that using visual debugging tools help understanding of program structure and their relation to problem to the problem domain concept" (p281).

The Model

This section lists the model that this study intended to build. Table 2 shows the completed model. It summarizes the ideas presented in the previous sections according to the three phases of the modified SDLC model that was presented in Figure 2 in this paper.

Table 2 – The Completed Version of a Model Learning to Program

	DIFFICULTY REASONS	STEPS TAKEN TO SIMPLIFY	ADDITIONAL STEPS
Planning Phase	• Unfamiliar Structure • Difficult concepts	• Planning tools (e.g. Flowchart, Hierarchy chart, pseudo code)	• Story boarding • Limited action tools
Coding Phase	• Rigid syntax • Vague error messages	• Advancements in program editors • Better message	• Eliminate syntax • Use visual objects
Output Phase	• Boring output • Time/output ratio	• Engage programmer • Use debugging tools	• Visual metaphors • Visual debugging tools

Summary and Future Research

This paper developed a model to simplify learning to program in beginner or introductory level programming courses. The model was created along a modified version of the Systems Development Life Cycle (or SDLC). The paper explained about the SDLC model, modified it into three

phases so it can fit easily with the level of teaching at introductory level programming courses. The paper then discussed each of the three levels of the modified SDLC model within three different perspectives: the points that make it difficult at that phase, what has been done to simplify it at the stage and more suggestions for additional steps at the same stage. Lastly, the paper presented the finding in a model that was presented in the previous section.

While working on this paper, a lot of details have been compromised. In many instances some examples could have been presented to further illustrate what was discussed in each section. But to do so, this will lead to writing a much longer paper than what is presented here. However, the setting for which this paper is submitted may not encourage a longer paper. Thus it is the intention of the author of this paper to present a more detailed study on the same topic of difficulty associated with learning to program. The intended study will give more detailed examples and different screen shots. It will also gives additional explanation of various tools and software in each phase of the model to illustrate concepts regarding the simplification of programming in beginner or introductory level programming courses.

References

Adams, J. (2008). *Alice in action: Computing through animation*. Boston, Massachusetts: Course Technology.

Ali, A., & Mensch, S. (2008). Issues and challenges for selecting a programming language in a technology update course. *Proceedings of the Information Systems Education Conference,* Phoenix, AZ 2008. Retrieved November 17, 2008 from. http://isedj.org/isecon/2008/020/index.html

Anewalt, K.(2008). Making CS0 fun: An active learning approach using toys, games and Alice. *Journal of Computing Sciences in Colleges*, *23*(3), 98-105. Retrieved March 28, 2008 from ACM Digital Library http://www.acm.org/dl

Baldwin, L.P., & Kuljis, J. (2001). Learning programming using program visualization techniques. *Proceedings of the 34th Hawaii International Conference on System Sciences* – 2001. Retrieved April 17, 2008 from IEEE Computer Society Digital Library http://www.computer.org/portal/

Carter, J., & Jenkins, T. (2002). Gender differences in programming?. *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education.* Retrieved April 15, 2008 from ACM Digital Library http://www.acm.org/dl

Dann, W., Copper, S., & Pausch, R. (2006). *Learning to program with Alice*. Upper Saddle River, NJ: Prentice Hall.

Guibert, N., Girard, P., & Guittet, L. (2004). Example-based programming: A pertinent visual approach for learning to program. *Proceedings of the Working Conference on Advanced Visual Interfaces,* 358 – 361. Retrieved March 30, 2008 from ACM Digital Library http://www.acm.org/dl

Hartman, W., Nievergelt, J., & Reichert, R. (2001). Kara, finite state machines, and the case for programming as part of general education. *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environment (HCC01)* Retrieved April 18, 2008 from IEEE Computer Society Digital Library http://www.computer.org/portal/

Herbert, C. (2007). *An introduction to programming with Alice*. Boston, Massachusetts: Course Technology.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environment and languages for novice programmers. *ACM Computing Surveys, 37*(2), 83-137. Retrieved March 28, 2008 from ACM Digital Library http://www.acm.org/dl

Laakso, M., Malmi, L., Korhnen, A., Rajala, T., Kaila, E., & Salakoski, T. (2008). Using roles of variables to enhance novice's debugging work. *Issues in Informing Science and Information Technology*, *5*, 281-294. Retrieved from http://proceedings.informingscience.org/InSITE2008/IISITv5p281-295Laakso523.pdf

Maraks, G. (2001). *Systems analysis and design. An active approach*. Upper Saddle River, NJ: Prentice Hall.

Marrero, W., & Settle, A. (2005). Testing first: Emphasizing testing in early programming courses. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education,* 4-8. Retrieved March 28, 2008 from ACM Digital Library http://www.acm.org/dl

Porter, R., & Calder, P. (2004). Patterns in learning to program: An experiment?. *Proceedings of the Sixth Conference on Australasian Computing Education, 30*, 241-246. Retrieved April 18, 2008 from ACM Digital Library http://www.acm.org/dl

Powers, K., Ecott, S., & Hirshfield, L. (2007). Through the looking glass: Teaching CS0 with Alice. *ACM SIGCSE Bulletin, 39*(1), 213-217. Retrieved March 28, 2008 from ACM Digital Library http://www.acm.org/dl

Schneider, D. (1999). *An introduction to programming using Visual Basic 6.0.* Upper Saddle, River, NJ: Prentice Hall.

Shelly, G., Cashman, T., & Vermont, M. (2003). *Discovering computers 2004: A gateway to information*. Boston, MA: Course Technology.

Vandever, K. (2008). Teaching the business of software development. *ACM SIGCSE Bulletin* 40(2) 90-92 Retrieved November 13, 2008 from ACM digital library http://www.acm.org/dl

Biography

Azad Ali, D.Sc., Associate Professor of Technology Support and Training at Eberly College of Business – Indiana University of Pennsylvania has 25 years of combined experience in areas of financial and information systems. He holds a bachelor degree in Business Administration from the University of Baghdad, an M.B. A. from Indiana University of Pennsylvania, an M.P.A. from the University of Pittsburgh, and a Doctorate of Science in Communications and Information Systems from Robert Morris University. Dr. Ali's research interests include object oriented languages, web design tools, curriculum design and service learning projects in technology programs.