Teaching Students to Program Using Visual Environments: Impetus for a Faulty Mental Model?

Edward Dillon
Clemson University
School of Computing (HCC Division)
100 McAdams Hall
Clemson, SC 29634
(864) 656-1266
edillon@g.clemson.edu

Monica Anderson-Herzog
The University of Alabama
Department of Computer Science
Box 870290
Tuscaloosa, AL 35487-0290
205-348-1667
anderson@cs.ua.edu

Marcus Brown
The University of Alabama
Department of Computer Science
Box 870290
Tuscaloosa, AL 35487-0290
205-348-5243
mbrown@cs.ua.edu

ABSTRACT
When learning to program, students are typically exposed to either a visual or command line environment. Visual environments are usually adopted to help engage students with programming due to their user-friendly feature capabilities. This article explores the effect of using visual environments such as *Integrated Development Environments* and *syntax-free tools* to teach students how to program.

Prior studies have shown that some visual environments can have a productive impact on a student's ability to learn and become engaged with programming. However, the functional behavior of visual environments may cause a student to develop a faulty mental model for programming. One possible reason is due to the fixed set of skills that a student acquires upon initial exposure to programming while using a visual environment.

Two systematic studies were conducted for exposing students to programming in introductory courses using both visual and command line environments. From the first study, it was found that visual environments can initially impose a lower learning curve for students. However, the second study revealed that visual environments may present a challenge for students to directly transfer their acquired skills to other programming environments after initial exposure.

Categories and Subject Descriptors
H.5.2 [**Information Interfaces and Presentation**]: *Interaction styles*; K.3.2 [**Computers and Education**]: Computer science education

General Terms
Design, Human Factors

Keywords
Visual Environments, Human-Computer Interaction, Education, Learning

1. INTRODUCTION
Programming can be considered a skill for solving problems computationally. However, teaching students how to program has been a challenge. It has been argued that students sometimes fail to develop an accurate mental model for programming [3, 18]. Because of this deficiency, students can encounter programming as a barrier and in many cases leave fields that typically view this skill as a necessity. For example, Computer Science departments generally face the challenge of retaining incoming majors. Beaubouef and Mason detailed many factors that could cause students to leave Computer Science with one being the lack of skills for problem solving [1].

Attention has been placed on ways to improve a student's ability to learn and apply programming skills. One area of focus has been programming environments. Guzdial advocates "the greatest contributions to be made in this field are not in building yet more novice programming environments but figuring out how to study the ones we have" [8]. Kelleher and Pausch noted that programming environments have been built since the 1960s with the purpose of making programming accessible to people of various ages and backgrounds [10]. Visual environments like integrated development environments (IDEs) and syntax-free tools have become more common for teaching programming. There have also been efforts to expose and engage students at earlier learning stages to programming using visual environments [11, 12, 13].

Because of their functional behavior, there is the potential concern whether visual environments cause students to develop a faulty mental model for programming. Visual environments are typically constructed in a way that hides basic programming behaviors (*ex. compilation, debugging,* and *execution)* under a GUI interface. This style of construction can restrict students from direct exposure to essential programming concepts and functionalities. For instance, syntax-free tools like Alice and Scratch can cause a student to learn a limited set of programming skills by restricting exposure to code syntax, program compilation, and file systems. IDEs can provide program compilation and file system scaffolding, but disguises these and related behaviors as GUI options that are embedded into a menu item, widget, or icon. It has been found that students can depend too much on the GUI options that an IDE offers with insufficient understanding of what they are doing [2].

This article explores visual environments and their potential effect on a student's productivity for programming. Section 2 discusses prior studies regarding visual environments and their effect on students. Section 3 expounds upon the construction, feature sets, and operation behavior of visual environments. Section 4 shows

two studies that were conducted to evaluate student behavior when a visual environment is used to teach programming. Section 5 gives the conclusion, threats to validity, and future work for this research.

2. RELATED WORK

Previous studies have shown the impact of using visual environments to teach students how to program. Below is a summary of studies that evaluated visual environments and their effect on students at introductory stages of programming (Figure 1). Measurements that were used to evaluate these environments were either subjective (ex. attitudes, motivation) or objective (ex. retention rates, time on task).

Visual Environment	Course Level	Effect on Novices		Specific Effect
		Positive	Negative/None	
Alice	CS1	X		Performance, Retention Rate, and Attitudes
BlueJ	CS1	X		Attitudes
CS1 Sandbox (with subsets)	CS1	X		Time on Task
Eclipse	CS2		X	Complexity of Usage
LEGO®Mindstorms	CS1		X	Extrinsic Motivation

Figure 1. Prior Evaluations of Visual Environments and their Effect on Novices

Moskal, Lurie, and Cooper [15] measured the effect of Alice, a syntax-free environment, on CS1 students over a period of two years. Their results showed that Alice had a positive impact on performance, retention, and attitudes of the students, especially those who were considered at-risk (students with little to no programming experience prior to CS1 enrollment or a weak mathematical background) [15].

Hagan and Markham [9] studied the effect of BlueJ, a Java IDE, for teaching CS1 students object-oriented programming. They found that initially students were indifferent towards BlueJ, but gradually their attitudes became more positive for using this environment as the semester progressed. The authors believed that the difficulty of installing and learning to use BlueJ might have influenced the students' initial attitude toward this environment [9].

DePasquale [4] evaluated the ease of use of the CS1 Sandbox IDE (with and without language subsets) against Microsoft Visual C++ .Net on CS1 students. He found that students were more efficient with their tasks when using CS1 Sandbox than Microsoft Visual C++ .Net when language subsets were applied. In addition, DePasquale discovered that students who used CS1 Sandbox at the beginning of the study later migrated more readily to using Microsoft Visual C++ .Net [4].

Chen and Marx [2] measured an Eclipse IDE against an IDE called Ready to Program in a CS2 course for a period of two years. During the first semester of this study, the students preferred Eclipse over Ready to Program due to their initial excitement for this environment during an in-class demonstration. However, many of these students chose to use Ready to Program to complete take-home projects. Some of the reasons for not using Eclipse were based on the lack of experience, installment issues, and the difficulty of using this environment in the absence of the instructor [2]. During the following two semesters, the students enrolled were given a CD that provided hands-on experience with using Eclipse. Chen and Marx found that these particular students showed slightly better attitudes toward Eclipse. During the final semester of this study, Chen and Marx expanded the study into CS1 by exposing students in this course to Eclipse. They found that students depended too much on the wizards that Eclipse offered with insufficient understanding of what they were doing. Therefore, no IDE was used for programming during the following semester but rather Notepad and the Command Prompt terminal. The reason for this change was to help the students get a broader understanding of compilation, execution, and editing of programs. The authors also believed that this change would help the students better understand the usefulness of an IDE [2].

McWhorter and O'Connor [14] performed a study on LEGO® Mindstorms to determine if this application could influence motivation (intrinsic or extrinsic) for students learning to program in a CS1 course. They found that students using LEGO® Mindstorms showed a barely significant decrease in their extrinsic motivation from the control group. McWhorter and O'Connor concluded that LEGO® Mindstorms scarcely had any substantial effect on their students' overall motivation for programming [14].

From these studies, there were different conclusions about the effect of visual environments on students while learning to program. Environments like Alice, BlueJ, and CS1 Sandbox were able to influence positive productivity in the students. On the other hand, Eclipse and LEGO® Mindstorms revealed a different outcome. In particular, Chen and Marx found that the appearance of Eclipse excited their students. However, its complexity and implied behavior for programming procedures caused the authors to move later students to a command line environment.

3. THE CONSTRUCT OF VISUAL TOOLS

Visual environments are typically built using a WIMP format (window, icon, menu, and pointing device) for operation. IDEs are composed of a menu bar with a list of menu options and icons, a text editor for writing code, a built-in compiler/interpreter, and a debugger for conducting programming tasks via a mouse. In many cases, these features are integrated into one window for operation (Figure 2). Syntax-free environments like Alice and Scratch are also constructed using the WIMP format with additional features for drag-and-drop coding.

