

**A STUDY ON THE EFFECT OF A PEDAGOGICAL PROGRAMMING  
ENVIRONMENT ON NOVICE PROGRAMMERS**

by

**KIMBERLY GRIGGS**

**A THESIS**

**Presented to the Department of Computer  
and Information Science  
and the Graduate School of the University of Oregon  
in partial fulfillment of the requirements  
for the degree of  
Master of Science**

**June 2006**

“A Study on the Effect of a Pedagogical Programming Environment on Novice Programmers,” a thesis prepared by Kimberly Griggs in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science. This thesis has been approved and accepted by:



\_\_\_\_\_  
Michal Young, Chair of the Examining Committee

\_\_\_\_\_  
June 2, 2006

Date

Committee in Charge:      Dr. Michal Young, Chair  
   Dr. Virginia Lo

Accepted by:




\_\_\_\_\_  
Dean of the Graduate School

© 2006 Kimberly Griggs

An Abstract of the Thesis of  
Kimberly Griggs for the degree of Master of Science  
in the Department of Computer and Information Science  
to be taken June 2006

Title: A STUDY ON THE EFFECT OF A PEDAGOGICAL PROGRAMMING  
ENVIRONMENT ON NOVICE PROGRAMMERS

Approved



Michal Young

Learning to program is hard. Pedagogical tools are designed with the goal of helping people learn to program. The designers of these tools face conflicting goals: making it easier for students to start programming, and giving students enough information to ease the transition to a general-purpose language. Little data is available on the effect pedagogical tools have on the programmer who was introduced to programming through a teaching tool and then wants to transition to a general-purpose language. In particular, it is not clear how closely teaching languages should resemble general languages to best prepare those students. This research is a first step in addressing these issues. In this thesis, we present VisualJava (VJ), a multi-paradigm programming environment. We empirically investigate how VJ compares to the traditional approach to introducing

programming and in particular how different language representations effect several factors that affect a programmer's transition success.

## CURRICULUM VITAE

NAME OF AUTHOR: Kimberly Griggs

PLACE OF BIRTH: Anchorage, Alaska

DATE OF BIRTH: 04/10/1974

### GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon

### DEGREES AWARDED:

Master of Science in Computer and Information Science, 2006, University of Oregon

Bachelor of Science in Computer and Information Science, 2004, University of Oregon

### AREAS OF SPECIAL INTEREST:

Software Engineering  
Human-Computer Interaction  
Pedagogical Tools

### PROFESSIONAL EXPERIENCE:

Graduate Teaching Fellowship, Department of Computer and Information Science, University of Oregon, 2004 - 2006

Business System Analysis Intern, NIKE, Beaverton, Oregon, 2005

Programmer Intern, BuzzMonkey Software, Eugene, Oregon, 2004

## ACKNOWLEDGMENTS

I wish to express sincere appreciation to Professors Young and Lo for their assistance in the preparation of this manuscript and their support through-out my graduate studies. I want to thanks the professors, the office staff and everybody else who made my time at the U of O wonderful. I would also like to thank those who took the time to edit this manuscript and those who took part in this study.

Finally, I also thank the members of my family and friends for leaving me alone long enough so I could finish this project. And a special thanks to my partner, Hugh Salkind, for supporting me in so many ways through-out my education.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION.....	1
1.1 Research Questions.....	2
1.2 Motivation.....	4
1.2.1 Consequences to US Job Market.....	5
1.2.2 Barriers to learning Programming.....	6
1.2.2.1 Mechanical Barriers.....	6
1.2.2.2 Sociological Barriers.....	8
1.2.2.2.1 Interest in Programming.....	8
1.2.2.2.2 Motivation to learn Programming.....	9
1.3 Hypotheses.....	10
II. RELATED WORK.....	13
1.1 Teaching Systems.....	13
1.1.1 Simplify Textual Programming Approach.....	14
1.1.1.1 DrJava.....	14
1.1.1.2 ObjectDraw.....	15
1.1.2 Provide Alternatives to Typing Approach.....	15
1.1.2.1 Alice.....	16
1.1.2.2 Leogo.....	16
1.1.3 Helping Students Visualize Programs.....	17
1.1.3.1 ToonTalk.....	17
1.1.3.2 Karel the Robot.....	18
1.2 Empowering Systems.....	18
1.2.1 Programming by Demonstration.....	19
1.2.1.1 Stagecast.....	19
1.2.2 End-Goals for Programming.....	20
1.2.2.1 Pinball Construction Set.....	20
1.3 Our Contribution.....	20
III. VISUALJAVA.....	22
1.1 User-Centered Design.....	24
1.1.1 Pre-Design Stage.....	26
1.1.2 Design Stage.....	30
1.2 Pedagogy Rationale.....	31
1.2.1 Educational content .....	35
1.2.2 Language Paradigms.....	37
1.2.2.1 Text-based Language.....	39
1.2.2.2 Iconic Language.....	39
1.2.2.3 Direct manipulation Language.....	40



Chapter	Page
1.3 Authoring Programs in VJ.....	40
1.3.1 Introduction to the VisualJava Environment.....	41
1.3.1.1 Step 1: Setting the Scene.....	42
1.3.1.2 Step 2: Scripting.....	47
1.3.2 The User Interface.....	54
1.3.2.1 The Direct Manipulation Editor.....	54
1.3.2.1.1 Canvas .....	55
1.3.2.1.2 Objects.....	57
1.3.2.1.3 Events.....	58
1.3.2.1.4 Pause Animation.....	59
1.3.2.2 The Icon Editor.....	60
1.3.2.2.1 Details Window.....	61
1.3.2.2.2 Variable Panel.....	61
1.3.2.2.3 Method panel.....	63
1.3.2.2.4 Method Editor panel.....	64
1.3.2.3 The Editor Panel.....	65
1.4 System Implementation.....	67
1.4.1 System Architecture.....	67
1.4.1.1 ObjectDraw Classes.....	70
1.4.1.2 VJ to ObjectDraw Glue.....	71
1.4.1.3 VJ Classes.....	71
1.4.1.4 User Scripts.....	72
1.4.2 Code Generation.....	73
1.4.3 Viability.....	77
IV. EMPIRICAL STUDY.....	80
1.1 Objective.....	81
1.2 Methodology.....	81
1.2.1 Participants.....	84
1.2.2 Procedure.....	85
1.2.3 The Traditional Approach.....	90
1.2.3.1 The Traditional group.....	91
1.2.4 The VJ Approach.....	92
1.2.4.1 The VJ Groups.....	92
1.3 Evaluation Results.....	94
1.3.1 Evaluation Implications.....	105
V. CONCLUSION.....	108

APPENDIX

A. STUDY ARTIFACTS.....113

B. PROGRAMMING TASKS .....123

C. PROTOCOL.....129

BIBLIOGRAPHY.....135

## LIST OF FIGURES

Figure	Page
1. Adding an event in Design View with the Event Builder dialog box.....	43
2. Adding an Object with the Object Panel .....	44
3. Naming the New Object.....	44
4. The Canvas's Code View .....	45
5. An Object's Pop-up Menu of Actions .....	45
6. Selecting Parameters for a pre-defined method .....	46
7. The Object's Details Panel .....	46
8. Naming a New Method.....	49
9. The Method Builder Panel .....	50
10. Making a new variable with the Variable Builder dialog box.....	50
11. Adding the variable to the method and setting it's value.....	51
12. Making a while loop expression with the Expression Builder.....	51
13. Adding a method to the while loop and setting it's parameter.....	52
14. Ending the while loop.....	52
15. Editing the Method in the iconic and code editor.....	53
16. The code view of the Square Object with a user defined method.....	53
17. MovingSquare.java after one mouse click and after another mouse click.....	54
18. Adding a Pause.....	56
19. The method editor.....	62
20. VJ System Architecture.....	67
21. VJ square object's class hierarchy.....	68

## LIST OF TABLES

Table	Page
1. User Analysis Questionnaire .....	28
2. Usability Goals for VJ.....	29
3. Pedagogic Goals for VJ .....	29
4. Nielsen's Top Ten Usability Heuristics .....	32
5. VJ Objects' built-in methods .....	58
6. VJ's built-in events .....	59
8. Generated Code Fragment: Main.java .....	76
9. Generated code fragment: Square.java .....	78

## LIST OF GRAPHS

Graph	Page
1. Comparison of concept comprehension scores ( VJ vs control groups) .....	94
2. Trend of concept comprehension scores of the VJ and traditional groups .....	96
3. Comparison of number of programming tasks and errors produced (VJ vs traditional groups) .....	97
4. Trend of concept comprehension scores (VJ vs VJ/Text groups) .....	99

## CHAPTER I

### INTRODUCTION

Since the 1960's researchers have built numerous systems and languages with the intention of making programming more accessible to people of all ages. Pedagogical tools, like simplified IDE's, special libraries, and microworlds, have been designed with the goal of helping people learn to program. Most of these tools focus on simplifying the mechanics of programming by removing syntax, making the language more human-centered, or exploring alternatives to text-based languages. The designers of these tools often juggle potentially conflicting goals: making it easier for students to start programming, and giving students enough information to ease the transition to a general purpose language [Kelleher & Pausch, 2005].

While these tools certainly make the process of learning to program less frustrating and improves a beginner's first programming experience, there is little evidence that the students have learned anything that they would not have from a more traditional approach. Specifically, little data exists on the impact of pedagogical tools on the student who was introduced to programming through a teaching tool and then wants to transition

to a general-purpose language (we will call them “intermediate programmers” for the rest of the thesis).

Many pedagogical tools use languages that are similar to the prevalent general-purpose language of the time. The designers of these tools make this choice to ease the transition from teaching language to general language. But, general-purpose languages were not designed for teaching and there is scant solid evidence on the most effective way to alter a general language for teaching purposes. While it may seem obvious that students need to understand the parallels between the programming constructs in teaching and general-purpose languages, it is not clear how closely a teaching language must resemble general languages.

Current research in pedagogic tools has left two important questions unanswered: how do pedagogic tools compare to the traditional approach to introducing programming and in particular how do language paradigms effect factors that affect a programmer's transfer knowledge to a general language.

## **1.1 Research Questions**

To investigate these questions I have implemented a pedagogical programming environment, VisualJava (VJ), which introduces students to introductory programming concepts through a mixture of three alternative programming paradigms: a direct manipulation language, an iconic language, and the text-based Java language. Our goal is provide an environment in which students can learn the kinds of problem-solving skills and the necessary concepts and skills for creating computer programs.

In this thesis, I empirically evaluate how VJ compares to a more traditional approach to introducing programming, in terms of program completion rate, numbers of errors, and concept comprehension. In addition, we examine how VJ's multiple language paradigms affect a programmer's concept and syntactic comprehension. The overall research questions are:

*RQ1: What effect does VJ have on intermediate programmers in terms of program completion rate, interest level, and concept comprehension compared to the traditional approach to introducing programming?*

*RQ2: How do different language paradigms affect novice programmers' concept comprehension and syntactic understanding?*

We hypothesize that first exposure to programming through VJ will be equally as successful as the traditional approach to introducing programming and that a language representation that more closely resembles a general language will increase syntactic understanding but have little effect on concept comprehension.

The research that has gone into exploring the above issues has been based on a two-pronged strategy:

**Usability Engineering:** Gather data from members of the target audience to engineer a pedagogical tool, VisualJava, through a user-centered approach. This will create a product that provides good usability and useful utility for the target audience, novice programmers.

**Empirical Evaluation:** Evaluate the hypotheses, by directly observing the target audience interact with the system and through the use of entrance/exit surveys. This will



uncover mistakes and design errors in the prototype's implementation and provide qualitative and quantitative data to evaluate our hypotheses.

One of the first questions that need to be answered when designing and evaluating a pedagogical tool is: what is wrong with the current approach to teaching introductory programming? It is my belief that, at the moment, the way in which programming is taught and learned is fundamentally broken. If the current approach no longer works, then something needs to change.

## 1.2 Motivation

Computer Science introductory programming courses (also known as CS1) are notoriously difficult for students. Research shows that first-year programming courses suffer an average failure rate between 30 to 40% worldwide, while other introductory science courses, such as physics and biology, experience an average failure rate of 15% [Kassboll, 2000]. Even though CS1 course listings claim that no programming experience is necessary, students often find themselves inadequately prepared. To succeed in CS1, students not only need to understand challenging programming concepts, they must also learn the mechanics involved in writing, testing, and debugging programs.

It may be tempting, and even ego-stroking, to believe that CS1 failure rates can be chalked up to “weeding out the weak.” It is sometimes argued that the students who find programming difficult are simply and solely those for whom programming *is* difficult. The problem with this school of thought is that then, only a small unrepresentative segment of our population can create new computer software. This is troubling for two reasons: 1) as the needs for technology continues to grow, we may be unable to fill

computer-related jobs and 2) if it is possible that "aptitude" for programming does not exist, then there must be other factors that affect the success of novice programmers.

### ***1.2.1 Consequences to US Job Market***

According to the US Department of Labor, 8 of the 10 fastest growing occupations between 2000 and 2010 will be computer related (Hecker, 2001). The Information Technology Association of America reported that in 2002, after the dot com bubble had burst, companies were unable to fill 425,000 computer-related jobs due to the lack of qualified applicants (ITAA 2001). Unless we can increase the numbers of students graduating with Computer Science degrees, the lack of qualified applicants for computer-related jobs may have serious consequences for US businesses.

It is important to not only increase the numbers of people qualified to enter computer related jobs, but also to make sure that those qualified people are a representative sample of today's students. To get a larger, broader group of people to enter the field of computer science, we need to get a larger, broader group of people succeeding at the first steps towards pursuing a career in computer science. That first step is learning to program.

An ACM task force describes the relationship between computer science and programming in the following way:

*While programming is a central activity to computer science, it is only a tool that provides a window into a much richer academic and professional field. That is, programming is to the study of computer science as literacy is to the study of literature (Tucker, Deek et al. 2002).*

Learning to program provides students with the basic skills necessary to pursue computer science.

### **1.2.2 *Barriers to learning Programming***

Few students find learning to program easy. There are many factors at work. Some are simply inherent in the subject while others have more to do with the modus operandi of teaching departments. Others are deeply interlinked with the expectations, attitudes, and previous experiences of the students and their peers.

If students struggle to learn something, it follows that this thing is for some reason *difficult* to learn. Two barriers present themselves as possibilities that might make learning to program difficult – mechanical and sociological.

#### **1.2.2.1 Mechanical Barriers**

"Programming" is a complicated business. An experienced programmer draws on many skills and much experience. Some of the skills required bear little obvious relevance to the process of producing program code. Some of the required skills are obvious; problem solving ability and some idea of the mathematics underlying the process are essential. But there are more. A programmer must be able to use the computer effectively, must be able to create the program in a file, compile it, and find the output. The program produced must be tested, and bugs found and corrected. The more time a student spends wrestling with the mechanics of programming, the less time they spend learning the concepts.

Programming, then, is not a single skill. It is also not a simple set of skills; the skills form a hierarchy, and a programmer will be using many of them at any point in

time. A student faced with learning a hierarchy of skills will generally learn the lower level skills first, and will then progress upwards [Bereiter & Ng, 1991]. In the case of coding (one small part of the skill of programming) this implies that students will learn the basics of syntax first and then gradually move on to semantics, structure, and finally style. This approach to learning is often reinforced by lectures that concentrate on the minutiae of syntax, and by textbooks that adopt the same approach.

Programming is not only more than a single skill; it also involves more than one distinct process. At the simplest level the specification must be translated into an algorithm, which is then translated into program code. The difficulty in learning to program for some students lies in "putting the pieces together", i.e., in designing the algorithm to solve the problem, and then programming the correct statements to accomplish the goal [Soloway, 1986]. Therefore, a student must master three distinct processes. Teaching (and learning), however, can concentrate on the low level issues of syntax at the expense of the higher level, more complex, process of designing an algorithm.

Programming is a new subject for many of the students who take programming courses. In his classic article on teaching programming, Dijkstra argues that learning is a slow and gradual process of transforming the "novel into the familiar" [Dijkstra, 1989]. He goes on to suggest that programming is what he terms a "radical novelty" in which this comfortable tried and tested learning system no longer works. The crux of the problem is, according to Dijkstra, that radical novelties are so "disturbing" that "they tend to be suppressed or ignored to the extent that even the possibility of their existence ... is more often denied than admitted".

A particular feature of programming (and one that reinforces Dijkstra's message) is that it is "problem solving intensive" [Perkins, et al, 1988] – it requires a significant amount of effort in several skill areas for often a very modest return. At the same time it is "precision intensive" [Perkins, et al] – the modest success that can be achieved by a novice programmer requires a very high level of precision, and certainly a much higher level than most other academic subjects. Dijkstra also notes that the "smallest possible perturbation" in a program of one single bit can render a program totally worthless. This is precision indeed.

To arrive in a setting where students are confronted with a totally new topic that does not respond to their habitual study approaches, and where a single semi-colon is the difference between glorious success and ignominious failure, surely represents a "radical novelty" in Dijkstra's terms. It is perhaps not study skills that novice programmers need, but coping skills.

### **1.2.2.2 Sociological Barriers**

Pure difficulty is not the only barrier that students face when learning programming. There are a variety of sociological factors that can keep students from succeeding in their CS1 course.

#### ***1.2.2.2.1 Interest in Programming***

Learning (or perhaps here "being taught") programming can be very dull. For example, in most computer science programs, the first few years are spent learning how to use different programming constructs and data structures typically in pursuit of completion of programming tasks with very little practical value or importance. A

program that displays the 50th number in the Fibonacci sequence or simulates the line at a bank using a queue is not very exciting to most students.

Students typically spend several years before they reach a level at which they can use programming for applications that interest them. Students who are not interested in what he/she can use programming to accomplish will not be especially motivated to learn programming. Learning scientists have found over and over again that engaging students is critical to deep learning [Soloway, 89].

#### **1.2.2.2.2      *Motivation to learn Programming***

Students who do the best in CS1 courses often have a genuine interest in programming (intrinsic motivation) versus their colleagues who see programming as a means towards a lucrative career (extrinsic motivation) [Sheard & Hagan, 1998]. Intrinsic motivation can be instilled in a novice programmer by demonstrating that programming can be an enjoyable, creative activity. Yet most CS1 courses and textbooks predominately feature programs that focus on text input/output or mathematical procedures that can seem pointless and unappealing to the average CS1 student. The problem is Computer science educators are using an outdated view of computing and students.

College students today have been called the “Nintendo generation” or the “MTV generation.” [Soloway, 89] Their perception of technology and media has been profoundly influenced by these sources. The implication has often been that they need to consume mass quantities of fast-paced sound, graphics, and animation. Perhaps there’s a more critical implication—that these are the kinds of media that Nintendo generation

students want to produce when learning computer science. Unfortunately most professors are not skilled graphic programmers and are more comfortable with teaching what they know. Often, educators teach computer science in much the same way as they learned it. But, computing is much different today. Animation of program execution can be used to help the student “put the pieces together.” Visualization is one approach to assisting the learner in finding out what task each piece can be expected to perform and how the pieces work together to perform the over all task of solving the problem at hand.

We have used “Hello, World!” for the past 25 years because text was the medium that was easiest to manipulate with the given technology. Today’s technology can manipulate sound, graphics, and video with the same responsiveness and ease. Today’s technology produces the media that students are consuming. These same students can produce their kind of media using today’s technology. In fact, they want to. And they’ll be motivated to learn programming to do it.

### **1.3 Hypotheses**

Ideally, we would like to be able to show that pedagogical tools, like VisualJava, are a good choice for introducing programming and have a positive effect on factors that affect a programmer's transition to a general-purpose language. We would like to provide statistical evidence that many students would greatly benefit from being introduced to programming through a pedagogical tool and that these tools do reduce many of the barriers to learning programming. We would also like to weigh in with quantitative data on how closely a teaching language should resemble a general-purpose language. But to truly answer these questions and obtain this data we would need to perform multiple

longitudinal studies and follow hundreds of students through their first few years in a computer science program. Unfortunately, to collect these kinds of numbers is currently unrealistic; I cannot wait three years. Also, since this is fairly early work, I need to be able to determine quickly whether or not VJ is successful at introducing programming concepts. Rather than tracking student's success with respect to computer science for many years, we will instead show through qualitative and quantitative methods that first exposure to programming through VJ will be equally as successful as the traditional approach to introducing programming and that a language representation that more closely resembles a general language will increase syntactic understanding but have little effect on concept comprehension. Because we will have limited time to work with the students in VJ, we will be looking at instant concept comprehension instead of longitudinal comprehension. Specifically, I will attempt to establish the following five hypotheses:

1. Novice programmers show an improvement in concept comprehension after using VJ.
2. Novice programmers have a more positive attitude towards Computer Science and programming after using VJ.
3. Intermediate programmers perform as well as traditional programmers in terms of program completion rate and numbers of errors when attempting to program in a general language, and experience concept comprehension equally.
4. Intermediate programmers exposed to a teaching language that more closely resembles a general-purpose language have fewer syntax errors when attempting to program in a general language.
5. Intermediate programmers exposed to a teaching language that does not closely resemble a general-purpose language experience the same, if not better, concept comprehension.



The rest of this thesis is structured as follows: Chapter II provides related work information and our contribution to this area of research. Chapter III describes the design and implementation of VJ. Section IV discusses the empirical study and results. Finally, Section V concludes the thesis.

## **CHAPTER II**

### **RELATED WORK**

Over the past 40 years, many researchers have attempted to make programming accessible to a broader range of people. Researchers have built many different tools that take a myriad of approaches to reduce the barriers to learning programming. While it is not reasonable to discuss all of the tools here, I will briefly summarize the main approaches that researchers have used in attempting to make programming accessible and give more detail about tools that have most influenced the design of VJ.

#### **1.1 Teaching Systems**

Learning to program consists of two main tasks: learning to form a syntactically correct program and learning to use logic and control structures to solve problems. A novice learning a general purpose language like C++ or Java must tackle these two challenges simultaneously; most teaching systems for novices attempt to simplify or provide additional support for one of these two tasks. The general philosophy behind these systems is that if the process of learning to program is less painful for novices, more people will succeed in learning to program. The majority of the systems discussed

in this section address the mechanics of programming: both expressing intentions to the computer and understanding the actions of the computer [Norman, 1986].

### ***1.1.1 Simplify Textual Programming Approach***

Textual programming languages for novice programmers often remove any unnecessary syntax, limit the size of the language, and choose understandable names for commands. A typical language in this category has a small number of carefully chosen constructs and commands and fewer syntactic requirements (e.g. parenthesis, semicolons, commas, etc) than most general-purpose languages. [Kolling, M., Quig, B., Patterson, A., & Rosenberg, J., 2003 & 1996a], [Hsia, J. I., Simpson, E., Smith, D., & Cartwright, R., 2005]

#### **1.1.1.1 DrJava**

DrJava is a pedagogic programming environment for Java that enables students to focus on designing programs, rather than learning how to use the environment. The environment provides a simple interface based on a “read-eval-print loop” that enables a programmer to develop, test, and debug Java programs in an interactive, incremental fashion. DrJava introduces students to Java through language levels, a hierarchy of sub-languages of Java, that shield students from the full complexity of the language while still allowing them to focus on learning to write programs. [Allen, E., Cartwright, R., & Stoler, B., 2002.]

