

**The Impact of Different Teaching Approaches and Languages on
Student Learning of Introductory Programming Concepts**

A Thesis

Submitted to the Faculty

of

Drexel University

by

Wanda M. Kunkle

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

September 2010

UMI Number: 3430595

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 3430595

Copyright 2010 by ProQuest LLC.

All rights reserved. This edition of the work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346



Office of Graduate Studies Dissertation/Thesis Approval Form

This form is for use by all doctoral and master's students with a dissertation/thesis requirement. Please print clearly as the library will bind a copy of this form with each copy of the dissertation/thesis. All doctoral dissertations must conform to university format requirements, which is the responsibility of the student and supervising professor. Students should obtain a copy of the Thesis Manual located on the library website.

Dissertation/Thesis Title: The Impact of Different Teaching Approaches and Languages on Student Learning of Introduction of Program Concepts

Author: WANDA M. KUNKLE

This dissertation/thesis is hereby accepted and approved.

Signatures:

Examining Committee

Chair

Row B. All

Members

Steph Gen

[Signature]

J. O. [Signature]

CCCC (DISSENT)
(Michael E. Atwood)

Academic Advisor

Row B. All

Department Head

Susan Wederbeck

Copyright 2010
Wanda M. Kunkle. All Rights Reserved.

DEDICATIONS

I dedicate this dissertation to the memory of

Nathan J. Kunkle

the loving father who encouraged me to become the person I was meant to be

and

the highly respected educator who inspired me to follow in his footsteps.



Nathan J. Kunkle (1907 – 1965)

ACKNOWLEDGEMENTS

I wish to thank my advisor, Robert Allen, and the members of my dissertation committee for their support and guidance in doing the research and writing required to complete this work.

I also wish to thank the Computer Science instructors who helped me recruit the students who participated in my research study, as well as the students themselves. Without the students, there would have been no study.

I especially wish to thank the friends and family who encouraged me to keep going when I began to falter because I was feeling overwhelmed by the amount of time and effort I needed to put in to complete this dissertation. Some of those friends are on my dissertation committee, so they are getting thanked twice.

Finally, a special thanks goes out to my cousin, Durand Kunkle, who regularly called to ask how I was progressing with my dissertation. He has been planning the celebration for some time now.

TABLE OF CONTENTS

LIST OF FIGURES	ix
ABSTRACT	xiii
CHAPTER 1:INTRODUCTION	1
1.1 Problem Statement	1
1.2 Novice Teaching Approaches in Practice	2
1.3 Contribution to Computer Science Education Research	5
CHAPTER 2:APPROACHES FOR TEACHING OBJECT-ORIENTED PROGRAMMING	6
2.1 Brief Descriptions	6
2.2 Detailed Descriptions	7
2.2.1 <i>Objects-First Approach Using BlueJ</i>	7
2.2.2 <i>Objects-First or Objects-Early Approach Using Alice</i>	9
2.2.3 <i>Imperative-First Approach Using Python</i>	12
2.3 Distinctive Features	14
CHAPTER 3:REVIEW OF NOVICE PROGRAMMER LITERATURE	16
3.1 Programming Language Knowledge	16
3.1.1 <i>Procedural and Functional Languages</i>	16
3.1.2 <i>Object-Oriented Languages</i>	17
3.2 Programming Misconceptions	18
3.3 Scaffolding for Learning Object-Oriented Programming	20
3.3.1 <i>Definition of Scaffolding</i>	20
3.3.2 <i>Key Features of Scaffolding</i>	21

3.3.3	<i>Scaffolding to Support Learning to Program</i>	21
3.4	Theories of Learning to Program	22
3.4.1	<i>Constructivism</i>	22
3.4.2	<i>Assimilation Encoding Theory</i>	24
3.4.3	<i>Advance Organizers</i>	25
3.5	Model of Object-Oriented Programming	26
3.6	Transfer in Computing Environments	26
3.6.1	<i>Definition of Transfer</i>	26
3.6.2	<i>Theories of Transfer</i>	26
3.6.3	<i>New Learning as Transfer from Previous Learning</i>	27
3.6.4	<i>Transfer Between Programming Languages and Text Editors</i>	28
CHAPTER 4: EDUCATIONAL ASSESSMENT		29
4.1	Assessment in Computing	29
4.2	Principles of Assessment	30
4.3	Approaches to Assessment	30
4.4	Existing Tools	31
4.4.1	<i>Mathematics, Science, and Engineering</i>	31
4.4.2	<i>Computing</i>	32
4.5	A Tool for Introductory Computing	33
4.5.1	<i>Rationale</i>	33
4.5.2	<i>Characteristics of the Tool</i>	34
4.5.3	<i>Tool Development</i>	41
4.5.4	<i>Reliability and Validity of the Tool</i>	42

CHAPTER 5:PILOT STUDY	46
5.1 Purpose	46
5.2 Participants	46
5.3 Task	46
5.4 Results	47
5.5 Analysis	49
CHAPTER 6: RESEARCH QUESTIONS AND HYPOTHESES	51
6.1 Research Questions	51
6.2 Related Hypotheses	52
CHAPTER 7:EXPERIMENTAL STUDY DESIGN	53
7.1 Purpose of Study	53
7.2 Description of Methodology (General)	53
7.2.1 <i>Time Dimension</i>	53
7.2.2 <i>Units of Analysis</i>	53
7.2.3 <i>Units of Observation (Participants)</i>	53
7.2.4 <i>Constructs</i>	54
7.2.5 <i>Terms or Components</i>	54
7.2.6 <i>Level of Measurement</i>	55
7.3 Description of Methodology (Detailed)	55
7.3.1 <i>Description</i>	55
7.3.2 <i>Research Questions</i>	55
7.3.3 <i>Related Hypotheses</i>	56
7.3.4 <i>Methodology</i>	56

7.3.5	<i>Data Analysis</i>	57
CHAPTER 8:MAIN STUDY DESCRIPTION AND RESULTS		58
8.1	Purpose	58
8.2	Participants	58
8.3	Task	60
8.4	Results	60
8.5	Assessment Tool	61
8.5.1	<i>Scoring Procedure</i>	61
8.5.2	<i>Item Response Pattern Analysis</i>	61
8.5.3	<i>Item Discrimination Analysis</i>	67
8.6	Student Performance	82
8.6.1	<i>Score Distribution Analysis</i>	82
8.6.2	<i>Mean Scores Analysis by Language Group</i>	83
8.6.3	<i>Topic Area Analysis by Language Group</i>	88
8.6.4	<i>Code-Completion Analysis by Language Group</i>	94
CHAPTER 9:DISCUSSION		96
9.1	Importance of Computer Science Education Research	96
9.2	Assessment Instrument Performance	97
9.3	Student Performance	100
9.4	Implications for Computer Science Education	105
9.5	Future Work	106

LIST OF REFERENCES	109
APPENDIX A: DEMOGRAPHIC SURVEY DATA FOR PILOT STUDY	123
APPENDIX B: PRE-TEST RESULTS FOR PILOT STUDY	124
APPENDIX C: POST-TEST RESULTS FOR PILOT STUDY	125
APPENDIX D: PRE-TEST QUESTION RESPONSES BY QUARTILE FOR MAIN STUDY	126
APPENDIX E: POST-TEST QUESTION RESPONSES BY QUARTILE FOR MAIN STUDY	127
APPENDIX F: ITEM DISCRIMINATION ANALYSIS FOR PRE-TEST FOR MAIN STUDY	128
APPENDIX G: ITEM DISCRIMINATION ANALYSIS FOR POST-TEST FOR MAIN STUDY	130
APPENDIX H: CORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR PRE-TEST FOR MAIN STUDY	132
APPENDIX I: CORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR POST-TEST FOR MAIN STUDY	134
APPENDIX J: INCORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR PRE-TEST FOR MAIN STUDY	136
APPENDIX K: INCORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR POST-TEST FOR MAIN STUDY	139
APPENDIX L: DEMOGRAPHIC SURVEY DATA FOR MAIN STUDY	142
APPENDIX M: DEMOGRAPHIC SURVEY FOR MAIN STUDY	145
APPENDIX N: ATTITUDE SURVEY FOR MAIN STUDY	146
APPENDIX O: COMPUTER CONCEPTS SURVEY FOR MAIN STUDY	148
APPENDIX P: COMPUTER CONCEPTS SURVEY REVIEW FORM	155
VITA	158

LIST OF FIGURES

Figure 2-1. An example BlueJ project: <i>people</i>	8
Figure 2-2. An example BlueJ class template: <i>Administration</i>	9
Figure 2-3. An example Alice 2.0 world: <i>First Encounter</i>	11
Figure 2-4. An example Alice 2.0 world: <i>First Encounter</i>	11
Figure 2-5. Source code for part of a Python program that calculates compound interest.....	13
Figure 2-6. GUI for a Python program that calculates compound interest.....	14
Figure 3-1. Model of Assimilation Encoding Theory (Mayer, 1979).....	25
Figure 4-1. Pseudocode for Euclidean Algorithm (Gersting, 2007).....	40
Figure 4-2. Pseudocode for BinarySearch Algorithm (Gersting, 2007).....	40
Figure 5-1. Mean pre-test (session one), post-test (session two), and percent change scores for pilot study (categorization scheme one).....	48
Figure 5-2. Mean pre-test (session one), post-test (session two), and percent change scores for pilot study (categorization scheme two).....	49
Figure 8-1. Proportion of students who chose each question response for session one (pre-test).....	62
Figure 8-2. Proportion of students who chose each question response for session two (post-test).....	63
Figure 8-3. Proportion of session two students who chose each response for questions 5, 9, 19, 20, and 23, divided into four groups by score (high to low).....	64
Figure 8-4. Session one (pre-test) and session two (post-test) item discrimination analysis for questions 1 through 11.....	70
Figure 8-5. Session one (pre-test) and session two (post-test) item discrimination analysis for questions 13 through 16, 18, and 20 through 24.....	72

Figure 8-6. Session one (pre-test) and session two (post-test) item discrimination analysis for questions 12, 17, and 19.	73
Figure 8-7. Session one (pre-test) reliability, average score, standard deviation, and standard error of measurement statistics for questions 1 through 24. Reliability is calculated using K-R21.	75
Figure 8-8. Session two (post-test) reliability, average score, standard deviation, and standard error of measurement statistics for questions 1 through 24 (left); and questions 1 through 24, minus questions 12 and 19 (right). Reliability is calculated using K-R21.	75
Figure 8-9. Cronbach’s alpha values for session one (pre-test) and session two (post-test).	77
Figure 8-10. Factors representing the construct “understanding of fundamental programming concepts,” extracted through principal component analysis.	81
Figure 8-11. Session one (pre-test) and session two (post-test) scores for the 61 students who completed both parts of the study.	82
Figure 8-12. Mean scores for session one (pre-test) and session two (post-test) by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.	84
Figure 8-13. Mean “basics” scores for pre-test and post-test by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.	89
Figure 8-14. Mean “logical expressions” scores for pre-test and post-test by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.	89
Figure 8-15. Mean “classes and objects” scores for pre-test and post-test by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.	92
Figure 8-16. Mean “code-completion” scores for pre-test and post-test by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.	94
Figure A-1. Demographic survey data for pilot study.	123
Figure B-1. Pre-test results for pilot study.	124
Figure C-1. Post-test results for pilot study.	125

Figure D-1. Proportion of pre-test students who chose each response for questions 1 through 12, divided into quartiles by score (high to low).....	126
Figure D-2. Proportion of pre-test students who chose each response for questions 13 through 24, divided into quartiles by score (high to low).....	126
Figure E-1. Proportion of post-test students who chose each response for questions 1 through 12, divided into quartiles by score (high to low).....	127
Figure E-2. Proportion of pos-test students who chose each response for questions 13 through 24, divided into quartiles by score (high to low).....	127
Figure F-1. Pre-test item discrimination analysis for questions 1 through 12.....	128
Figure F-2. Pre-test item discrimination analysis for questions 13 through 24.....	128
Figure F-3. Reliability, average score, standard deviation, and standard error of measurement statistics for pre-test.....	129
Figure G-1. Post-test item discrimination analysis for questions 1 through 12.....	130
Figure G-2. Post-test item discrimination analysis for questions 13 through 24.....	130
Figure G-3. Reliability, average score, standard deviation, and standard error of measurement statistics for post-test.....	131
Figure H-1. Correct response percentages for questions 1 through 8 of pre-test.....	132
Figure H-2. Correct response percentages for questions 9 through 16 of pre-test.....	132
Figure H-3. Correct response percentages for questions 17 through 24 of pre-test.....	133
Figure I-1. Correct response percentages for questions 1 through 8 of post-test.....	134
Figure I-2. Correct response percentages for questions 9 through 16 of post-test.....	134
Figure I-3. Correct response percentages for questions 17 through 24 of post-test.....	135
Figure J-1. Incorrect response percentages for questions 1 through 8 of pre-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.....	136
Figure J-2. Incorrect response percentages for questions 9 through 16 of pre-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.....	137
Figure J-3. Incorrect response percentages for questions 17 through 24 of pre-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.....	138