Visual environments are usually constructed differently from command line environments. Command line environments use a text editor to write and edit code but depend upon an external command terminal for code compilation/interpretation, debugging, and execution (Figure 3). In addition, students may be required to learn a variety of command arguments to effectively operate a command terminal. There are cases where certain text editors may provide a WIMP-oriented background to create and edit a program (Figure 4), but still require a command terminal to generate the program's output.

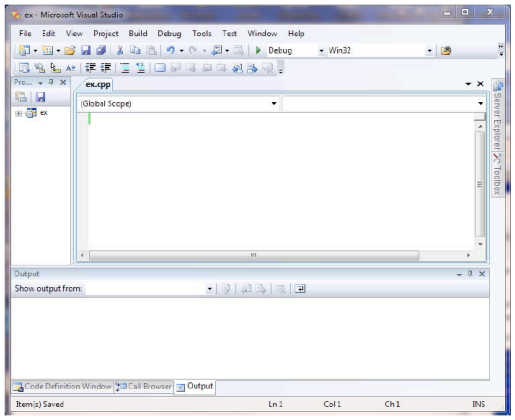

Figure 2: Microsoft Visual Studio IDE 2008

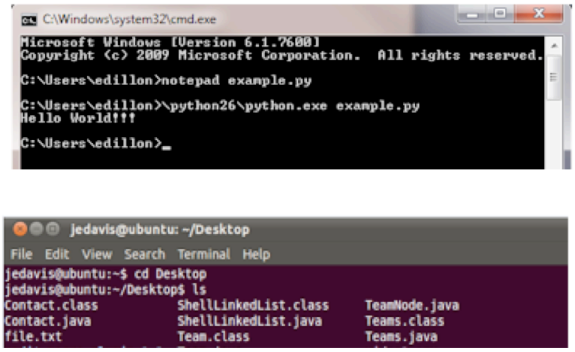

Figure 3: Command Terminals for Windows and Linux Platforms (respectively)

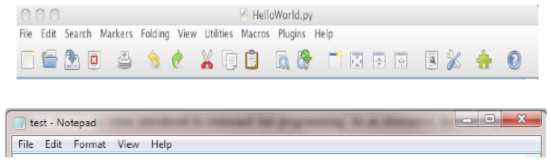

Figure 4: Text Editors for JEDIT and Notepad (respectively)

3.1 Feature Sets

The feature sets within visual environments typically provide a higher level of assistance to students when learning to program [5]. For example, IDEs can provide a large quantity of features that are designed specifically to assist users with programming; these include *syntax highlighting, error highlighting, auto completion, mouse usage,* and *integrated compilation/execution*. Usually, command line environments are not built with these capabilities, which restrict students to use a fixed set of features for operation. The next subsection provides more detail about the different levels of assistance that occur between visual and command line environments.

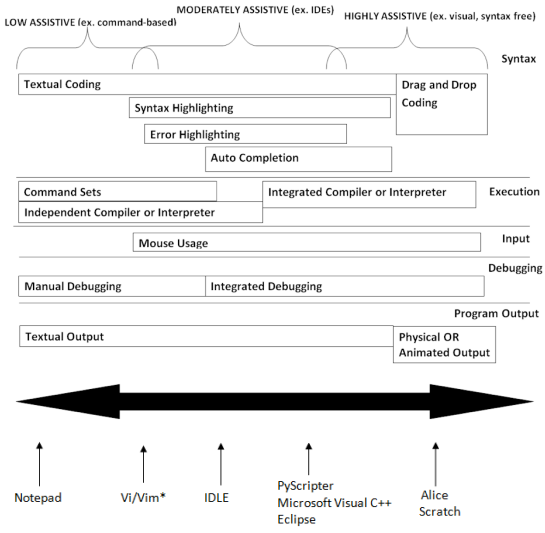

Figure 5: Programming Environments: Feature Sets Continuum [5]

Feature set can readily be altered

3.1.1 Continuum

Figure 5 illustrates a continuum of basic feature sets that can be seen amongst visual and command line environments [5]. Feature sets enable these environments to provide low, moderate, or high assistance to a programmer. The continuum provides clarity for how specific environments are categorized based on their default feature sets. There are cases where individual features can be enabled or disabled within environments (notice the asterisk beside the Vi/Vim editor in Figure 5). This can alter an environment's behavior, which can also cause an environment to shift either left or right on the continuum.

Low assistive environments (left region of the continuum) typically possess basic essential features for programming. Some of these environments may only provide the user with an editing window and a window for compilation/execution or interpretation. These environments typically allow the user to perform textual coding, command usage, and manual debugging. Users depend on some independent compiler or interpreter to run a written program that usually generates a textual output. Example environments that provide low assistance are plain text editors and text editors with very limited features. As listed on the continuum, Vi/Vim is an example text editor that provides limited features, which include syntax highlighting and mouse usage for programming. In addition, environments that represent this region of the continuum tend to be command-line oriented [5].

Moderately assistive environments (middle region of the continuum) can provide a larger quantity of assistive features for programming. Some of these features consist of syntax highlighting, error highlighting, auto completion, mouse usage, integrated compilation/execution (or interpretation), and integrated debugging. Usually, these environments can also provide textual feedback. There are some full-featured environments that possess similar traits seen in low assistive environments. These traits include: command sets, independent window for compiling/executing (or interpreting), and manual debugging. Example environments that represent this region of the continuum are rich-featured editors, intermediate and advanced/commercial IDEs [5].

Highly assistive environments (right region of the continuum) can also possess a larger quantity of assistive features for programming. Usually, these environments are built specifically to teach novices how to program. Therefore, many of these environments can also provide features that restrict the user to foundational programming concepts. Some highly assistive environments also require the user to perform drag and drop programming rather than syntax programming. In addition, physical or animated output can be used as an alternative to textual output. Example environments that represent this region of the continuum are graphical environments like Alice and Scratch, and pedagogical IDEs [5].

For additional details about the feature set continuum, see our paper published in the ***Journal of Computing Sciences in Colleges*** [5].

3.1.2 Familiarity

As part of feature assistance, there are features within programming environments, particularly those that are visual (Figure 6), that can provide a student with a familiar clue or *affordance* of how a particular action can be performed while programming [17]. Some of these features can also be seen in common software applications that provide service to users with

different levels of computational experience, which include *Microsoft Office suites, Internet Safari,* and *ITunes* (Figure 7).

It is likely that students have been exposed to these software applications to *surf the web*, *chat online*, *write an essay or term paper*, or *listen to music* prior to their first programming class. Because of these similar features, there is the potential for a visual environment's behavior to be familiar to students while learning to program. For example, a student could perceive the procedures for using a visual environment to be relative to a word processor. This sense of familiarity could also lessen the learning curve for understanding the operations of a visual environment upon initial exposure.

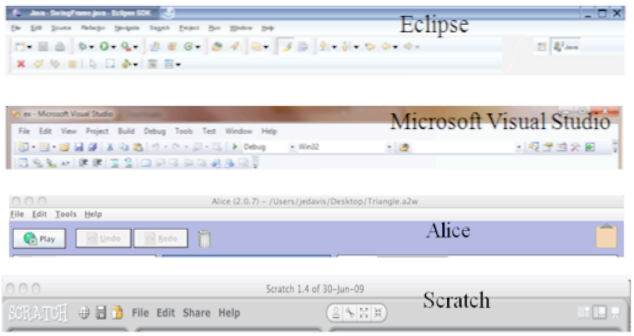

Figure 6: Examples of Visual Environments and their Relative Features [6]

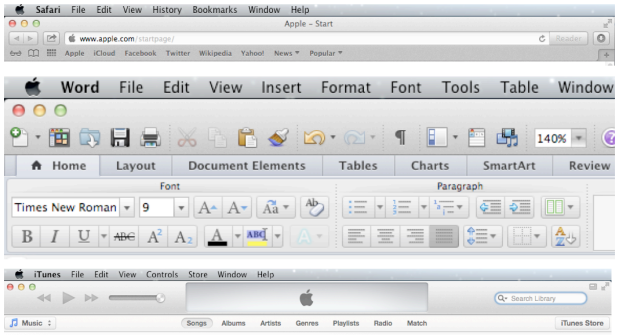

Figure 7: Software Applications and their Relative Features to Visual Environments (*Internet Safari, Microsoft Word, and iTunes*)

3.2 Operation Behavior

While students are learning to program, understanding programming concepts or language syntax is one aspect. Another is becoming accustomed to the procedures for operating a programming environment. When operating a command line environment, students typically cannot bypass one procedure and complete another. This is not the case for many visual environments. Instead, programmers can perform certain procedures automatically with a click of the mouse. The next subsections discuss the operation of command line environments, IDEs, and syntax-free environments respectively along with a brief discussion about their potential effect on students.