### 1.1.1.2 ObjectDraw

The ObjectDraw library was developed to support teaching Java to novice programmers. It is designed to support an “objects from the beginning” approach to CS 1. It supports truly object-oriented graphics and makes it possible to incorporate event-driven programming techniques from the beginning. Graphical objects serve as both excellent examples of objects and provide visual feedback that makes it easier for students to determine the effects of their code and to detect errors in their programs. The graphical objects provided in the ObjectDraw library also have the advantage that results of the creation and modification of objects appear immediately on the screen without the need to invoke a paint or repaint method. [Bruce, K. B., Danyluk, A., & Murtagh, T. 2001a-b. & 2005b.]

### *1.1.2 Provide Alternatives to Typing Approach*

Despite the attempts to make programming languages simpler and more understandable, many novices still struggle with syntax: remembering the names of commands, the order of parameters, whether or not they are supposed to use parentheses or braces, etc. Rather than starting with a textual programming language and attempting to improve it, these systems attempt to provide novices with non-textual methods for creating programs. There are two common approaches. The first approach allows novices to connect graphical or physical objects representing elements of a program such as commands control structures or variables. By moving and combining graphical or physical objects, novices can create programs. The second approach allows novices to create programs by selecting from a concrete set of actions in an interface, often by

pushing a sequence of buttons or filling in a form. [Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., & Doyle, K., 1988], [Sheehan, R., 2004], [Tanimoto, S. & Runyan, M., 1986]

#### **1.1.2.1 Alice**

Alice is a programming system for building 3D virtual worlds, typically short animated movies or games. In Alice users construct programs by dragging and dropping graphical command tiles and selecting parameters from drop-down menus. Students can also add standard programming control structures such as if-statements and loops by dragging *if* and *loop* tiles from the tool bar. Unlike many no-typing programming systems, Alice allows students to gain experience with all of the standard constructs taught in introductory programming classes in an environment that prevents them from making syntax errors. [Conway, M., Audia, S., Burnette, T., Cosgrove, D., & Christiansen, K., 2000] , [Cooper, S., Dann, W., and Pausch, R., 2000.]

#### **1.1.2.2 Leogo**

Leogo is a system that produces drawings similar to the Logo turtle [Papert, 1980]. However, rather than concentrating on one method for creating programs, it provides three: a typed syntax similar to Logo, a direct manipulation interface in which the turtle is dragged around and his actions are recorded, and an iconic language which contains templates for defining structures and using common turtle commands. Motions are expressed in all code styles simultaneously; when the turtle is dragged forward 15 units, the text window shows forward 15, and the iconic window shows forward 15 in

icons so it is possible to learn some of the iconic and typed languages using direct manipulation.[Cockburn, A. & Bryant, A, 1997 & 1998]

### ***1.1.3 Helping Students Visualize Programs***

Most program execution is invisible: values in variables change and perhaps the program occasionally prints to the screen, but the steps the program takes. But, if the program does not behave as the programmer anticipated, it can be difficult to determine what is going wrong. The systems in this category try to help students understand what happens during the execution of programs, either by placing programming into a concrete setting so that students can more readily imagine what the execution of a program “looks like” or by providing a simulation to allow students to watch their programs execute.[Fenton, J. & Beck, K., 1989], [Guibert, N., Girard, P., & Guittet, L., 2004], [Shaffer, S. C., 2005]

#### **1.1.3.1 ToonTalk**

ToonTalk has a physical metaphor for program execution. In ToonTalk, cities and the creatures and objects that exist in cities represent programs. Most of the computation takes place inside of houses; trainable robots live inside the houses. Communication between houses is accomplished with birds that carry objects back to their nests. The ToonTalk environment places the user within the city (program). Using interaction techniques commonly found in videogames, users can navigate around the space, pick up tools, and use tools to affect other objects. By entering the thought bubbles of robots and

showing them what they should do using standard ToonTalk tools, users construct programs [Kahn, K., 1996].

### **1.1.3.2 Karel the Robot**

Karel the Robot is one of the most widely-used mini-languages, originally designed for use at the beginning of a programming course, before the introduction of a more general purpose language. Karel is a robot that inhabits a simple grid world with streets running east-west and avenues running north-south. Karel's world can also contain immovable walls and beepers. Karel can move, turn, turn himself off, and sense walls half a block from him and beepers on the same corner as him. A Karel simulator allows students to watch the progress of their programs step by step. The syntax was designed to be similar to Pascal to ease the transition from Karel to Pascal after the first few weeks of an introductory programming course. [Bergin, J., Stehlik, M., Roberts, J., & Pattis, R., 2001 & 1996], [Pattis, R., 1981]

## **1.2 Empowering Systems**

While it is helpful to make the beginning steps of learning to program as painless as possible, it has not been sufficient to drastically change the number or kinds of people that enroll in programming classes or choose to pursue computer-related careers. The difficulty of learning to program is not the only reason that people choose not to pursue programming, either in school or as a career. Some people do not immediately see a compelling reason to learn to program. The systems in this category are built with the belief that the important aspect of programming is that it allows people to build things that are tailored to their own needs. Consequently, the designers of these systems are not

concerned with how well users can translate knowledge from these systems to a standard programming language. Instead, they focus on trying to create languages and methods of programming that allow people to build as much as possible.

### ***1.2.1 Programming by Demonstration***

Many researchers have examined the problem of making languages more understandable and usable for novices. While progress has been made making programming languages more understandable, there still are many barriers for novices trying to build their own programs. The systems in this category examine ways that users can program a system by showing the system what to do through manipulating the interface, without relying on a programming language. [Cohen, P. R., Dalrymple, M., Moran, D. B., Pereira, F. C., & Sullivan, J. W., 1989], [Cypher, A., 1993], [McDaniel, R. G. & Myers, B. A., (1999)], [Smith, D., 1993],[Smith, D. C., Cypher, A., & Tesler, L., 2000]

#### **1.2.1.1 Stagecast**

Stagecast is a programming by demonstration (PbD) environment for creating simulations. Users are presented with a grid-based world in which they can create their own actors. Users define rules for the simulation by selecting a before condition from the grid world and then demonstrating how that condition should change. When the simulation is started, when a section of the grid matches a condition of one of the rules, the rule is applied. [Cypher, A. & Smith, D. C., 1995]



### **1.2.2 End-Goals for Programming**

For students who are not inherently interested in the computer as a machine, it is crucial to answer the question “Why should I program?” A few children’s games and children’s programming systems have attempted to embed programming within a “fun” context. For many of the systems, programming is presented either as a way to solve a particular puzzle or as a way to create simulations and simple games. [Repenning, A., 1993]

#### **1.2.2.1 Pinball Construction Set**

The Pinball Construction Set was written in 1983 to allow users to design and build their own pinball machine simulations. It provided a construction space, a set of pinball parts, and bitmap editing capabilities to allow users to build themed pinball machine simulations. Physical laws and behaviors were written into each part; each part provided could be seen as acting on balls that collide with it in defined ways. In this system, users can program by placing pinball parts in well-defined relationships. For example, users may want to specify that when a ball hits a certain target, it is diverted onto a ramp, and its path affected by a magnet.[Budge, B., 1983]

### **1.3 Our Contribution**

Because of these tools we can now more easily introduce beginners to programming. Most of these systems focus on one of two types of audiences: children and end-users. What about people who want to be more than novice programmers or end users? Rather than focusing on making the process of learning to program less frustrating for novices and

children, I will focus on the intermediate programmers; students who are introduced to programming through a system designed for beginners. Specifically, we contribute evidence on the effect of a pedagogic tool on intermediate programmers in terms of program completion rate, number of errors, and concept comprehension. We also provide data on how pedagogical tools compare to a traditional approach of introducing programming concepts. Because of the reasons we gave in the motivation section, we believe that studying the intermediate programmer is a valuable contribution to the field.

In addition, through VJ's multiple language paradigms we provide data on how different language paradigms *affect* a novice programmer's concept comprehension and syntactic understanding. Addressing this question will provide the field with valuable insight on the impact of presenting issues of syntax and program expression earlier or later in the process of learning programming.

Finally, due to the empirical study (see Chapter IV) we are also contributing valuable data through quantitative empirical methods that can be used by future researchers and educators when designing and evaluating new teaching systems.

## CHAPTER III

### VISUALJAVA

VisualJava (VJ) is a pedagogical programming environment for teaching introductory programming concepts. VJ uses object-oriented concepts as its foundation and teaches event driven programming as a way to motivate and empower its users. VisualJava is designed to supplement, not supersede, the traditional approach to introducing programming. It is my intention that students would use VJ as a tool to prepare them for a CS1 course. It is my belief that an initial introduction to programming through VJ will be beneficial to students who face any of the mechanical or sociological barriers discussed in Chapter I.

VJ's user interface and functionality was designed with a user-centered approach. We applied good user interface (UI) principals to not only the interface, but also to the *process* of programming. The key idea is to combine a powerful, but intuitive, graphical user interface (GUI) with a simplified Java 2D graphics library, ObjectDraw.

VJ introduces students to the mechanics of writing event-driven Java programs through a mixture of three alternative programming paradigms: a direct manipulation language, an iconic language, and the text-based Java language. Its three programming metaphors allow students to choose a method of program expression that best suits their

skill level. To reinforce the equivalence of the languages, VJ supports *user-interface equal opportunity* [Smith, Cypher, & Schmucker, 1996] -the input actions in one programming paradigm cause corresponding outputs in the other two programming paradigms. In addition, students can create their own example programs by using the direct manipulation or iconic language to solve a problem and then view the Java code that VJ generates. This provides a learning-by-doing environment that is beneficial to novice programmers [Mayer, 89].

VisualJava's environment provides error-prevention support, such as forcing students to end loops before moving on (syntax-error prevention) and structured templates for forming correct expressions (semantic-error prevention.) VJ eliminates many of the unrelated skills necessary to start programming and reduces the burden of learning seemingly nonsense syntax. Skilled programmers can also use the tool to enhance their problem-solving skills and algorithmic understanding.

In addition to addressing the mechanical barriers students face, VJ also addresses some of the sociological ones. VJ provides students an environment to make interactive graphics-driven programs immediately. The system comes with a set of actions and objects that can be used by the student so that they can make interesting programs their first time in the environment. The "Hello World" program can be replaced by displaying a graphical object on a canvas. This new version teaches the same concepts the "Hello World" program does, but is visually more gratifying.

This section describes the design, functionality, and implementation of VisualJava in more detail.

## 1.1 User-Centered Design

One of the reasons for creating software systems is to make it possible for people to achieve their goals more efficiently. Designers seek to do this by providing functionality which supports people in tasks for accomplishing those goals. However, producing systems that can help people *effectively* and which are *fully adopted* by them is a complex process. Part of that process involves designing and implementing the system. Another part involves users learning, employing and finally adopting the system. Good system design is key for a system seeking to help users effectively. Critical factors for creating systems with good design are [Norman, 1996]:

a) The system must provide functionality that enables users do what they need.

That is, the system must have good *utility*.

b) The system must be done in a way that facilitates users learning and efficiently exploiting its functionality. That is, the system must have good *usability*.

In this thesis, *utility* is defined as the question of whether the functionality of the system can do what is needed, and *usability*, as the question of how well users can exploit that functionality. Therefore usability applies to all aspects of the system with which a human might interact. Neilson describes usability as mainly composed by five *usability attributes* [Neilson, 1993]:

- a) Learnability.
- b) Efficiency of Use.
- c) Memorability.
- d) Error Handling and Prevention.
- e) User Satisfaction.

In order to practice good design, I decided to follow the user-centered approach to designing systems. The notion of user-centered design involves a whole set of concepts and principles seeking to create products that are highly useful to the users. Much research has been done in the last decade within user-centered design with respect to software systems. Gould and Lewis (1985) give a good description of the main principles behind it:

- Establish an early focus on users and the tasks they perform to achieve their goals.
- Perform empirical measurement of the software's usability using simulations and prototypes.
- Perform cycles of iterative design, where a cycle consisting of design, usability evaluation and redesign should be repeated as many times as necessary.

Nielsen (1993) describes a series of activities that should be followed by people developing a system with the user-centered approach. These activities can be divided into three stages:

1. A *pre-design stage*, which should provide essential concepts to be used through out the lifecycle of the software. This stage comprises the following activities:
  - a) User and task analysis.
  - b) Performing a competitive analysis.
  - c) Setting usability goals.
2. A *design stage*, which guides the design of the user interface. This stage comprises the following activities:
  - a) Including users in design.
  - b) Applying guidelines and heuristic rules.
  - c) Prototyping.
  - d) Performing empirical testing.
  - e) Performing iterative design.

3. A *post-design* stage, in which feedback from the use of the system in the field is collected.

We have so far completed stage one and two of the design process. The following sections describes the activities and results from the pre-design and design stage.

### ***1.1.1 Pre-Design Stage***

*User and task analysis* means analyzing the intended user population of the system in order to learn about them as well as the tasks with which the software is intended to help. User differences and variability in tasks are the two factors with the biggest impact on usability. The concept of users should include all people that will be affected by the software system; these users are the stakeholders of the system. Learning about the tasks should include observing the users' goals, needs, and how they approach tasks. Also, it is necessary to analyze if users' tasks could be improved by the use of the system.

Task analysis centered on users generates concrete statements of representative user tasks that provide good coverage of the functionality that the system should provide and that can be used for usability testing later. Also, task analysis centered on users emphasizes human factors more than traditional analysis by attempting to identify where users waste time or are made uncomfortable. Early focus on users and tasks is one of the most important activities in order to reach high usability.

The intended user population for VJ is easily identified: novice programmers. But other groups of people like, CS1 instructors, Computer Science departments and IT companies, are also stakeholders in the design of an educational programming tool. We

conducted interviews to understand the target population, novice programmers, and their experiences with learning programming.

Students who had recently completed the introductory programming courses at the University of Oregon were interviewed to elicit information about their attitude about CS and their experiences with learning to program. We also asked about coursework and how the assignments affected their success and interest in learning programming. Finally, we asked about their use and opinions of programming environments and languages they used when learning to program. Table 1 contains the information gathering questionnaire.

*Performing a competitive analysis* refers to analyzing similar products according to established usability guidelines in order to notice their strengths and weaknesses. This can provide ideas for the new software. Lewis and Rieman (1993) list some points in favor of using other product's ideas:

- a) It is not easy for developers to come up with design ideas as good as those already implemented in high quality products.
- b) Using ideas from other known products can improve the learning of the intended software because they are more likely to be already known by users.
- c) It can save design time.



### User Analysis Questionnaire

1. **Interest in Computer Science (CS)**
  1. What interest you most about CS?
  2. What interests you the least?
  3. What are the projects you are drawn to?
1. **Attitudes on CS**
  1. What skills do you find necessary to be successful in CS? Do you have them?
  2. Answer True or False:
    1. You have to be a good programmer to be successful in Computer Science.
    2. There are interesting problems to solve in CS
2. **Programming Experience**
  1. What was the first programming language you learned?
  2. What programming language are you the most comfortable programming in?
  3. Do you program for fun (besides assigned work)? What type of applications do you program for fun?
  4. What was harder for you when you started programming: Solving the problem or coding the solution? Explain.
  5. What helped you get better at either solving the problems or coding the solution?
3. **Integrated Development Environments (IDE) examples: textpad, eclipse, Jedit**
  1. What IDE did you first learn to program in?
  2. The following questions ask you to indicate on a scale of 1 to 7 your opinions on the IDE you learned in. For each of the questions, please indicate what you think about your most used DE. 1 indicates "Totally Disagree", 4 indicates "Neutral", and 7 indicates "Totally Agree"
    1. The IDE interface was easy to use
    2. I only used the IDE features that were most apparent
    3. I spent time tinkering with the IDE to learn about the features
    4. I used most of the features the IDE offered
    5. I used the tutorials / help that came with the IDE to learn about the features
  3. Do you have any ideas on IDE features that would make learning how to program easier?
4. **Programming Assignments**
  1. Can you tell me about a programming assignment / project that you really enjoyed? Why?
  2. Can you tell me about a programming assignment / project that you really hated? Why?
  3. Can you tell me about the types of programming assignments in your first CS class?
    1. Did you enjoy those assignments? Why / Why not?
    2. Did they get you excited about CS? Why / Why not?
    3. Were you successful at those first programming assignments?
    4. What helped or hindered your success with those first programming assignments?
  4. Do you have any ideas on what would make for interesting first programming assignments?

Table 1: User Analysis Questionnaire

The intention is not to steal particular copyrighted ideas but rather to take general ideas and try to improve them. The related works section (Chapter II) describes the systems analyzed in the design of VJ.

*Setting usability goals* refers to establishing concrete goals that the system ideally has to comply with before it is released. Many times a trade off exists between different

usability attributes, therefore the goals must be based on the results of the user and task analysis.

The underlying goal in developing VisualJava is to enhance traditional approaches to teaching and learning programming. This objective can be decomposed into two broad categories of pedagogical and usability goals. These categories are not completely orthogonal in that poor usability can mask pedagogical rewards, and few pedagogical benefits can make efforts towards usability irrelevant. Satisfying goals in both categories, however, greatly improves the effectiveness of any educational aid. Table 2 and Table 3 list the usability and pedagogical goals for VisualJava.

- (U1) Easy to learn. Straightforward and efficient to use.
- (U2) Make everything relevant to the interface visible.
- (U3) Establish a cause-effect relationship between user actions and system semantics.
- (U4) Support copying and modifying vs creating from scratch, i.e., allow users to copy and modify existing items in a system as a way to create new ones.
- (U5) Support seeing and pointing vs remembering and typing, i.e., "direct manipulation."
- (U6) Support concrete vs abstract objects in the system.

Table 2: Usability Goals for VJ

- (P1) Increase student understanding of the target domain, i.e., introductory programming concepts.
- (P2) Support different learning abilities, learning styles and levels of knowledge.
- (P3) Motivate and generate interest in the subject matter.
- (P4) Promote active engagement with the tools.
- (P5) Support various scenarios of learning, including examples, tutorials and exploration.

Table 3: Pedagogic Goals for VJ

### 1.1.2 Design Stage

*Including users in design* refers to have some users (a) criticizing developers' designs, (b) capturing problems with current developers' concepts, and (c) contributing with other ideas. This step is usually good at capturing mismatches between users' actual tasks and developers' model of the tasks. Also, users seem to be very good at reacting to designs they don't like or won't work in practice. However as Nielsen describes, "users are not designers" therefore this step consisted of having users contribute their opinions on sketches of the interface design. The ideas and opinions gathered from this free-form usability study provided valuable information. Ideas that matched our usability guidelines were incorporated into the current design of VJ.

*Applying guidelines and heuristic rules* has as its objective to implement well-known principles in the current system design in order to improve its usability. There are many well-known principles and guidelines that can provide improvements in the usability of a system. We followed Jakob Nielsen's top ten usability heuristics, listed in Table 4, while *iteratively designing* VJ.

A key step for developing usable systems is the development of an early *prototype* or simulation of the user interface. Prototypes can range from paper sketches to working programs. VisualJava has evolved from conceptual sketches to a functioning prototype. The design of a large system goes through many revisions and VJ is no exception. The system used for the empirical evaluation presented in this thesis is a *vertical* prototype, a prototype that includes in-depth functionality of only certain parts of the interface.

Once a system design has been decided, it is necessary to *empirically test* it in order to capture usability problems. Testing is usually done using prototypes. One reason

behind a formative evaluation is the concept that it is too difficult to design a complex system and get it correct the first time. Some of the reasons are (a) the current limitations in psychological theory that don't permit people yet to accurately predict users' behavior with the system, and (b) the difficulty for developers at the start of a project to have a complete understanding of the entire context in which a system will be used in the field. The empirical evaluation of VJ is described in Chapter 4.

## 1.2 Pedagogy Rationale

Norman and Spohrer (1996) argue that educational systems need to satisfy three fundamental requirements: engagement, effectiveness and viability. *Engagement* at the interface critically affects student motivation levels. *Effectiveness* concerns the educational content of the environment. *Viability* concerns the practicality of the system: whether it is affordable, extensible, and so on.

The viability of VJ is discussed later in Section 3.7.4. This section focuses on our approach to satisfying the engagement and effectiveness requirement. Object-Oriented Event-Driven Programming

The underlying goal in developing VisualJava is to enhance traditional approaches to learning programming. The text-based language choice for VJ is an important factor in meeting that goal. Little research has been done on whether one language is better for teaching than another. Currently, Java is the most widely taught language in University Computer Science departments.

**Visibility of system status**

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

**Match between system and the real world**

The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

**User control and freedom**

Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

**Consistency and standards**

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

**Error prevention**

Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

**Recognition rather than recall**

Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

**Flexibility and efficiency of use**

Accelerators -- unseen by the novice user -- may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

**Aesthetic and minimalist design**

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

**Help users recognize, diagnose, and recover from errors**

Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

**Help and documentation**

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

Table 4: Nielsen's Top Ten Usability Heuristics

Though, with the current failure rate of CS1 courses, Java is not proving to reduce any of the barriers discussed in Chapter I. The difficulty of choosing a language for VJ becomes how to prepare students for a Java-based course while still meeting the pedagogy goals of the system.

While Java meets some of the guidelines, it is not proving to be effective at meeting P3: motivate and generate interest in learning to program. We believe to do this; students need to be able to create more interesting programs from the start. Graphics and event programming can be important tools in building interesting programs. Most programs students use today are highly interactive. Writing programs that are similar to those they use is both more interesting and more “real” to the students. Also, graphical programs provide students with visual feedback when they make programming errors. The design of interactive graphical programs helps students to both use objects and write methods early while designing and implementing interesting programs. While Java has an extensive graphical library and event-driven capabilities, the complexity of using those libraries is beyond the skill of novice programmers.

The objects first event-based style is increasingly common in modern end-user programming languages. Visual Basic, Macromedia’s Director, web scripting languages, as well as many recent novice-g geared research prototypes such as Alice [9], provide event-based constructs and user interfaces, enabling programmers to create highly interactive environments.

Objects-first “emphasizes the principles of object-oriented programming and design from the very beginning.... [The strategy] begins immediately with the notions of objects and inheritance....[and] then goes on to introduce more traditional control structures, but always in the context of an overarching focus on object-oriented design” [8, Chapter 7]. An objects-first strategy is intended to have students work immediately with objects. This means students must dive right into classes and objects, their encapsulation (public and private data, etc.) and methods (the constructors, accessors,

modifiers, helpers, etc.). All this is in addition to mastering the usual concepts of types, variables, values, and references, as well as with the often-frustrating details of syntax. Now, add event-driven concepts to support interactivity! Our approach meets the challenges by incorporating a specially designed library into the VisualJava environment.