Figure K-1. Incorrect response percentages for questions 1 through 8 of post-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.....	139
Figure K-2. Incorrect response percentages for questions 9 through 16 of post-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.....	140
Figure K-3. Incorrect response percentages for questions 17 through 24 of post-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.....	141
Figure L-1. Demographic information for the 14 Java students; the highlighted records represent students who only completed the first part of the study.....	142
Figure L-2. Demographic information for the 29 Visual Basic students; the high- lighted records represent students who only completed the first part of the study.....	143
Figure L-3. Demographic information for the 26 C++ students; all of the C++ students completed both parts of the study.....	144
Figure M-1. Demographic survey, pages 1 and 2.....	145
Figure N-1. Attitude survey, pages 1 and 2.....	146
Figure N-2. Attitude survey, page 3.....	147
Figure O-1. Computer concepts survey, pages 1 and 2.....	148
Figure O-2. Computer concepts survey, pages 3 and 4.....	149
Figure O-3. Computer concepts survey, pages 5 and 6.....	150
Figure O-4. Computer concepts survey, pages 7 and 8.....	151
Figure O-5. Computer concepts survey, pages 9 and 10.....	152
Figure O-6. Computer concepts survey, pages 11 and 12.....	153
Figure O-7. Computer concepts survey, pages 13 and 14.....	154
Figure P-1. Computer concepts survey review form, page 1.....	155
Figure P-2. Computer concepts survey review form, page 2.....	156
Figure P-3. Computer concepts survey review form, page 3.....	157

Abstract

The Impact of Different Teaching Approaches and Languages on Student Learning of Introductory Programming Concepts

Wanda M. Kunkle
Robert B. Allen, Ph.D. Supervisor

Many students experience difficulties learning to program. They find learning to program in the object-oriented paradigm particularly challenging. As a result, computing educators have tried a variety of instructional methods to assist beginning programmers. These include developing approaches geared specifically toward novices and experimenting with different introductory programming languages. However, having tried these different methods, computing educators are faced with yet another dilemma: how to tell if any of these interventions actually *worked*?

The research presented here was motivated by an interest in improving practices in computer science education in general and improving my own practices as a computer science educator in particular. Its purpose was to develop an instrument to assess student learning of fundamental and object-oriented programming concepts, and to use that instrument to investigate the impact of different teaching approaches and languages on students' ability to learn those concepts.

Students enrolled in programming courses at two different universities in the Mid-Atlantic region during the 2009-2010 academic year participated in the study. Extensive data analysis showed that the assessment instrument performed well overall. Reliability estimates ranged from 0.65 to 0.79. The instrument is intrinsically valid since the questions are based on the core concepts of the *Programming Fundamentals* knowledge

area defined by the 2008 ACM/IEEE curricular guidelines. Support for content validity includes: 71% of correct responses varied directly with the students' scores; all possible responses were selected at least once; and 21 out of 24 questions discriminated well between high and low scoring students. CS faculty reviewers indicated that 19 out of 24 questions reflected basic concepts and should be used again "as is" or with "minor changes." Factor analysis extracted three comprehensible components, "methods and functions," "mathematical and logical expressions," and "control structures," suggesting the instrument is on its way to effectively representing the construct "understanding of fundamental programming concepts."

Statistical analysis revealed significant differences in student performance based on language of instruction. Analyses revealed differences with respect to overall score and questions involving assignment, mathematical and logical expressions, and code-completion. Language of instruction did not appear to affect student performance on questions addressing object-oriented concepts.

CHAPTER 1: INTRODUCTION

1.1 Problem Statement

Learning to program is a difficult undertaking for many students. Of the programming language paradigms currently taught, the object-oriented paradigm is especially hard for beginning programmers to grasp. Object-oriented languages such as Java and C++ require that students think abstractly to create software objects that model real-world objects. Personal experience teaching object-oriented programming (specifically, C++ and Java) has taught me that this type of thinking does not come easily to most students. Students struggling to master difficult object-oriented concepts become frustrated, losing confidence in their ability to program.

In an effort to help students learn to program in the object-oriented paradigm, computing educators have developed teaching approaches geared specifically toward novices. These approaches range from high-level approaches that focus on creating and manipulating classes and objects to low-level approaches that focus on writing source code. Because of their novelty, these approaches are often accompanied by programming environments designed specifically to support them (Kelleher and Pausch, 2005). Discussions of these innovative approaches will therefore include descriptions of their supporting environments.

In addition to developing novel teaching approaches, computing educators have experimented with using different programming languages to facilitate the process of learning to program. Languages that have been used to introduce students to programming include Basic, Visual Basic, Pascal, C++, Java, Python, Scheme, and many

others. Unfortunately, some of these languages come with development environments that present students with yet another obstacle to overcome. Students learning to program in Visual Basic, for example, must use Microsoft Visual Studio (Microsoft Visual Studio, 2010). Visual Studio is a professional programming environment equipped with a plethora of features that novice programmers often find overwhelming.

Having tried different teaching approaches and programming languages to alleviate the beginning programming experience, computing educators are faced with yet another dilemma: how to tell if any of these interventions actually *worked*?

Unfortunately, as Goldman, et al. (2008) point out, “there remains a notable lack of rigorous assessment tools in computing.” Consequently, there is presently no objective way to accurately determine the impact of diverse teaching approaches and languages on student learning of introductory programming concepts.

The purpose of this research was thus twofold:

- To develop an instrument to assess student learning of fundamental programming concepts and specific object-oriented programming concepts.
- To use the instrument developed to investigate the impact of different teaching approaches and programming languages on students’ ability to learn the above concepts.

1.2 Novice Teaching Approaches in Practice

It seems reasonable to suppose that a teaching approach that emphasizes the mastery of high-level concepts may very well hinder the mastery of low-level concepts, and vice-versa. For example, researchers at Monash University, Melbourne, Australia, and the Mærsk Institute at the University of Southern Denmark advocate an “objects-

first” approach to teaching object-oriented programming. The BlueJ environment they developed to support their approach therefore emphasizes interacting with static visualizations of classes and objects to facilitate the learning of object-oriented concepts (Barnes and Kölling, 2005; BlueJ, 2005; Kölling, Quig, Patterson, and Rosenberg, 2003). Ragonis and Ben-Ari (2005b) used BlueJ to investigate the learning of object-oriented programming by high school students over the course of an academic year. They found that while BlueJ helped students understand object-oriented programming concepts, it did not help them understand the overall execution (or “flow”) of a program. These findings suggest that the students were unable to trace the program’s source code, an ability required to understand its execution. I noted a similar phenomenon in the *Java for Object-Oriented Programmers* course I took over when a colleague became ill. Specifically, a student who learned to program in Java using BlueJ did not realize that in order to create an application he could run and test, all he had to do was implement a “main” method for a new or pre-existing class. Similarly, another student in the same course (who also learned Java programming using BlueJ) did not seem to understand that a complete (i.e., executable) Java program requires a “main” method. He thought it was sufficient to be able to instantiate an object of a class, then right-click on the graphical representation of the object that BlueJ creates to select and run its various methods.

The authors of *Learning to Program with Alice* (Dann, Cooper, and Pausch, 2006) support either an “objects-first” or an “objects-early” approach to teaching object-oriented programming. Alice (Alice v3.0, 2010) is a three-dimensional (3-D) graphics programming environment designed to enhance students’ initial experience with learning to program by enabling them to build virtual worlds without having to write source code.

Unlike the BlueJ environment, Alice preceded its accompanying teaching approach. Users of Alice report experiences similar to those reported by users of BlueJ. Powers, Ecott, and Hirshfield (2007) first used Alice to teach their CS0 (introductory) programming course in the Fall 2005 semester. They followed the approach of *Learning to Program with Alice* (Dann, Cooper, and Pausch, 2006) and postponed introducing programmer-defined variables until late in the Alice portion of the course. They found that this approach initially contributed to student understanding of loops, conditionals, and events. Once variables were introduced, however, the difficulties students generally encounter with these constructs resurfaced.

Clancy (2004) reports a similar experience with a course in which Karel the Robot (Pattis, 1981) was used to acquaint college students with introductory programming concepts before moving to the Pascal language. Karel The Robot is a robot simulator that introduces students to programming using a language similar to Pascal, except for the absence of variables. One term, a colleague of Clancy's spent more time than was typically spent on Karel before transitioning to Pascal. His students exhibited better than usual comprehension of control flow and procedural decomposition. Once his students moved to Pascal, however, they experienced difficulty with procedures that involved parameter passing because they did not understand the concept of a variable.

When object-oriented programming first appeared in the computer science curriculum, it was taught using what might be described as an "objects-late" or an "imperative-first" approach (Joint Task Force, 2001). Courses that introduced students to object-oriented programming began by teaching them fundamental procedural programming concepts such as variables, assignment, conditionals, loops, and

procedures/functions early in the course, followed by classes and objects later in the course. Personal experience has taught me that this approach did not work particularly well. Students struggled to grasp the concept of abstract objects modeling real-world objects, as well as to master the syntax of the code required to create those objects.

1.3 Contribution to Computer Science Education Research

Computer science educators have been motivated by both the complexity of object-oriented languages and the complexity of professional tools for programming in them to develop approaches and environments specifically to help students learn object-oriented programming. Unfortunately, too little is known about why students find learning to program so hard. In order to help them resolve their difficulties, more research needs to be done in pinpointing the source of those difficulties. It is hoped that by studying the impact of different teaching approaches on student learning of introductory programming concepts and by building a tool to assess that learning the research conducted here will have made a meaningful contribution to the body of computer science education research. According to the Computing Research Association (CRA), enrollment in American computer science programs during the 2007-2008 academic year increased for the first time in six years (Zweben, 2009). It should go without saying that any efforts to continue that trend will be welcomed by the computer science community.

CHAPTER 2: APPROACHES FOR TEACHING OBJECT-ORIENTED PROGRAMMING

2.1 Brief Descriptions

Numerous approaches for teaching introductory object-oriented (OO) programming currently exist. Three examples should suffice to demonstrate a diversity of approaches. Two focus on creating and manipulating classes and objects, while the third focuses on writing source code. The first represents an “objects-first” approach supported by the BlueJ environment. BlueJ is an easy-to-use integrated development environment (IDE) that allows beginning Java programmers to visualize and interact with classes and objects before ever seeing or writing a line of code (Barnes and Kölling, 2005; BlueJ, 2005; Kölling, Quig, Patterson, and Rosenberg, 2003). The second represents either an “objects-first” or an “objects-early” approach supported by the Alice environment. Alice is a three-dimensional (3-D) graphics programming environment that enables students to build virtual worlds by dragging and dropping objects, methods, and control structures into a suitable editor (Alice v3.0, 2010; Dann, Cooper, and Pausch, 2006). The third represents a more traditional “programming-first” or “imperative-first” approach using Python. Python is an object-oriented scripting language that is being used increasingly to teach introductory programming because of its simple and flexible syntax and support for immediate feedback (Grandell, Peltomäki, Back, and Salakoski, 2006; Python, 2007).

2.2 Detailed Descriptions

2.2.1 *Objects-First Approach Using BlueJ*

The “objects-first” approach advocates introducing students to concepts such as classes, objects, methods, and parameters at the very beginning of a course in object-oriented programming. It encourages students to interact with objects and their methods before acquainting them with the code that created them.

BlueJ (2005) is an integrated development environment designed specifically for teaching and learning introductory object-oriented programming in Java. It is basically a front end developed to run on top of Sun Microsystems’ Java 2 Standard Edition (J2SE) Development Kit (Java Platform, Standard Edition (J2SE), 2005) to provide an environment that addresses three key shortcomings identified by its developers of existing programming environments for object-oriented teaching (Kölling, Quig, Patterson, and Rosenberg, 2003).

These shortcomings are:

- The environment does not reflect the object-oriented paradigm.
- The environment has been designed for professionals and is too complex for novices.
- The environment focuses on building graphical user interfaces (GUIs).