3.2.1 Command Line Programming

When conducting command line programming, students are usually directed to an editing window to begin composing (or *writing*) their program. Students must also save their program as a file for the remaining procedures. Next, students should test the correctness of their written program by compiling their saved file. Since a command terminal is typically used for compilation, students are required to use command sets for operation. Based on the command terminal and language being used for programming, there are certain commands that will enable the students to compile their program file. Upon compilation, students are faced with one of two scenarios: 1) If a syntax error(s) is detected during compilation, this error must be corrected before proceeding to the next step. 2) If no errors are detected during compilation, the program file undergoes the process of linking. When linking occurs, the program file is converted into an executable file in preparation for execution. After program linking is completed, the students must type a certain command in the terminal to invoke the execution of their program. Upon execution, the students are faced with one of two more scenarios: 1) If a semantic (or logical) error(s) occurs, this error must be corrected and would require the students to repeat the compilation and linking process again. 2) If no errors are detected during execution, the output of the program would be generated and viewed in the terminal window. Figure 8 provides an outline of the typical operations for command line programing. Table 1 provides a summarized list of these operations in their respective order.

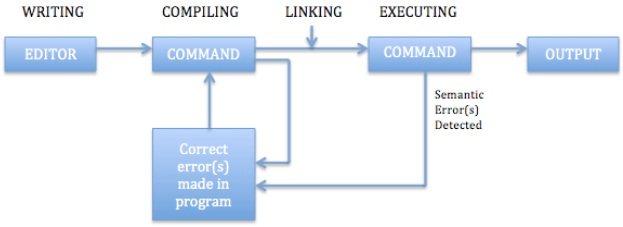

Figure 8: Outline of Command Line Programming

Table 1. Command Line Programming Operations

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is compiled and checked for syntax errors. (Students must use the appropriate command to invoke this behavior)
Step 3*	The file of the program is converted into an executable file for execution.
Step 4*	The executable file of the written program is executed to acquire the intended output. (Students must use the appropriate command to invoke this behavior)
Step 5	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

It is also important to note that certain languages are not compiled, but rather interpreted. The operations for interpreted languages are almost identical to a compiled language with exception to the procedures for compiling and linking the program file. Instead, the program file containing the written code has to be interpreted. There are certain commands that will enable students to interpret the code in their program file. Upon interpretation, the students are faced with one of three scenarios: 1) If a syntax error(s) is detected

during interpretation, this error must be corrected before proceeding to the next step. 2) If a semantic (or logical) error(s) occurs, this error must be corrected before proceeding to the next step. 3) If no errors are detected during interpretation, the output of the program would be generated and viewed in the terminal window. Figure 9 provides an outline of the typical operations for command line programming with interpreted languages. Table 2 provides a summarized list of these operations in their respective order.

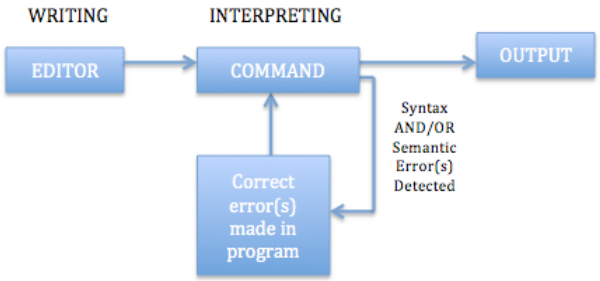

Figure 9: Outline of Command Line Programming (using an Interpreted Language)

Table 2. Command Line Programming Operations (Interpreted Language)

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is interpreted and checked for syntax and semantic errors. (Students must use the appropriate command to invoke this behavior)
Step 3	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

3.2.2 IDE Programming

Similar to command line programming, students are directed to an editing window to begin composing their program in an IDE. Students must also save their program as a file for the remaining procedures. Next, students must test the correctness of their written program. Depending upon the IDE, this can occur in different ways. For example, many IDEs provide a menu option that enables students to automatically compile, link, and execute their program file with a single mouse click. During this process, students are faced with one of three scenarios: 1) If a syntax error(s) is detected during compilation, this error must be corrected before the file automatically proceeds to the linking phase. 2) If a semantic (or logical) error(s) occurs, this error must be corrected before the file is successfully executed. 3) If no errors are detected during this process, the output of the program would be generated and viewed either within the same window of the editor or in an independent window.

Other IDEs follow a similar procedure seen in command line environments, which allow students to compile (and link) their program independently of execution. Instead of using a command terminal to do so, a menu option is provided to conduct this procedure. The output generated during execution from these particular IDEs can also be viewed either within the same window of the editor or in an independent window.

For languages that are interpreted, certain IDEs are built to interpret a written language using a menu option that invokes this behavior using a single mouse click. Upon interpretation, the output is also generated and viewed either within the same window of the editor or in an independent window. Figure 10 provides an outline of IDE programming that includes all three styles of operation. Table 3-5 provides a summarized list of each style of IDE operation respectively.

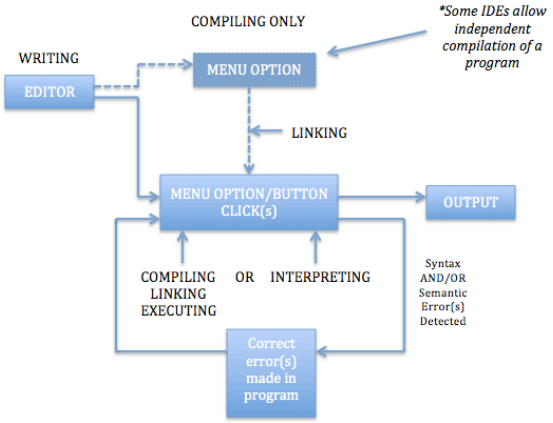

Figure 10: Outline of IDE Programming

Table 3. IDE Programming Operations (Compiling, Linking, and Executing automatically)

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is compiled, linked, and executed based upon the correctness of the written code. (Students must use the appropriate menu option to invoke this behavior). During this process, the file is checked for syntax and semantic errors.
Step 3	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

Table 4. IDE Programming Operations (Compiling/Linking and Executing independently)

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is compiled and checked for syntax errors. (Students must use the appropriate menu option to invoke this behavior)
Step 3*	The executable file of the written program is executed to acquire the intended output. (Students must use the appropriate menu option to invoke this behavior)
Step 4	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

Table 5. IDE Programming Operations (using an Interpreted Language)

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is interpreted. (Students must use the appropriate menu option to invoke this behavior). During this process, the file is checked for syntax and semantic errors.
Step 3	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

3.2.3 Syntax-Free Programming

Syntax-free programming also provides students with an editing window to create their program. Instead of using syntax as a method for composing a program, students in many cases must *drag* snippets of code from other windows of the environment and *drop* them into the editing window. Once a program has been created, students must also save their program as a file for the upcoming procedures. Similar to IDEs, syntax-free environments provide a menu option for students to test the correctness of their written program. During this process, students are faced with one of two scenarios: 1) If a semantic (or logical) error(s) occurs, this error must be corrected before the program can be executed. In many cases, errors can be corrected by either *discarding* inappropriate code from the composed program or dragging/dropping additional snippets of code into the same program. 2) If no errors are detected during this process, the output of the program would be generated and viewed either within the same window of the editor or in an independent window. Figure 11 provides an outline of syntax-free programming. Table 6 provides a summarized list of these operations in their respective order.