Kim Bruce developed the Java library, ObjectDraw, that supports an "Object-Oriented from-the-beginning" approach specially for teaching CS 1. The use of real graphics objects and event driven programming are important components of the ObjectDraw library. ObjectDraw provides classes for objects that can be produced on a display: Lines, Rectangles, Ovals, Text, etc. When one of these objects is constructed using the "new" operator, it appears on the screen, no need to call paint (or draw). In addition, there is a list of methods that can be used to modify each of these objects: move, setColor, setWidth, etc. Again, if any of these methods are invoked, the screen is updated to reflect the requested change. This collection of graphical objects and the methods associated with them provide an excellent framework for introducing the notion of objects to novice programmers. The fact that the objects produced by constructors are not abstract, but are concrete and visible on the screen, makes it very easy for students to appreciate the connection between their code and its behavior.

In addition to the graphics features, the ObjectDraw library provides support for the use of an event-driven style of programming. Introducing event-driven programming simplifies the process of preparing students to use object-oriented techniques. In particular, the use of the event-driven style helps familiarize students with the definition of methods and the overall structure of classes. The definition of event-handling methods also has the advantage that the introduction of formal parameters is separated from the

introduction of actual parameters. At the same time, students are actively using actual parameters when invoking the constructors and methods of the graphics library. In conjunction with the use of objects through the graphics library, the event-handling style of programming provides an excellent way to prepare students for the introduction of classes. By the time students are ready to define their own classes, they have already used all of the required language mechanisms. Instead of learning about parameter passing or class syntax, they can focus on the role of objects.

ObjectDraw has the ability to limit the complexity faced by beginning programmers and to present objects first programming from the very start. But, it is our belief that it is not perfect for novice programmers. The text-based language for VJ extends the ObjectDraw library with the following changes:

- Up/Down/Left/Right replaces XYZ coordinates in movement commands.
- Bigger/Smaller/Longer/Shorter replaces double tuples for sizing commands.

Students already have a vocabulary for moving and resizing objects. Yet ObjectDraw continues to use traditional mathematical names. Instead VJ uses LOGO style [Papert], object-centric names. This tiny change is probably the most important aspect of VJ's text language. By using object-centric words in lieu of XYZ and doubles, we relieved the users of having to perform the cognitive mapping between numbers and movement by employing a model they are already used to.

### ***1.2.1 Educational content***

One of the four goals we stated for evaluation was that the students will increase their knowledge of introductory programming concepts. The ACM is in the process of



creating a curriculum for Computer Science education on the K-12 level (Tucker, Deek et al. 2002). Included in the draft are recommendations of what concepts and activities should be introduced at different ages.

#### *Grades 9 – 10*

In early high school, students should take a year-long course that is designed to impart general knowledge about computer hardware, software, languages, networks, and their impact in the modern world. This course should include a programming component that introduces the following concepts:

- Variables, data types, and the representation of data in computers
- Managing complexity through top-down design
- Procedures and parameters
- Sequences, conditionals and loops (iteration)
- Tools for expressing design – flowcharts, pseudocode, UML

#### *Grades 11 – 12*

In later high school, students should take a year-long course that focuses on developing programming skills, particularly algorithm development, problem solving and programming using software engineering principles. The programming components of this course should include:

- Methods (functions) and parameters
- Recursion
- Objects and classes (arrays, vectors, stacks, queues, and their uses in problem solving)
- Graphics Programming
- Event-driven and interactive programming

Currently, VJ introduces students to the following concepts, most of which are on the ACM curriculum for either the 9-10 grades or the 11-12 grade computer science classes:

- Variables and data types (9-10 ACM)
- Procedures and parameters (9-10 ACM)
- Conditionals and loops (9-10 ACM)
- Objects and classes (11 – 12 ACM)
- Methods and parameters (11 -12 ACM)
- Graphics Programming (11 – 12 ACM)
- Event-driven and interactive programming (11 – 12 ACM)
- Inheritance and Recursion (11 – 12 ACM)

To validate that these concepts would help prepare a student for a CS1 course, I analyzed some of today's popular CS1 textbooks. I found that most of the above concepts are taught in the first four chapters of the textbooks. Concepts like objects, inheritance and recursion are taught around the middle of the books. Graphics and event-driven programming is usually not taught until the end of the book, if at all. It should be noted that some text-books are starting to adopt the objects-first approach and often use graphics and event-driven programs from the start. This further validates that VJ is on the right track, not only with the concepts it teaches but also the approach it takes in teaching introductory programming concepts.

### ***1.2.2 Language Paradigms***

No single programming paradigm is ideal for all uses and users. One portion of a program may be best defined through animated programming by demonstration, another portion through static graphics (such as state transition diagrams), and another portion by traditional text-based programming instructions. Furthermore, the preferred mechanisms for expressing and representing programs will differ between programmers and programming skill.

VisualJava allows users to articulate their programming tasks through any mixture of three alternative programming paradigms: a direct manipulation language for building animated programs, by clicking buttons and choosing menu items in an iconic language, and a text-based subset of Java. To reinforce the user's perception of equivalence between the three programming styles, VJ provides support for user-interface equal opportunity [Smith, Cypher, and Schmucker, 1996] -the input actions in one programming paradigm cause corresponding outputs in the other two programming paradigms.

Most pedagogical tools support only a single syntax for expressing input: for instance, a text-based command language, or a direct manipulation graphical user interface. In an environment that supports more than one way of articulating programming tasks (with equal opportunity between them), the user is free to select whichever paradigm(s) that best suits their current needs. Additionally, by supporting multiple means for articulating programming tasks, an educational system can span a wide range of educational abilities. In VJ, for instance, novice programmers can program using the direct manipulation mechanisms, and more advanced students can use the more abstract but powerful text based language.

We believe that equal opportunity is also a powerful tool for self-directed learning. Traditional programming environments require a novice programmer to work forwards, from the program towards the desired solution. This can be a frustrating and disheartening process. With equal opportunity the user can state (or demonstrate) the required result, and view a program that produces the same result. This provides a *learning-by-doing* environment that is beneficial to novice programmers because it produces concrete models which provide students with prerequisite knowledge [Mayer,

1981]. We contend that users may gain an accelerated and deeper understanding of the mechanisms of programming by seeing how the alternative programming paradigms achieve equivalent effects.

The following is a short description of VJ's three language paradigms. More information about the languages can be found in Section 3.3.

#### **1.2.2.1 Text-based Language**

VJ's text-based programming paradigm is equivalent to standard Java programming. A full dialect of Java is supported. Program lines are typed into the text-editor, and the program is executed when the user clicks the *Compile* button. The system generated code is in the text-based Java language.

#### **1.2.2.2 Iconic Language**

The iconic language for VJ constructs are as follows: mechanisms for defining new procedures, parameters, and sample parameter values; mechanisms for defining variables, their values, and incrementing values; constructs for creating repeat loops; constructs for generating conditional statements; and icons allowing the user to invoke user-defined and built-in procedures with or without parameter values. Constructs built in the iconic language cause corresponding constructs to be generated in the text-based programming window and are available in the direct manipulation programming editor.

Parameter values for any procedure invocation are set through a drop-down menu alongside the procedure icons. The value list in the menu contains the valid values for the procedure. Variable values are set in the same manner.

Expressions, such as: `size < 5`, are supported in the iconic environment through VJ's expression builder. When the user clicks to add the conditional if expression, the expression builder pops up. The builder provides a template that allows users to enter expressions, which include any of the local variables declared. The variable, the logic and the test value is set by clicking on items in three drop-down boxes. Expressions can also be used for the bounds of while loops.

### **1.2.2.3 Direct manipulation Language**

Users construct programs in VJ by dragging and dropping objects, clicking on menus and then selecting parameters from a list of valid choices. Users can add events and stop-motion actions to create animated event-driven programs. Corresponding programming actions that produce identical actions are simultaneously generated for the text-based code view.

## **1.3 Authoring Programs in VJ**

The creation of an interactive graphics program can be broken down roughly into the following phases of development:

1. The placement of objects for the scenes in the program.
2. The definition of how behaviors are changed in the face of user interactions (mouse and keyboard commands)
3. The definition of the behaviors of the dimensional objects.

Steps 1 and 2 are setup steps often done more easily through direct manipulation while Step 3 is a definition step, which is more accessible through textual/scripting

means. Therefore, authoring in VJ consists of two phases: creating the scenes and scripting the actions. This same two-phase workflow is seen in some commercial tools, including other Event-Driven Programming (EDP) tools such as VisualBasic.

The metaphor of single cell animation motivated the actions of *creating scenes*. A user creates a scene in the canvas view of the system using the direct manipulation language. The animator can create single frames of an animation by inserting a pause between method calls and then composite them into a complete program.

Once the objects are placed, the user can use the iconic or text-based languages to script additional functionality. The user iteratively scripts and creates scenes, until the program is completed.

### ***1.3.1 Introduction to the VisualJava Environment***

This extended example is meant to show what an average VisualJava interaction is like to a user. Of course, a static presentation of VJ can't capture everything important about the VJ experience, but it can help make the explanations that follow in the next sections more concrete.

The following example shows how a user might develop code for making an interactive program we'll call MovingSquares. In this program, every time a user clicks on the canvas, a blue rectangle that moves about randomly is created. To make this program the user will have to make a mouse event, instantiate an object, set its attributes, and create the animation method that will instruct the square to move around randomly.

### 1.3.1.1 Step 1: Setting the Scene

When a student first starts a program in VJ, they can add an object, add an event, add an animation pause, or change to the code view (design view is the default). Since we want the square to be created in a mouse event, the *Add Event* button is selected and a box appears which prompts the user to choose a mouse event (see figure 1). If the user selects an event and clicks *OK*, the system switches to event mode: until the event is ended, all actions will occur in the canvas's event method. If the user selects cancel, the system stays in begin mode: all actions occur in the canvases begin method. Notice that in figure 1 and 2, the user can not add another event or compile the program (since the buttons are inactive) until the event is ended. Enabling and disabling buttons is a practice widely used in VJ's interface to enforce the concept of scope and to reduce common syntax errors, like forgetting to close curly brackets.

To add an object the user clicks on the *Add Object* button. The object panel opens on the right-hand side of the interface (see figure 2). When a user adds an object to the canvas the system requests a unique name for the object (see figure 3). The user gives the object a unique name meeting Java standards. If the user clicks *OK* with out giving the object a unique and correct name, an error is returned. If the user selects *Cancel* no action occurs and the dialog box closes. If the user gives an appropriate name to the object and clicks *OK* the dialog box closes and the object is added to the canvas and instantiated in the canvas's code (see figure 4).

To finish setting the scene by dragging the square to where we want it to first appear on the canvas and set its attributes. To set the squares attributes the user right-clicks on the square. A pop-up menu appears that lets the user call the object's methods. If the method

has parameters, the pop-up extends to a pre-defined list of possible parameters values (see figure 5 and 6). If the method that is called is a pre-defined method the action occurs immediately (see figure 7).

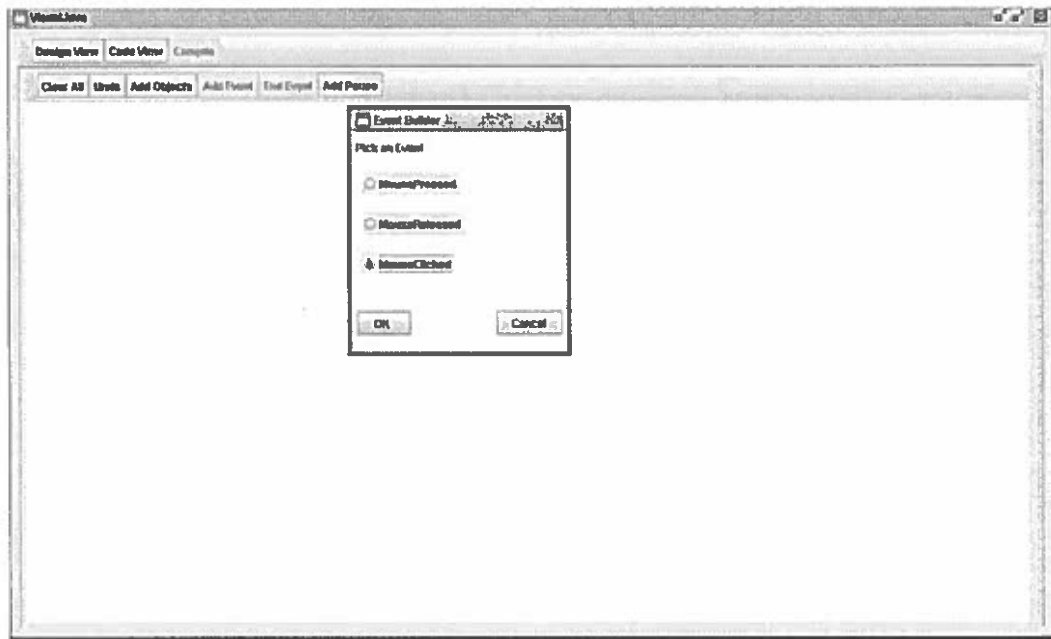


Figure1: Adding an event in Design View with the Event Builder dialog box



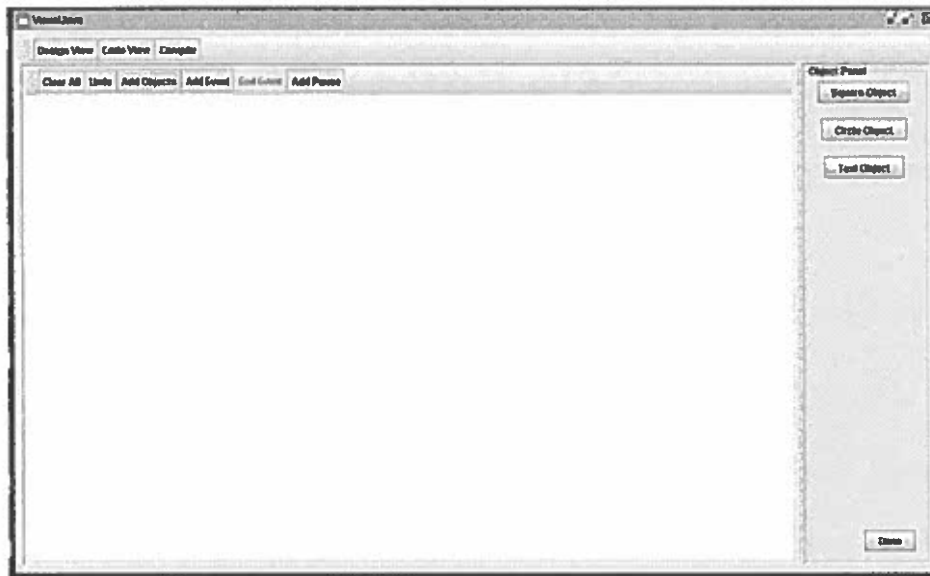


Figure 2: Adding an Object with the Object Panel



Figure 3: Naming the New Object

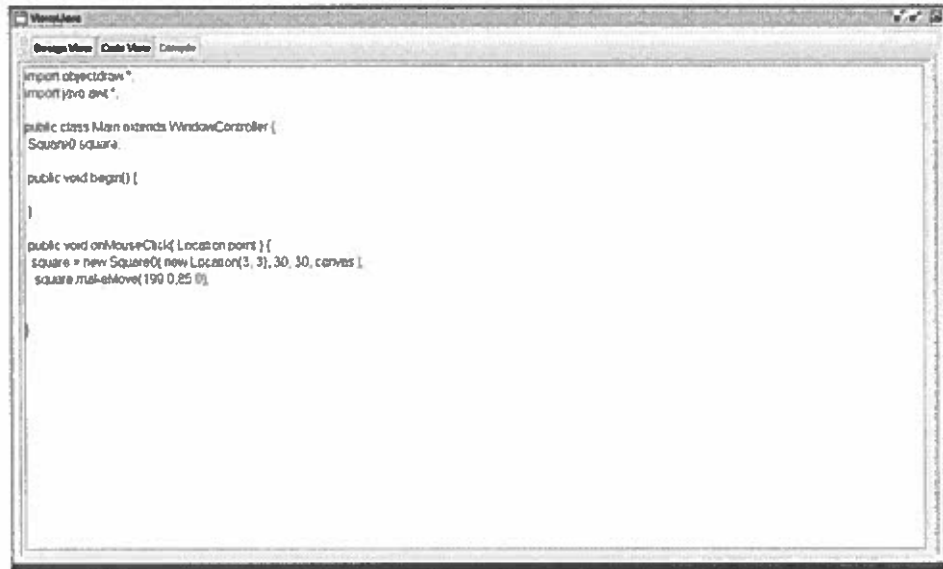


Figure 4: The Canvas's Code View



Figure 5: An Object's Pop-up Menu of Actions



Figure 6: Selecting Parameters for a pre-defined method

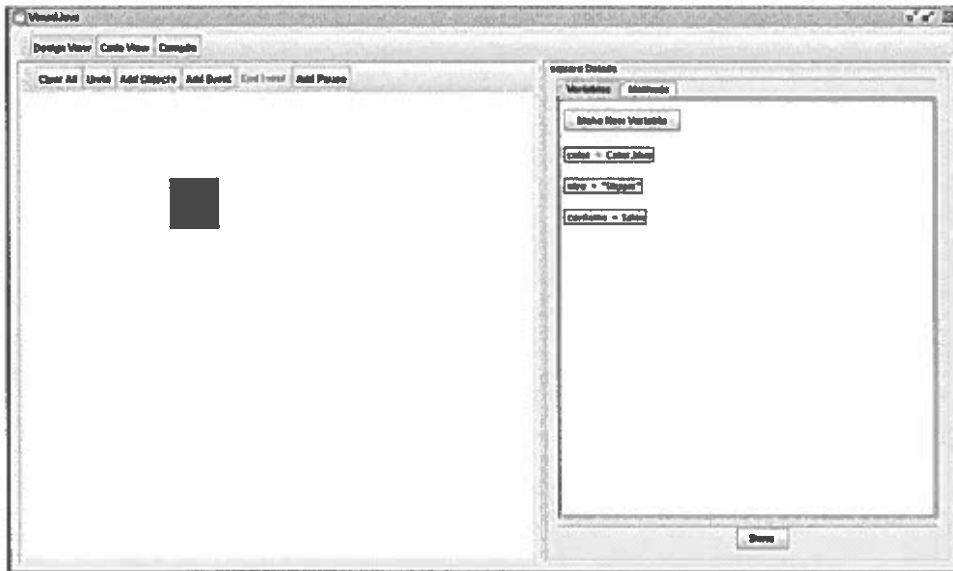


Figure 7: The Detail Window

Having established our (very simple) opening scene, we can move on to experimenting with fragments of VisualJava script that we will use in our animation behavior.

### 1.3.1.2 Step 2: Scripting

To make a new method the user scripts in either the text or iconic language. This example will build the method in the iconic language and view the generated code in the text editor. To script a new method the user can either double click on the object or select the *Make New Method* option from the object's pop-up menu. When the user double clicks, the details panel opens on the right-hand side of the interface (see figure 7). The user can define new global variables and create new methods in the details panel. The user clicks on the Methods Tab and clicks the Make New Method button (see figure 8). A dialog box opens, requesting a unique and Java correct method name (see figure 8). If the user enters a non-unique or an incorrect name and clicks OK an error is returned hinting at the problem. If the user enters a unique correct name, the Method Builder Panel opens in the bottom half of the details panel (see figure 9). The variables and methods are also given the ability to be added to the new method (see figure 9).

To create the animation method the user will need to create a new variable to control the animation loop. The user clicks on the Variable Tab, and clicks on Make New Variable (see figure 7). The variable builder dialog box opens up. The user types in a

variable name, chooses its type, and sets its initial value (see figure 10). The types have default values for the user to choose from. For the case of Booleans, the user can only choose true and false. For integers and Strings the user can also type in a value (see figure 10). When a user clicks on OK, the system checks to see if there are any variables with that name. If there is an error is returned. If there is not, then the variable is added to the object's global variables and the details panel (see figure 11 and 16).

The user can add variables or methods to a method by clicking on the item's *Add* button. Figure 11 depicts the method after the user-defined variable has been added. In this figure the variable is being incremented by a value. Integers can be assigned or incremented by a value, all other variables can only be assigned a value. The list of values the user can choose from is determined by the variable's type and any user-defined values.

An object's animation method basically consists of a while loop that defines the animation. To add a while loop, the user clicks on the *While-Loop* button (see figure 11). When a while or if statement is added the system goes into loop mode: until the loop is ended, all actions will occur in the loop. To build the loop's expression the expression builder dialog box opens (see figure 12). The expression builder provides the user with a template to create correct expressions. The user selects the variable to test from the first drop-down box, then that variable's type determines the logic list in the second drop-down, finally, the third drop down is populated with the variable's pre-defined and user-defined values. When an item is selected from the list, the expression is built in an uneditable text-box. The user can clear the expression at any time to start over. When the

user has built a correct expression and clicks OK the expression is added to the method (see figure 13). The user can add variables or call methods to the loop statement by clicking on the item's *Add* button (see figure 13). In figure 13 the move method is being added. When a method is added the user selects the parameters by choosing an item from a drop-down menu. When a user ends a loop the statement is surrounded by a box (see figure 14). Once the user is satisfied with the results, we can save our work by clicking on the Save button. The user can then edit or view their method by clicking on the method's Edit button (see figure 15). When a user clicks on the Edit button, the method builder opens. The user can also edit the code in the code view (see figure 15). The user clicks *Done* to close the details panel.

When the scene has been set and the scripting done the user can compile and execute the program by clicking on the Compile button. Before or after compiling the user can view the complete canvas and object code. Figure 16 depicts the code view for the object.

When the program is executed, the user is rewarded with a sophisticated graphics program that responds to mouse events (see figure 18).