BlueJ seeks to overcome these shortcomings by providing beginning programming students with an easy-to-use environment that allows them to visualize and interact directly with classes and objects. Visualization is supported through UML-like diagrams. Interaction is supported through dialogue boxes and popup menus. Templates support code generation. Finally, BlueJ intentionally shields students from some of the more advanced concepts of the Java language (for example, the “main” method with its

required signature “public static void main(String[] args)” until the instructor is ready to introduce them. A snapshot of BlueJ after opening the project *people*, one of the example projects that comes with the standard BlueJ distribution, is shown in Figure 2-1.

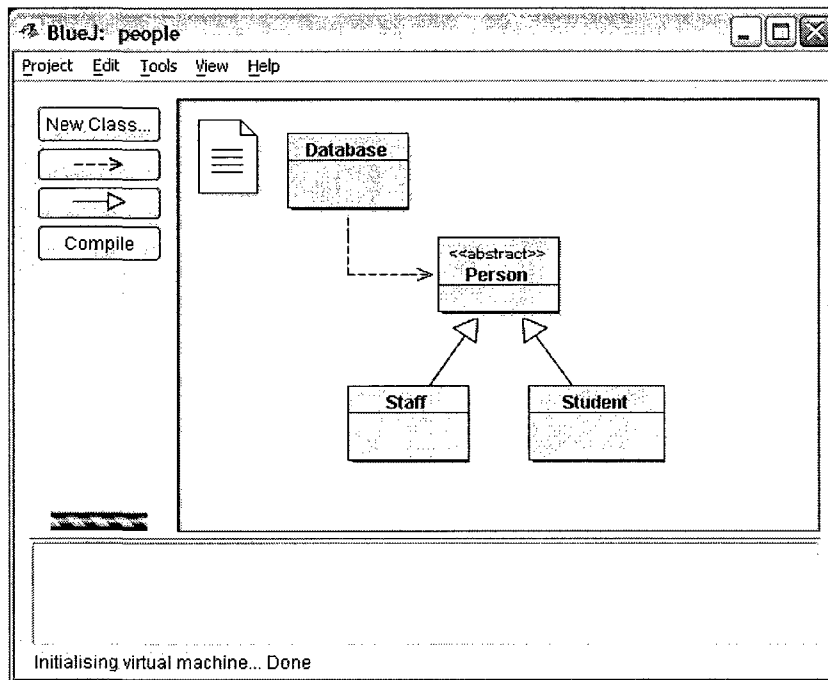


Figure 2-1. An example BlueJ project: *people*.

Figure 2-2 shows a template for an *Administration* class, not yet implemented.

```

/**
 * Write a description of class Administration here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Administration
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class Administration
     */
    public Administration()
    {
        // initialise instance variables
        x = 0;
    }

    /**
     * An example of a method - replace this comment with your own
     *
     * @param y    a sample parameter for a method
     * @return     the sum of x and y
     */
    public int sampleMethod(int y)
    {
        // put your code here
        return x + y;
    }
}

```

changed

Figure 2-2. An example BlueJ class template: *Administration*.

2.2.2 *Objects-First or Objects-Early Approach Using Alice*

The “objects-first” and the “objects-early” approaches differ only in the amount of time that elapses before students are introduced to actual programming language code. The wait is slightly longer for “objects-first” than it is for “objects-early.”

Alice (Alice v3.0, 2010) is a three-dimensional (3-D) graphics programming environment for building virtual worlds. Originally designed for undergraduate students

with no 3-D graphics or programming experience, the focus of the Alice project has expanded to include middle school and high school students. The goal of the Alice project is to enhance students' initial experience with learning to program in two ways:

- Remove needless frustration.
- Provide an environment in which novice programmers of both genders can create compelling programs.

Needless frustration is minimized by allowing students to create programs without ever typing a line of code, thereby preventing syntax errors. Programs, or worlds, as they are called in Alice, are constructed primarily by dragging and dropping objects, their associated methods, and standard control structures into the appropriate editor. An environment for creating simple, yet exciting programs is provided by using “storyboarding” as a metaphor for designing computer programs that take the form of 3-D movies and games. Students can test their programs at any point in the design process by simply pressing the Play button on the Alice tool bar.

Figure 2-3 shows an example Alice world, First Encounter. A snapshot of the world as it plays appears in Figure 2-4 (Dann, Cooper, and Pausch, 2006).

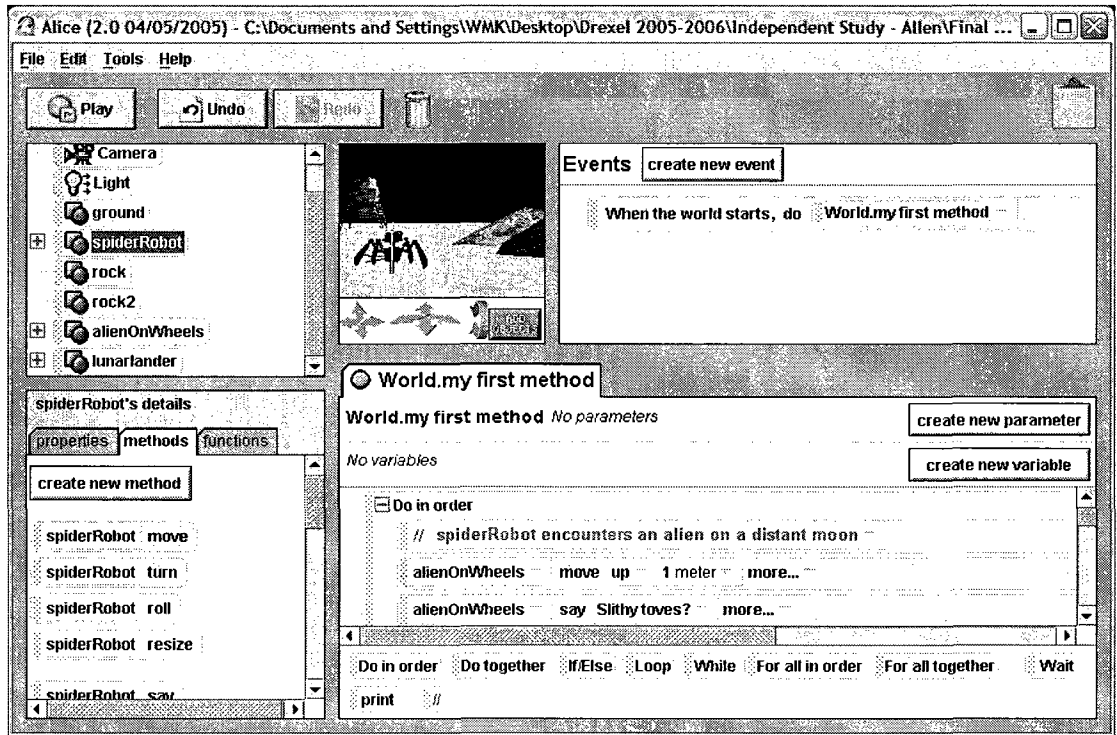


Figure 2-3. An example Alice 2.0 world: *First Encounter*.

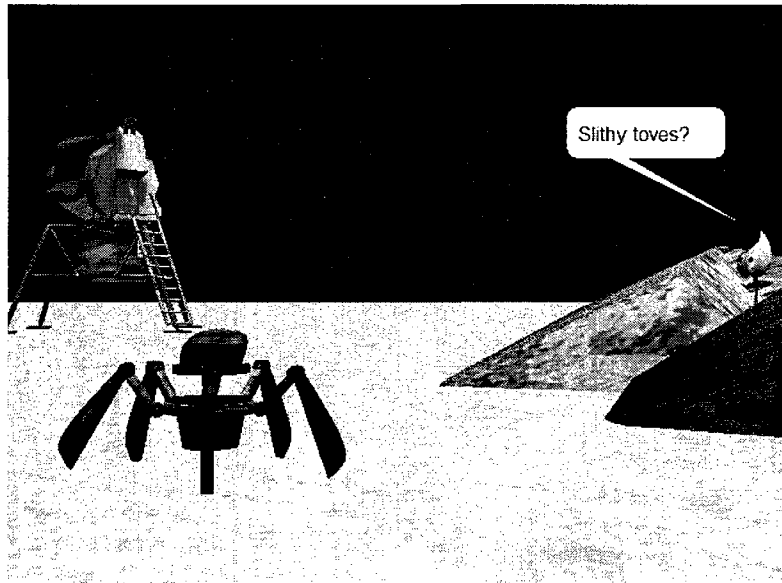


Figure 2-4. An example Alice 2.0 world: *First Encounter*.

2.2.3 Imperative-First Approach Using Python

The “imperative-first” approach begins by teaching students procedural programming techniques to learn about the imperative aspects of an object-oriented language, followed by object-oriented techniques to learn about its object-oriented qualities. Imperative aspects include expressions, control structures, procedures, and functions. Object-oriented qualities include classes, objects, methods, and inheritance.

Python (2007) is an object-oriented scripting language that is being used increasingly to teach introductory programming. (JavaScript is another, but it must be embedded in a host environment such as a Web page to use it (JavaScript, 2007).)

Features that make the language particularly suitable for beginning programmers include:

- Simple syntax.
- Enforced program structure.
- Dynamic typing of variables.
- Powerful built-in types, such as lists.
- Easy-to-use graphics libraries.
- Interactive mode for experimenting with code.

The source code for a Python program that calculates compound interest is shown in Figure 2-5. Figure 2-6 displays the corresponding GUI (Agarwal and Agarwal, 2005).

```

*compIntCalculator.py - C:\Documents and Settings\Wanda\Desktop
File Edit Format Run Options Windows Help
#The Tk interface allows you to set up a GUI.
from Tkinter import *

#This is executed when the left button of the mouse ("Button-1") is pressed.
def getValue(event):
    stringValue1=text1.get()
    Principal=float(stringValue1)

    stringValue2=text2.get()
    Rate=float(stringValue2)

    stringValue3=text3.get()
    NumYears=float(stringValue3)

    Amount=Principal * (1 + Rate) ** NumYears

    label4=Label(smallWindow, text="The compounded value is:$" + "%.2f" % Amount)
    label4.pack()

#Set up a new window called smallWindow with the title
#"Compound Interest" of size 200 by 200 pixels.
smallWindow=Tk()
smallWindow.title("Compound Interest")
smallWindow.geometry("200x200")

#Display a label followed by a box for each data entry.
label1=Label(smallWindow, text="Enter Principal:")
label1.pack()

text1=Entry()
text1.pack()

label2=Label(smallWindow, text="Enter Rate (real):")
label2.pack()

text2=Entry()
text2.pack()

label3=Label(smallWindow, text="Enter Number of Years:")
label3.pack()

text3=Entry()
text3.pack()

button=Button(text="Calculate")
button.pack()
button.bind("<Button-1>", getValue)

mainloop()
Ln: 50 | Col: 0

```

Figure 2-5. Source code for part of a Python program that calculates compound interest.

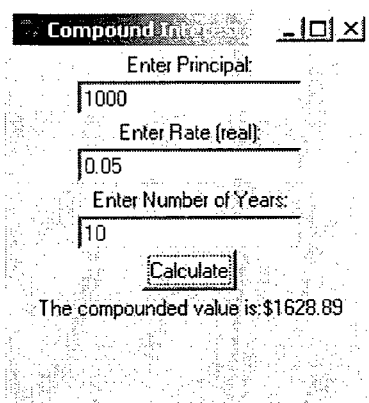


Figure 2-6. GUI for a Python program that calculates compound interest.

2.3 Distinctive Features

The teaching approaches described above each have characteristics that their proponents claim make their approach superior to the others for teaching object-oriented programming. Advocates of the “objects-first” approach using BlueJ believe that focusing on high-level concepts (graphical representations of classes and objects) contributes to a better overall understanding of object-oriented programming than focusing on low-level concepts (lines of source code) (Kölling, Quig, Patterson, and Rosenberg, 2003). Advocates of the “imperative-first” approach using Python claim that its simple syntax, easy-to-use graphics libraries, and support for small- as well as large-scale object-oriented program development make it particularly suitable for teaching introductory programming (Agarwal and Agarwal, 2005, 2006). Finally, Alice’s advocates believe that blending traditional problem-solving techniques with Hollywood-style storyboarding will enable students to readily create compelling object-based programs, thereby improving motivation, reducing attrition, and smoothing the road to

learning professional object-oriented languages such as C++ or Java (Dann, Cooper, and Pausch, 2006).

CHAPTER 3: REVIEW OF NOVICE PROGRAMMER LITERATURE

3.1 Programming Language Knowledge

3.1.1 *Procedural and Functional Languages*

Early studies of programmers focused on novices learning procedural and functional languages. These studies frequently involved identifying what concepts novice programmers are actually learning (or failing to learn) and what misconceptions they have.