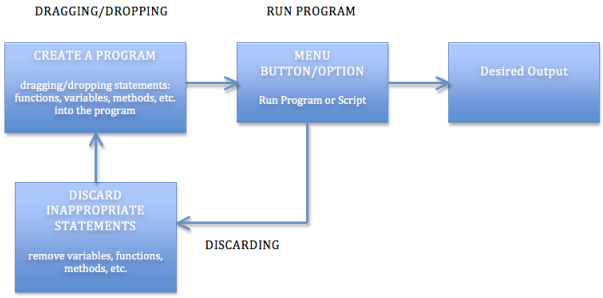

Figure 11: Outline of Syntax-Free Programming

Table 6. Syntax-Free Programming Operations

Step 1	Editing window is used to compose program. (Program should be saved as a file)
Step 2*	The file of the composed program is tested. (Students must use the appropriate menu option to invoke this behavior). During this process, the file is checked for semantic errors.
Step 3	The program's output is generated and viewed.
**May require multiple attempts due to semantic errors.*	

3.2.4 Discussion

Command line programming directly exposes students more to basic procedures for programming, such as compiling a written program, generating an executable file of a program through linking, and executing the executable file to generate the program's output. Students have to manually perform each procedure using certain commands to obtain the output of their written program. In contrast, visual environments can potentially provide a shorter process for students to conduct the same behavior. Because visual environments are usually operated using menu bars, icons, and mouse clicks, students are exposed to a higher level of abstraction for operation and navigation while programming. However, this style of construct may misrepresent some of the basic procedures for programming. For example, a student who is initially exposed to programming through an IDE may get the impression that clicking the appropriate menu option magically makes their program work while disregarding the actions of compiling, linking, executing, or interpreting.

4. STUDIES

To further examine the effects of visual environments on students while learning to program, a study was conducted on a CS1 lab and lecture course respectively at *The University of Alabama*. Section 4.1 discusses the first study that was conducted on the CS1 lab. Section 4.2 talks about the second study that was conducted as a semester-long assessment on the CS1 lecture course.

4.1 Study #1

The first study was conducted as a one-day pilot study for measuring the initial effects of visual and command line programming on students. The CS1 lab course generally introduces students to robotic programming through a syntax-free environment called PREOP that allows them to program real robots using drag-and-drop procedures in Alice. This particular course has no prerequisites and two or three sections are usually offered per semester. Three sections were offered during the time of this study (Spring 2011).

4.1.1 Methods & Procedures

For this study, each section received an environment to conduct Python programming: Section 1 received an *IDLE IDE* (Figure 12), Section 2 was given a *PyScripter IDE* (Figure 13), and Section 3 used *Notepad/Command Prompt* (Figure 14). Three measures were conducted for student assessment: *Computer Programming Self-Efficacy Scale* [16], *a time on task assessment,* and a *usability survey*.

The number of students enrolled in the CS1 lab course was 133. There were 45, 45, and 43 students enrolled in the IDLE, PyScripter, and Notepad sections respectively. The student population for this study varied for each procedure. This was due to students either arriving late to class or not correctly following the instructions. Therefore, the student population represented in this study ranged from 91-102 students. Tables 7-20 (with exception to Table 13) list the numbers of students who participated during each assessment.

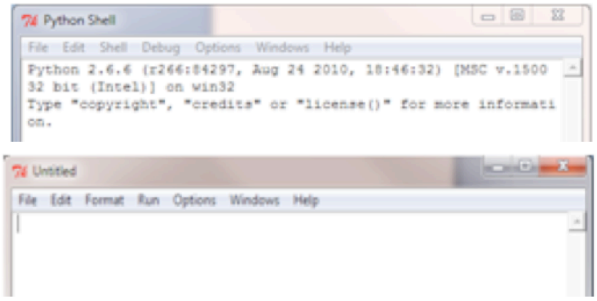

Figure 12: IDLE IDE version 2.6.6

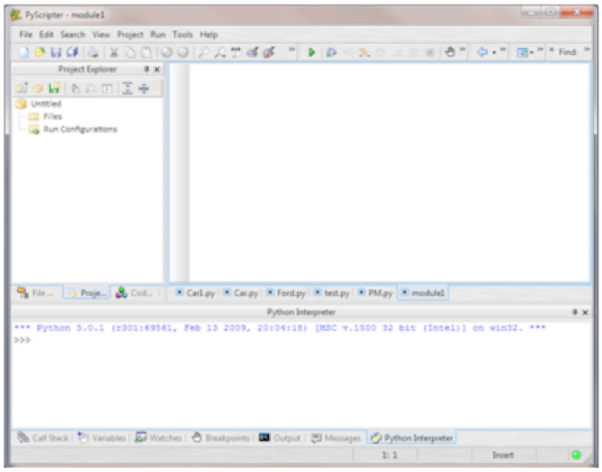

Figure 13: PyScripter version 1.9.9.6

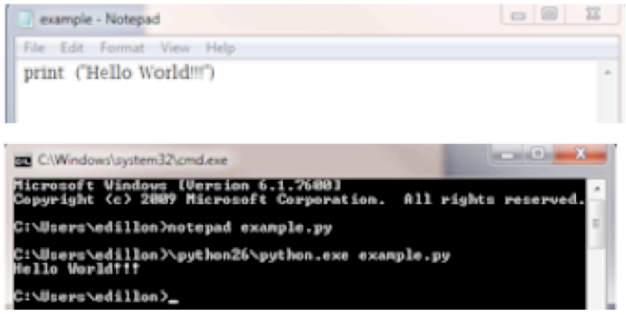

Figure 14: Notepad/Command Prompt – Windows Platform

To begin the study, each student received a self-efficacy survey. This survey consisted of 31 questions from the Computer Programming Self-Efficacy Scale. The responses were given on a 7-point Likert scale that ranged from *not confident at all* to *absolutely confident*. As part of this survey, a demographics section was provided in order to acquire feedback about the students, which included *academic major*, *classification,* and *prior programming experience*.

Next, the students received an introductory lecture on the Python language. This lecture introduced basic Python concepts that the students would need to complete the exercise. Students were exposed to concepts of *code syntax and semantics*, *selection*, and *information hiding*. Topics that were covered included *print statements, variable usage and assignment*, *reserved keywords* and *mathematical operations (with inferences on division and modulus usage)*. The lecture concluded by showing an example program using every topic. *This program converted* ***x*** *number of minutes into* ***h*** *hours and* ***m*** *minutes remaining.* The behavior of this program resembled the assignment that the students would be asked to write.

After the lecture, the students received a demonstration on how to use their respective environment and were required to write a small program that converted 700 days into *y* years, *m* months, and *d* days remaining. During this process, their time to complete this task was measured. The objective was to measure the students' time on task for writing the required program using their respective environment. For the IDLE group, a process monitoring application was used to measure time on task. In order to access their time logs, the students first accessed the process monitoring application before using IDLE, and then remain logged onto their computers after completing the assignment. However, some students did not follow these directions correctly which resulted in their time logs being lost. Therefore, the remaining two sections did not use the software. Instead these students were required to start at the same time and were required to raise their hands upon completing the assignment. The time on task for these sections was calculated by subtracting time of completion from the starting time.

After the time on task assessment, a usability survey was issued. This survey was composed of questions that directly focused on the students' experience with their respective tool. These questions measured subjective attributes regarding attitudes and feelings about using these environments respectively.

4.1.2 Results

The student demographics consisted of different majors at varying classification levels with contrasting levels of prior programming experience (Tables 7 - 10). For instance, the PyScripter group had more *Electrical Engineering* majors than *Computer Science.* The PyScripter and Notepad groups had significantly more *juniors* than the IDLE group *($p<0.05$)*. The IDLE group had less prior programming experience than the PyScripter group, which was also significantly less *($p<0.05$)* than the Notepad group. In addition, the Notepad group had a higher percentage *(50%)* of students who were taking the CS1 lecture course in conjunction with this lab. Traditionally, CS1 teaches Python programming using the VIM command editor on the Linux platform.