Figure 8: Naming a New Method

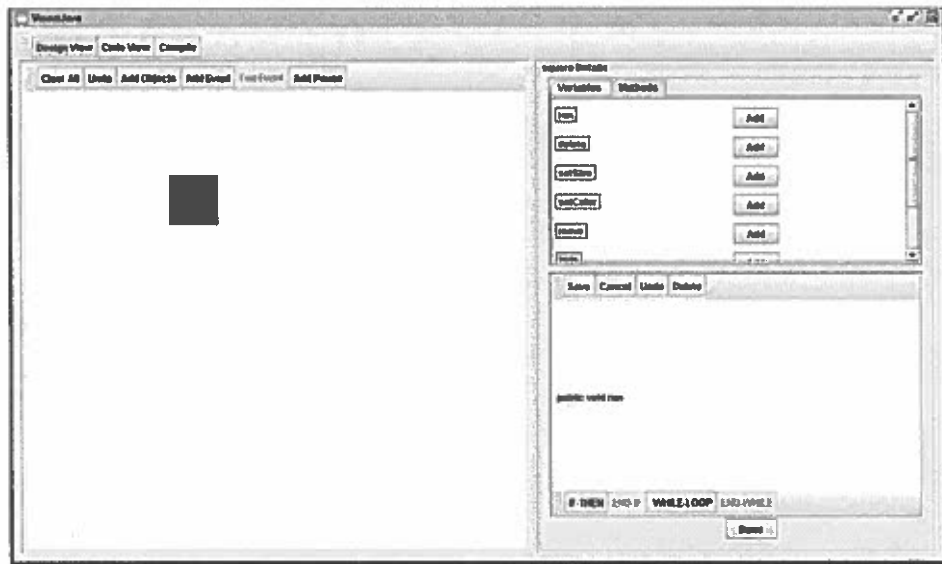


Figure 9: The Method Builder Panel

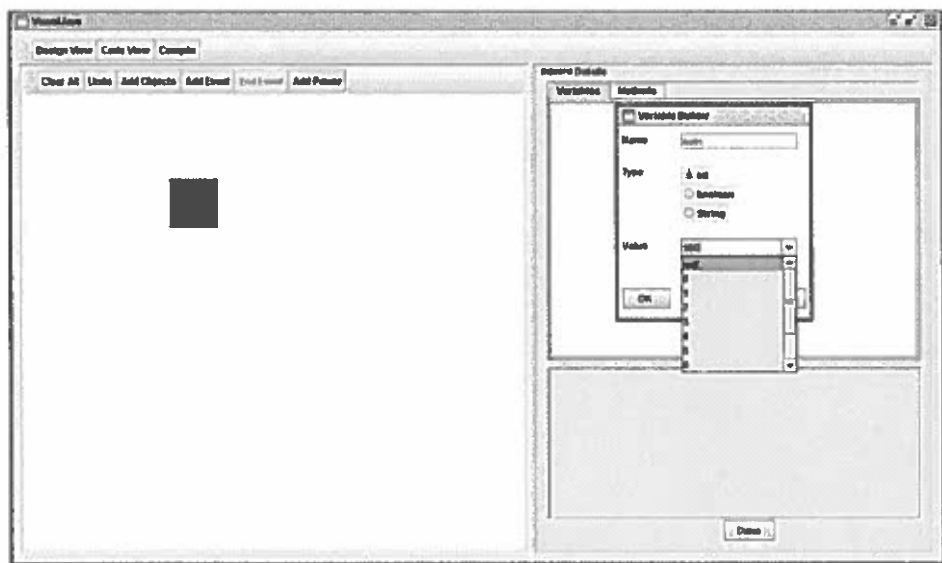


Figure 10: Making a new variable with the Variable Builder dialog box

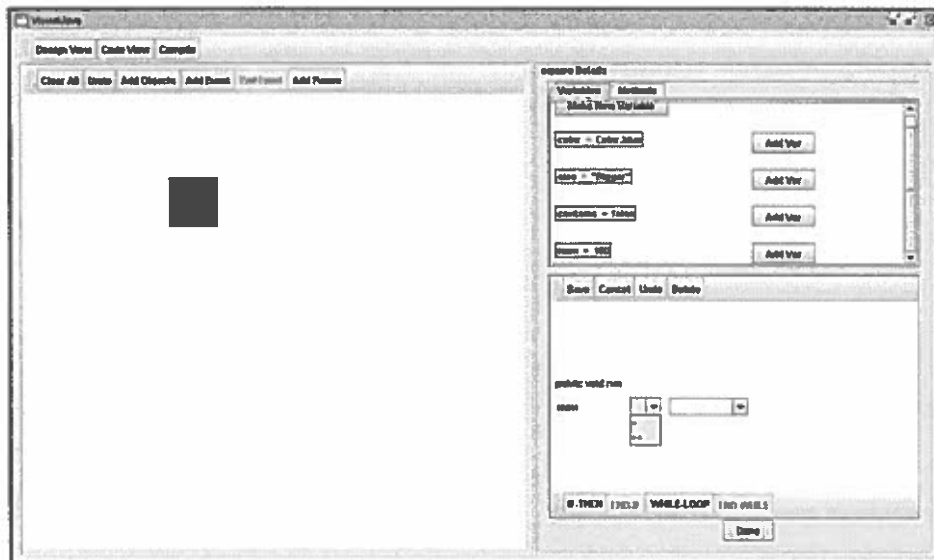


Figure 11: Adding the variable to the method and setting its value



Figure 12: Making a while loop expression with the Expression Builder



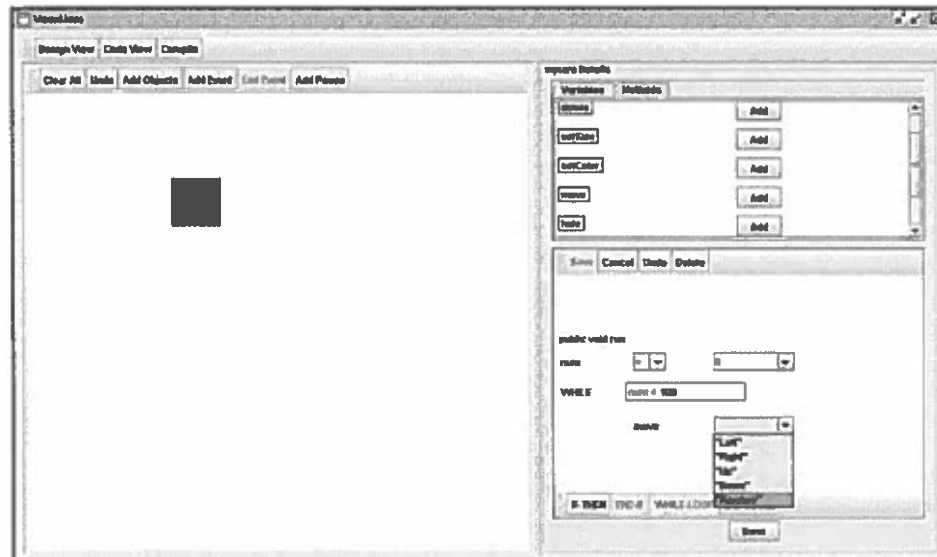


Figure 13: Adding a method to the while loop and setting its parameter

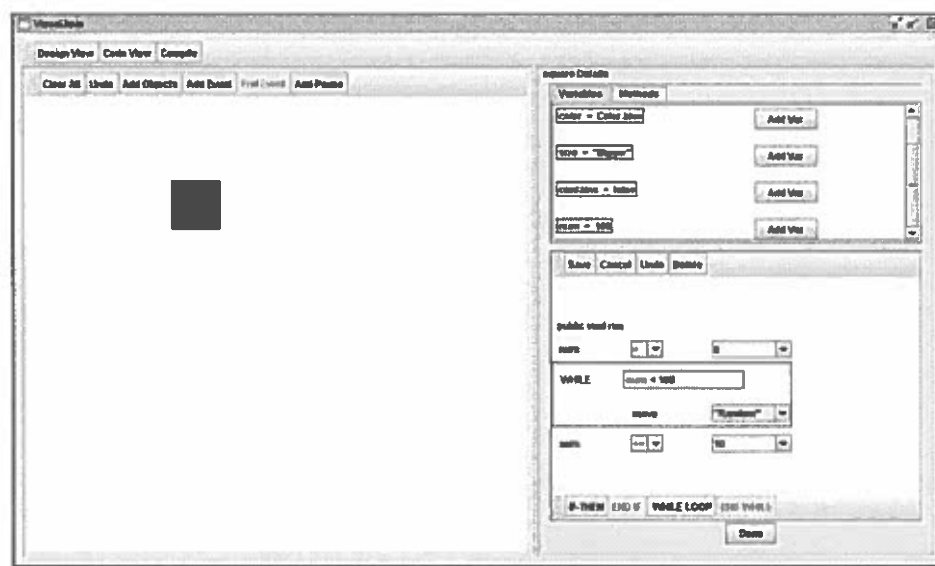


Figure 14: Ending the while loop

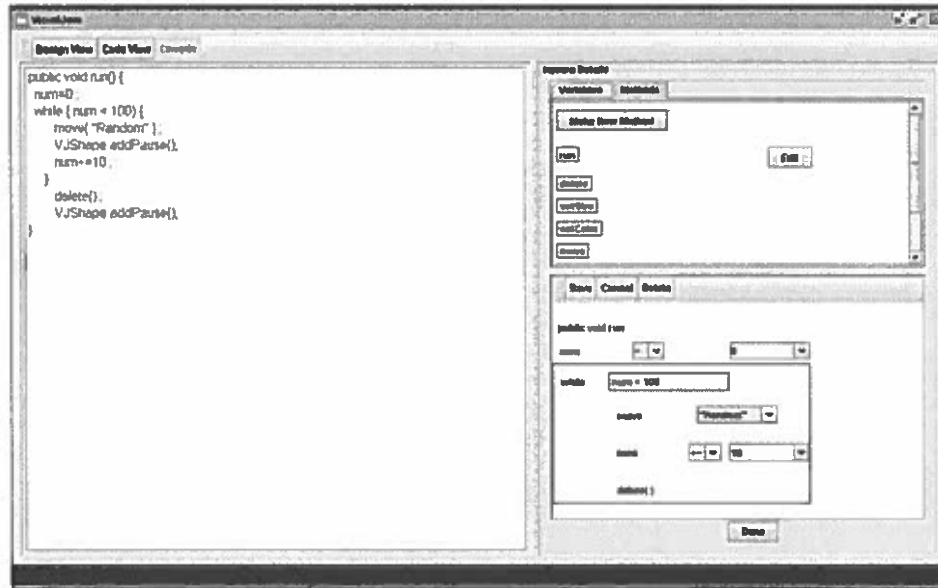


Figure 15: Editing the Method in the iconic and code editor.

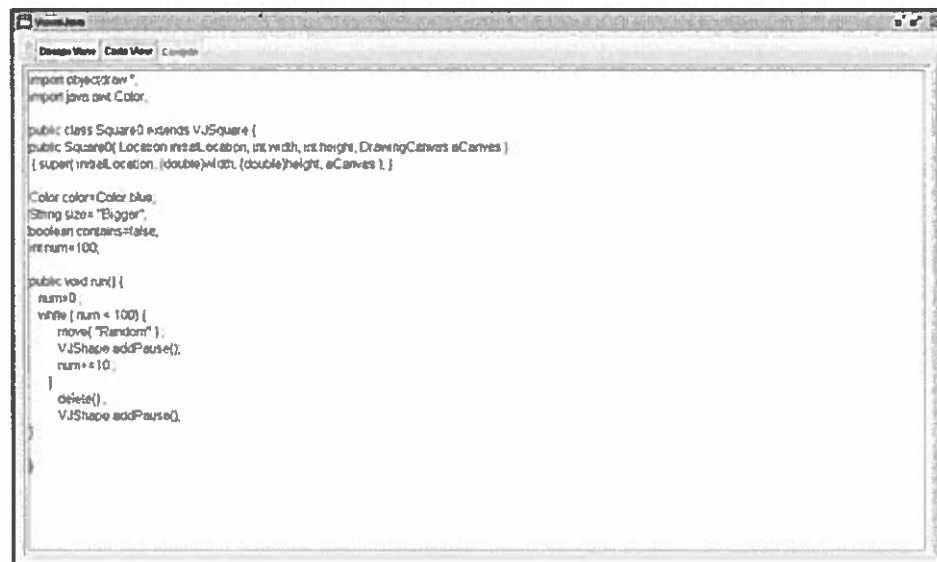


Figure 16: The code view of the Square Object with a user defined method

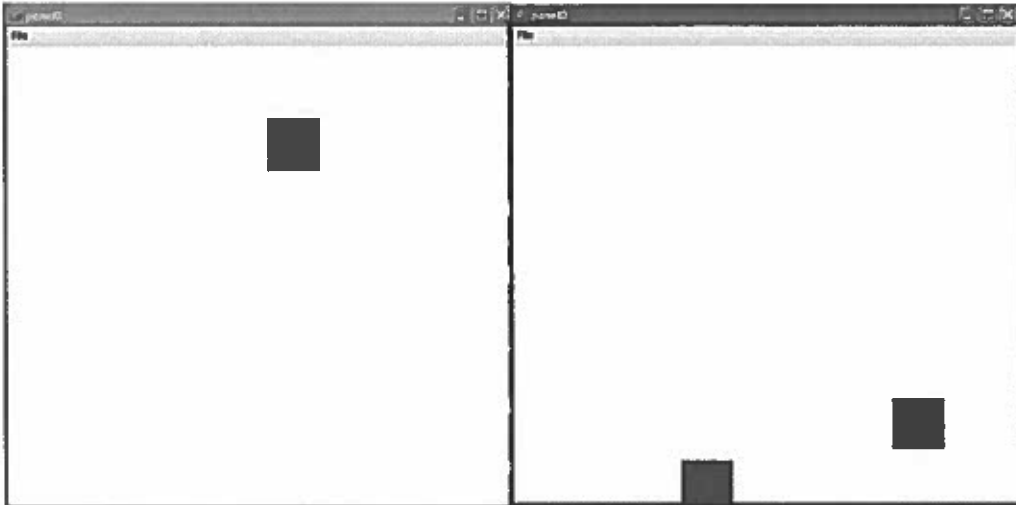


Figure 17: MovingSquare.java after one mouse click and after another mouse click

### 1.3.2 *The User Interface*

The user interface (UI) consists of three main components: 1) the direct manipulation editor, 2) the iconic editor, and 3) the text editor. Each component supports a different language and provides different functionality. The following sections describe the UI components in more detail.

#### 1.3.2.1 **The Direct Manipulation Editor**

The direct manipulation panel (DME) provides a drag-and-drop editor and functions to create event-driven programs. The DME is one of two views that make up the Interaction Panel. The other view, the Editor Panel, is discussed in section 3.3.2.3. The drag-and-drop editor prevents students from making syntax errors that are prevalent for beginners. Users can use/modify objects and author programs that generate events and animations. This

enables students to a) build an event-driven graphics program by just dragging and clicking from examples to build prerequisite knowledge, and c) observe the immediate effects of their changes in their program.

Because users construct programs in VJ by dragging and dropping objects, clicking on menus and then selecting parameters from a list of valid choices, they cannot make syntax errors. By preventing users from making syntax errors, VJ allows users to gain experience with and understanding of the logic and control structures used in programming, independently of learning to type syntactically correct program. At any time a user can switch to view the generated Java code to build prerequisite knowledge of programming concepts.

#### **1.3.2.1.1 Canvas**

The most prominent feature of the DME is the canvas. Clicking on the Canvas View button accesses the canvas. The canvas is the drag-and-drop editor for the user's program and a visualization of the program's backdrop. A user can add objects, add events, or add an animation pause to the canvas. They can also end an event, clear all, and undo the last command.

Adding an event and an object was discussed in Section 3.3.1 and will not be repeated here. To add an animation pause a user clicks on the *Add Pause* button. A confirmation box opens asking the user to affirm their decision to add a pause (see figure 18). Adding a pause does two things: 1) inserts an animation pause, which allows us to see the changes to the object, and 2) takes a snapshot of the scene.

When a user clicks the *Clear All* button all the objects from the canvas are removed. *Clear All* effectively erases the canvas and clears the canvas code of any reference to the objects. It provides the user a fresh slate to start a new program.

If at any time a user does something that they would like to reverse, they can press the *Undo* button. *Undo* engages an infinite-level undo mechanism capable of undoing nearly any operation in VJ all the way back to the beginning of a VJ programming session. *Clear All* is the only command that can not be undone. If the last command was



Figure 18: Adding a Pause

adding an object or event, all references to the item is removed from the canvas and the code. If the last command was a call to an object's method, the call is undone and the object is returned to its original state. For example, if a user changes a blue square to red and then clicks *Undo* the square turns back to blue and the message is erased from the canvas code.

### *Objects*

A user needs to set the stage by populating the world with objects, placing them in their initial locations, and setting their physical attributes. Objects are added to the canvas by clicking on the Add Object button and then selecting from the object panel library (see figure 2). Each object requires a unique name. Once an object is added, the user can right-click on the object to view and call the object's methods. This process was already detailed in section 3.3.1.

Once the object is initialized, the program code is created using the drag-and-drop canvas editor. Using the mouse, an object is dragged around the editor and mouse-clicked to view a drop-down menu, which allows the user to select from primitive methods that send a message to the object. All commands in VJ are animated by default whenever it is semantically reasonable. A student can write his/her own user-defined methods and functions, and these are automatically added to the drop-down menus. Because users construct programs in VJ by dragging and dropping objects, clicking on menus and then selecting parameters from a list of valid choices, they cannot make syntax errors. By preventing users from making syntax errors, VJ allows users to gain experience with and understanding of the logic and control structures used in programming, independently of learning to type syntactically correct program statements.

Each object encapsulates its own data (its private properties such as height, width, and location) and has its own member methods. This allows students to experiment with modifying existing classes/programs. In general the methods can be divided into two categories: those that tell the object to perform a motion and those that change the

physical nature of the object. Table 2 contains the current list of VJ object's primitive methods.

<b>hide()</b>	hides the object
<b>show()</b>	displays the hidden object
<b>sendToBack()</b>	Sends the object to the back.
<b>sendToFront()</b>	Sends the object to the front.
<b>move(Left/Right/Up/Down/Random)</b>	Moves the object within the canvas in the direction specified as the parameter.
<b>setColor(Color.c)</b>	Sets the color of the object.
<b>setSize(Bigger/Smaller/Normal)</b>	Sets the width and height of the object bounding the object.
<b>delete()</b>	Permanently removes the object from the canvas

Table 2: VJ Objects' built-in methods

#### 1.3.2.1.3 *Events*

A user can add mouse events to the canvas by clicking on Add Event. This process was described in section 3.1.1. Table 3 lists the mouse events VJ currently supports. If a user adds an event, the system switches to event mode; until the event is ended, all actions will occur in the canvas's event method. By default when a user starts a program the system is in begin mode; all actions occur in the canvas's begin method. The user does not see any difference in the DME when the system switches modes. The difference is when the action will occur and how the code is generated. If a user adds an event and wants an object to be created when that event occurs, as in our example program, the system needs to know to add the code to the event method. Adding and

ending events simulates the functionality of curly brackets. When a user ends an event the system is notified to insert an ending curly bracket and adds any further statements to the begin method. This reduces the novice mistake of forgetting to end methods. To enforce the actual ending of the event, the user is prohibited from adding another event, or compiling the program until the *End Event* button is clicked. While a user can still build programs they did not mean to, they will either realize their mistake when they try to compile and notice that they need to end an event before they can, or when they execute the program and it does not behave in the expected way. We believe that some trial and error learning is beneficial to students. Our goal in error support is to try to shield the student's from common syntax and semantic errors.

onMousePressed(Location pt); Mouse is clicked down.
onMouseReleased(Location pt); Mouse is released.
onMouseClicked(Location pt); The combination of onMousePressed and onMouseReleased.

Table 3: VJ's built-in events

#### **1.3.2.1.4    *Pause Animation***

When a user is creating an animation they must insert a pause between methods calls that they want to differentiate. This is conceptually like the animation technique



used in stop or single cell animation. It works by adding a pause, setting up the object and the scene, and then adding another pause, until the desired program is demonstrated.

The pause serves an important purpose. It allows us to see the movement of the object when the program is executed. The computer works at such high speed that if we didn't explicitly pause between changes to the object we would hardly see it.

### 1.3.2.2 The Icon Editor

An object's functionality can be extended using the iconic language in the icon editor. The mechanism for generating the new code relies on visual formatting rather than details of punctuation. The gain from this no-type editing mechanism is a reduction in complexity. Students are able to focus on building methods and variables, rather than dealing with the frustration of parentheses, commas, and semicolons. We hasten to note that program structure is still part of the visual display and the semantics of instructions are still learned. At any time a user can switch to view the generated Java code to support a later transition to C++/Java syntax.

The use of buttons, menus, drop-down boxes and formatted text fields to build programs provides a *controlled exposure to power* [Pausch, 2003] to the user. This characteristic allows users to become experts by incrementally adding to what they know, rather than forcing them to learn entirely new commands or constructs. This enables students to a) be actively engaged with the tool, b) learn at a level that is appropriate to the skill, and c) learn programming concepts through a novel format.

#### **1.3.2.2.1 *Details Widow***

The details window provides a button and menu based icon editor to create new methods and variables for an object. The details window consists of three panels: the variable panel, the method panel, and the method editor panel.

The details window is accessed through either double clicking on an object or selecting the *Make New Method* option from an object's pop-up menu. The procedure of double-clicking was already described. When a user chooses the *Make New Method* option the system requests a method name before it opens the details window. The method name is checked for uniqueness and Java correctness before it is accepted. The details window opens with the method editor panel active (see figure 19). A user can immediately start building a new method. The user can make new variables at any time. When a user is finished working in the details window they can click *Done* to close the window. The interaction panel's current view enlarges to fill the space.

#### **1.3.2.2.2 *Variable Panel***

The variable panel lists the object's global variables. When a user is not building a method the variable panel displays the names of the variables and the variables current value. If that value is changed in the DME the value is updated in the variable panel immediately. The values cannot be manipulated in the variable panel. They can only be changed in a method or by calling one of the object's setter methods.

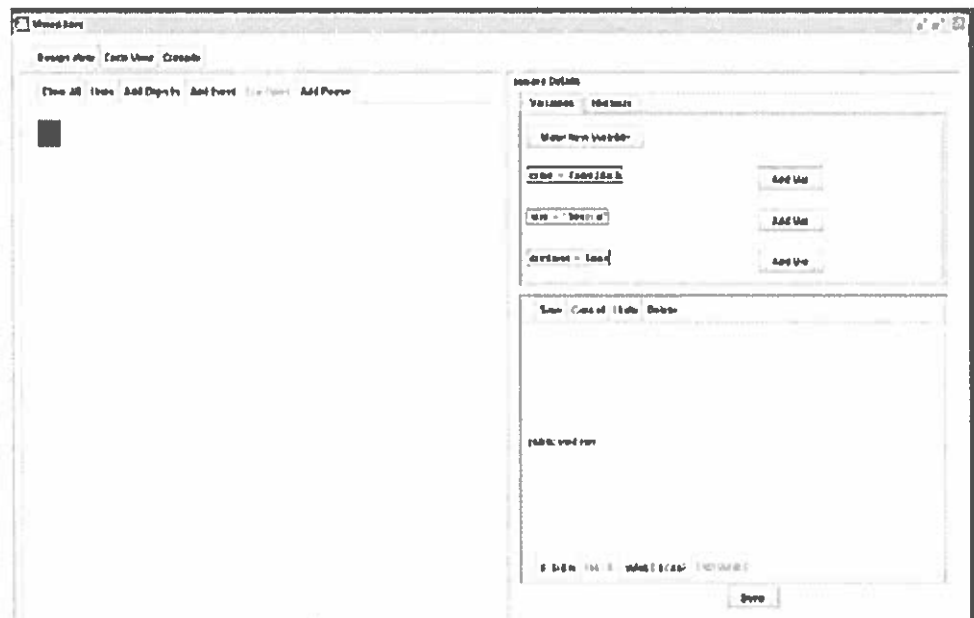


Figure 19: The method editor

When a user is creating a method, the variables can be added and then manipulated in the method editor panel. A variable is added to a method by clicking on the Add button next to the variable. When the Add button is clicked a visual representation of the variable appears in the method editor. The user can select a valid value for the variable from a drop down list. This process was discussed in detail in section 3.1.1.