Bonar and Soloway (1983) used video-taped interview studies that focused on bugs and buggy programs to gain insight into how novices use a programming system. They found that while novices generally understand what task a program is to carry out, they have difficulty translating that task into program code.

Soloway, Bonar, and Ehrlich (1989) explored the relationship between the strategies Pascal programmers use when solving problems involving loops and the looping constructs they choose to implement their solutions. Results of their study indicated that people prefer a looping strategy that reads and processes the i^{th} element on the i^{th} pass through the loop, as opposed to a strategy that processes the i^{th} element and reads the $(i + 1)^{\text{st}}$ element on the i^{th} pass through the loop. Soloway, Bonar, and Ehrlich also found that people are more likely to write correct programs when programming in languages that facilitate their preferred cognitive strategy.

Kessler and Anderson (1989) investigated novice learning of control structures within the context of a LISP-like programming language called SIMPLE. Specifically, they studied students' ability to write recursive functions and iterative functions and to

transfer between these two programming skills. Experimental results showed positive transfer from writing iterative to recursive functions, but no transfer from writing recursive to iterative functions. In general, the students had difficulty understanding flow-of-control concepts.

3.1.2 Object-Oriented Languages

More recent studies have focused on novices learning to program in the object-oriented paradigm. Two of the four described below were longitudinal studies.

Ragonis and Ben-Ari (2005) investigated high school students learning object-oriented programming using Java and BlueJ over the course of two academic years. Specific goals of their study involved identifying key concepts of object-oriented programming that novices should be taught, the order in which these concepts should be taught, and the conceptions novices build as well as the difficulties they encounter trying to learn these concepts. Study results indicated four main categories of object-oriented concepts: class vs. object, instantiation and constructors, simple vs. composed classes, and program flow. A total of fifty-eight conceptions and difficulties were identified.

Stamouli and Huggard (2006) conducted a phenomenographic study over an entire academic year to investigate first-year undergraduate computer science students' understanding of the principles of object-oriented programming using Java. They found that over half of the students did not attain a mature understanding of what learning to program really means. They also found that most of the students perceived program correctness as syntactic (language-based) and functional (problem-based) correctness. The researchers propose that their findings suggest a relationship between the two themes.

Wiedenbeck and Ramalingam (1999) compared the comprehension of small object-oriented and procedural programs by novices. Their goal was to determine the differences, if any, between novices' mental representations of object-oriented programs and procedural programs, and the focus of each. Results of their study indicated that novices' mental representations of small object-oriented programs were strongest in knowledge related to the function of the program, whereas novices' mental representations of small procedural programs were strongest in knowledge related to the program text.

Evidence that Alice provides support for teaching and learning object-oriented programming is still largely anecdotal. Cooper, Dann, and Pausch (2003a) observed that novice programmers using Alice developed a firm sense of objects and inheritance, as well as a strong sense of design. They also noticed that although Alice's drag-and-drop editor prevents syntax errors, most students were able to reproduce the proper syntax when required to do so in code they wrote for exams (Cooper, Dann, and Pausch, 2003b).

3.2 Programming Misconceptions

Clancy (2004) and Soloway and Spohrer (1989) are both excellent resources for learning about the misconceptions that beginning programmers commonly hold and the kinds of errors they typically make. Clancy surveyed research into programming misconceptions and their causes, followed by suggestions for addressing these misconceptions. Based on the body of research, he divided programming misconceptions into two main categories, those caused by inappropriate transfer, and those caused by confusion about computational models.

According to Clancy, sources of inappropriate transfer include:

- English (natural language).
- Mathematical notation.
- Previous programming experience.
- Overgeneralizing from examples.
- Modification of correct rules.
- Misapplication of analogy.

Sources of confusion about computational models include:

- Input.
- Constructors and destructors.
- Recursion.
- Execution of PROLOG programs.

Many of the examples of research addressing programming-related misconceptions that Clancy (2004) cited may be found in *Studying the Novice Programmer* (Soloway and Spohrer, 1989). Bonar and Soloway (1989) discovered that novice programmers learning Pascal confused the meaning of “while” in natural language with its meaning in the Pascal programming language. One subject inferred that since “while” is typically used as a continually active test in natural language, it can be used similarly in the Pascal while loop. Putnam, et al. (1989), found that high school students learning BASIC could not reconcile what they had learned in algebra with what they were learning in BASIC. One student in particular indicated that the statement $LET C = C + 1$ did not make sense and must be an error. The researchers also found that these students generally experienced difficulty with the concept of reading data (input).

DuBoulay (1989) cautions programming instructors against the use of analogies. For example, comparing a variable to a “box” may lead students to believe that a variable, like a box, can hold multiple values.

Fleury (2000), also cited by Clancy (2004), identified misconceptions related specifically to object-oriented programming. She found that students taking an introductory Java course thought that they could assign values to an object’s instance variables using a mutator function without first allocating memory for the object using a constructor. She also found that they had apparently generated a set of overgeneralized rules related to Java classes and objects based on a limited set of examples. Specifically:

- Different classes must have different method names.
- Arguments to methods must be numeric types.
- The dot operator must be applied to methods; it cannot be applied to instance or class variables.

3.3. Scaffolding for Learning Object-Oriented Programming

3.3.1 Definition of Scaffolding

Scaffolding is a form of support that enables a student to solve a problem, carry out a task, or achieve a goal that would be beyond his capabilities without assistance (Wood, Bruner, and Ross, 1976). An important characteristic of scaffolding is that it facilitates the student learning how to perform these activities independently when the support is removed. Scaffolding was originally used to describe support provided by a parent or a tutor. Scaffolding is now used increasingly to describe support provided by software tools, curricula, and other learning resources (Puntambekar and Hübscher, 2005).

3.3.2 *Key Features of Scaffolding*

The key features discussed here are based on the historical and theoretical roots of scaffolding and are considered critical to its success (Puntambekar and Hübscher, 2005).

These are:

- **Intersubjectivity** – Intersubjectivity, or shared understanding of a task, is attained when the tutor and the student arrive at a shared understanding of the goal of the task the student needs to perform.
- **Ongoing diagnosis** – The tutor must engage in an ongoing diagnosis of the student's current level of understanding to provide him with appropriate support. This requires that the tutor reconcile his theory of the task and how it may be completed with his theory of the performance characteristics of the student (Wood, Bruner, and Ross, 1976).
- **Tailored assistance** – The tutor adjusts the amount and type of support he provides based on his ongoing assessment of the student's understanding. This requires continual interactions between the tutor and the student.
- **Fading** – This involves gradually removing the support as the student takes responsibility for his own learning until he can function on his own. At this point the student is able to generalize the process whereby he completed the task and apply it to similar tasks.

3.3.3 *Scaffolding to Support Learning to Program*

Over the years, educators have experimented with various forms of scaffolding to aid students in learning to program. The teaching approaches and supporting

environments discussed in Section 2 are examples of computer-based scaffolds developed to help students learn to program in object-oriented languages.

3.4 Theories of Learning to Program

It is common knowledge that students have difficulty learning to program. Theories have been put forth as to why this is so and suggestions made as to how to remedy the situation.

3.4.1 Constructivism

Constructivism is a theory of learning which claims that students actively construct knowledge rather than passively absorb it from textbooks and lectures. New knowledge is created by combining experiential knowledge with existing knowledge, or by reflecting on existing knowledge (Ben-Ari, 1998).

Constructivist ideas date back to the time of Socrates (Constructivism as a Paradigm for Teaching and Learning, 2004). Although many have contributed to constructivist theories, Ernst von Glasersfeld (1990) credits Jean Piaget with being “the great pioneer of the constructivist theory of knowing today.” Piaget was a trained biologist who developed theories of intellectual development in the child by studying his own children (Hergenhahn and Olson, 2001). Greatly influenced by Piaget’s ideas was Seymour Papert, probably best known as the inventor of LOGO, a programming language he created to teach mathematics to elementary school children (Papert, 1993). Papert’s pioneering work has led to the widespread use of computer technology within constructivist environments (Constructivism as a Paradigm for Teaching and Learning, 2004).

Constructivism has been applied successfully to learning topics in mathematics and science (e.g., Cobb, et al., 1991; Cobb and Steffe, 1983). Pea, Soloway, and Spohrer observed that a student learning to program also engages in constructivist activities. Specifically, he actively builds a system of conceptual and procedural knowledge related to programming. In “The Buggy Path to the Development of Programming Expertise” they explored how this knowledge-building activity may lead to “errors” or “misconceptions” in programming and how to increase instructors’ awareness with regard to this phenomenon (Pea, Soloway, and Spohrer, 1987).

Ben-Ari (1998) explores the extent to which constructivism is applicable to computer science education in general. Ben-Ari contends that many of the difficulties experienced by a beginning computer science student stem from the fact that he possesses no effective model of a computer. In other words, he has no existing knowledge in which to integrate the information about computers and programming he hears in class or reads in his textbook. Consequently, he must construct his own model of a computer. As an alternative, Ben-Ari suggests explicitly teaching the model.

Hadjerrouit (1999) describes a pedagogical framework for teaching introductory object-oriented design and programming based on constructivism. He believes that constructivism is a particularly appropriate approach for teaching programming in the object-oriented paradigm because it takes into account a student’s previous programming knowledge and stresses the need for him to actively participate in the construction of object-oriented knowledge. Hadjerrouit’s motivation for developing this teaching approach was the realization that students experience great difficulty learning abstract

object-oriented concepts, and that their learning may be impeded by the presence of procedural programming concepts.

3.4.2 Assimilation Encoding Theory

Mayer (1989) discusses how to promote meaningful learning of computer concepts by novices. Meaningful learning takes place in much the same way that the constructivists claim new knowledge is created. Specifically, meaningful learning is a process whereby a learner connects new information with knowledge already existing in memory. The connecting process is called “assimilation.” The existing knowledge to which new information is connected is called a “schema.” Learning about computers and computer programming requires assimilation to schema of new technical information.

Mayer’s Assimilation Encoding Theory provides a three-stage model (shown in Figure 7) of how technical information must be processed to support meaningful learning (Mayer, 1979). First, the information is “received” in working memory from short-term memory (arrow “a”). Second, the information is transferred from working memory to long-term memory if there is knowledge “available” in long-term memory to which to relate it (arrow “b”). Third, the information is “actively” integrated with relevant existing knowledge that has been transferred from long-term memory to working memory during learning (arrow “c”).

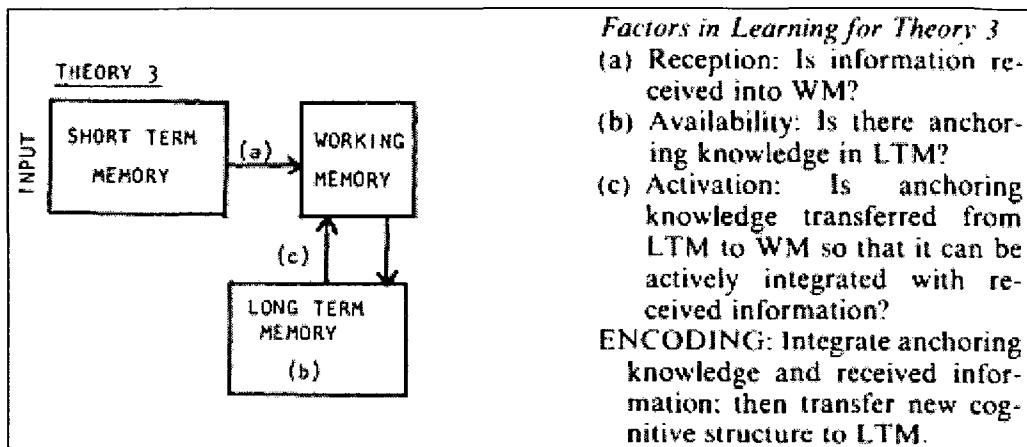


Figure 3-1. Model of Assimilation Encoding Theory (Mayer, 1979).

Mayer (1989) explored techniques for increasing novices' understanding of computer programming that involved activating relevant existing knowledge in long-term memory. Called "relevant anchoring ideas" by Ausubel (1968), two techniques that Mayer explored are: 1) provide the learner with a concrete model of the computer and 2) encourage the learner to explain technical information in his own words. Results of a series of studies indicated that both techniques may lead to enhanced conceptual learning of technical information, as measured by learners' ability to solve novel problems.

3.4.3 Advance Organizers

The term "relevant anchoring ideas" derives from Ausubel's work on advance organizers (Ausubel, 1960, 1968). Ausubel (1968) described advance organizers as "appropriately relevant and inclusive introductory materials" presented before the learning material itself and at a higher level of abstraction, generality, and inclusiveness. The purpose of the organizer was to provide "ideational scaffolding" for incorporating and retaining the detailed information in the learning material that followed. While

Ausubel restricted advance organizers to abstract stimuli, Mayer and other researchers extended them to include concrete physical analogies (Mayer and Bromage, 1980).