4.1.2.1 Self-Efficacy

The self-efficacy survey was used as an indicator for initially determining the students' self-efficacy for programming prior to their participation in this study. This survey measured the students' confidence for performing certain programming procedures ranging from *writing syntactically correct programs* to *writing a program that someone else could successfully comprehend.* The students' scores on this survey reflected their self-efficacy, meaning that a high score indicated an individual to have a high self-efficacy toward programming (and vice versa). The highest score that could have been made on this survey was 217. From this survey, the students showed an overall mean self-efficacy score of *114.85* out of 217 (with a normalized mean of *0.51* on a scale of 0 to 1).

The mean self-efficacy scores (see Table 11) amongst the three sections were tested using a one-way ANOVA. The ANOVA showed a significant variation amongst the three sections *($p<0.01$)*.

Table 7. CS1 Lab Demographics
**Number of responses before Time on Task was conducted.*

Participants (N=94*)	
Major	Computer Science - 33% Electrical Engineering - 29% Computer Engineering - 15% MIS - 3% Math - 5% Other - 18%
Classification	Freshmen - 41% Sophomore - 32% Junior - 22% Senior - 3% Other - 1%
Programming Experience	CS1 programming - 31% High School programming - 26% Another College Course - 18% No Experience - 26%

Table 8. CS1 Lab Demographics – IDLE
**Number of responses before Time on Task was conducted.*

IDLE Group (N=30*)	
Major	Computer Science - 37% Electrical Engineering - 27% Computer Engineering - 23% MIS - 7% Math - 7% Other - 7%
Classification	Freshmen - 57% Sophomore - 37% Junior - 7% Senior - 0% Other - 0%
Programming Experience	CS1 programming - 17% High School Course - 17% Another College Course - 17% No Experience - 40%

Table 9. CS1 Lab Demographics – PyScripter
**Number of responses before Time on Task was conducted.*

PyScripter Group (N=38*)	
Major	Computer Science - 24% Electrical Engineering - 42% Computer Engineering - 13% MIS - 3% Math - 3% Other - 18%
Classification	Freshmen - 32% Sophomore - 37% Junior - 39% Senior - 0% Other - 3%
Programming Experience	CS1 programming - 34% High School Course- 16% Another College Course - 24% No Experience - 26%

Table 10. CS1 Lab Demographics – Notepad
**Number of responses before Time on Task was conducted.*

Notepad Group (N=26*)	
Major	Computer Science - 42% Electrical Engineering - 12% Computer Engineering - 8% MIS - 4% Math - 8% Other - 31%
Classification	Freshmen - 38% Sophomore - 19% Junior - 31% Senior - 12% Other - 0%
Programming Experience	CS1 programming - 50% High School Course - 27% Another College Course - 8% No Experience - 15%

The ANOVA test was followed by T-tests to determine whether specific differences existed amongst the sections. The T-tests showed a significant difference between the IDLE and PyScripter groups *(p<0.01)* as well as the IDLE and Notepad groups *(p<0.01)* respectively. There was no significant difference between the PyScripter and Notepad groups. This indicated that students in the IDLE group were less confident in their programming abilities than their counterparts in the PyScripter and Notepad groups respectively.

Table 11. Self-Efficacy Descriptive Data for CS1 Lab

Group	N	Mean Score (Possible Score)	StdDev	Normalized Mean (scaling from 0 to 1)
IDLE	30	88.30 (out of 217)	38.91	0.42
PyScripter	38	125.63 (out of 217)	49.57	0.53
Notepad	26	129.73 (out of 217)	38.90	0.59
All Groups	94	114.85 (out of 217)	46.83	0.51

4.1.2.2 Time on Task

Overall, the average performance time for students to complete the assignment was *24.63 minutes* (Table 12). A one-way ANOVA showed a significant difference *(p<0.01)* between the average performance times amongst the three sections. The ANOVA test was followed by T-tests which showed a significant difference between the IDLE and PyScripter groups *(p<0.05)*, the IDLE and Notepad groups *(p<0.01)*, and the PyScripter and Notepad groups *(p<0.01)*. This indicated that students who used PyScripter finished their required task quicker than the students using IDLE and Notepad respectively. At the same time, students who used IDLE completed their task quicker than the students using Notepad.

Table 12. Time on Task Descriptive Data for CS1 Lab

Group	N	Average Time	StdDev
IDLE	21	23.05 minutes	12.62
PyScripter	40	15.88 minutes	10.89
Notepad	30	34.97 minutes	16.83
All Groups	91	**24.63 minutes**	**13.45**

4.1.2.3 Environment Usability

This survey was composed of several attributes to measure the environments' usability. Questions in the survey are listed in (Table 13). The results that were generated from the students' response to each question are also discussed in further detail. Tables 14-20 provide statistical analysis for each attribute measured.

Table 13. Usability Attributes
(OE = Open Ended; MC = Multiple Choice)

Attribute	Question
Initial Impression of Environment	OE
Comfort with Environment	MC
Confidence with Doing Another Assignment with Environment	MC
Fondness of Environment	MC
Easiest Attributes about the Environment	OE
Hardest Attributes about the Environment	OE
Experiences with Other Environments (besides PREOP)	OE

Initial Impression about the Environment. The responses were quantified into three categories: positive, non-positive, and no response. Non-positive responses consist of either neutral/confused or negative feelings about the environment. For quantification, the positive responses received a value of 1, and the non-positive and no responses received a value of 0.

A one-way ANOVA indicated a significant difference *(p<0.01)* amongst the three groups. Afterwards, T-tests indicated a significant difference for each T-test: IDLE vs. PyScripter *(p=0.05)*, IDLE vs. Notepad *(p=0.05)*, PyScripter vs. Notepad *(p<0.01)*. These results showed that the Notepad group had a less positive initial impression than the IDLE and PyScripter groups respectively. In addition, students in the IDLE group had a less positive initial impression than the PyScripter group. Table 14 provides further analysis about this measure.

Table 14. Initial Impression of Environment

Group	N	Mean	StdDev
IDLE	34	0.35	0.49
PyScripter	38	0.55	0.50
Notepad	30	0.17	0.38
All Groups	102	0.37	0.48

The mean was calculated by labeling Positive Responses = 1, and Non-Positive and No Responses = 0.

Comfort with Environment. Based on the response choices ranging from *not comfortable at all* to *absolutely comfortable*, a one-way ANOVA indicated a significant difference *(p<0.01)*. Afterwards, T-tests indicated a significant difference for two of the pairings: IDLE vs. PyScripter *(p<0.01)* and IDLE vs. Notepad *(p<0.05)*. These results showed that the IDLE group was less comfortable with using IDLE than the PyScripter group with PyScripter and the Notepad group with Notepad respectively. The PyScripter and Notepad groups showed no significant difference between each other. Table 15 provides further analysis about this measure.

Table 15. Comfort with Environment

Group	N	Mean	StdDev
IDLE	34	3.44	1.52
PyScripter	38	4.63	1.53
Notepad	30	4.30	1.74
All Groups	102	4.14	1.65

*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Not Comfortable at All** to **7 = Absolutely Comfortable***

Confidence with Doing Another Assignment with the Environment. Based on the response choices ranging from *not confident at all* to *absolutely confident*. A one-way ANOVA indicated a significant difference *($p<0.01$)*. Afterwards, T-tests indicated a significant difference for two of the pairings: IDLE vs. PyScripter *($p<0.01$)* and IDLE vs. Notepad *($p<0.05$)*. These results showed that the IDLE group was less confident with using IDLE to do another assignment than the PyScripter group with PyScripter and the Notepad group with Notepad respectively. The PyScripter and Notepad groups showed no significant difference between each other. Table 16 provides further analysis about this measure.