A user can make a new variable at any time by clicking on the Make New Variable button in the variable panel. A variable builder replaces the list of variables in the panel. The variable builder provides a template to build new global variables for the object. The user types in a name for the variable and then chooses from an integer, String or boolean type for the variable. The type the user chooses populates the value list for the variable. An integer variable by default can have the initial value of null or 0-9. A user can also

type an integer into a formatted text box that only allows numbers. A String value by default can be null or set to the word “string”. A user can also type in a string value into a formatted text box that only allows characters. A boolean variable can only be initialized to null, true or false.

A user can use the tabs at the top of the details panel to toggle between the variable and the method panel.

#### **1.3.2.2.3 *Method Panel***

The method panel lists the object's methods, both built-in and user defined. When a user is not building a method the variable panel displays the names of the methods. The methods cannot be manipulated in the method panel. They can only be called in a method or in the DME.

When a user is creating a new method, the methods can be added and parameters set in the method editor panel. A method is added to a new method by clicking on the Add button next to the method. When the Add button is clicked a visual representation of the method appears in the method editor. The user can select a valid value for the method's parameter from a drop down list. This process was discussed in detail in section 3.1.1.

A user can make a new method by clicking on the *Make New Method* button in the method panel. The system requests a new name for the method and checks to see that it is unique and Java correct by looking through the object's lists of methods and checking the name against all the method names. To check for Java correctness we use Java regular

expressions to compare the given name against an acceptable pattern. This technique is used for all the dialog boxes that require a name for an item.

#### **1.3.2.2.4 Method Editor Panel**

The most useful feature of the details panel is the method editor panel. The method editor is used to build new methods that extend the basic object's functionality. A user can add and manipulate variables, call the objects methods, and build control flow and conditional statements. They can also end loops, delete the new method, undo the last command, and save and cancel all changes to the program.

We've already discussed adding and manipulating variables and methods in the method editor. To build control flow a user can select from adding an *if* or a *while* statement by clicking on their respective buttons. The process of adding a *while* statement was described in section 3.1.1, and the process of adding an *if* statement is identical so we will not repeat the information here. When a user is done building the while loop and if statement they must click on their respective *End* button to end the function. A user cannot save the method until all while loops of if statements are ended. When a user ends a statement, a black border is inserted around the code. The box serves as a visual representation of the functionality of curly brackets creating scope in a method. Currently, a user cannot build nested if/while statements. They can nest one inside the other though. The system uses disabling and enabling buttons to control when and what can be added to the system.

Clicking on the *Delete* button erases the entire method. No reference to the method exists and the user is free to create a new method with the same name. When a user clicks *Delete* the method editor panel closes and the details panel returns to its inactive state.

Clicking on the *Cancel* button returns the program to the state it was the last time it was saved. In other words it erases all recent changes. When a user clicks *Cancel* the method editor panel closes and the details panel returns to its inactive state, but the method is still listed in the method panel and still appears in the object's pop-up menu. The method can be further edited by clicking on the *Edit* button next to it. Only user-defined methods can be edited in the method editor.

Clicking on the *Save* button saves all recent changes to the program. When a user clicks *Save* the system asks them to confirm their choice and then commits the changes to the object's program. The method editor closes and the details panel returns to its inactive state.

The *Undo* button operates in the same manner as the *Undo* in the DME. A user can undo all commands added to the method. A user can only undo to the point of the method creation they cannot delete the method through undo.

### **1.3.2.3 The Editor Panel**

At present, the editor panel is just a simple MS Wordpad-like text editor, but we would like to extend the prototype to provide support for context-sensitive coloring (command tokens in red, comments in green, strings in blue) of the text-based language. Currently the editor does support cut/paste, undo/redo and save/cancel. The editor panel

shares space on the interaction panel with the DME. Clicking on the Code View button accesses the editor panel.

The editor is used to view and edit the canvas's code, the object's code, and the user-defined method code. While a user is creating their program in either the DME or the Iconic editor, code is being generated to populate the Editor Panel with the Java code that comprises the user's program. At any time a user can switch to the code view and see the code that has been generated by their engagement with the tool. The text editor enables students to a) increase their understanding of programming concepts, b) modify existing code examples, and c) use the system at a more advanced level. To view the canvas's code a user clicks anywhere on the canvas that does not contain an object and then on the Code View button. The editor opens in the space that was occupied by the canvas.

To view the object's code a user performs the same action, but clicks on a specific object instead of the canvas. The method code is viewed by clicking on the Edit button next to the method in the method panel.

The text editor serves two functions: 1) as a window to view the generated text and 2) as an editor to write Java programs. The text editor is the last resource in providing a scaffolding approach to learning programming concepts. Instructional scaffolding is the provision of sufficient supports to promote learning when concepts and skills are being first introduced to students [Cohen, P. R, et al, 1989]. These supports are gradually introduced and removed as students develop autonomous learning strategies, thus promoting their own cognitive, affective, and psychomotor learning skills and knowledge

[Mayer, 89]. A user can choose to do some of the programming in the text editor after gaining experience and confidence from building programs in the DME and Icon Panel. In VisualJava the resources are never taken away. A user can switch between the program creation languages and editors at any time they need.

## 1.4 System Implementation

We specifically did not seek to create breakthroughs in programming languages, new GUI toolkits, or graphics tools. Instead, we took for granted that these technologies would exist, and instead focused our energies on the problem of ease-of-authorship.

Our development strategy was to look for the best “off the shelf” solutions for each of VJ’s components while not becoming wedded to any one solution. Our goal was to keep the interfaces between layers small and well controlled, allowing us extreme flexibility.

### 1.4.1 System Architecture

In terms of the latest implementation, the architecture is shown below:

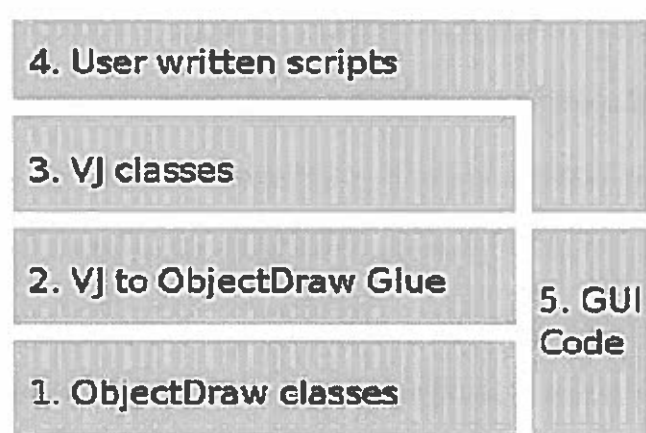
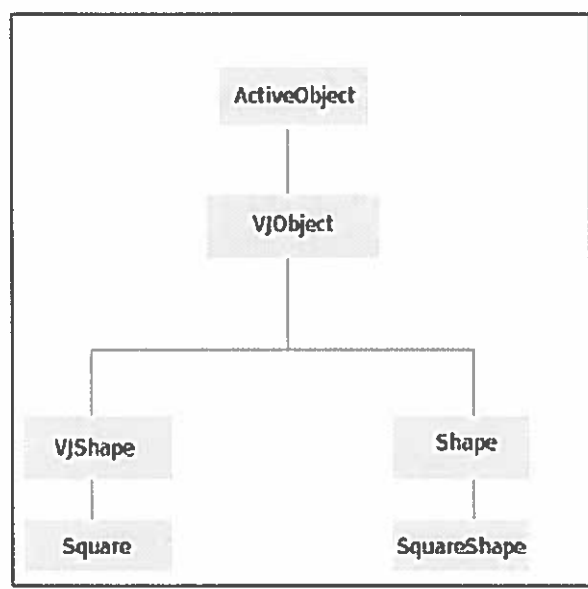


Figure 20: VJ System Architecture



All programs built by a user in the VJ environment extend a VJ class, which in turn extends an ObjectDraw class. For example the class hierarchy of a square object is illustrated below.

Figure 21: VJ square object's class hierarchy.



From top to bottom and left to right, these classes are:

#### *ActiveObject* Class

This class defines the basic animation API for an object. The code that is to be animated should be placed in the body of a parameterless run method defined within a class definition that extends *ActiveObject*. When the start() method of an *ActiveObject* is invoked, the system will begin to execute whatever code was placed in this run method.

*ActiveObject* is a simple extension of Java's native Thread class. *ActiveObjects* are automatically managed so that they are suspended and restarted when an application is

stopped and started. In addition, the sleep method provided in the Thread class is replaced with a similar pause method that does not require exception handling.

### *VJObject Class*

This class serves as the glue between the ActiveObject class and the VJShape Class. The VJObject class implements the glue code that makes the ObjectDraw library callable from the VJShape class. The VJObject class translates the user's actions in the environment to ObjectDraw actions.

### *VJShape Class*

This class defines the widest part of the VJ API, and is the primary source of commands that VJ programmers have to know. This class is the abstract class for the visual representation of all VJ objects in a user's program. In other words this is the class that creates and maintains any object that is placed on the canvas and then manipulated by the user. Note that the abstraction is that of a single class, though the actual functionality is implemented in several superclasses of the classes of ObjectDraw's drawable objects, none of which the end programmer ever sees. Novice users might make calls to `SendToBack`, never knowing or caring that this function is not (directly) a VJShape method. OOP becomes an implementation detail, a decidedly good thing for our target audience.

### *Shape Class*

The Shape class is another abstract class that extends the VJObject class. The shape class creates new objects in the Java paint method that can be used in the VJ environment. Users never directly encounter the Shape class or its descendants.

### *Square Class*

The Square class is the program the system generates through the actions of the user in the VJ environment. A user's program extends a VJShape class. The square class is implemented in a subclass of the VJShape class. The Square base class is extended by the user's scripts to make new instances of a Square.

### *SquareShape Class*

The SquareShape class extends the Shape class. The shape the users drag around and manipulate in the VJ environment is an instance of a Java graphics object. The object is created in the Shape's overloaded paint method.

#### **1.4.1.1 ObjectDraw Classes**

At the lowest level (layer 1), VJ depends on the ObjectDraw graphics library. This layer provides basic services for managing the canvas, the objects and their attributes. The ObjectDraw API is the primary set of classes used in the construction of the programs that users build in VJ. This set of object-oriented classes (WindowController, ActiveObject, FilledRect, FilledOval, Text, and the classes they extend) give VJ programmers access to Animation control, movement and resizing, and a drawable canvas.

WindowController is a Swing compatible class designed to be extended to produce programs that handle simple mouse events and draw graphics in a single window on the screen. In a program written by extending this class, mouse-event response is executed in the bodies of the methods onMousePress, onMouseRelease, and onMouseClick. Each of

these methods is passed a Location describing the location of the mouse at the time of the event. Initialization code can be placed in the body of the definition of a begin method. Throughout the definition of a class that extends WindowController, the name "canvas" can be used to refer to the program's DrawingCanvas.

FilledRect, FilledOval and Text are the graphics objects that can be added and manipulated on the canvas. The VJ layer uses these classes as the base objects available in the environment. These classes provide the same functionality as the VJShape classes.

#### **1.4.1.2 VJ to ObjectDraw Glue**

The next level up (layer 2) contains classes which implement the glue code that makes the ObjectDraw library callable from VJ. This layer provides the code that enables the system to turn a user's program in VJ into an executable program. The glue code translates the user's action in the DME or Icon editor to classes that extend the ObjectDraw classes.

#### **1.4.1.3 VJ Classes**

Above the glue layer is the VJ layer (layer 3). This layer contains the most used components of the VJ system: the object classes and the canvas classes. These classes implement the scenes in a program to which all other things are attached.

The VJShape classes are abstract classes for the visual representation of the objects in a user's program. In other words these are the classes that create and maintain any object that is placed on the canvas and then manipulated by the user. These abstract classes are also the primary source of commands that are available to VJ programmers. The objects' methods fall into several general categories:

**Geometric Manipulation:** Move, MoveTo, SetSize, sendToBack ( ) ,  
sendToFront()

**Property and State Query:** Contains

**Coloring and Rendering:** Get/SetColor

**Miscellaneous:** Show, Hide, and Delete

This layer also contains the classes that builds and maintains the canvas. The canvas listens for users' actions in the DME and interacts with the glue classes to build users' program. The canvas and object classes are the most visible classes and the classes most encountered by the users. A user extends these classes by interacting with the VJ environment to build their own programs.

#### 1.4.1.4 User Scripts

Users can provide any special-purpose functionality to a VJ object through the scripting editors (layer 4). VJ allows users to script new functionality to extend VJ's base object classes. The VJ scripting language allows functions implemented in the iconic editor or text editor to be callable from the any of the editors, including itself.

The scripts made in the iconic editor are guaranteed to be error-free while the scripts made in the text-editor have no such guarantee. The system guarantees the correctness of the scripts made in the iconic editor by providing templates for making new variables and creating conditional statements. It also controls the user's actions by supplying valid value lists for variables and method parameters, and by enabling/disabling actions. A user's script created in the iconic editor is not fool-proof though. A user can create scripts that do not behave as they expected. Unfortunately, users' scripts can not be tested until they are compiled and executed in a program.

### 1.4.2 Code Generation

During the user and task analysis stage of designing VJ we asked our participants the following questions (see Table 1): Q2.5) What helped you get better at either solving programming problems or coding the solutions? And Q3.3) Do you have any ideas on IDE features that would make learning how to program easier?

The most common response to these questions was *examples*. That is, having examples helped our participants we interviewed learn to program, and that an IDE for novices should provide examples. This led us to include the following two goals for VJ: U4) Support copying and modifying vs. creating from scratch, i.e., allow users to copy and modify existing items in a system as a way to create new ones and P5) Support various scenarios of learning, including examples, tutorials and exploration. One way we meet these goals is to provide generated Java code for VJ's users.

When a user performs actions in the direct manipulation editor (DME) or scripts new functionality in the iconic editor, the system translates those actions into their Java counterparts.

The Java code can then be viewed at anytime in the text editor. It is this code that makes up the user's executable program.

When a user is programming in the DME they are building the Main program that extends the WindowController class. As a user adds and manipulates objects the system keeps four global array lists. The first array list contains a list of Shapes (Java graphics objects). This list is used by the GUI and is not part of the code generation process. The second list contains the VJObjects that have been added to the canvas. The third and fourth lists contain the methods that have been called. Each method knows who it belongs to and

all other needed information: parameter list, value, and name. Which list a method call is added to depends on whether the system is in begin mode or event mode. If the system is in event mode then all actions are added to the event list, else they are added to the begin list.

When a user starts a program the system is in begin mode by default; all actions will occur in the program's begin method. The begin method is a mouse-independent event-handling method. When a begin method is defined in a program, it is executed once each time the program begins to execute. The begin method provides a way to specify instructions the computer should follow to set things up before the user begins interacting with the program. In other words, it creates the opening scene for the user's program. The user can switch to the while mode by adding an event to the Main program. The event methods are mouse dependent; the actions occur every time the user performs the mouse event in the executed program.

The way that the object method calls are generated differs from the way the canvases methods are generated. The rest of this section is intended to show how the system translates the user's action into the generated code.

When a user adds an object, the system does three things: 1) creates the VJObject and adds it the VJObject array list, 2) creates the Java graphics object that is visible to the user and adds it to the Shape array list, and 3) instantiates the object in either the begin method or an event method, depending on the mode. When a user adds an event the system does the following two things: 1) switches to event mode and 2) adds the event to the event array. If a user calls an object's method or adds a pause method the system does one of the following: 1) if in event mode, adds the method to the event list, else 2) if the event was

ended, adds the method to the begin list. When a user ends an event the system: 1) adds the method to the events list and 2) switches to begin mode. While performing these actions the user can switch to the code view and see the generated code at anytime.

The following code fragments will be used in a more detailed description of code generation process. The code fragments are from the example we provided in Section 3.3.1. The user's actions were detailed in that section and will not be repeated here.

When a user clicks on the canvas and switches to code view the system calls the CodeGenerator class' buildMainClass method. This method expects a list of VJObjects, and the two lists of executed methods. The buildMainClass method maintains a private string variable that builds the Main class. The string is initially set to the code that comprises the first four lines from Table 5. The method then loops through the object list and adds an instance variable for each object to the string. A VJObject knows its name, its type, and all its methods and variables information. The code generator extracts the information to generate the code that builds the 5th line in Table 5. The algorithm then checks to see if the array list of methods that were added while in begin mode is empty or not. If not, each method is taken out of the list and code is generated depending on which method it is. Different types of methods need to be generated differently because of who they belonged to, the canvas or the object, and whether they had parameters or not. The code generator extracts the information to generate the code that builds the method calls. This adds lines 7 – 10 to the string. Next, the event list is checked. If the event list is not empty the code generator loops through the list looking for different types of method calls. When it encounters an event method the string is appended with the method call (line 12). The methods that follow an event method in the array list are then built according to their type



(line 13) . When an end event is encountered the string is appended with an ending curly bracket (line 14). Before the string is returned line 16 is added.

```
1 import objectdraw.*;
2 import java.awt.*;
3
4 public class Main extends WindowController {
5     Square0 square;
6
7     public void begin() {
8         square = new Square0( new Location(3, 3), 30, 30, canvas );
9         square.setSize("Bigger");
10    }
11
12    public void onMouseClick( Location point ) {
13        square.setColor(Color.blue);
14    }
15
16 }
```

Table 5: Generated Code Fragment: Main.java

The object code is generated by calling the CodeGenerator class' buildObjectClass method. This class expects the VJObject that was clicked on as a parameter. The method maintains a string that is initially set to line 10 in Table 6, simply by extracting the type of the VJObject that was passed in. The method then loops through the VJObject's variable list to check for any user-defined variables. If there are any, the method builds the new variable instance and appends it to the string (line 11). The method extracts the information the variable knows about itself to generate the variable statement. The method then checks the VJObject's userMethods list to see if the user scripted any new methods. If the list is not empty the method extracts the VJMethod methodBody string to generate the rest of the objects class. The methodBody is built by calling the codeGenerator's buildMethodBody

method (lines 13-20). Before returning the string, the method appends an ending curly bracket to the class (line 22).

The `buildMethodBody` is called when a user clicks on the Save button from the details window. The `buildMethodBody` requires the `VJMethod` it is building. `VJMethods` know whom they belong to, their user-given names, and a string that contains the method body. The method body is built in one of the scripting editors. When it is made in the text editor there is no need to generate any code. But when the method is created in the iconic editor, the user's actions must be translated into their Java counterparts. To do this the system keeps a global list of the variables and the methods that are added to the new method. The variables and methods keep track of their user-defined values. Conditional statements are controlled in the same manner as event methods. The system uses modes and separate lists to know how to generate the code.

### ***1.4.3 Viability***

Viability is concerned with the practicality of the system: whether it is affordable, extensible, and so on. VJ is currently a research project and is neither for sale, nor are there any commercial prospects for the system. If VJ is offered to the public it will be under the GNU General Public License (GPL) and therefore free of charge.

As we stated at the beginning of this section our goal while implementing VJ was to keep the interfaces between layers small and well controlled, allowing us extreme flexibility. We also made the individual layers easily extensible. The VJ classes are built so that it can support added functionality. The current system only implements a small portion of the systems functionality.

```
1 import objectdraw.*;
2 import java.awt.Color;
3
4 public class Square0 extends VJSquare {
5     public Square0( Location initialLocation, int width, int height, DrawingCanvas aCanvas)
6         { super( initialLocation, (double)width, (double)height, aCanvas ); }
7
8     Color color = Color.blue;
9     String size = "Bigger";
10    boolean contains = false;
11    int num = 100;
12
13    public void run() {
14        num = 0 ;
15        while ( num < 100) {
16            move( "Random" );
17            VJShape.addPause();
18        }
19        num += 10;
20    }
21
22 }
```

Table 6: Generated code fragment: Square.java

Adding more object types or more built-in methods is as easy as duplicating the code for an implemented object or method and simply changing key portions. The educational content can also be easily extended in the same manner. The code generator needs only be told if it needs to build the objects or methods in a different fashion as already available.

The current system is a proof of concept. The system has been implemented to show that indeed the concepts are viable. However, it was important though to make the system robust since it was going to be used in an empirical evaluation with novice programmers.

We did not want to confuse or frustrate the programmers by having a poorly behaving environment.

## CHAPTER IV

### EMPIRICAL STUDY

An empirical study is designed in order to test hypotheses. Three techniques are applied in this study: (a) *background information questionnaires*, (b) *pre/post-study surveys*, and (c) *thinking aloud usability testing*.

Thinking aloud usability testing is used both (a) to look for problems that users experience when learning and using the tool, and also (b) to explore how well users who had been given the tool were capable of using its functionality. Both are achieved by observing a group of users performing some tasks typical of the tool's design intent. User and task analysis is used to find a right group of users and a good set of tasks for the thinking aloud usability testing technique. Thinking aloud usability testing is also used to obtain information about how the users had employed the tool.

Our goal for the usability study is to find out if our hypotheses are valid. The methodology we are using to evaluate the hypotheses is a common approach seen both in social sciences and Human-Computer Interaction (HCI) research. The participants were

chosen to match our target audience, and the tasks are representative of the steps a student performs while first learning to program.

## 1.1 Objective

Earlier I stated five objectives:

1. Show that novice programmers experience an improvement in programming concept comprehension after using VJ.
2. Show that novice programmers have a more positive attitude towards Computer Science and programming after using VJ.
3. Show that intermediate programmers perform as well as traditional programmers in terms of program completion rate and numbers of errors when attempting to program in a general language, and experiences concept comprehension equally.
4. Show that intermediate programmers exposed to a teaching language that more closely resembles a general-purpose language have fewer syntax errors when attempting to program in a general language.
5. Show intermediate programmers exposed to a teaching language that does not closely resemble a general-purpose language experience the same, if not better, concept comprehension.

I will assess my success in achieving these four objectives using two primary instruments: surveys and the programs the participants create during the study.

## 1.2 Methodology

1. *Show that novice programmers experience an improvement in programming concept comprehension after using VJ.*

I will collect and analyze the pre/post-study surveys completed by the participants.