3.5 Model of Object-Oriented Programming

A beginning computer science student possesses no effective model of a computer. Ben-Ari and others propose providing the student with one to use to integrate new incoming information about computers and programming. An alternative approach would be to provide him with support (i.e., some form of scaffolding) for constructing his own model. One of the purposes of this research is to identify ways of doing precisely that.

3.6 Transfer in Computing Environments

3.6.1 Definition of Transfer

Merriam-Webster's Collegiate Dictionary (2003) defines transfer as “the carryover or generalization of learned responses from one type of situation to another” (definition 2b). Although the definition seems perfectly straightforward, researchers interpret transfer of learning in many different ways. According to Barnett and Ceci (2002), “there is little agreement in the scholarly community about the nature of transfer, the extent to which it occurs, and the nature of its underlying mechanisms.”

3.6.2 Theories of Transfer

In spite of the controversy, the idea of transfer is still traditionally based on theories of common elements. Thorndike (1906) proposed a theory of identical elements which stated that training in one type of mental function or activity would transfer to another only if the two activities shared common stimulus-response elements. The common elements in Thorndike's view involved associations between concrete entities.

Singley and Anderson (1989) resurrected Thorndike's theory of identical elements and recast it in abstract form. Using the ACT* theory of skill acquisition (Anderson, 1983), they redefined Thorndike's identical elements as mental elements instead of physical elements. Essentially, they replaced Thorndike's concrete stimulus-response pairs with abstract condition-action pairs called productions. According to Singley and Anderson's modified version of the theory, transfer occurs to the extent that sets of productions for different tasks overlap.

An alternative model of transfer that may prove beneficial to this research bears mentioning. It is the "actor-oriented transfer" model proposed by Lobato (2003), which emerged from her work with design experiments that explored transfer of algebraic concepts. The classical approach to transfer focuses on the degree to which a learner applies knowledge learned in one situation to a new situation based on the researcher's (or another expert's) model of performance. The actor-oriented approach to transfer de-emphasizes degree, and focuses rather on the influence of any previous activities on a learner's behavior in a novel situation, including the ways in which learning generalizes (Lobato, 2006).

3.6.3 New Learning as Transfer from Previous Learning

The constructivists believe that students actively construct new knowledge by combining experiential knowledge with existing knowledge. Bransford, Brown, and Cocking (1999) paraphrase this as "all learning involves transfer from previous experiences." They point out that previous learning can help as well as hinder students' ability to learn new information. For example, children's experiences with counting and putting things together when they play help them to learn addition and subtraction of

whole numbers when they begin school. Unfortunately, these same experiences undermine children's abilities to learn operations with fractions, which are not based on the same underlying principles as whole numbers (e.g., you can't generate a fraction by "counting things"). With respect to computer programming, Hadjerrouit (1999) found that students' ability to learn object-oriented programming was somewhat impeded by their previous experience with procedural programming.

3.6.4 Transfer Between Programming Languages and Text Editors

Studies have been conducted to investigate transfer of skills between programming languages and transfer of skills between text editors. Scholtz and Wiedenbeck (1990a) studied moderately experienced programmers transferring to a new programming language to determine the kinds of knowledge that transfer and the areas in which programmers experience difficulties. They found that programmers encounter the most difficulty trying to figure out how to implement a problem solution in the new language, whereas issues of syntax and semantics present them with minimal difficulty.

Singley and Anderson (1985, 1988, 1989) studied computer-naïve subjects from a local secretarial school learning to edit text using three different text editors. Two of these were classified as line editors, while the third was classified as a screen editor. Singley and Anderson's goal was to study transfer of text-editing skill between the line editors and from the line editors to the screen editor. In general, they found positive transfer of skill both between the line editors and from the line editors to the screen editor.

CHAPTER4: EDUCATIONAL ASSESSMENT

4.1 Assessment in Computing

In spite of its unpopularity with students, testing is still the preferred method for ascertaining the effectiveness of educational endeavors. An instructor who wishes to assess his students' learning constructs a test which he then administers to those students. The results of that test provide the instructor with information about what *his* students learned in *his* classroom based on *his* instruction. This scenario reflects the current state of affairs in the field of computer science in which we have no standardized tools to measure student learning of fundamental programming concepts.

Developing a standardized tool for a field as dynamic as computer science presents special challenges. Computing educators use a variety of approaches and languages to introduce students to computer programming. A “language independent” tool that could be administered at both the beginning and at the end of a course would enable educators to objectively measure student learning and assess the effectiveness (or ineffectiveness) of their particular methodologies. For example, do students learn object-oriented concepts better when taught with an “objects-first” or a “programming-first” approach? Does programming language make a difference when teaching students about modular programming (the practice of breaking a program into manageable chunks that can be easily re-used)?

Several years ago, researchers at another educational institution began work on a language independent tool to assess learning of fundamental computer science concepts. An experimental version of this tool was used in the pilot study. Although I was aware

that the developers of the tool were now in the process of validating it, they were not willing to share it. Since their experimental tool was the best computer concepts assessment tool I could find, I adapted it for use at the two universities where the study was conducted.

The chapter proceeds by reviewing assessment in general, followed by a discussion of tools to assess student learning in mathematics, science, and engineering, fields closely related to computer science. It continues by examining recent efforts to develop assessment tools for computer science. The remaining sections focus on the process used to develop the computer concepts survey used in this research.

4.2 Principles of Assessment

Educators use a variety of tools and techniques to evaluate student performance in a particular subject area. These tools and techniques reflect two the key principles of assessment (Bransford, Brown, and Cocking, 1999):

- What is assessed must be in agreement with one's learning goals.
- It should provide opportunities for feedback and revision.

4.3 Approaches to Assessment

The two principles cited above correspond to two disparate approaches to assessment, one traditional, the other contemporary. The first principle corresponds to the traditional approach, called "summative" assessment. The second principle corresponds to the contemporary approach, called "formative" assessment (Bransford, Brown, and Cocking, 1999; Even, 2005).

Summative assessment typically involves giving students a paper-and-pencil test at the end of a unit of study to measure what they have learned. The grades assigned on

the test represent the students' level of achievement and may also be used to classify or rank the students.

Formative assessment generally takes place in the classroom as a part of instruction. Examples include teachers' comments on students' work in progress, such as math worksheets, computer programs, or drafts of papers. Formative assessment emphasizes providing feedback to increase students' learning and transfer and obtaining feedback to inform teachers' instructional decisions.

Ideally, teachers should use a combination of summative and formative assessment in their teaching. In reality, summative assessment is still the norm in many classrooms (Bransford, Brown, and Cocking, 1999). This is because many teachers, parents, and students still equate effective learning with memorizing facts and procedures, a belief bolstered by the many standardized tests that overemphasize these same skills. Determining the depth of students' conceptual understanding requires formative assessment techniques, however.

4.4 Existing Tools

4.4.1 Mathematics, Science, and Engineering

Many tools exist or are under development to assess student learning in subjects such as mathematics, science, and engineering. The *Algebra End-of-Course Assessment* (2009) is a multiple-choice test that K-12 math teachers can use to measure student understanding of first-year algebra concepts. The *Force Concept Inventory (FCI)* is a multiple-choice test that college physics instructors can use to assess student understanding of fundamental concepts in Newtonian physics (Hestenes, Wells, and Swackhamer, 1992). Thermal-science concept inventories have been jointly developed

by personnel from the University of Wisconsin-Madison and the University of Illinois Urbana-Champaign. Designed specifically for mechanical engineering undergraduate students, the three concept inventories evaluate student understanding of concepts in thermodynamics, fluid mechanics, and heat transfer (Mitchell and Martin, 2007). Assessment tools are also under development in biology, calculus, and chemistry, to name but a few (Allen, 2007).

4.4.2 Computing

Assessment tools for computing are sorely lacking. There are two reasons for this unfortunate state of affairs. The first is that the field of computing, by its very nature, is highly dynamic. The second is that while the ACM and IEEE provide curricular guidelines for undergraduate programs in computing (CS2008 Review Taskforce, 2008; Joint Task Force on Computing Curricula, 2001), there are no actual standards per se as there are in fields like math and science. A step was taken in that direction, however, when, in April 2000, the International Technology Education Association (ITEA) and its Technology for All Americans Project (TfAAP) published *Standards for Technological Literacy: Content for the Study of Technology*. This document defines technological literacy, and articulates the knowledge and abilities students in grades K-12 need to become technologically literate. It does not provide a curriculum, however (ITEA, 2007). The International Society for Technology in Education (ISTE) also publishes the one-page *National Educational Technology Standards (NETS•S) and Performance Indicators for Students* (NETS•S, 2007), which outlines six areas in which students should possess technological competency. These include understanding technology concepts, systems, and operations, among others.

The Educational Testing Service (ETS) maintains the Carl C. Brigham Library Test Link database of more than 25,000 tests created by authors inside and outside of the ETS. Querying the database for the search term *comput** in the title field yielded over 100 matches which included:

- Advanced Placement Program: Computer Science A Exam
- Certificate in Computer Programming Examinations
- Computer Literacy Examination : Cognitive Aspect
- Computer Science Placement Exam
- Computer Science Test for Grades Nine Through Twelve
- Major Field Tests: Computer Science
- Student Occupational Competency Achievement Testing: Computer Programming

Unfortunately, based on their descriptions, none of these tests is appropriate for assessing fundamental programming knowledge. In other words, there is no computer science equivalent of the *Force Concept Inventory*.

4.5 A Tool for Introductory Computing

4.5.1 Rationale

The computer concepts survey used in the pilot study (described in Chapter 5) was a slightly modified version of an experimental tool developed at another educational institution (The name is being withheld in the interest of confidentiality.). In order to validate the results of the pilot study and to further investigate the impact of different teaching approaches on students' ability to learn both fundamental programming concepts and specific object-oriented programming concepts, it would be better to use a tool that has been thoroughly tested. Unfortunately, no such tool is currently available.

Although I was aware that the developers of the experimental tool were now in the process of validating the tool, they were not willing to share it. Consequently, I modified the experimental tool yet again to better suit the student population with which it would be used. Adjustments were made in accordance with the guidelines laid out in the next section.

4.5.2 *Characteristics of the Tool*

A list of guidelines for creating a computer concepts assessment tool was developed by drawing on personal experience teaching computer science and programming, the literature on computer concepts inventories under development (Tew and Guzdial, 2010; Goldman, et al., 2008), and the literature on novice programmers' errors and misconceptions (Clancy, 2004; Soloway and Spohrer, 1989). A review of this literature combined with personal experience suggests that an instrument designed to assess student learning of introductory programming concepts should possess certain characteristics. Specifically, it should consist of questions and response choices:

- 1) Based on topics that students generally encounter in a first course in computer programming (i.e., *topics taught in a first programming course*).
- 2) Based on topics that experts agree students should master by the end of a first course in computer programming (i.e., *topics agreed upon by experts*).
- 3) That contain distracters (incorrect answer choices intended to mislead the test taker) based on the misconceptions that students learning to program commonly hold and the kinds of errors they typically make (i.e., *distracters based on novice errors and misconceptions*).

- 4) Written in a language that is “programming language independent” (i.e., *language independent pseudocode*).

- 1) *Topics taught in a first programming course*

A good sense of topics generally taught in a first computer programming course can be obtained by examining sample syllabi from institutions offering such courses. I looked at syllabi from two different introductory programming courses taught at Rowan University in Glassboro, NJ (Hartley, 2009; Provine, 2009), and two different introductory programming courses taught at Drexel University in Philadelphia, PA (Medlock, 2008; Popyack, 2009). While the topics covered in the four courses were not exactly the same, they did commonly include: variables, conditionals, loops, arrays, methods, and objects.

Tew and Guzdial (2010) recently reported on their progress in creating a language independent CS1 assessment instrument. Section 4 of the paper discusses their strategy for defining the instrument’s content. They began by analyzing the tables of contents of twelve of the most widely adopted CS1 textbooks to identify a set of common concepts, then used the Computing Curricula 2001 Computer Science guidelines for conceptual content of introductory programming courses to refine the list.