Table 16. Confidence with Doing Another Assignment with Environment

Group	N	Mean	StdDev
IDLE	34	3.38	1.67
PyScripter	38	4.74	1.64
Notepad	30	4.37	1.96
All Groups	102	4.18	1.82

*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Not Confident at All** to **7 = Absolutely Confident***

Like the Environment. Based on the response choices ranging from *not at all* to *absolutely like*. A one-way ANOVA indicated a significant difference *($p<0.01$)*. Afterwards, T-tests indicated a significant difference for two of the pairings: IDLE vs. PyScripter *($p<0.01$)* and PyScripter vs. Notepad *($p<0.01$)*. The students in the IDLE and Notepad groups liked IDLE and Notepad respectively less than the PyScripter group with PyScripter. No significant variations were noted between the IDLE and Notepad groups. Table 17 provides further analysis about this measure.

Table 17. Fondness of Environment

Group	N	Mean	StdDev
IDLE	34	3.41	1.73
PyScripter	38	4.87	1.66
Notepad	30	3.77	1.79
All Groups	102	4.06	1.81

*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Not at All** to **7 = Absolutely Like***

Easiest Attributes about the Environment. The responses were quantified into five categories: *Python Attributes*, *Environment Attributes*, *Familiarity*, *Nothing/No Response* and *I Don't Know*. Python Attributes represented students who gave a response about the Python language. Environment Attributes represented students who gave a response about their respective environment based on its features. Familiarity represented students who responded based on a previous experience with programming. The categories of Nothing/No Response and I Don't Know represented students who actually provided such responses. For quantification, responses that were categorized as Environment Attributes received a value of 1. All other responses received a value of 0.

A one-way ANOVA indicated no significant difference amongst the three groups. Since many of the students were not exposed to Python prior to this study, several of them responded more frequently about the easiest attributes of the Python language itself rather than their respective environment. A T-test indicated a significant difference *($p<0.05$)* between responses towards the Python language and the respective environments. Additional T-tests were used to determine any significant differences within each group. The results indicated a significant difference *($p<0.01$)* for only the IDLE group. These results showed that the IDLE group responded more frequently about the easy attributes of the Python language rather than the IDLE environment. The frequency of responses to Familiarity, Nothing/No Response, and I Don't Know were insignificant. Table 18 provides further analysis about this measure.

Table 18. Easiest Attributes of the Environment

Group	N	Mean	StdDev
IDLE	34	0.18	0.37
PyScripter	38	0.37	0.49
Notepad	30	0.37	0.49
All Groups	102	0.30	0.46

The mean was calculated by labeling Environment Attributes = 1 and all other categories = 0.

Hardest Attributes about the Environment. The responses were also quantified using the same categories as shown for the easiest attributes. For quantification, responses that were categorized as Environment Attributes received a value of 1. All other responses received a value of 0. A one-way ANOVA indicated a significant difference *($p<0.01$)*. Afterwards, T-tests indicated a significant difference for two of the pairings: IDLE vs. Notepad *($p<0.01$)* and PyScripter vs. Notepad *($p<0.01$)*. These results showed that Notepad received more responses concerning its hard attributes than IDLE and PyScripter respectively.

In regards to the responses about the Python language itself, a one-way ANOVA indicated a slight significant difference *($p=0.054$)*. Afterwards, T-tests indicated a significant difference for two of the pairings: the IDLE group vs. the Notepad group *($p=0.01$)* and the PyScripter group vs. the Notepad group *($p<0.05$)*. These results showed that students in the Notepad group gave fewer responses about the hardest attributes of the Python language than the IDLE and PyScripter groups respectively. The frequency of responses to Familiarity**,** Nothing/No Response, and I Don't Know were insignificant. Table 19 provides further analysis about this measure.

Table 19. Hardest Attributes of the Environment

Group	N	Mean	StdDev
IDLE	34	0.06	0.24
PyScripter	38	0.11	0.31
Notepad	30	0.40	0.50
All Groups	102	0.18	0.38
The mean was calculated by labeling Environment Attributes = 1 and all other categories = 0.			

Experiences with Other Environments (Besides PREOP). This particular question was asked in conjunction with another question: *Was the environment mandatory for a course?* Statistical analyses were conducted for both questions.

A one-way ANOVA was used to determine if certain sections had more prior experience with other environments besides PREOP. The results indicated a significant difference *($p<0.01$)*. Afterwards, T-tests were used to compare each group against another. A significant difference was found for two of the T-tests: the IDLE group vs. the PyScripter group *($p<0.01$)* and the IDLE group vs. the Notepad group *($p<0.01$)*. These results showed that the IDLE group has less experience with using other environments (besides PREOP) than the PyScripter and Notepad groups respectively.

A one-way ANOVA was also used to determine if these other environments were mandatory for another course. The results indicated a significant difference *($p<0.05$)*. Afterwards, T-tests indicated a significant difference for only one of the pairings: the IDLE group vs. the PyScripter group *($p<0.01$)*. These results not only showed that the PyScripter group had more experience with other environments than the IDLE group, but also that they were mandatory for another course. The PyScripter and Notepad groups showed no significant difference amongst each other. Table 20 provides further analysis about this measure.

Table 20. Experiences with Other Environments (besides PREOP)

Group	N	Mean	StdDev
IDLE	34	0.26	0.45
PyScripter	38	0.68	0.47
Notepad	30	0.50	0.51
All Groups	102	0.49	0.50
The mean was calculated by labeling Yes = 1 and No = 0.			

An additional T-test was used for the PyScripter group to determine whether their experience with other environments were actually IDEs. For the PyScripter group, the results were significant *($p<0.01$)*. These results showed that most of these students (68%) had prior experience with IDEs. As previously mentioned, many of the students in the PyScripter group were ECE majors. Traditionally at this university, all ECE majors must take CS285, which teaches the C language using the CodeBlocks IDE. Similar to PyScripter, CodeBlocks is an IDE rich with features. Out of the 68% of these students who had prior exposure to IDEs, 90% of them had experience with CodeBlocks prior to this study.

4.1.3 Discussion

The IDLE group had less prior programming experience than their counterparts in the PyScripter and Notepad groups. This factor may have impacted a majority of the results seen from this group. They were found to be less confident in their programming abilities, less comfortable with IDLE after using it, and less confident about doing another assignment. They also did not like IDLE as much as students who liked PyScripter. Their lack of programming experience was obvious when asked about the ease or difficulty of using IDLE. Instead of providing positive responses about IDLE, they expressed comfort about the Python language. Despite lacking programming experience, the IDLE group completed their task significantly faster than the Notepad group.

Students in the PyScripter and Notepad groups showed no differences in their programming experience. They also showed no differences in their comfort with their respective environments as well as their confidence of doing another assignment. However, the PyScripter group had a more positive initial impression, more of a fondness with PyScripter, and a faster completion time than the students using Notepad. Students in the Notepad group (not significantly) had more prior exposure to command line programming through CS1. However, they frequently showed difficulties with using Notepad, which influenced their time to complete the required exercise. In contrast, students using PyScripter rarely demonstrated difficulties about using PyScripter, and a majority of them had prior exposure to IDEs. In addition, 45% of the PyScripter group had a non-positive initial impression. On the other hand, 70% of the Notepad group had a non-positive initial impression. Fifty-three percent of the IDLE group showed a non-positive impression. However, many of the IDLE students did not have prior programming experience unlike the other groups.

4.2 Study #2

This study was conducted as part of a larger empirical evaluation of visual and command line programming in CS1 over the course of a semester. As previously mentioned, the CS1 course at the University of Alabama traditionally teaches Python using the VIM command line environment on the Linux platform. During the Fall 2011 semester, this course was altered to allow certain sections to use IDLE (in Linux) as an alternative to VIM. Four sections were offered during this particular semester; two sections were taught programming using VIM (Figure 15) and one section used IDLE (Figure 16). The remaining section, an honors section, was given the option of either tool. During the latter part of the semester, the non-honor sections were required to switch environments.