Specifically, I will compare the answers on the surveys and look for an increase in

understanding of the following concepts: object instantiation, use of action commands with and with out parameters, defining and assigning variables, understanding and correct use of Java syntax, and understanding and construction of if/while statements. An increase in understanding will be defined by a higher score on the post-study survey than the pre-study survey. To factor out the impact of taking the same survey twice, I will ask a control group of participants who will not participate in the usability study to take the survey two times, separated by the length of the study.

2. *Show that novice programmers have a more positive attitude towards Computer Science and programming after using VJ.*

The instruments used to compile information about students' attitudes toward computers and programming came from two sets of data points: a Computer Attitude Survey (CAS) and background questions. The CAS consisted of items on a 5-point Likert-type response format. The background questions also collected are used to verify use of computers, a self-rating of computer and programming skill and future plans of the student's programming academics. I will add questions to the post-survey to specifically examine participants' interest and confidence in continuing to use VJ and other programming related activities. Furthermore, I will analyze behavioral differences and comments from the think-aloud study.

3. *Show that intermediate programmers perform as well as traditional programmers in terms of program completion rate and numbers of errors when attempting to program in a general language, and experiences concept comprehension equally.*

The participants will be split up in two main groups: 1) the traditional group and 2) the VJ group. I will collect and analyze the final programs created by the two groups during the user study. Specifically, I will compare the number of programming tasks each

participant completes and the number of errors they produce while completing the tasks. The tasks will consist of instantiating an object, creating simple methods, sending messages, and defining and using variables. Further, I will look for differences in participants' answers to the pre/post-study surveys and compare the concept comprehension scores of the two groups (see 1).

4. *Show that intermediate programmers exposed to a teaching language that more closely resembles a general-purpose language have fewer syntax errors when attempting to program in a general language.*

The VJ group will be further split up into the following two groups: 1) the VJ group and 2) the VJ/Text group. The VJ group will use the full VJ environment which provides a language paradigm that resembles a general purpose language. The VJ/Text group will only use the editors which do not resemble a general language. I will collect and analyze the programs created by the two groups during the study. Specifically, I will compare the number of syntax errors they generate while completing the final programming task.

5. *Show intermediate programmers exposed to a teaching language that does not closely resemble a general-purpose language experience the same, if not better, concept comprehension.*

I will collect and analyze the pre/post-study surveys completed by the participants in the VJ and VJ/Text groups. Specifically, I will compare the answers on the surveys and look for an increase in understanding of introductory programming concepts (see 1 and 2).



### ***1.2.1 Participants***

Twenty-Five participants altogether took part in our study. We recruited the participants through personal contacts, posters that were placed in various locations in the Computer Science department and emailed to group mailing lists, and the pre-CS1 programming courses offered through the department. About half of the total group were students in the pre-CS1 class and were given extra-credit by their professor for participating in the study. To participate in the study the participants had to self-identify with having little to no programming skill and no experience with Java.

Twelve of the participants were female and thirteen were male. Their ages ranged between 18 to 30 years old. In an entry questionnaire we asked them about their prior experience with computers and programming as well as their current GPA. The average GPA for the group was 3.3 (+- .3). The group was split 4-7-7-7. Four participants are in the control group that does not participate in the usability study. Seven participants were in each of the traditional, VJ, and VJ/Text groups. The groups were split-up randomly to ensure an even mixture of participants in the groups.

All of the participants rated their programming skill below 2 on a scale of 1 to 5, with five being advanced. They rated their computer skill between 3 and 4. Three of the twenty-five participants intend to major in Computer Science. Basically, all the participants had average computer skills, but little to no experience with programming and the concepts related to this study.

All the participants signed a consent form informing them that participation was voluntary, that they could terminate the study at any time and that all personal information would be kept confidential.

Once participants were screened they were asked to schedule an appointment to participate in the usability study.

### **1.2.2 Procedure**

Each appointment consisted of the following steps:

All participants are asked to first fill out the pre-study survey. The survey consists of: 1) background information questions, 2) attitude assessment questions in a 5-point Likert-type response format, and 3) a knowledge-based quiz on introductory programming concepts. The survey was emailed to the participants before they come in to the usability study. The participants are instructed via email that the survey is intended to gather background information about their experience with and attitudes of computers and programming. The participants are also told that the survey is to determine their current concept comprehension on introductory programming concepts. They are asked not to seek answers from outside sources and to only spend twenty minutes on the quiz section of the survey. Finally, they are ensured that they were not being tested and if they did not know the answers to simply state that they did not know.

Once the survey is emailed back to the investigator, the participants receive an introduction to object-oriented programming manual. All participants are emailed the 25-page manual that introduces basic programming concepts [Bruce, K., 2005b]. For each concept the statements are presented, grammar rules for the statement (e.g., definitions

of legal parameters) are provided, an example of the statement as it might occur in a line of a program is given. Subjects in both groups are given the same manual to read at their own rates before they come in for the usability study.

The control group completes the pre-study survey and reads the manual. Two hours later the participants complete the post-study survey. The data from the control group will factor out the impact completing the survey twice, separated by reading the manual, has on concept comprehension and attitude about programming.

Next, the participants who are not in the control group meet with the investigator in a private work space to participate in the usability section of the study. The participants are shown around the work area and the artifacts that are used in the study. The investigator gives a short description of the tasks they will be doing. The participants are given the opportunity to ask questions about the study. The artifacts shown to the participants are: the user's tasks and support documents, the computer, and either the VJ environment or the traditional environment. In addition to the protocols and questionnaires, we videotape sections of the study for later evaluation. After the introduction the participants are then asked to read and sign the consent form mentioned above. All the artifacts and documents used in the study can be found in Appendix A and B.

In the actual usability study the participants complete three main tasks: 1) interact with and manipulate existing example programs, 2) complete the post-study survey, and 3) complete a final programming task. Steps 1 and 3 are videotaped and use the think-aloud technique.

In step 1 the participants complete the first set of programming tasks. The participants are asked to use the think-aloud technique. In this technique the participants are asked to verbalize their thoughts while performing the tasks with the system. This set of tasks consists of three sub-tasks in which the participants modify the example programs in the manual. This step is the participant's initial introduction to authoring Java programs. The first sub-task is a simple program that paints a square in response to a mouse event. This program is an example of a simple event-driven program and is our equivalent to the "Hello World" program. The program teaches object instantiation, adding a mouse event and calling an object's method. The second sub-task is a program that simulates a stoplight. This program introduces classes, the if statement, and concepts from the first sub-tasks. The third sub-task is the most complex example program. This program creates a falling ball animation. This program involves problem-solving, making a new method, the while statement, creating a new variable, and use of the skills acquired from the first two sub-tasks.

It is worth noting that there is a small split in the procedure the three groups follow at this stage. The VJ group will create the example programs in the environment and then modify the program in the text editor – making use of all of VJ's language paradigms and learner support. The VJ/Text group will create the example programs but then modify the programs in the iconic editor – not using or viewing any text-based languages. The traditional group will interact with and modify existing programs in MS WordPad – following the traditional approach for introducing programming. This split in procedure is to highlight the different approaches' strengths and weaknesses.

All participants are asked to complete the following steps in all the sub-tasks: 1) Interact with the example program then answer questions. The VJ groups will create the example programs before answering the questions. 2) Modify the program and then answer questions, and 3) execute and interact with the modified program then answer questions. The modifications they make depend on the task. Each sub-task is meant to teach new concepts while enforcing the old concepts. The modifications focus on the new concepts, while the questions focus on the new and old concepts.

While the participants modify the example programs the investigator asks questions about the participant's understanding of the programming concepts in the examples. The questions are either direct that have a right or wrong answer or indirect, which are subjective. A direct question is like: what does line X of the code do. Indirect questions are like: what do you think will happen when you do X. The purpose of these questions is two-fold: 1) to remind the participants to think-aloud, and 2) to motivate the participants to think about the task instead of just going through the motions. The comments are also data that is used in the evaluation of the hypotheses.

The participants are then asked to complete the post-study survey. One goal of administering the post-study survey is to determine if the participant's concept comprehension increases after their initial introduction to programming. Along with some of the questions from the pre-study survey, the post-study survey also includes some new quiz questions. The new questions include: one new programming task and questions about the concepts taught in the example programs. The new programming task asks the participants to write a program, with pen and pencil, which is similar, but different from the falling ball example. They are asked to write it in Java code as much as possible. The

participants are given a copy of the API that was included in the manual. The concept questions concentrate on animation concepts.

Another goal of the post-study survey is to determine if the participants experience any change in attitude about computers and programming after their initial introduction. Therefore the post-survey repeats the attitude assessment questions from the pre-survey. Furthermore, the post-survey extends the CAS to include questions inquiring about the participants' interest and confidence in continuing to use VJ and other programming related activities.

Finally, the participants are asked to complete one final programming task. Both groups are asked to finish a partially implemented program. The program is a simple game that requires the users to complete 8 sub-tasks. The sub-tasks are no more than a couple lines of code each and use of the concepts introduced in the first programming tasks. Both groups will program in the traditional environment, i.e., the command line and MS WordPad. Because the traditional group will have had more experience using this environment, the VJ group is given a short tutorial of the environment. We believe that because the investigator provides assistance with the mechanics of using the environment to both groups, neither group has any advantages over the other that will effect the data.

Once the participants complete the study they are given the opportunity to ask any questions or provide any final input. They are also encouraged to make suggestions on how the VJ system could be improved. Finally, they are thanked for their time.

### ***1.2.3 The Traditional Approach***

One of our goals in this research is to show that intermediate programmers perform as well as traditional programmers in terms of certain factors. A traditional programmer is a programmer that is introduced to programming through a traditional approach. To introduce the participants to programming through the traditional approach, we must first define what the traditional approach is specifically. This section describes our interpretations of the traditional approach.

One of the first steps to describe the traditional approach is to determine the unit of knowledge. Traditionally, the main units of knowledge used in CS1 courses have been: 1) the statements, such as PRINT, IF, FOR, and 2) a program, such that the program contains the just taught statements. In order to teach these two units of knowledge, instructional sequences traditionally contain the following types of framework: 1) statement definition text devoted to presenting format and formal definition of the statement, 2) statement grammar text devoted to the grammatical rules relating to a statement, 3) program example a program that uses the statements and rules described in the statement definition and statement grammar frames, 4) programming exercises questions asking the learner to write or interpret a program containing statements discussed earlier. Generally, statements are defined and examples are given; the learner is encouraged to engage in "hands on" experience such as modifying programs and seeing what output comes out.

The difference between the traditional programmer and the intermediate programmer is their experiences in the hands-on step in learning programming. "Hands

on” programming is what we consider the initial introduction to programming. It is only in real working programs does a student reach glorious success or dishearten failure. This is a critical point in a novice programmer’s instruction, but often ignored. The traditional approach has been to use null and boring programs, archaic programming environments, and limited error support. Students are usually given code examples and asked to make specific changes to see the effect and learn the statements. Students rarely experience the joy of authoring a complete program from start to finish. Nor do they get to focus on the higher-level concepts since they must first struggle with the precision of the program's language.

The approach to introducing programming is the variable that we change to determine the effect each approach has on the participant's concept comprehension. Quantitative tests assume that your change in the independent variable (for example, the environment) influences the dependent variable (number of errors on the final program). This influence is called the experimental effect. However, if other factors are introduced into the design, the effect may be confounded, that is, not statistically valid due to tainting by the other factors.

#### **1.2.3.1 The Traditional Group**

Keeping with the traditional approach, the participants in the traditional group will use a simple text editor MS WordPad, and a DOS command line compiler as their environment. During the initial hands-on programming tasks, the participants in this group will use the environment to complete the following steps: 1) interact with the executed example program, 2) modify the code in a specific way, and 3) interact with the



modified program. The participants will modify the code by typing text and copying/cutting/pasting text.

To take into account possible confounding factors and eliminate possible sources of tainting, we break the traditional approach by using animated event-driven programs instead of the traditional I/O-based program. Evaluating the data would be impossible without this cohesion between the two groups.

As I stated, one of the hypotheses we want to evaluate is how a traditional programmer compares to an intermediate programmer a student introduced to programming through a pedagogical approach, in this case the VJ approach.

#### ***1.2.4 The VJ Approach***

A student's initial introduction to hands-on programming can have a great effect on the student's future with programming. The underlying goal of VJ is to support the traditional approach to introducing programming. We believe that introducing hands-on programming through the VJ approach supplements the traditional approach in a meaningful way.

The VJ approach uses programs that are similar to the types of programs the students are used to. Therefore, the programs are more interactive and object-oriented. The VJ programming environment supports novice programmers by: a) being simple and easy to use, b) providing error support, and c) supporting different styles of learning.

##### **1.2.4.1 The VJ Groups**

Participants in the VJ groups will perform the first hands-on programming tasks in the VJ environment. Before using the environment, the participants are given a short

tutorial of the system. The investigator introduces the participants to VJ's different editors and their functionality. Since VJ does not currently have a built in help, the participants are instructed that they can ask the investigator any questions and the investigator will answer it if he/she thinks a computer help system would be able to. We should mention that not a single participant ever asked any questions of the investigator while performing the tasks. This either attests to the usability of VJ or the idea that computer users do not use help systems.

For the programming tasks all the participants in the VJ groups will use the environment to complete the following steps: 1) interact with the executed example program, 2) re-create the program in one or more of VJ's editors and interact with the executed program, 3) modify the code in one of the two scripting editors, and 4) interact with the modified program.

How the VJ groups re-create and modify the example programs differentiates the VJ group and the VJ/Text group. The VJ group will use the DME and the iconic editor to recreate the programs. The participants will be asked to intermittently look at the generated code and answer questions about the code and their actions. They will also be asked to write down code fragments. These steps are to ensure that the participants are making full use of the text editor. Also, when participants in the VJ group modify the programs they will use the text editor. The participants will modify the code by typing text and copying/cutting/pasting text. This will force them to use real Java code.

The participants in the VJ/Text group will use the DME and iconic editor to re-create the programs as well. But, the participants will never look at or type a line of real

Java code. Instead, they will be asked to either modify the code in the iconic editor or asked to verbalize how the modification would effect the program. The VJ/Text group uses a language paradigm that does not closely resemble a general-purpose language. By limiting the participant to these two views they are not exposed to a general-purpose like language. The VJ group, on the other hand, is required to use a general-purpose like languages.

The two groups will provide us with the data to answer the other objective; to evaluate how language representations affect a programmer's concept and syntactic understanding. Specifically, we want to show that programmers introduced to programming with a language that more closely resembles a general language (the VJ group) has fewer syntactic errors when attempting to program in a general language than a programmer introduced to programming with a language that does not closely resemble a general language (the VJ/Text group). We also want to show that VJ/Text programmers experience the same, if not better, concept comprehension than the VJ group.

### **1.3 Evaluation Results**

We used three different sources for our evaluation results: 1) the surveys, 2) the example programs, and 3) data from the usability study, i.e., video recordings and investigator's notes.

The surveys provide us with the following: 1) subjective opinions of the participants, 2) preference data that allow us to quantify opinions using numerical scales, and 3) quantitative data that can be analyzed using statistics much as raw performance

data. The example programs gave us more quantitative data on the participant's performance. Performance data, like error rates and task completion, are evaluated by performing statistical analysis on the data set.

One purpose of the empirical experiment was to observe the target audience in the VJ environment. The video tape recordings gave us objective data about how effective VJ meets its usability goals. Think-aloud comments are especially useful for evaluating a system.

Data were analyzed using a one-way analysis of variance (ANOVA) to ascertain if there was a significant pre-study knowledge difference or pre-study attitude difference between the three groups we evaluate in the experiment. Alpha level was set at .05. Results of the ANOVA indicated there were no significant differences between the pretest scores of the students in the VJ groups and students in the control group and traditional group. These results were to be expected, as the students had not received prior instruction concerning areas specifically covered in the test and because the groups were randomized. The data also indicates that at the beginning of the study the knowledge of the students concerning introductory programming was similar.

At the beginning of this chapter, we presented some hypotheses we wanted to prove. The following discusses how successful we were in this regard.

*1. Novice programmers show an improvement in concept comprehension after using VJ.*

In order to evaluate this hypothesis, we look at data from the knowledge-based tests in the pre/post-study surveys. For this thesis an improvement in concept comprehension

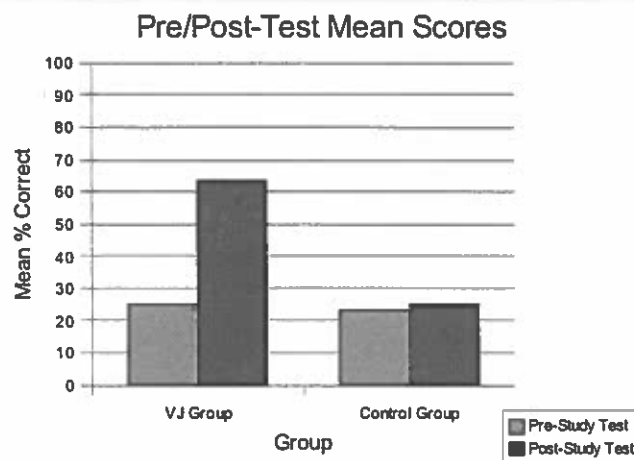
is shown through an increase in the score between the knowledge-based pre-test (test 1) and the knowledge-based post-test (test 2). To be able to conclude that any difference in the scores of test 1 and test 2 is due to the activities the participants completed in the VJ environment, we test the VJ group's data against the control group's data. The control group is to factor out the impact the other activities (reading the manual and taking the survey twice) have on the group's scores. The data from the same test 1 and test 2, were analyzed to answer the following question: Does the VJ Group show an increase in score on the knowledge-based section of the pre/post-study survey that is not due to extraneous factors?

The knowledge-based tests were independently scored according to a solution with a specific guide for assigning points. The test consisted of true/false, multiple choice, and right/wrong coding problems. The resulting scores are based on percent of problems correct per survey. The individual scores of each participant in a group was aggregated and a mean score was extracted per survey.

In order to ascertain if there was a significant knowledge difference between the VJ group and the control group after the study, data were analyzed using an analysis of covariance (ANCOVA). Alpha level was set at .05. Results of the ANCOVA indicate that a statistically significant relationship exists between the VJ group and control group's adjusted mean knowledge-based post-test (test 2) scores with the knowledge-based pretest (test 1) as the covariate.

The VJ group increased their scores on test 2 (64 %) from test 1 (25%) - showing a 39% increase in concept comprehension. The control group, on the other hand, only

showed an increase of 1.5% (from 23.5% to 25%). Graph 1 presents the magnitude of the environments impact on the participant's concept comprehension.



Graph 1: Comparison of concept comprehension scores

Two conclusions can be drawn from this data: 1) the use of VJ as a learning aid is effective in increasing novice programmers' knowledge of introductory programming concepts. 2) VJ is a more effective tool for increasing students' knowledge than just reading an instructional manual.

*2. Novice programmers have a more positive attitude towards Computer Science and programming after using VJ.*

In order to evaluate this hypothesis, we look at data from the computer attitude survey (CAS) in the pre/post-study surveys. For this thesis a more positive attitude towards Computer Science is shown through an increase in the score between the pre-study attitude survey (survey 1) and the post-study attitude survey (survey 2). The survey was based on a five-point Likert scale. Participants were asked to respond to statements concerned with their attitude towards computers and programming. Five on the scale

corresponded with strongly agree, three corresponded with neither disagree nor agree, and one corresponded with strongly disagree.

To be able to conclude that any difference in the scores of survey 1 and survey 2 is due to the activities the participants completed in the VJ environment, we test the VJ group's data against the control group's data. The control group is to factor out the impact the other activities (reading the manual and taking the survey twice) have on the group's scores. The data from the same survey 1 and survey 2, were analyzed to answer the following question: Does the VJ Group show an increase in score on the attitude section of the pre/post-study survey that is not due to extraneous factors?

Data from the pre-attitude survey (survey 1) and the post-attitude survey (survey 2) were analyzed using an analysis of covariance (ANCOVA) to ascertain if a significant attitude difference between the VJ and control groups existed after the study (see Table 7). Alpha level was set at .05. The ANCOVA indicated a statistically significant relationship between the VJ group and control group's adjusted mean scores on post-attitude survey (survey 2) with pre-attitude survey (survey 1) as the covariate

The VJ group increased their scores on survey 1 (mean score 4.25) from survey 1 (mean score 1.75) - showing a 44% improvement in attitude regarding computers and programming. The control group, on the other hand only showed an increase of 4% (from mean score 1.80 to 2).

Two conclusions can be drawn from this data: 1) Users experience an improvement in attitude regarding computers and programming after being introduced to programming through VJ. (2) VJ is a more effective tool for improving students' attitude about computers and programming than just reading an instructional manual.

3. *Intermediate programmers perform as well as traditional programmers in terms of program completion rate and numbers of errors when attempting to program in a general language, and experiences concept comprehension equally.*

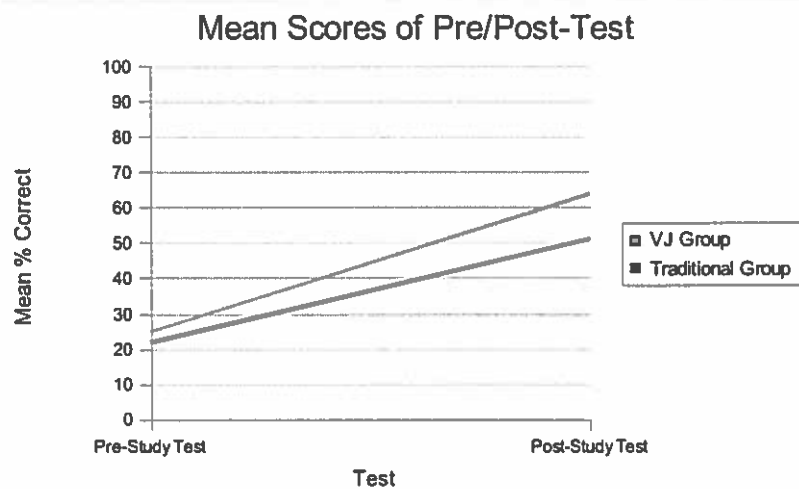
In order to evaluate this hypothesis, we look at data from the knowledge-based tests in the pre/post-study surveys and the data from the final programming task in the study.

To be able to conclude whether the VJ group and traditional group experience equal concept comprehension after the study, we compare the VJ group's knowledge-based tests against the control group's tests. The data from the same knowledge-based pretest (test 1) and the knowledge-based post-test (test 2), were analyzed to answer the following question: Does the VJ Group and traditional group show a similar increase in score on the knowledge-based section of the pre/post-study survey?

In order to ascertain if a significant knowledge difference existed between the VJ group and the traditional groups after the study, data were analyzed using an analysis of covariance (ANCOVA). Alpha level was set at .05. Results of the ANCOVA indicate that no statistically significant relationship existed between the VJ group and the traditional group's adjusted mean knowledge-based post-test (test 2) scores with the knowledge-based pretest (test 1) as covariate. However, the final mean score of the VJ group was higher than the mean score of the traditional group on the post-test (64% vs. 51.5%).