- 2) *Topics agreed upon by experts*

Efforts are being made at both the undergraduate and pre-college (K-12) levels to arrive at a consensus regarding topics in computing that should constitute a standardized curriculum in computer science education. In 2006, the Computer Science Teachers Association (CSTA) published the second edition of the ACM Model Curriculum for K-

12 Computer Science. According to the executive summary, the purpose of this report is to propose “a model curriculum that can be used to integrate computer science fluency and competency throughout primary and secondary schools, both in the United States and throughout the world.” In addition, “It provides a framework within which state departments of education and school districts can revise their curricula to better address the need to educate young people in this important subject area, and thus better prepare them for effective citizenship in the 21st century.” (CSTA, 2005)

As discussed in the section on assessment tools for computing, the ACM and IEEE provide curricular guidelines for undergraduate programs in computing (CS2008 Review Taskforce, 2008; Joint Task Force on Computing Curricula, 2001). The *Programming Fundamentals* knowledge area defines the skills and concepts that students must master to become proficient programmers regardless of paradigm. Of the eight core topics, fundamental constructs, algorithmic problem solving, data structures, recursion, object-orientation, and secure programming are the six regularly included in introductory programming courses. (The remaining two are event-driven programming and foundations of information security.)

In 2008, Goldman, et al. (2008) employed a Delphi process to identify important and difficult concepts in college-level introductory computing courses. The experts who participated in the study agreed upon eleven topics for the particular area of programming fundamentals. Arranged in order from most important to least important, these are:

- Procedure design.
- Conceptualize problems, design solutions.
- Issues of scope, local vs. global.
- Functional decomposition, modularization.
- Designing tests.
- Parameter scope, use in design.
- Debugging, exception handling.
- Abstraction/pattern recognition and use.
- Recursion, tracing and designing.
- Inheritance.
- Memory model, references, pointers.

In light of the current emphasis on object-oriented programming, it is worth noting that only one of the eleven topics, inheritance, is exclusively object-oriented.

The Goldman, et al., study was conducted as part of a multi-institution project led by Craig Zilles to develop concept inventories for three introductory computer science subjects: discrete math, digital logic design, and programming fundamentals (Concept Inventories, 2010). Using a Delphi process to identify important and difficult concepts is step one of the four-step process the project investigators plan to use to develop the inventories. Steps two through four are: identify common misconceptions for those

concepts, design questions using those misconceptions, and validate the concept inventory (Goldman, et al., 2008). According to the information posted on the publicly accessible portion of their Web site (Concept Inventories, 2010), the only concept inventory on which they have done extensive work is the concept inventory for digital logic design. Herman, Loui, and Zilles (2010) describe the process used to create and evaluate an alpha version of the digital logic concept inventory that was administered to 203 students from the University of Illinois at Urbana-Champaign in Spring 2009.

3) *Distracters based on novice errors and misconceptions*

Clancy (2004) and Soloway and Spohrer (1989) are good resources for learning about beginning programmers' errors and misconceptions. This literature was reviewed in Chapter 3.

The focus here is on the use of novice errors and misconceptions to inform the development of incorrect answer choices. However, errors commonly made and misconceptions commonly held can also inform the development of the questions themselves. A good example is a question that asks a student to trace the code for a "while" loop in which the loop control variable never gets incremented (a *very* common error!).

4) *Language independent pseudocode*

A good model for an appropriate pseudocode is that employed in *Mathematical Structures for Computer Science* (Gersting, 2007). Gersting defines pseudocode as a compromise form for describing algorithms that is readily understood by nonprogrammers. It is, in her words, "a form that is a middle ground between a purely verbal description in paragraph form and a computer program written in a programming

language.” The pseudocode that Gersting employs is a cross between English and a Pascal-like procedural language. It is particularly suitable for students with minimal programming experience. Two samples from her book appear in Figures 4-1 and 4-2.


```

GCD(positive integer  $a$ ; positive integer  $b$ )
// $a \geq b$ 

Local variables:
integers  $i, j$ 

   $i = a$ 
   $j = b$ 
  while  $j \neq 0$  do
    compute  $i = qj + r, 0 \leq r < j$ 
     $i = j$ 
     $j = r$ 
  end while

// $i$  now has the value  $\text{gcd}(a, b)$ 
return  $i$ ;

end function GCD

```

Figure 4-1. Pseudocode for Euclidean Algorithm (Gersting, 2007).

```

BinarySearch(list  $L$ ; integer  $i$ ; integer  $j$ ; itemtype  $x$ )
//searches sorted list  $L$  from  $L[i]$  to  $L[j]$  for item  $x$ 
  if  $i > j$  then
    write("not found")
  else
    find the index  $k$  of the middle item in the list  $L[i]-L[j]$ 
    if  $x =$  middle item then
      write("found")
    else
      if  $x <$  middle item then
        BinarySearch( $L, i, k - 1, x$ )
      else
        BinarySearch( $L, k + 1, j, x$ )
      end if
    end if
  end if

end function BinarySearch

```

Figure 4-2. Pseudocode for BinarySearch Algorithm (Gersting, 2007).

4.5.3 Tool Development

The assessment tool used in this research was created by modifying the experimental tool used in the pilot study. The topic areas are based on the core topics defined in the ACM/IEEE guidelines' *Programming Fundamentals* knowledge area (CS2008 Review Taskforce, 2008), i.e., the *Topics agreed upon by experts*, as well as the *Topics taught in a first programming course*, discussed in Section 4.5.2. In addition, it seemed important to include a question asking students to rank the difficulty level of the computer concepts survey using a Likert-type scale. The current tool employs a multiple-choice format in which questions consist of a stem (actual question) and four possible responses. Of the four responses, one is correct, while the others are distracters (incorrect responses).

Analysis of the pilot study results revealed features of the experimental tool that needed to be adjusted for further use with Rowan and Drexel students. A few of the questions were too easy, while others were too hard. In addition, many distracters were never chosen. Finally, the pseudocode was unlike the pseudocode that Rowan and Drexel students are accustomed to seeing in their computer programming classes.

Adapting the experimental tool entailed modifying the question stems, responses, and pseudocode. Ideas for questions to replace those that were too easy or too hard were drawn from programming textbooks, personal experience teaching programming, and, in one case, a sample major field test for computer science. Ideas for distracters to replace those that were never chosen were drawn from the literature on novice programmers' errors and misconceptions (discussed in *Distracters based on novice errors and misconceptions*, Section 4.5.2), personal teaching experience, and personal experience

doing SAT test preparation. The pseudocode was modeled after the pseudocode used by Gersting (discussed in *Language independent pseudocode*, Section 4.5.2) and the pseudocode used by a member of the Drexel Computer Science faculty in a diagnostic test he developed.

4.5.4 *Reliability and Validity of the Tool*

Measurement experts currently associate validity with a set of scores rather than with the assessment tool used to produce them (American Educational Research Association, American Psychological Association & National Council on Measurement in Education, 1999). Simply put, validity has to do with the meaning of the scores and how we use them. It also has to do with the item responses that are aggregated to form the scores (Haladyna, 2004). According to Ebel and Frisbie (1991), validity has two aspects: what is measured and how consistently it is measured. Area of measurement refers to a particular cognitive ability of interest. Consistency of measurement refers to reliability. Reliability is a necessary but not sufficient condition to ensure validity.

A variety of methods exist for estimating score reliability (Ebel and Frisbie, 1991). These include test-retest, equivalent forms, split halves, Kuder-Richardson, and Cronbach's alpha. The methods used in this research estimated reliability using information internal to the test. They are:

- Kuder-Richardson – Kuder and Richardson (1937) developed two of the most widely accepted formulas for estimating reliability, K-R20 and K-R21. K-R20 uses number of test items, proportion of correct/incorrect responses for each item, and variance of test scores to estimate reliability. K-R21 uses test mean in place of proportion of correct/incorrect responses for each item to estimate the value of K-R20.
- Cronbach's alpha – This method, unlike K-R20, can be used to estimate reliability for tests that employ weighted scoring. The formula is similar to K-R20 and uses number of test items, variance of each item, and variance of test scores to estimate reliability.

Researchers developing assessment tools comparable to the computer concepts assessment tool have used either K-R21 or Cronbach's alpha to measure reliability (Herman, Loui, and Zilles, 2010; Nugent, Soh, Samal, and Lang, 2006), hence the choice of internal consistency measures. K-R21 is easy to use, but may underestimate the reliability coefficient depending on the average scores of the individual test items (Ebel and Frisbie, 1991). Cronbach's alpha was therefore used to obtain a second measure of reliability. The K-R21 and Cronbach's alpha formulas are similar, so their results will be similar, but the Cronbach's alpha reliability estimates will usually be a little higher.

The decision was also made to obtain separate reliability ratings for the pre-test and the post-test instead of obtaining a single, overall reliability rating. Analysis of the

demographic surveys indicated that some of the students had not been exposed to all of the programming concepts before taking the pre-test. By the time they took the post-test, they would have been introduced to all of the concepts. Consequently, it was decided to measure reliability separately for the pre- and post-test.

Validating test scores requires gathering evidence to support how the scores are interpreted and used. It is more difficult to demonstrate than reliability because there is no simple prescription for determining the forms of evidence that will be sufficient. The three general categories of validity evidence are content-related, criterion-related, and construct-related (Ebel and Frisbie, 1991). The two most appropriate for this research are:

- Content-related – This type of evidence deals with how well the content of the test represents the domain of knowledge, skills, and tasks the test is intended to measure.
- Construct-related – This type of evidence deals with what the test scores mean as a psychological construct. For example, if a test is intended to measure the psychological construct “understanding of fundamental programming concepts,” is there sufficient evidence to show that it does?

Approaches used in this research to demonstrate content validity included:

- Insuring that the questions making up the computer concepts inventory addressed core concepts of the *Programming Fundamentals* knowledge area defined by the 2008 ACM/IEEE curricular guidelines.
- Psychometric analysis techniques (section 8.5.3) to show that a majority of the inventory questions effectively discriminated between students who performed well on the inventory from those who did not.

- Psychometric analysis techniques (section 8.5.2) to show that all distracters were chosen and that many of the distracters exhibited desirable response patterns (i.e., the percentage of students choosing distracters was inversely related to the students' scores).
- Expert reviews of the questions and corresponding answer choices for appropriateness.

Approaches used in this research to investigate construct validity included:

- Factor analysis to determine which questions should be removed from the computer concepts inventory because they did not meet the acceptable load threshold of 0.4.
- Factor analysis to determine how well the inventory questions grouped into the constructs they were intended to measure.

CHAPTER 5: PILOT STUDY

5.1 Purpose

A pilot study was conducted to investigate the effectiveness of disparate teaching approaches currently being used to help students learn to program in object-oriented languages.

5.2 Participants

Fourteen students enrolled in two different C++ courses (CS 131, CS 171) during the Fall 2008-2009 and Winter 2008-2009 terms at Drexel participated in the study. Students in these particular courses were targeted for recruitment because of their previous programming experience. Students enrolled in CS 131 generally take an earlier course (CS 130) that introduces them to programming using *Alice*. *Alice* is a three-dimensional (3-D) graphics programming environment that minimizes the frustration of learning to program by enabling students to create compelling programs without ever writing a line of code. Students enrolled in CS 171 generally take courses that introduce them to programming using languages such as *JavaScript* or *Python*. Students learn to program in these languages by writing source code in text-based environments.

5.3 Task

All fourteen participants completed the study, which consisted of two sessions. They were paid for their participation. The first session was scheduled near the beginning of the term, while the second session was scheduled near the end of the term (or at the start of the following term). During the first session, each student was asked to complete three surveys: a demographic survey to obtain information about his

background; an attitude survey to gain insight into his attitude toward computer science and programming; and a computer concepts survey to assess his knowledge of computer programming concepts. Students who participated in the study during the Winter 2008-2009 term were also asked to solve a simple programming problem at the computer. During the second session, each student was asked once again to complete the attitude and computer concepts surveys. Those who participated during the Winter 2008-2009 term were asked to solve a slightly harder programming problem. Most of the students completed each session in an hour or slightly more. The sessions were not timed.

5.4 Results

The data obtained from the demographic and computer concepts surveys is of most interest to me. However, after recording the participants' responses to the questions on the computer concepts survey, it seemed necessary to find a way to compute pre- and post-test scores for each participant that accounted for correct responses, incorrect responses, and non-responses (i.e., omissions). The algorithm used to compute a raw score on the multiple-choice portions of the SAT seemed appropriate. Specifically, "1" is assigned for each *correct* response, " $-1/(\text{number of incorrect responses})$ " is assigned for each *incorrect* response, and "nothing" (i.e., "0") is assigned for *non-responses*. Applying that algorithm yielded the summarized results for sessions one and two that appear in Appendices B and C. The results of the demographic survey appear in Appendix A.

Ten of the fourteen participants majored in areas other than Computer Science and Software Engineering. These students possessed the programming backgrounds

closest to what I had been looking for in study participants. Consequently, the subsequent summary and analysis will focus on the performance of these ten students.