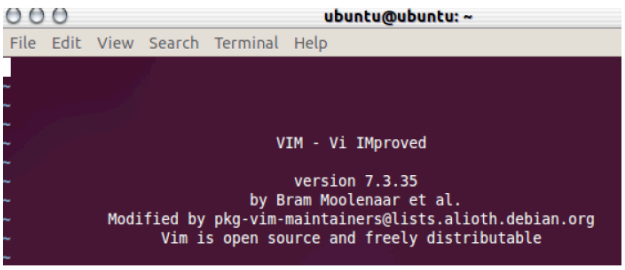

Figure 15. VIM version 7.3.35

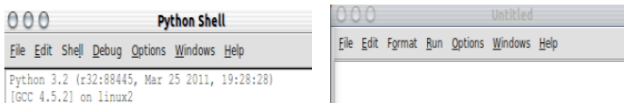

Figure 16. IDLE-Python 3.2

4.2.1 Methods/Procedures

As part of this empirical assessment, a demographic survey, three usability surveys and a protocol analysis were given during the semester. The usability surveys were administered twice before switching environments and once afterwards. After the environment switch, a protocol analysis was conducted on a small group of students to study their mental model for operating a visual or command line environment.

The number of students enrolled in the CS1 course was 179. There were 46, 88, and 45 students enrolled in the IDLE, two VIM, and honor sections respectively. Tables 21-27 list the numbers of students who participated during each assessment.

4.2.2 Results

The demographics shown in Tables 21-24 respectively are a representation of the CS1 student population *(N=119)* at the beginning of the semester. However, there were students who stopped attending class, dropped the CS1 course, or became agitated with participating in this study. These factors influenced a decrease in sample representations and student participation as the semester progressed, especially during the final assessments of this study.

Table 21. CS1 Demographics

Participants (N=119)	
Major	Computer Science - 61% Electrical Engineering - 3% Computer Engineering - 3% MIS - 1% Math - 6% Other - 22% Double Major (including CS) - 1% Double Major (excluding CS) - 3%
Classification	Freshmen - 40% Sophomore - 32% Junior - 19% Senior - 8% Other - 3% **one student did not provide an answer*
Programming Experience	High School programming - 16% Another College Course - 16% No Prior Experience - 68% **three students did not provide an answer*

Table 22. CS1 Demographics - IDLE Section

IDLE Section (N=33)	
Major	Computer Science - 85% Electrical Engineering - 0% Computer Engineering - 0% MIS - 0% Math - 6% Other - 9% Double Major (including CS) - 0% Double Major (excluding CS) - 0%
Classification	Freshmen - 34% Sophomore - 42% Junior - 15% Senior - 9% Other - 0%
Programming Experience	High School programming - 9% Another College Course - 25% No Prior Experience - 66%

Table 23. CS1 Demographics – VIM Sections

VIM Sections (N=46)	
Major	Computer Science - 49% Electrical Engineering - 2% Computer Engineering - 0% MIS - 2% Math - 9% Other - 29% Double Major (including CS) - 2% Double Major (excluding CS) - 7% **one student did not provide an answer*
Classification	Freshmen - 31% Sophomore - 27% Junior - 29% Senior - 11% Other - 2% **one student did not provide an answer*
Programming Experience	High School programming - 11% Another College Course - 9% No Prior Experience - 80%

Table 24. CS1 Demographics – Honor Section

Honor Section (N=40)	
Major	Computer Science - 56% Electrical Engineering - 5% Computer Engineering - 7% MIS - 0% Math - 5% Other - 27% Double Major (including CS) - 0% Double Major (excluding CS) - 3%
Classification	Freshmen - 55% Sophomore - 28% Junior - 10% Senior - 3% Other - 5%
Programming Experience	High School programming - 25% Another College Course - 17% No Prior Experience - 58%

4.2.2.1 Usability

One of the attributes measured in this survey was *Tool Mishandling* (Tables 25 and 26). Tool Mishandling was defined on the basis of how often students found themselves making errors, due to using IDLE or VIM incorrectly. This attribute was based on a 7-point Likert scale (*where 1 = absolutely often & 7 = absolutely NOT often*). *The results discussed are strictly based on the behavior of the non-honor sections.*

In the IDLE section, the results from a one-way ANOVA indicated a significant difference ($p<0.05$). Afterwards, T-tests indicated a significant difference for two of the pairings: 1st vs. 3rd surveys *($p<0.05$)* and 2nd vs. 3rd surveys *($p<0.01$)*. The results indicated two things: students in the IDLE section mishandled VIM more often than IDLE and the mishandling of a tool increased significantly after the switch. In the VIM sections, the results from a one-way ANOVA and T-Tests showed no significant difference. These results indicated students in the VIM sections did not mishandle one tool more often than the other.

Table 25. Tool Mishandling Results – IDLE Section
Avg = Average; SD = standard deviation

IDLE Section (IDLE to VIM)				
Tool	**Survey**	**N**	**Avg**	**StdDev**
IDLE	1st	31	4.10	1.42
IDLE	2nd	26	4.42	1.36
VIM	3rd	13	3.08	1.75
*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Absolutely Often** to **7 = Absolutely Not Often***				

Table 26. Tool Mishandling Results – VIM Sections
Avg = Average; SD = standard deviation

VIM Sections (VIM to IDLE)				
Tool	**Survey**	**N**	**Avg**	**StdDev**
VIM	1st	29	3.90	1.40
VIM	2nd	49	4.22	1.21
IDLE	3rd	39	4.41	1.58
*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Absolutely Often** to **7 = Absolutely Not Often***				

When comparing the average mishandling score between both groups after the environment switch (Table 27), the VIM sections showed a significantly higher average than the IDLE section *($p < 0.05$)*. This indicated that the VIM sections mishandled IDLE less often than the IDLE section did with VIM.

Table 27. Tool Mishandling Results *(after environment switch)*
Avg = Average; SD = standard deviation

Section	Tool	N	Avg	StdDev
IDLE	VIM	13	3.08	1.75
VIM	IDLE	39	4.41	1.58
*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Absolutely Often** to **7 = Absolutely Not Often***				

*For further details about these results and other attributes measured during the usability assessment, see our paper published in the **Proceedings of the Human Factors and Ergonomics Society 56th Annual Meeting*** [7].

4.2.2.2 Protocol Analysis

This assessment was conducted during the week of the environment switch. The structure of this assessment allowed for the collection of both qualitative data and first-hand information about the CS1 students' mental model for programming. The objective was to determine whether certain features within these respective environments could shape the students' mental model for programming. The selection process for this assessment was based on random volunteers.

There were seven students who volunteered to participate in this study (all from non-honor sections); four were enrolled in the VIM sections and three were registered in the IDLE section. The same programming assignment was given to each student. Table 28 provides background information about each student. Similar to the assignment given during the CS1 lab study, the students had to write a program that converted 700 days into y years, m months, and *d* days remaining. A video camera was used to record the behavior of each student while completing this assignment. During the recording, each student had to "think aloud" about their approach for writing this program using their new environment. Each student was given 30 minutes to complete the assignment.

Table 28. Subject Background Information

**Student #4 was in the IDLE section but chose to use VIM in the course; **Student #6 was repeating the CS1 course;*

Student	Gender	Ethnicity	Prior Programming Experience	Environment (after switch)
S1	M	Caucasian	None	IDLE
S2	M	Caucasian	HTML	VIM
S3	M	Caucasian	HTML	VIM
S4	F	African American	None	IDLE**
S5	F	Caucasian	None	IDLE
S6	F	African American	VIM*	VIM
S7	M	African American	VI, C++, Java, Fortran	VIM

Each student who used VIM in this study (original IDLE users) indicated prior exposure to some form of programming before taking CS1. Each student from the VIM sections indicated otherwise. After conducting this assessment, the results showed that the students from the VIM sections had less challenges with using IDLE. Two of these particular students completed their assignment within the allotted time. The other two students' inability to complete the assignment was due to the difficulty of the assignment rather than IDLE. The three students from the IDLE section were not able to complete the assignment due to the challenges of using and understanding the VIM command editor. Table 29 provides a summarized description of the subjects' behavior during assessment.