Having no statistical significance for this data is as expected. We are only trying to show that the VJ group performs as well as the traditional group. Results are summarized in Graph 2.





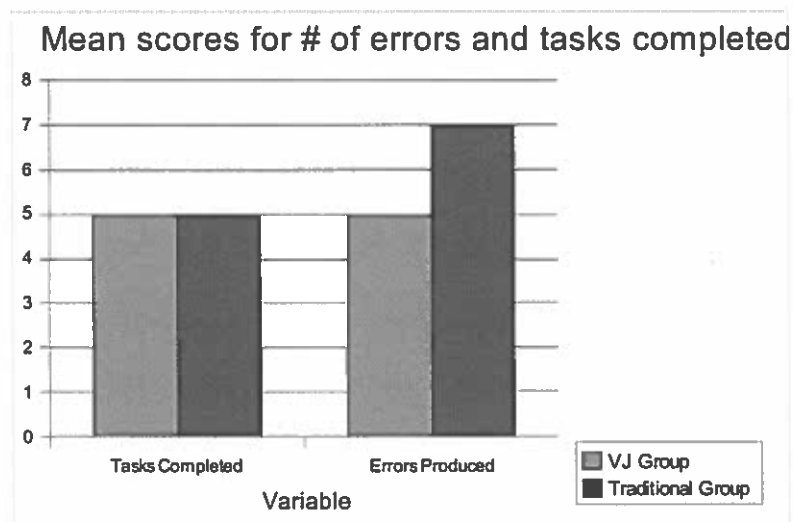
Graph 2: Trend of concept comprehension scores of the VJ and traditional groups

We can conclude the following from this data: 1) Intermediate programmers and traditional programmer's experiences concept comprehension equally. 2) An initial introduction to programming through VJ results in programmers that have a similar increase of programming concept comprehension as programmers taught the same concepts through a traditional approach.

To be able to conclude whether the VJ group performs as well as the traditional group in terms of program completion rate and numbers of errors produced when programming in a general language. The VJ group's mean number of tasks completed and number of errors produced on the final programming task, and the traditional groups same mean numbers, were analyzed to answer the following question: Does the VJ Group and control group complete an equivalent number of tasks and produce a similar number of errors?

In order to ascertain if a significant difference in tasks completed and number of errors produced existed between the VJ group and the traditional group after

programming in a general-purpose language, data were analyzed using an analysis of covariance (ANCOVA). Alpha level was set at .05. Results of the ANCOVA indicate that no statistically significant relationship existed between the VJ group's adjusted mean scores with the traditional group's mean scores as the covariate. However, the final mean score of the number of errors was less for the VJ group (5 vs. 7), while completing the same number of tasks (5 vs. 5), than the traditional group. Graph 3 presents the magnitude of the environments impact on the number of tasks completed and errors produced.



Graph 3: Comparison of number of programming tasks and errors produced

We can conclude the following from this data: 1) Intermediate programmers and traditional programmers perform equally in terms of number of tasks completed and errors produced when first programming in a general language. 2) An initial introduction to programming through VJ results in programmers that can transition to a general language and perform as well as traditional programmer.

4. *Intermediate programmers exposed to a teaching language that more closely resembles a general-purpose language have fewer syntax errors than when attempting to program in a general language.*

In order to evaluate this hypothesis, we look at data from participants' final programming task in the study. We use the data from the VJ group and compare it to the data from the VJ/Text group's data. Syntax errors are defined as: a mistake in a statement's set of allowed reserved words and their parameters and the correct *word order* in the statement, i.e., a missing expected semi-colon.

To be able to conclude whether the VJ group performs better than the VJ/Text group in terms of numbers of syntax errors when attempting to program in a general language. The VJ group's mean number of syntax errors produced on the final programming task, and the traditional group's same mean number, were analyzed to answer the following question: Does the VJ Group produce a smaller number of errors than the VJ/Text group?

In order to ascertain if a significant difference in number of errors produced existed between the VJ group and the VJ/Text group after programming in a general-purpose language, data were analyzed using an analysis of covariance (ANCOVA). Alpha level was set at .05. Results of the ANCOVA indicate that no statistically significant relationship existed between the VJ group's adjusted mean scores with the VJ/Text group's mean scores as the covariate. However, the final mean score of the number of tasks completed was more for the VJ group (6 vs. 4), while producing the same number of errors (5 vs. 5), than the VJ/Text group.

We can not make any statistically significant conclusions from this data. The small numbers of participants in each group and the small number of lines of code the participants had to write may have affected the results. Further work will need to be done to conclude on whether programmers who are introduced to programming through a language that more closely resembles a general language produce fewer syntax errors than programmers introduced with a less general-like language.

*5. Intermediate programmers exposed to a teaching language that does not closely resemble a general-purpose language experience the same, if not better, concept comprehension.*

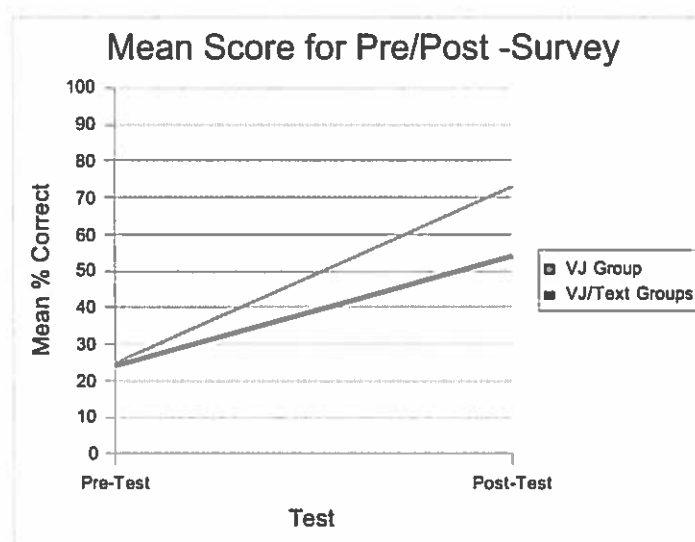
In order to evaluate this hypothesis, we look at data from the participant's final programming task in the study. We use the data from the VJ group and compare it to the data from the VJ/Text group's data. To be able to conclude whether the VJ group and VJ/Text group experience equal concept comprehension after the study, we compare the VJ group's knowledge-based tests against the VJ/Text group's tests. The data from the same knowledge-based pretest (test 1) and the knowledge-based post-test (test 2), were analyzed to answer the following question: Does the VJ Group and VJ/Text group show a similar increase in score on the knowledge-based section of the pre/post-study survey?

In order to ascertain if a significant knowledge difference existed between the VJ group and the VJ/Text group after the study, data were analyzed using an analysis of covariance (ANCOVA). Alpha level was set at .05. Results of the ANCOVA indicate that no statistically significant relationship existed between the VJ group and the VJ/Text group's adjusted mean knowledge-based post-test (test 2) scores with the knowledge-

based pretest (test 1) as covariate. However, the final mean score of the VJ group was higher than the mean score of the VJ/Text group on the post-test (test 2).

Having no statistical significance for this data is as expected. We are only trying to show that the VJ/Text group performs as well as the VJ group in terms of concept comprehension. The VJ group does have a slightly higher post-test score than the traditional group (73% vs. 54%). Results are summarized in Graph 4.

We can conclude the following from this data: 1) Programmers introduced to programming with a language that does not resemble a general language experiences concept comprehension equally as programmer introduced with a more general-like language. 2) An initial introduction to programming through a language that is not like a general-purpose language results in programmers that can transition to a general language and perform as well as those introduced with a more general-purpose like language.



Graph 4: Trend of concept comprehension scores

### **1.3.1 Evaluation Implications**

To answer the research questions we posed in the beginning of this thesis we implemented a pedagogical environment, VJ, conducted an experiment with novice programmers, and performed statistical methods on quantitative data from the experiment to conclude on specific hypotheses. In this section we discuss the implications the results from the experiment have on our research questions.

*RQ1: What effect does VJ have on intermediate programmers in terms of program completion rate, interest level, and concept comprehension compared to the traditional approach to introducing programming?*

As we expected VJ has a similar effect as the traditional approach in terms of program completion rate, interest level, and concept comprehension. We have statistically shown the following: 1) Intermediate programmers introduced to programming through VJ experience the same increase in concept comprehension as traditional programmers, 2) intermediate programmers introduced to programming with VJ perform as well as traditional programmers in terms of task completion and errors produced when first programming in a general-purpose language, and 3) the use of VJ as a learning aid is effective in increasing novice programmers knowledge of introductory programming concepts. From this data we can conclude: 1) an initial introduction to programming through VJ results in programmers that can transition to a general language and perform as well as traditional programmer. 2) An initial introduction to programming through VJ results in programmers that have a similar increase of programming concept comprehension as programmers taught the same concepts through a traditional approach.

Showing that the VJ approach is as good as the traditional approach in introducing introductory programming concepts is a positive result. The underlying goal in developing VisualJava is to enhance traditional approaches to teaching and learning programming. The environment lends no support to learning programming if it does not enhance a student's understanding of programming concepts at least as well as a more traditional approach. Similarly, if programmers, who learn the concepts through VJ, can not transfer that knowledge to a general-purpose language, then we have done the students a great disservice. While not significant, the raw data from the study indicates that VJ outperforms the traditional approach in both concept comprehension and transfer of knowledge in the programming tasks.

*RQ2: How do different language paradigms affect novice programmers' concept comprehension and syntactic understanding?*

We expected that languages that more closely resemble a general-purpose language will have a positive effect on novice programmers' syntactic understanding and concept comprehension. While a language that does not closely resemble a general language will have a positive effect on novice programmers' concept comprehension, it has a negative effect on their syntactic understanding. We have shown that: Programmers introduced to programming with a language that does not resemble a general language experiences concept comprehension equally as programmer introduced with a more general-like language. This lets us conclude: An initial introduction to programming through a language that is not like a general-purpose language results in programmers that can transition to a general language and perform as well as those introduced with a more general-purpose like language. We can not make any statistically significant conclusions on whether

programmers who are introduced to programming through a language that more closely resembles a general language produce fewer syntax errors than programmers introduced with a less general-purpose like language. The raw data does suggest that programmers introduced with a more general-purpose like language do produce fewer syntax errors per number of programming tasks completed.

Showing that the language representation does not affect programmers concept comprehension is a positive result. By showing this we have shown that the language representation is not important in teaching higher level concepts. Students that are introduced to programming in a less syntax strict and more user-centered language show an equal increase in understanding of introductory programming concepts as students introduced with a more general-purpose like language.

These results imply that educators can choose to wait to introduce syntax until after they have covered higher level concepts. Learners can make use of the myriad of programming tools that teach programming through a no-type approach or programming by demonstration, and be confident that they will be able to transfer the knowledge they learn in those systems to learning a more general-purpose language. Finally, educators can feel confident using a pedagogical tool without a general-purpose like language as a teaching aid.



## CHAPTER V

### CONCLUSION

We presented pedagogical tools as a way to increase the number of students who succeed in CS1 courses by lowering the barriers novice programmers face. The main focus of this thesis was to answer two questions: 1) What effect does a pedagogical tool have on intermediate programmers in terms of program completion rate, interest level, and concept comprehension compared to the traditional approach to introducing programming, and 2) How do different language paradigms affect novice programmers' concept comprehension and syntactic understanding?

In order to do this we designed and implemented a pedagogical programming environment, VisualJava (VJ), conducted an experiment with VJ and novice programmers, and performed statistical analysis on quantitative data from the experiment.

VisualJava (VJ) is a pedagogical programming environment for teaching introductory programming concepts. VJ uses object-oriented concepts as its foundation and teaches event driven programming as a way to motivate and empower its users.

VisualJava is designed to supplement, not supersede, the traditional approach to

introducing programming. The underlying goal in developing VisualJava is to enhance traditional approaches to teaching and learning programming.

VJ's user interface and functionality was designed with a user-centered approach. We conducted pre-design interviews and performed user and task analysis to lead the design of VJ. A paper-prototype was used in an informal user study with the target population. Results from the user study and standard HCI heuristics were used in an iterative design of the current VJ interface. Task and competitive analysis, literature review, and previous research dictated how and what programming concepts VJ teaches.

The key idea of VJ is to combine a powerful, but intuitive, graphical user interface (GUI) with a simplified Java 2D graphics library, ObjectDraw. VJ introduces students to the mechanics of writing event-driven Java programs through a mixture of three alternative programming paradigms: a direct manipulation language, an iconic language, and the text-based Java language. Authoring a program in VJ consists of two steps: (1) setting scenes, and (2) scripting additional functionality. A user can author programs in any of VJ's three programming languages. VJ generates the code for the user to view and edit in the text-based language.

VisualJava's architecture's development strategy was to look for the best "off the shelf" solutions for each of VJ's components while not becoming wedded to any one solution. Our goal was to keep the interfaces between layers small and well-controlled, allowing us extreme flexibility. The current system is comprised of a Java Swing GUI, the ObjectDraw library, the VisualJava classes, the glue code that makes the ObjectDraw library callable in the VJ and user classes.

We used the VJ prototype to conduct an experiment with novice programmers to evaluate the following five hypotheses:

- 1) Novice programmers experience an improvement in programming concept comprehension after using VJ.
- 2) Novice programmers have a more positive attitude towards Computer Science and programming after using VJ.
- 3) Intermediate programmers perform as well as traditional programmers in terms of program completion rate and numbers of errors when attempting to program in a general language, and experiences concept comprehension equally.
- 4) Intermediate programmers exposed to a teaching language that more closely resembles a general-purpose language have fewer syntax errors when attempting to program in a general language.
- 5) Intermediate programmers exposed to a teaching language that does not closely resemble a general-purpose language experience the same, if not better, concept comprehension.

We have statistically shown the following:

- Intermediate programmers introduced to programming through VJ experience the same increase in concept comprehension as traditional programmers.
- Intermediate programmers introduced to programming with VJ perform as well as traditional programmers in terms of task completion and errors produced when first programming in a general-purpose language.
- The use of VJ as a leaning aid is effective in increasing novice programmers' knowledge of introductory programming concepts.
- Programmers introduced to programming with a language that does not resemble a general language experiences concept comprehension equally as programmer introduced with a more general-like language.

From this data we concluded:

- An initial introduction to programming through VJ results in programmers that can transition to a general language and perform as well as traditional programmer.
- An initial introduction to programming through VJ results in programmers that have a similar increase of programming concept comprehension as programmers taught the same concepts through a traditional approach.
- An initial introduction to programming through a language that is not like a general-purpose language results in programmers that can transition to a general language and perform as well as those introduced with a more general-purpose like language.

Pedagogical tools are not a new topic. Research in the area has been conducted for the last 40 years resulting in hundreds of pedagogical languages, tools, and environments, each taking a slightly different approach. As far as we know we are one of the first to look at the effect of a pedagogic tool on intermediate programmers – programmers who are introduced to programming through a pedagogical tool and then want to apply that experience to learning a general-purpose language. We provided data on how pedagogical tools compare to a traditional approach of introducing programming concepts. In addition, through VJ's multiple language paradigms we provided data on how different language paradigms affect a novice programmer's concept comprehension and syntactic understanding. Finally, we contributed valuable data through quantitative empirical methods that can be used by future researchers and educators when designing and evaluating new teaching systems.

We believe that pedagogical tools, like VJ, can be effective in introducing programming to novice programmers. Pedagogical tools can help a novice prepare for a CS1 type course. They can also be used by educators in programming courses at any introductory level as an instructional aid or as the preferred teaching language or

environment. Event-driven programming and the object-first approach is an effective way to interest and motivate beginning programmers. We believe tools or approaches that use these paradigms should be encouraged in introductory programming courses.

**APPENDIX A**  
**STUDY ARTIFACTS**

This appendix provides the artifacts that were used in the evaluation study.

## Pre-Test Survey

### Participants' Background Questionnaire

This questionnaire is designed to elicit specifics about your history and experiences in Computer Science and programming. If you do not want to answer any of the questions, please just indicate N/A. All information will be kept in confidentiality.

#### Personal Information

Age:

Major:

Year in School:

GPA:

Highest Math Class:

#### Please explain you background in programming:

1. What programming classes have you taken (college and K-12)?
2. Have you ever programmed in Java?
  - If yes, please explain (when, what, why).
3. What other programming languages have you been exposed to?
4. What is the biggest program you have written by yourself (approximate the number of lines of code)?
5. What is harder for you when solving programming problems; coming up with a solution or Coding the Solution? Explain.
6. Please indicate on a scale of 1(beginner), 4(average), and 7 (advanced):
  - My level of programming is
  - My level of problem-solving is
  - My level in Mathematics is

## CAS (survey 1)

This Survey is designed to elicit information about your attitude and interests in regards to programming and Computer Science.

### Directions

Please note that your answers will be kept confidential.

On the following pages are a series of statements.

1. Read each statement.
2. Think of the extent to which you agree or disagree with each statement
3. Mark your response on the answer sheet

Please remember:

- There are no right or wrong answers. Don't be afraid to put down what you really think.
- Don't spend a lot of time on any one item. Move quickly!
- Complete all of the items.

**Respond to the following questions, using the following scale:**

- 5) strongly agree
- 4) agree, but with reservations
- 3) neutral, neither agree nor disagree
- 2) disagree, but with reservations
- 1) strongly disagree

1. I am interested in computer science.
2. I am sure that I can learn programming.
3. I have a lot of self-confidence when it comes to programming.
4. I'm no good at programming.
5. I don't think I could do advanced computer science.
6. I'm not the type to do well in computer programming.
7. Knowing programming will help me earn a living.
8. Computer science is a worthwhile and necessary subject.
9. I will use programming in many ways throughout my life.
10. Programming is of no relevance to my life.
11. Programming will not be important to me in my life's work.
12. I see computer science as a subject I will rarely use in my daily life.
13. Taking computer science courses is a waste of time.
14. I like writing computer programs.
15. Programming is enjoyable and stimulating to me.
16. Figuring out programming problems does not appeal to me..
17. Programming is boring.



## Knowledge-Based Test (test 1)

This test is designed to determine your knowledge of introductory programming concepts. Please note that your score will be kept confidential.

### Directions

**You have 30 minutes to complete the test.**

On the following pages are a series of problems. The problems are in the form of multiple choice, true and false, and coding questions. Please mark an answer to each question – I do not know is an acceptable answer. Answer the coding questions in code, pycdo-code, or plain English.

Please remember:

- *You* are not being tested. Your score will only determine your current concept comprehension.
- Don't spend a lot of time on any one item. Move quickly!

**Multiple-Choice Questions: Circle the most correct response.**

**Q 1:** What are semicolons ; used for in Java?

- a) Statement Terminators
- b) Statement Separators
- c) Variable Terminators
- d) I don't know

**Q 2:** What are curly brackets {} used for in Java?

- a) To define scope
- b) To end methods
- c) To start a class
- d) I don't know

**Q 3:** What is a Java object?

- a) An instance of a class variable.
- b) An instance of a software bundle of associated information and related behavior.
- c) A template that describes the data and behavior associated of a certain type of thing.
- d) I don't know

**Q 4:** What is Java class?

- a) An instance of a software bundle of associated information and related behavior.
- b) An instance of a class variable.

- c) A template that describes the data and behavior associated with all objects of a certain kind.
- d) I don't know

**Q 5:** What are variables?

- a) Information that is passed to a method
- b) The data associated with a class or object
- c) The behavior associated with a class or object
- d) I don't know

**Q 6:** What are methods?

- a) The behavior associated with a class or object
- b) A message that is called from another class
- c) The data associated with a class or object
- d) I don't know

**Q 7:** What are parameters?

- a) A variable that is passed to a method
- b) A method that is called from another class
- c) Information about an object
- d) I don't know

**Q 8:** What is the difference between an if statement and a while statement?

- a) The actions in an if statement occurs only once if the condition is true, the actions in a while statement occur until the condition is false
- b) Nothing
- c) The actions in a while statement occurs only once if the condition is true, the actions in a if statement occur until the condition is false.
- d) I don't know

**True/False Questions: Answer True or False**

**Q 9:** If the object sun is declared and created inside of a method you can then call sun.move() in a different method.

**Q 10:** A variable of type int can be assigned to a word.

**Q 11:** An object of type Square can be assigned to a variable of type Circle.

**Q 12:** All methods require parameters.

**Q 13:** All Java statements must end in a semicolon ;

**Coding Questions: write your response in the space below.**

**Q 13:** Use mutator methods to write the following messages for our light variable:

```
FilledOval light = new FilledOval( new Location(100, 100), 10, 10, canvas );
```

- a. make the light bigger

- b. move the light from where it is at to the corner of a 300 by 300 canvas
- c. make the light hide

**Q 14:** a. Declare an instance variable for a new FilledRect object, name it square.

b. Write an assignment statement for the square variable.

**Q 15:** Write a method that draws an oval when the mouse is clicked on the canvas. The circle should fit in a box that is 100 by 100. Place the circle at location 20, 80.

## Post-Test Survey

### CAS (survey 2)

This Survey is designed to elicit information about your attitude and interests in regards to programming and Computer Science.

#### Directions

Please note that your answers will be kept confidential.

On the following pages are a series of statements.

1. Read each statement.
2. Think of the extent to which you agree or disagree with each statement
4. Mark your response on the answer sheet

Please remember:

- There are no right or wrong answers. Don't be afraid to put down what you really think.
- Don't spend a lot of time on any one item. Move quickly!
- Complete all of the items.

**Respond to the following questions, using the following scale:**

- (1) strongly agree
- (2) agree, but with reservations
- (3) neutral, neither agree nor disagree
- (4) disagree, but with reservations
- (5) strongly disagree

1. I am interested in computer science.
2. I am sure that I can learn programming.
3. I have a lot of self-confidence when it comes to programming.
4. I'm no good at programming.
5. I don't think I could do advanced computer science.
6. I'm not the type to do well in computer programming.
7. Knowing programming will help me earn a living.
8. Computer science is a worthwhile and necessary subject.
9. I will use programming in many ways throughout my life.
10. Programming is of no relevance to my life.
11. Programming will not be important to me in my life's work.
12. I see computer science as a subject I will rarely use in my daily life.
13. Taking computer science courses is a waste of time.
14. I like writing computer programs.
15. Programming is enjoyable and stimulating to me.
16. Figuring out programming problems does not appeal to me..
17. Programming is boring.
18. I am sure I could use a tool like VJ to learn programming
19. I am interested in using a tool like VJ
20. I liked programming in VJ

## Knowledge-Based Test (test 2)

This test is designed to determine your knowledge of introductory programming concepts. Please note that your score will be kept confidential.