The ten students of interest included five Physics majors, one Information Technology major, two Mathematics majors, and two Digital Media majors. The Physics and Information Technology majors had programmed mainly in *Python*. The Mathematics and Digital Media majors had programmed mainly in *Alice*. The Excel spreadsheets that follow (Figures 5-1 and 5-2) display the mean pre- and post-test scores for these two groups of students, as well as the mean pre- and post-test scores for all of the students who participated in the study. The spreadsheets also display the percent change in these scores. Scores were calculated for categories of questions and for the computer concepts survey as a whole. The two spreadsheets show the different ways in which questions were grouped into categories. The shaded regions highlight findings that I found to be of particular interest.

Session 1		Scores										
Means	Basics	Logical Operators	If	Loops (w/o arrays)	Loops (w/ arrays)	Functions	Arrays (only)	Objects	Overall	Definition	Trace	Write
Overall Mean	1.715	0.715	1.667	1.310	0.857	-0.356	0.548	1.715	8.169	4.025	2.215	1.929
Alice Mean	1.334	1.667	2.667	1.417	0.334	-0.666	0.334	1.834	8.919	4.085	2.668	2.167
Python Mean	1.534	0.467	0.867	1.400	1.200	-0.399	0.533	1.257	6.870	3.202	2.467	1.201
Session 2		Scores										
Means	Basics	Logical Operators	If	Loops (w/o arrays)	Loops (w/ arrays)	Functions	Arrays (only)	Objects	Overall	Definition	Trace	Write
Overall Mean	1.762	0.762	1.857	1.762	0.762	0.239	0.810	1.810	9.764	4.739	2.668	2.358
Alice Mean	2.000	0.334	2.000	2.667	0.334	1.001	0.334	1.167	9.836	4.751	1.668	3.417
Python Mean	1.134	0.934	1.667	0.867	0.667	-0.466	1.000	1.800	7.603	3.402	2.801	1.401
Percent Change		Scores										
Means	Basics	Logical Operators	If	Loops (w/o arrays)	Loops (w/ arrays)	Functions	Arrays (only)	Objects	Overall	Definition	Trace	Write
Overall Mean	2.78%	6.67%	11.42%	34.53%	-11.10%	167.01%	47.81%	5.55%	19.52%	17.74%	20.42%	22.21%
Alice Mean	-9.87%	-79.98%	-24.99%	88.20%	0.00%	250.28%	0.00%	-36.35%	10.28%	16.32%	-37.48%	57.62%
Python Mean	-26.07%	99.91%	92.23%	-38.07%	44.43%	-16.68%	87.48%	42.08%	10.68%	6.25%	13.51%	16.66%

Figure 5-1. Mean pre-test (session one), post-test (session two), and percent change scores for pilot study (categorization scheme one).

Session 1 Scores												
Means	Basics		Logical					Overall	Definition Trace		Write	
	Operators	If	For	While	Functions	Arrays	Objects					
Overall Mean	1.715	0.715	1.667	1.409	0.762	-0.356	0.548	1.715	8.169	4.025	2.215	1.929
Alice Mean	1.334	1.667	2.667	1.084	0.667	-0.666	0.334	1.834	8.919	4.085	2.668	2.187
Python Mean	1.524	0.467	0.867	1.667	0.934	-0.399	0.593	1.267	6.870	3.202	2.467	1.203
Session 2 Scores												
Means	Basics		Logical					Overall	Definition Trace		Write	
	Operators	If	For	While	Functions	Arrays	Objects					
Overall Mean	1.762	0.762	1.857	1.953	0.572	0.239	0.310	1.810	9.764	4.739	2.668	2.358
Alice Mean	2.000	0.334	2.000	2.000	1.000	1.001	0.334	1.167	9.836	4.751	1.668	3.417
Python Mean	1.134	0.934	1.667	1.400	0.134	-0.466	1.000	1.800	7.503	3.402	2.801	1.401
Percent Change												
Means	Basics		Logical					Overall	Definition Trace		Write	
	Operators	If	For	While	Functions	Arrays	Objects					
Overall Mean	2.78%	6.67%	11.42%	38.96%	-24.98%	167.01%	47.81%	5.55%	19.52%	17.74%	20.42%	22.21%
Alice Mean	49.97%	-79.98%	-24.99%	84.57%	49.96%	250.28%	0.00%	-36.35%	10.28%	16.32%	-37.48%	-57.67%
Python Mean	-26.07%	99.91%	92.23%	-15.99%	-85.07%	-16.68%	87.48%	42.08%	10.68%	6.25%	-13.51%	16.66%

Figure 5-2. Mean pre-test (session one), post-test (session two), and percent change scores for pilot study (categorization scheme two).

5.5 Analysis

The findings were both interesting and unexpected. (Note: For the purpose of this discussion, the students who had programmed mainly in *Python* will be referred to as the “*Python* programmers,” and the students who had programmed mainly in *Alice* will be referred to as the “*Alice* programmers.”) For example, the *Alice* programmers outperformed the *Python* programmers on the questions involving objects in session one (as predicted), but the situation was reversed in session two. Overall, the *Python* programmers exhibited a 42.1% increase in performance on the objects questions, whereas the *Alice* programmers exhibited a 36.4% decrease in performance. The situation was similar for the questions involving tracing code. In session one, the *Alice* programmers once again outperformed the *Python* programmers, but in session two, the opposite occurred. Overall, the *Python* programmers exhibited a 13.5% increase in performance on the code tracing questions, whereas the *Alice* programmers

exhibited a 37.5% decrease in performance. (The prediction was that the code-based, in this case Python, programmers would prevail on questions involving program flow.)

The results from the questions involving writing code were unexpected (although the write questions merely required students to recognize the correct programming solution, not actually author it). The *Alice* programmers outperformed the *Python* programmers in both sessions (*not* as predicted). In fact, from session one to session two, the *Alice* programmers exhibited a 57.7% increase in performance on the code writing questions, whereas the *Python* programmers exhibited only a 16.7% increase in performance.

Both groups of programmers performed similarly in session one on the questions involving basics, loops (without arrays), functions, and arrays (only, no loops). These results did not carry over to session two, however. From session one to session two, the *Alice* programmers exhibited an increase in performance on the basics, loops (without arrays), and functions questions, whereas the *Python* programmers exhibited a decrease in performance on those questions. Interestingly, the increase in performance demonstrated by the *Alice* programmers on the functions questions was an incredible 250.3%!

The overall improvement in performance for the two sessions was comparable, 10.3% for the *Alice* programmers and 10.7% for the *Python* programmers.

CHAPTER 6: RESEARCH QUESTIONS AND HYPOTHESES

6.1 Research Questions

Explorations into the different teaching approaches and programming languages currently being used to teach introductory object-oriented (OO) programming led to questions regarding the impact of these approaches and languages on students' ability to learn both fundamental and OO programming concepts. Assessing these abilities is best accomplished using a computer concepts inventory. Since no such tool exists, one was developed for the purpose of this research. The research questions that follow resulted from the processes of exploration and tool development described above. They are:

- Q 1: In a field like computer science where there are educational guidelines but no actual standards, is it possible to create a tool to assess student knowledge of fundamental and object-oriented programming concepts that is sufficiently “generic” to be adopted for general use?
- Q 2: What effect does the teaching approach and/or programming language have on students' ability to grasp fundamental programming concepts (e.g., assignment, iteration, functional decomposition, program flow, etc.)?
- Q 3: What effect does the teaching approach and/or programming language have on students' ability to grasp specific object-oriented programming concepts (e.g., class creation, method invocation, object instantiation, etc.)?
- Q 4: What effect does the teaching approach and/or programming language have on students' ability to select the correct code to complete the solution to a programming problem when provided with multiple options?

6.2 Related Hypotheses

The next chapter will describe the design of the experiment conducted to investigate the research questions posed above. The hypotheses that were tested with respect to the research questions follow.

The hypothesis that was tested with respect to the first research question (Q 1) was:

- H 1: The computer concepts assessment tool created for use in this study will prove effective if a majority of its questions discriminate well between high and low scoring students.

The hypothesis that was tested with respect to the second research question (Q 2) was:

- H 2: Students taught to program using different approaches and/or languages will exhibit different strengths and weaknesses with respect to their performance on questions that test their knowledge of fundamental programming concepts.

The hypothesis that was tested with respect to the third research question (Q 3) was:

- H 3: Students taught to program using the “objects-first/objects-early” approach will perform better on problems that test their knowledge of object orientation than students taught to program using the “programming-first” approach.

The hypothesis that was tested with respect to the fourth research question (Q 4) was:

- H 4: Students taught to program using highly structured languages will perform better on code-completion problems than students taught to program using less structured languages.

CHAPTER 7: EXPERIMENTAL STUDY DESIGN

7.1 Purpose of Study

This research study examined the performance of students enrolled in freshman level programming courses for the purpose of investigating the research questions outlined in the preceding chapter. It was specifically designed to explore the impact of different teaching approaches and languages on students' grasp of fundamental and object-oriented programming concepts using the assessment tool developed for the study.

7.2 Description of Methodology (General)

7.2.1 *Time Dimension*

The study was an exploratory study conducted over the course of two quarters (Drexel University) and one semester (Rowan University).

7.2.2 *Units of Analysis*

The dual nature of the study necessitated two units of analysis. The first was the computer concepts inventory developed to assess student learning of fundamental and object-oriented programming concepts. The second was the teaching approach and/or programming language used in the introductory programming courses.

7.2.3 *Units of Observation (Participants)*

The units of observation were students enrolled in freshman level programming courses at Rowan University in Glassboro, New Jersey, and Drexel University in Philadelphia, Pennsylvania. Details about the students and the courses from which they were recruited are presented in Chapter 8.

7.2.4 *Constructs*

The indicators described in the *Terms or Components* section below were used to evaluate the effectiveness of the computer concepts inventory and to assess the impact of the different teaching approaches and languages on students' grasp of fundamental and object-oriented programming concepts.

7.2.5 *Terms or Components*

Standard item analysis techniques for multiple-choice tests were used to evaluate the quality of the computer concepts inventory. These techniques are discussed in parallel with the results of the analysis in Chapter 8.

The computer concepts inventory was developed because indicators were needed to measure the abstract ideas I wished to study. Specifically, I hoped to identify features of approaches and languages for teaching introductory object-oriented programming that fostered/hampered students' ability to:

- Grasp fundamental programming concepts such as assignment, iteration, functional decomposition, and program flow.
- Grasp specific object-oriented programming concepts such as class creation, method invocation, and object instantiation.
- Choose the correct code to complete the solution to a programming problem when provided with multiple options.

A variety of statistical analysis techniques were used to analyze the students' performance on the computer concepts inventory. These included the repeated measures ANOVA; the one-way, between-subjects ANOVA; the Tukey test; and the within-subjects paired samples *t*-test.

Quantitative techniques were also employed to interpret the students' responses to the demographic and attitude surveys.

7.2.6 Level of Measurement

The level of measurement was predominantly scalar.

7.3 Description of Methodology (Detailed)

7.3.1 Description

This experiment explored the impact of different teaching approaches and languages on students' grasp of fundamental and object-oriented programming concepts using the assessment tool developed for the study.

7.3.2 Research Questions

- Q 1: In a field like computer science where there are educational guidelines but no actual standards, is it possible to create a tool to assess student knowledge of fundamental and object-oriented programming concepts that is sufficiently "generic" to be adopted for general use?
- Q 2: What effect does the teaching approach and/or programming language have on students' ability to grasp fundamental programming concepts (e.g., assignment, iteration, functional decomposition, program flow, etc.)?
- Q 3: What effect does the teaching approach and/or programming language have on students' ability to grasp specific object-oriented programming concepts (e.g., class creation, method invocation, object instantiation, etc.)?
- Q 4: What effect does the teaching approach and/or programming language have on students' ability to select the correct code to complete the solution to a programming problem when provided with multiple options?

7.3.3 *Related Hypotheses*

- H 1: The computer concepts assessment tool created for use in this study will prove effective if a majority of its questions discriminate well between high and low scoring students.
- H 2: Students taught to program using different approaches and/or languages will exhibit different strengths and weaknesses with respect to their performance on questions that test their knowledge of fundamental programming concepts.
- H 3: Students taught to program using the “objects-first/objects-early” approach will perform better on problems that test their knowledge of object orientation than students taught to program using the “programming-first” approach.
- H 4: Students taught to program using highly structured languages will perform better on code-completion problems than students taught to program using less structured languages.