Table 29. Subject Behavior

Student	Completed Assignment *YES*	*NO*	Reason for NOT Completing Assignment
S1	X		
S2		X	S2 spent the entire time trying to understand the functionality of the VIM editor.
S3		X	S3 spent most of her time trying to understand the functionality of the VIM editor.
S4		X	S4 struggled with understanding how to approach the assignment; She encountered several syntactical errors and struggled with correcting them.
S5	X		
S6		X	S6 struggled with understanding how to approach the assignment; She encountered semantic errors, which was due to her inability to determine the appropriate conversions for her program.
S7		X	S7 spent most of his time trying to understand the functionality of the VIM editor.

Another notable observation from this assessment relates to the subjects' tendency of reverting back to familiar procedures from their original tool if they felt lost or confused while using the new one. For example, the recording showed two of the original IDLE users attempting to use the menu bar of the command terminal assuming that VIM possessed relative features to IDLE. One of the original VIM users began using the command terminal to interpret her program when she felt unsure about performing this procedure in IDLE, but managed to complete this assignment.

We concluded from this assessment that feature sets in programming environments could play a role in shaping a novice's perception of programming. This study also showed that visual environments could potentially enable students to develop an inaccurate depiction of programming. For further detail about the results from this assessment, see our paper published in the ***Proceedings of the 50th Annual ACM Southeast Conference*** [6].

4.2.3 Discussion

Students from the IDLE section showed a significant decrease in their ability to use a different tool after being exposed to IDLE. However, students from the VIM sections showed a slight increase in their ability to use a different tool after their exposure to VIM. After switching environments, the mean score for mishandling tools in the VIM sections remained significantly higher than the IDLE section. These results also support the findings from the protocol analysis. Participants from the IDLE section found it more challenging to transition to a command line tool after using IDLE, while students in the VIM sections had a better transitioning to a visual tool after exposure to VIM.

5. CONCLUSION

The objective of this article was to study visual environments and their potential effect on students who are learning to program. Prior studies have shown that visual environments can have both productive and unprofitable effects on a student's ability to become accustomed to programming. From our studies, it was shown that visual environments could provide students with a lower learning curve for operation, while having the potential of placing limitations on their mental depiction of programming.

In the first study, the familiarity of features in IDLE and PyScripter possibly played a role in lowering the learning curve for the students in the CS1-lab course. By the same token, some of these features may have placed a limitation on the skills that the IDLE students in the CS1 course acquired during the second study. Table 30 summarizes the outcomes from both studies.

The question remains of whether visual environments are "ideal" for teaching students how to program. Even though prior studies have shown visual environments to promote student retention [15], positive attitudes [9], and motivation [11] during exposure, our findings show that these environments may also cause students to develop a faulty mental model for programming. These results also support Chen and Marx's reasoning for moving their students from an IDE to command line programming [2]. Certain visual environments may be too restrictive for learning specific programming concepts and procedures. In this case, it may be necessary for students to be exposed to other programming environments that are more inclined to round out their skill sets.

As an alternative solution, it may be appropriate to train students to understand the implied behavior of visual environments. For instance, students may need to receive appropriate training for understanding programming procedures before being exposed to a

Table 30. Study Outcomes

	Outcome	Reason
Study 1	Visual environments can initially impose a lower learning curve	The IDLE group completed their programming tasks significantly faster than their counterparts who used Notepad despite having less prior experience and a lower self-efficacy for programming. Students in the PyScripter and Notepad groups had more prior programming with using IDEs and command line environments respectively, however the PyScripter group completed their programming tasks significantly faster.
	Outcome	**Reason**
Study 2	Visual environments may impose a greater challenge for a student to directly transition to a command line environment	From the usability assessment, it was found that the students from the IDLE section showed a significant decrease in their ability to use VIM after being exposed to IDLE. From the protocol analysis, it was found that all of the IDLE participants were unable to complete their tasks due to struggling with using and understanding the VIM editor.

visual environment. By understanding these underlying factors, it may be possible for a student to avoid the acquisition of a faulty mental model for programming while also being able to make a smoother transition to other types of environments.

5.1 Threats to Validity

There are potential threats that could affect the validity of our findings from these studies. One threat is the finite set of environments that were evaluated during these assessments. Every visual environment that is used to teach programming was not evaluated during these studies. Instead, our studies were conducted while using theories, prior conclusions, and anecdotal evidence as point of references. Another threat relates to the short-term duration of the CS1 Lab study. This particular study was only composed of a one-day assessment. A third issue relates to the low students samples during the latter assessments in the CS1 lecture course. As previously mentioned, there were students who stopped attending class, dropped the course, or showed agitation toward participation in this study due to the repeated assessments.

5.2 Future Work

One future work is to improve student participation during these empirical assessments. This could be done by adjusting the amount of instruments employed during a study to obtain a high number of responses at a consistent level. A related future work is to assess students at particular times of the semester when the attendance rate tends to be high on a consistent basis.

Another area of future work relates to the actual programming environments. Some of the environments used during the CS1 lab and lecture studies consisted of tools primarily for Python programming. A primary future work is to apply evaluations to environments outside of the Python language.

6. FUNDING SOURCE

This work was conducted independent of any financial support.

7. REFERENCES

[1] Beaubouef, T. & Mason, J. (2005). Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *SIGCSE Bulletin, 37(2)*, 103-106.

[2] Chen, Z. & Marx, D. (2005). Experiences with Eclipse IDE in programming courses. *Journal of Computing Sciences in Colleges, 21(2)*, 104-112.

[3] Crosby, M. E. & Stelovsky, J. (1990). How Do We Read Algorithms? A Case Study. *Computer 23(1)* 24-35.

[4] Depasquale, P. J. (2003) Implications on the Learning of Programming Through the Implementation of Subsets in Program Development Environments. *Doctoral Thesis. UMI Order Number: AAI3095195., Virginia Polytechnic Institute and State University.*

[5] Dillon E., Anderson M., & Brown M. (2012). Comparing Feature Assistance Between Programming Environments and Their Effect on Novice Programmers. *Journal for Computing Sciences in Colleges, 27(5),* 69-77.

[6] Dillon E., Anderson M., & Brown M. (2012). Comparing Mental Models of Novice Programmers when using Visual and Command Line Environments. *In Proceedings of the 50th Annual ACM Southeast Conference,* 142-147.

[7] Dillon E., Anderson M., & Brown M. (2012). Studying the Novice's Perception of Visual Vs. Command Line Programming Tools in CS1. *In Proceedings of the Human Factors and Ergonomics Society 56th Annual Meeting, vol. 56(1),* 605-609.

[8] Guzdial, M. (2004). Programming environments for novices. *In Computer Science Education Research. S. Fincher and M. Petre (Eds.). Swets and Zeitlinger. Chapter 3.*

[9] Hagan, D., & Markham, S. (2000). Teaching Java with the BlueJ environment. *In 17th Annual Proceedings of Austrailian Society for Computers in Learning in Tertiary Education.*

[10] Kelleher, C. & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys. 37(2),* 83-137.

[11] Kelleher, C. Pausch, R., & Kiesler, S. (2007). Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems,* 1455-1464.

[12] Lewis, M. (2010). How programming environment shapes perception, learning and goals: logo vs. scratch. In *Proceedings of the 41st ACM technical symposium on Computer Science Education,* 346-350.

[13] Maloney,J., Peppler K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: urban youth learning programming with scratch. *SIGCSE Bulletin, 40(1),* 367-371.

[14] McWhorter, W. I.& O'Connor, B. C. (2009). Do LEGO® Mindstorms® motivate students in CS1*?. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education.* 438-442.

[15] Moskal, B., Lurie, D. & Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. *In Proceedings of the 35th ACM Technical Symposium on Computer Science Education,* 75-79.

[16] Ramalingam, V. & Wiedenbeck, S. (1997). An empirical study of novice program comprehension in the imperative and object-oriented styles. *In Papers Presented At the Seventh Workshop on Empirical Studies of Programmers*,124-139.

[17] Sharp, H., Rogers, Y., & Preece, J. (2007). *Interaction Design: Beyond Human-Computer Interaction* . Hoboken, NJ: John Wiley & Sons Inc.

[18] Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers, 11(3),* 255-282.