### Directions

**You have 20 minutes to complete the test.**

On the following pages are a series of problems. The problems are in the form of multiple choice, true and false, and coding questions. Please mark an answer to each question – I do not know is an acceptable answer. Answer the coding questions in code, psudo-code, or plain English.

Please remember:

- *You* are not being tested. Your score will only determine your current concept comprehension.
- Don't spend a lot of time on any one item. Move quickly!

**Multiple-Choice Questions: Circle the most correct response.**

**Q 1: What are semicolons ; used for in Java?**

- a) Statement Terminators
- b) Statement Separators
- c) Variable Terminators
- d) I don't know

**Q 2: What are curly brackets {} used for in Java?**

- a) To define scope
- b) To end methods
- c) To start a class
- d) I don't know

**Q 3: What is a Java object?**

- a) An instance of a class variable.
- b) An instance of a software bundle of associated information and related behavior.
- c) A template that describes the data and behavior associated of a certain type of thing.
- d) I don't know

**Q 4: What is Java class?**

- a) An instance of a software bundle of associated information and related behavior.
- b) An instance of a class variable.
- c) A template that describes the data and behavior associated with all objects of a certain kind.
- d) I don't know

**Q 5:** What are variables?

- a) Information that is passed to a method
- b) The data associated with a class or object
- c) The behavior associated with a class or object
- d) I don't know

**Q 6:** What are methods?

- a) The behavior associated with a class or object
- b) A message that is called from another class
- c) The data associated with a class or object
- d) I don't know

**Q 7:** What are parameters?

- a) A variable that is passed to a method
- b) A method that is called from another class
- c) Information about an object
- d) I don't know

**Q 8:** What is the difference between an if statement and a while statement?

- a) The actions in an if statement occurs only once if the condition is true, the actions in a while statement occur until the condition is false
- b) Nothing
- c) The actions in a while statement occurs only once if the condition is true, the actions in a if statement occur until the condition is false.
- d) I don't know

**Q 9:** What is the Pause method for?

- a) To show the changes to a graphical object
- b) To stop the program
- c) To make the graphical objects stop
- d) I don't know

**True/False Questions: Answer True or False**

**Q 9:** If the object sun is declared and created inside of a method you can then call

sun.move() in a different method.

**Q 10:** A variable of type int can be assigned to a word.

**Q 11:** An object of type Square can be assigned to a variable of type Circle.

**Q 12:** All methods require parameters.

**Q 13:** All Java statements must end in a semicolon ;

**Coding Questions: write your response in the space below.**

**Q 13:** Use mutator methods to write the following messages for our light variable:

```
FilledOval light = new FilledOval( new Location(100, 100), 10, 10, canvas );
```

a. make the light bigger

b. move the light from where it is at to the corner of a 300 by 300  
canvas

c make the light hide

**Q 14:** a. Declare an instance variable for a new FilledRect object, name it square.

b. Write an assignment statement for the square variable.

**Q 15:** Write a method that draws an oval when the mouse is clicked on the canvas. The circle should fit in a box that is 100 by 100. Place the circle at location 20, 80.

**Q 16:** Write a class, Elevator that meets the following specifications: an elevator is a rectangle that is longer than it is shorter. It starts from the bottom and goes up to the top floor (the height of the canvas minus the height of the elevator) and stops. Reminder: the canvas is 300 by 300.

**APPENDIX B**  
**Programming Tasks**



## Traditional Group's First Programming Tasks

### Programming Task 1: 20 minutes per task

For each task please complete the following sub-task:

1. Interact with the executed example program.
2. Modify the code
3. Interact with the modified code

#### Programming Task 1: ClickMeApp program

After interacting with the program, view the program's code.

- What method is the Square created in?
- Write down the code that creates the square.
- Write down the code that sets the squares color

Change the ClickMeApp program in the following ways:

- Enter smaller or larger values where we had used the numbers 10 and 10.
  - How will the modified program behave?
  - What will be different?
- Interchange the bodies of the two methods so that the construction appears in onMouseRelease and the canvas.clear is in onMousePress.
  - How will the modified program behave?
  - What will be different?
- Run the modified code

#### Programming Task 2: FallingBall.java

After interacting with the program, view the program's code.

- What method is the Circle created in?
- Write down the code that creates the Circle.
- Will the circle be on the canvas when the program starts?
  - If no, when will the circle be displayed?

Change the FallingBall.java program in the following ways:

- Change the code so that the light starts at green and when the mouse is pressed turns yellow and when it is released turns red.
- Change the code so that the light is located 20 pixels left and 20 pixels down from the canvas top left corner. Hint: top left corner is at 0,0.

#### Programming Task 3: FallingBall.java

Change the FallingBall program in the following ways:

- Have the FallingBall move left-to-right by twice its current speed .
- Change the pause method to a smaller and bigger value.
  - What is the effect of the squares movement?
- Change the while loop to an if statement.
  - How is the program effected?

## VJ Group's First Programming Tasks

### **Programming Task 1: 50 minutes total**

**For each task please complete the following sub-tasks:**

1. Interact with the example program
2. Re-Create the program
3. Modify the program
4. Interact with the modified program

Learn how to make and run a program in VisualJava **[give short tutorial]**.

### **Programming Task 1: ClickMeApp program**

After interacting with the program, re-create the program by following these steps:

- Add a Square Object
- Click on the Canvas and look at the Code in Code View
  - What method is the Square created in?
  - Write down the code that creates the square.
- Right Click on the object and set its color
- Add a Mouse Pressed Event
- Drag the Square away from the corner of the canvas
- Right click on the square and hide it
- End the Event
- Add a Mouse Released Event
- Right click on the square and show it
- End the Event
- Run the Code: What happened?

Change the ClickMeApp program in the following ways:

- Enter smaller or larger values where we had used the numbers 10 and 10.
  - How will the modified program behave?
  - What will be different?
- Interchange the bodies of the two methods so that the construction appears in onMouseRelease and the canvas.clear is in onMousePress.
  - How will the modified program behave?
  - What will be different?

### **Programming Task 2: FallingBall.java**

After interacting with the program, re-create the program by following these steps:

- Clear the canvas
- Add a Mouse Click Event
- Add a Circle object
- Look at the Canvas Code
  - What method is the Circle created in?
  - Write down the code that creates the Circle.
- Move it to the top and over from the corner of the canvas
- End the Event
- Right click on the circle and create a new method for the ball: call it run
- Make a new variable: bottom, int, 300
- Add the bottom variable to the run method and set it to 0
- Click on the Methods tab
- Create a While Loop with the condition that bottom <300
- Add the move method, set it to move "down"
- Add the bottom variable and add 10 to it
- End the While loop
- Add the delete method
- Save the run method
- Click on the Methods tab, and edit your run method
- Look at the run method's Code
  - Write down the while loop:
- Click Save, Done, then Design view
- Click on the canvas and look at the Canvas code
  - Will the circle be on the canvas when the program starts?
  - If no, when will the circle be made?
- Compile the code:What is going to happen?
- Run the Example:What happened?

### Programming Task 3: FallingBall.java

- In the Circle's Java Code make the Following changes
  - Change the While loop to an if.
    - What will be different?
    - Compile / Run?
    - Whats different?
  - Change the condition to test if bottom is less than a smaller or larger value
    - What will be different?
    - Compile / Run?
    - Whats different?

## All Group's Final Programming Task

### Final Programming Task : 20 min

For our final program we will use the concepts we just learned to finish a simple game that randomly displays and removes “creatures” on a canvas. The point is to try to catch as many of the creatures as you can, by clicking on them. But, the player should also get points deducted for missing. The game needs to display the player's current point. And include instructions to begin and restart the game. The game ends when either the points equal 10 or points is less than zero.

Open `CatchtheCreature.java` and `Creature.java` You will see places in the code with the following text:

```
//Fill in the code to do ( a description)
```

The description will include a very high level explanation of what you need to do. In these sections you are to add the code to try to accomplish the goal. It is recommended that you work on a piece of code and then compile to check your changes before you move on to another part of the code. You may need to add in variables or make new methods to accomplish the goal of the description.

## **APPENDIX C**

### **Protocol**

(Selection)

## Script

### Introductions

Members: Kim Griggs

#### Purpose

“This is for a Mater Thesis project in the Computer and Information Science dept. ”

#### Background Information

“We have developed a programming environment that teaches introductory programming concepts to novice programmers. We would like to test how usable our new tool is by having some people use it to perform some tasks we have outlined. Remember, we are testing the product, not you; so if you have trouble, it's the product's fault, not yours.”

### OK to quit

“Though we don't know of any reason this should happen, if you would like to quit at any time, you are free to do so.”

### Equipment

#### Camera

“We will be recording this session with a video camera. We will focus the camera on the computer screen, keyboard and your hands, avoiding any shots of your body and face. We will also use the camera to record your voice.”

#### Computer

“Our project is a programming environment that teaches introductory programming concepts to novice programmers. We will use this laptop to interact with the development environment. **[pause, follow up if participant hesitates]**

### Think Aloud

“We ask that you think aloud as you work through a task, saying what comes to mind as you work. It might be a little awkward at first, but it's really easy once you get used to it. All you have to do is speak your thoughts as you work. If you forget to think aloud, I'll remind you to keep talking. Would you like me to demonstrate?” **[demonstrate think aloud]**

### No Helping

“Once you begin the tasks we provide, I will not be able to help you. Even though I cannot answer your questions, please ask them anyway. It's very important that I capture all of your questions and comments. Remember, we are testing the product, not you. So if there is a problem you cannot readily solve it is the product's fault, not yours. After you complete all of the tasks, I will be happy to answer any remaining questions at that time.”

### Tasks

“I will provide for you a short list of tasks to complete. There will be a brief description of the task and specific goals for you to accomplish. Please try to complete the tasks in the order provided.” **[show all the tasks]**

### Consent

“We ask that you sign a standard consent form. It basically states that...”  
**[summarize consent form, give consent form for signature]**

### Questions?

“Remember: once you begin, I will not be able to answer any questions. Also remember to think aloud as you work. Before we begin, do you have any questions?” **[answer any questions]**

### [Begin Observation]

#### Task1: First Programming Tasks

Length: 40 minutes – Traditional Group, 50 minutes – VJ Groups

Artifacts: Programming tasks, environment, and computer

Purpose: An initial introduction to programming

#### Task 2: Post-Study Survey

Length: 20 minutes

Artifacts: test 2 and survey 2

Purpose: To determine concept comprehension

#### Task 3: Final Programming Task

Length: 20 minutes

Artifacts: Programming tasks, MS wordpad and DOS command line

Purpose: To determine transition success

### [Conclude Observation]

“Thank you. Do you have any remaining questions?” **[answer questions]**

### Post-evaluation

“During your work you did **[behavior]** which I found interesting. Can you explain what this was about?”

“How would you improve this product? Would you ever use something like this?”



## CONSENT FORM

You are invited to participate in a research study conducted by Kimberly Griggs, from the University of Oregon, Computer and Information Science (CS) department. The data I gather from this study will be used in my Master's thesis.

If you decide to participate, you will take part in a user study in which you will test the effectiveness of a new programming environment. The user study will include an introduction to the environment, and tasks that you will be asked to complete in the environment. The study will take place at 205 Deshutes hall, and will include one, two-hour session. The study poses no known risk or inconveniences to you, but you are able to leave at any time during the study if you so choose.

Any information that is obtained in connection with this study and that can be identified with you will remain confidential and will be disclosed only with your permission. Subject identities will be kept confidential by the use of an ID that will not be connected to your name or information. While the information is being analyzed it will be kept in my private office at 247 Deschutes hall. Once the analysis is completed all information from you will be destroyed.

Your participation is voluntary. Your decision whether or not to participate will not affect your relationship with the PI or University in any means. If you decide to participate, you are free to withdraw your consent and discontinue participation at any time without penalty.

If you have any questions, please feel free to contact the principal investigator, Kim Griggs at 346- 0375, 1202 University of Oregon Eugene, OR 97403, or her advisor, Michal Young at 346- 4140. If you have questions regarding your rights as a research subject, contact the Office for Protection of Human Subjects, University of Oregon, Eugene, OR 97403, (541) 346-2510. You have been given a copy of this form to keep.

Your signature indicates that you have read and understand the information provided above, that you willingly agree to participate, that you may withdraw your consent at any time and discontinue participation without penalty, that you have received a copy of this form, and that you are not waiving any legal claims, rights or remedies.

Print Name \_\_\_\_\_

Signature \_\_\_\_\_

Date \_\_\_\_\_

## **ASSENT FORM**

### **Cover Letter for Anonymous, Non-Sensitive Questionnaires**

I would appreciate your assistance with this research project on lowering the barriers to programming. The project is being conducted by graduate student Kimberly Griggs, from the University of Oregon Computer and Information Science (CS) department, for the purpose of gaining data for a Master's thesis. The research will help me understand factors that effect the success of novice programmers in the CS major and tools that can be used to lower the barriers to programming.

All you need to do is complete this short questionnaire, which should take approximately one hour. Your participation is voluntary. If you do not wish to participate, simply discard the questionnaire. Responses will be completely anonymous; your name will not appear anywhere on the survey. Completing and returning the questionnaire constitutes your consent to participate.

Keep this letter for your records. If you have any questions regarding the research, contact the principal investigator, Kim Griggs at 346-0375, 1202 University of Oregon Eugene, OR 97403, or her advisor, Michal Young at 346-4140. If you have any questions regarding your rights as a research subject, please contact the Office for Protection of Human Subjects at the University of Oregon at (541) 346-2510. Thank you again for your help.

**DATA COLLECTION FORM**

**Participant ID:**

**Task Number:**

**Comments made by participant:**

**Errors or problems observed ( include any assistance that was needed):**

**Other:**

## BIBLIOGRAPHY

- Allen, E., Cartwright, R., & Stoler, B. (2002). DrJava: a lightweight pedagogic environment for Java. *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky, February 27 - March 03, 2002). SIGCSE '02. ACM Press, New York, NY, 137-141.
- Bereiter, C. & Ng, E. (1991). Three Levels of Goal Orientation in Learning. *Journal of the Learning Sciences*, Vol. 1, pp 243-271.
- Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. (2001). *Karel J. Robot: A gentle introduction to the art of object-oriented programming*. Retrieved April, 6, 2006, from: <http://csis.pace.edu/bergin/KarelJava2ed/Karel++JavaEdition.html>.
- Bergin, J., Stehlik, M., Roberts, J., & Pattis, R (1996). *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley & Sons, Inc., New York, NY.
- Bruce, K. B. (2005). Controversy on how to teach CS 1: a discussion on the SIGCSE-members mailing list. *SIGCSE Bull.* 37, 2 (Jun. 2005), 111-117.
- Bruce, K. B., Danyluk, A., & Murtagh, T. (2001). A library to support a graphics-based object-first approach to CS 1. *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education* (Charlotte, North Carolina, United States). SIGCSE '01. ACM Press, New York, NY, 6-10.
- Bruce, K. B., Danyluk, A., & Murtagh, T.(2001). Event-Driven Programming Facilitates Learning Standard Programming Concepts. *Conference Proceedings on Object-Oriented Programming, Systems, Languages, and Applications* ( Canada, Act 24-28 2001). OOPSLA '04. ACM Press, New York, NY, 119-125.
- Bruce, K. B., Danyluk, A., & Murtagh, T. (2005). *An Eventful Approach*. Prentice Hall press, Englewood Cliffs, NJ.
- Budge, B. (1983). *Pinball Construction Set*, Available through: Exidy Software.

- Cockburn, A. & Bryant, A. (1998). Cleogo: Collaborative and multi-metaphor programming for kids. *3rd Asia Pacific Conference on Computer Human Interaction* (Japan). 187–192.
- Cockburn, A. & Bryant, A. (1997). Leogo: An equal opportunity user interface for programming. *J. Visual Lang. Comput.* 8, 5-6. 601–619.
- Cohen, P. R., Dalrymple, M., Moran, D. B., Pereira, F. C., & Sullivan, J. W. (1989). Synergistic use of direct manipulation and natural language. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Wings For the Mind* K. Bice and C. Lewis, Eds. CHI '89. ACM Press, New York, NY, 227-233.
- Conway, M., Audia, S., Burnette, T., Cosgrove, D., & Christiansen, K. (2000). Alice: lessons learned from building a 3D system for novices. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (The Hague, The Netherlands, April 01 - 06, 2000). CHI '00. ACM Press, New York, NY, 486-493.
- Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Nevada, USA, February 19 - 23, 2003). SIGCSE '03. ACM Press, New York, NY, 191-195.
- Cooper, S., Dann, W., & Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts. *Proceedings of the Fifth Annual CCSC Northeastern Conference on the Journal of Computing in Small Colleges* (Ramapo College of New Jersey, Mahwah, New Jersey, United States). J. G. Meinke, Ed. Consortium for Computing Sciences in Colleges. Consortium for Computing Sciences in Colleges, 107-116.
- Cypher, A. & Smith, D. C. (1995). KidSim: end user programming of simulations. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Denver, Colorado, United States, May 07 - 11, 1995). I. R. Katz, R. Mack, L. Marks, M. B. Rosson, and J. Nielsen, Eds. Conference on Human Factors in Computing Systems. ACM Press/Addison-Wesley Publishing Co., New York, NY, 27-34.
- Cypher, A. (1993). *Watch what I do: Programming by Demonstration*. MIT Press, Cambridge, MA.
- Dijkstra, E.W. (1989). On the Cruelty of Really Teaching Computing Science, *CACM*, Vol. 32, No. 12, December 1989, page 1404.

- Fenton, J. & Beck, K. (1989). Playground: an object-oriented simulation system with agent rules for children of all ages. *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (New Orleans, Louisiana, United States, October 02 - 06, 1989). OOPSLA '89. ACM Press, New York, NY, 123-137.
- 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* (Gallipoli, Italy, May 25 - 28, 2004). AVI '04. ACM Press, New York, NY, 358-361.
- Hsia, J. I., Simpson, E., Smith, D., & Cartwright, R. (2005). Taming Java for the classroom. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (St. Louis, Missouri, USA, February 23 - 27, 2005). SIGCSE '05. ACM Press, New York, NY, 327-331.
- Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., & Doyle, K. (1988). Fabrik: a visual programming environment. *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (San Diego, California, United States, September 25 - 30, 1988). N. Meyrowitz, Ed. OOPSLA '88. ACM Press, New York, NY, 176-190.
- Kaasboll, J. (2000). *Learning and Teaching Programming*. Lecture at Interface 2000, University of Pretoria, 19-20 May, 2000.
- Kahn, K. (1996). ToonTalk - An Animated Programming Environment for Children, *Journal of Visual Languages and Computing*, 197-217.
- Kelleher, C. & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.* 37, 2 (Jun. 2005), 83-137.
- Kolling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *J. Comput. Science Educ., Special Issue of Learning and Teaching Object Technology* 12, 4, 249-268.
- Kolling, M., Quig, B., Patterson, A., & Rosenberg, J. (1996a.) Blue—A language for teaching object-oriented programming. *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, PA. 190-194.
- Kolling, M. & Rosenberg, J. (1996b). An object oriented program development environment for the first programming course. *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, PA. 83-87.

- Lewis, T. L. & Smith, W. J. (2005). The computer science debate: it's a matter of perspective. *SIGCSE Bull.* 37, 2 (Jun. 2005), 80-84.
- Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13, 1 (Mar. 1981), 121-141.
- McDaniel, R. G. & Myers, B. A. (1999). Getting more out of programming-by-demonstration. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: the CHI Is the Limit* (Pittsburgh, Pennsylvania, United States, May 15 - 20, 1999). CHI '99. ACM Press, New York, NY, 442-449.
- Moritz, S. H. & Blank, G. D. (2005). A design-first curriculum for teaching Java in a CS1 course. *SIGCSE Bull.* 37, 2 (Jun. 2005), 89-93.
- Nielsen, J. 1993 *Usability Engineering*. Morgan Kaufmann Publishers Inc, San Fransisco, CA.
- Norman, D. (1986). *Cognitive engineering*. In Norman, D. and Draper, S., Eds. *User Centered System Design: New Perspectives on Human- Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Norman D.A. & Spohrer J.C., (1996). Learner Centered Design: Introduction, *Comm. ACM* vol. 39 N.4, April 1996, pp. 24-27
- Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, NY.
- Pattis, R. (1981). *Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal*. John Wiley & Sons, New York, NY.
- Perkins, D. N., Schwart S., & Simmons, R. (1988). *Instructional Strategies for the Problems of Novice Programmers*. In R. E. Mayer (ed), *Teaching and Learning Computer Programming*, pp 153-178, Lawrence Erlbaum Associates.
- Repenning, A. (1993). Agentsheets: A tool for building domain-oriented visual programming. *Conference on Human Factors in Computing Systems*, 142-143.
- Repenning, A. & Ambach, J. (1996). Tactile programming: A unified manipulation paradigm supporting program comprehension, composition, and sharing. *IEEE Symposium on Visual Languages*, Boulder, CO. 102-109.
- Shaffer, S. C. (2005). Ludwig: an online programming tutoring and assessment system. *SIGCSE Bull.* 37, 2 (Jun. 2005), 56-60.
- Sheard, J. & Hagan, D. (1998) Our Failing Students: A Study of a Repeat Group. *Proceedings of ITiCSE 98*, pp 223-227.

- Sheehan, R. (2004). The Icicle programming environment. *Proceeding of the 2004 Conference on interaction Design and Children: Building A Community* (Maryland, June 01 - 03, 2004). IDC '04. ACM Press, New York, NY, 147-148.
- Smith, D. (1993). *Pygmalion*. In Cypher, A., Ed. *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA.
- Smith, D. C., Cypher, A., & Spohrer, J. (1994). KidSim: programming agents without a programming language. *Commun. ACM* 37, 7 (Jul. 1994), 54-67.
- Smith, D. C., Cypher, A., & Tesler, L. (2000). Programming by example: novice programming comes of age. *Commun. ACM* 43, 3 (Mar. 2000), 75-81.
- Smith, D. C., Cypher, A., & Schmucker, K. (1996) Making programming easier for children. *Interactions*, 3(5):58-67, 1996.
- Spohrer, J. & Soloway, E. (1989). *Novice Mistakes: Are the Folk Wisdoms Correct?* In E. Soloway and J. C. Spohrer (eds), *Studying the Novice Programmer*, pp 401-416, Lawrence Erlbaum Associates.
- Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (Sep. 1986), 850-858.
- Tanimoto, S. & Runyan, M. (1986). Play: An iconic programming system for children. In Chang, S. K., Ichikawa, T. and Ligomenides, P. A., Eds. *Visual Languages*. Plenum Publishing Corp. 191-205.
- Tucker, A., Deek, F., Jones, J., McCowan, D., Stephenson, C. & Verno, A. (2002). A Model Curriculum for K-12 Computer Science: *Report of the ACM K-12 Education Task Force Computer Science Curriculum Committee – Draft*. Retrieved on April 5, 2006 at: <http://www.acm.org/k12/k12Draft1101.pdf>.