7.3.4 *Methodology*

Students enrolled in freshman level Java, C++, and Visual Basic programming courses at Rowan University in Glassboro, New Jersey, and Drexel University in Philadelphia, Pennsylvania, participated in the study. They were assessed using the techniques described below and were assured that their performance on study-related tasks would not adversely affect their course grade in any way.

- At the start of the course, I asked the students to complete three surveys:
 - A demographic survey that asked them to provide background information about their academic level, major, gender, ethnicity, age, programming language experience, and SAT and ACT math scores.

- An attitude survey that asked them to respond to a series of statements designed to gain insight into their attitudes toward computer science and programming.
- A computer concepts survey that asked them to answer twenty-five multiple-choice questions designed to assess their knowledge of computer programming concepts (the computer programming concepts tool described in Chapter 4).
- At the end of the course, I asked the students once again to complete the attitude survey and computer concepts survey for purposes of comparison.

7.3.5 *Data Analysis*

The data gathered during the study was analyzed for the purpose of drawing conclusions regarding the stated hypotheses. The dual nature of the study necessitated two different forms of analysis:

- Item response pattern analysis, item discrimination analysis, and reliability analysis to evaluate the effectiveness of the computer concepts assessment instrument.
- Standard statistical techniques that included the repeated measures ANOVA; the one-way, between-subjects ANOVA; the Tukey test; and the within-subjects paired samples *t*-test to analyze the students' performance on the assessment instrument.

The above techniques are discussed in parallel with the results of the analysis in Chapter 8.

Quantitative techniques were also employed to interpret the students' responses to the demographic and attitude surveys.

CHAPTER 8: MAIN STUDY DESCRIPTION AND RESULTS

8.1 Purpose

A research study was conducted to investigate the following research questions:

- Q 1: In a field like computer science where there are educational guidelines but no actual standards, is it possible to create a tool to assess student knowledge of fundamental and object-oriented programming concepts that is sufficiently “generic” to be adopted for general use?
- Q 2: What effect does the teaching approach and/or programming language have on students’ ability to grasp fundamental programming concepts (e.g., assignment, iteration, functional decomposition, program flow, etc.)?
- Q 3: What effect does the teaching approach and/or programming language have on students’ ability to grasp specific object-oriented programming concepts (e.g., class creation, method invocation, object instantiation, etc.)?
- Q 4: What effect does the teaching approach and/or programming language have on students’ ability to select the correct code to complete the solution to a programming problem when provided with multiple options?

8.2 Participants

Sixty-nine students participated in the study. Sixty-four of these were from Rowan University in Glassboro, New Jersey. The remaining five were from Drexel University in Philadelphia, Pennsylvania. The Rowan students were enrolled in courses in Java (CS 113, CS 114), Visual Basic (CS 141), and C++ (CS 103) during the Spring 2009-2010 semester. The Drexel students were enrolled in courses in C++ (CS 132, CS

172, SE 103) during the Spring 2009-2010 quarter. Students in these particular courses were targeted for recruitment because of the level of the course and the programming language in which it was taught. Specifically, I sought to recruit students taking freshman level programming courses that employed a variety of languages and teaching approaches. I also sought to recruit entire classes where possible in order to insure a wide range of abilities and a diversity of majors among the student participants.

The above students were recruited by contacting the instructors of the courses of interest, who then advertised the study on my behalf or who graciously invited me into their classes so that their students, if willing, could complete my surveys. The CS 113 (Java), CS 141 (Visual Basic), and CS 103 (C++) students represented a substantial proportion of their respective classes. The CS 114 (Java), CS 132 (C++), CS 172 (C++), and SE 103 (C++) represented a much smaller proportion of their respective classes.

All of the students who participated in the study were enrolled in courses that required prior programming experience. Although the prerequisites are not always enforced, it was clear from the student responses to the demographic survey that most had programmed before. Languages previously studied included Alice, Basic, C, C++, Java, JavaScript, and Visual Basic.

The sixty-one students who completed both parts of the study represented a variety of majors. Fourteen were Computer Science (CS) majors. Twenty-one were Management Information Systems (MIS) majors; two of the MIS majors had second majors in Marketing (MKT) and Radio, Television, and Film (RTF), respectively. Nine were Electrical and Computer Engineering (ECE) majors. Ten were Mechanical Engineering (ME) majors. Three were Mathematics (MATH) majors; two of the MATH

majors had second majors in Physics (PHYS) and Economics (ECON), respectively. Of the three remaining students, one was a Software Engineering (SE) major, one was a Liberal Arts (LA) major, and one was non-matriculated (N/A). Interestingly, seven of the eight students who completed only the first part of the study were MIS majors; the eighth was a CS major.

Additional information about the research study participants, such as the fact that only seven out of sixty-one were female, can be viewed in Appendix L.

8.3 Task

Sixty-one of the original sixty-nine participants completed the study, which consisted of two sessions. The first session was scheduled near the beginning of the term, while the second session was scheduled near the end of the term. During the first session, each student was asked to complete three surveys: a demographic survey to obtain information about his background (Appendix M); an attitude survey to gain insight into his attitude toward computer science and programming (Appendix N); and a computer concepts survey to assess his knowledge of computer programming concepts (Appendix O). During the second session, each student was asked once again to complete the attitude and computer concepts surveys. Most of the students completed each session in an hour or less. The sessions were not timed. Each student who completed the study received a stipend of \$15.00.

8.4 Results

Because of the dual nature of the research study, the presentation, analysis, and discussion of the data gathered is divided into two parts. The first part will focus on the

assessment tool. The second part will explore how useful the tool was in assessing student performance.

8.5 Assessment Tool

8.5.1 Scoring Procedure

The scoring procedure for the version of the computer concepts survey used in the pilot study employed the algorithm used to compute a raw score on the multiple-choice portions of the SAT. This algorithm assigns a “1” for each *correct* response, a “ $1/(\text{number of incorrect responses})$ ” for each *incorrect* response, and a “0” for each *nonresponse*. However, after reviewing additional literature on constructing and scoring multiple-choice tests (DeAyala, Plake, and Impara, 2001; Ebel and Frisbie, 1991; Haladyna, 2004), it seemed more appropriate to use an algorithm that assigns a “1” for each *correct* response, a “0” for each *incorrect* response, and a “0.5” for each *nonresponse*. In a study that investigated the effect of omitted responses on an examinee's ability estimate, DeAyala, Plake, and Impara (2001) found that substituting 0.5 for omitted responses yielded ability estimates nearly as accurate as those that used complete data. They also concluded that treating blank responses as incorrect is probably the worst practice.

8.5.2 Item Response Pattern Analysis

Multiple techniques exist for evaluating the effectiveness of assessment tools. One approach for evaluating the quality of tools composed of multiple-choice questions is to study the item (question) response patterns. Every multiple-choice question exhibits a response pattern that is either desirable or undesirable. High scoring students tend to choose correct answers, while low scoring students tend to choose incorrect answers.

Research has revealed a pattern to the relationship between distracter (incorrect answer) choice and overall test score (Haladyna, 2004). Consequently, it is important to examine both correct and incorrect answer choices. Our starting point for evaluating the effectiveness of the computer concepts assessment tool constructed for this research is to determine:

- The proportion of students who answered each question.
- The proportion of students who answered each question correctly.
- The efficiency of the distracters.

Figure 8-1 shows the results of this analysis for session one of the study. Figure 8-2 shows the results for session two. Only the responses for the 61 students who completed both parts of the study were used in the analysis.

Response %	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
A	0.61	0.02	0.02	0.30	0.10	0.20	0.05	0.13	0.49	0.18	0.39	0.25
B	0.10	0.05	0.74	0.05	0.13	0.31	0.03	0.61	0.15	0.05	0.15	0.23
C	0.18	0.80	0.20	0.08	0.36	0.02	0.67	0.10	0.13	0.03	0.28	0.31
D	0.10	0.13	0.03	0.54	0.36	0.46	0.23	0.07	0.11	0.69	0.03	0.11
E												
Total	0.98	1.00	0.98	0.97	0.95	0.98	0.98	0.90	0.89	0.95	0.85	0.90

Response %	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Q23	Q24
A	0.39	0.39	0.31	0.18	0.21	0.28	0.28	0.52	0.16	0.11	0.15	0.08
B	0.10	0.05	0.13	0.31	0.36	0.07	0.11	0.18	0.39	0.25	0.25	0.20
C	0.05	0.33	0.23	0.15	0.28	0.31	0.25	0.03	0.18	0.21	0.38	0.46
D	0.43	0.10	0.25	0.25	0.07	0.23	0.18	0.18	0.07	0.30	0.11	0.13
E												
Total	0.97	0.87	0.92	0.89	0.92	0.89	0.82	0.92	0.80	0.87	0.89	0.87

Correct Response:

Figure 8-1. Proportion of students who chose each question response for session one (pre-test).

Response %	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10	Q 11	Q 12
A	0.69	0.05	0.10	0.30	0.11	0.23	0.10	0.10	0.51	0.15	0.48	0.33
B	0.15	0.16	0.70	0.05	0.23	0.25	0.07	0.70	0.13	0.07	0.18	0.20
C	0.08	0.69	0.13	0.21	0.30	0.11	0.59	0.15	0.20	0.18	0.28	0.30
D	0.03	0.10	0.05	0.44	0.34	0.39	0.25	0.05	0.10	0.56	0.02	0.11
E												
Total	0.95	1.00	0.98	1.00	0.98	0.98	1.00	1.00	0.93	0.95	0.95	0.93

Response %	Q 13	Q 14	Q 15	Q 16	Q 17	Q 18	Q 19	Q 20	Q 21	Q 22	Q 23	Q 24
A	0.44	0.49	0.36	0.25	0.23	0.23	0.38	0.41	0.23	0.18	0.21	0.03
B	0.11	0.03	0.16	0.31	0.26	0.30	0.11	0.20	0.38	0.16	0.26	0.23
C	0.05	0.31	0.23	0.26	0.34	0.13	0.16	0.13	0.11	0.30	0.39	0.39
D	0.38	0.08	0.18	0.11	0.13	0.28	0.20	0.16	0.15	0.33	0.07	0.21
E												
Total	0.98	0.92	0.93	0.93	0.97	0.93	0.85	0.90	0.87	0.97	0.93	0.87

Correct Response:

Figure 8-2. Proportion of students who chose each question response for session two (post-test).

The rows labeled “Total” show the proportion of students who answered each question on the test for each of the two sessions. The rose-highlighted cells show the proportion of students who answered each question correctly. This value is referred to as the *item facility* (Elvin, 2004) or *p value* (Haladyna, 2004). The remaining cells show the proportion of students who chose distracters (incorrect responses). Although in some cases the proportions are low, it is important to note that all possible responses, correct as well as incorrect, were selected at least once. Response E was an option only for Question 25, which asked students to rate the difficulty level of the test using a Likert-type scale.

A more detailed breakdown of student responses that considers test score distribution provides additional information about response patterns for correct as well as incorrect answers. Student records were arranged in descending order by score, then divided into four approximately equal-sized groups. For each group, the proportion of students who chose each of the four possible responses was calculated. Figure 8-3

displays the breakdown for a subset of questions and corresponding responses for session two. These particular questions were chosen as exemplars because their response patterns represent a variety of patterns typically seen in multiple-choice questions with four answer choices. (See Appendices D and E for the breakdowns for all 24 questions and their responses for both sessions.) For each question, the proportion of correct responses should vary directly with the students' scores (Haladyna, 2004). For each of the question's distracters, the proportion of responses should vary inversely. As we shall see shortly, however, this is not necessarily the case.

Quartiles per response

Response %	Q 5	Q 9	Q 19	Q 20	Q 23
A	0.03	0.21	0.16	0.20	0.00
	0.02	0.15	0.10	0.11	0.05
	0.03	0.07	0.03	0.07	0.07
	0.03	0.08	0.08	0.03	0.10
B	0.05	0.00	0.02	0.00	0.02
	0.05	0.03	0.02	0.02	0.08
	0.07	0.05	0.03	0.08	0.07
	0.07	0.05	0.05	0.10	0.10
C	0.00	0.00	0.02	0.00	0.21
	0.07	0.05	0.03	0.03	0.08
	0.08	0.05	0.05	0.03	0.08
	0.15	0.10	0.07	0.07	0.02
D	0.16	0.02	0.03	0.05	0.00
	0.11	0.00	0.07	0.03	0.02
	0.05	0.05	0.08	0.03	0.02
	0.02	0.03	0.02	0.05	0.03
Total	0.98	0.93	0.85	0.90	0.93

Correct Response:

Figure 8-3. Proportion of session two students who chose each response for Questions 5, 9, 19, 20, and 23, divided into four groups by score (high to low).

Desirable response patterns for correct answers are exhibited by questions 5, 9, 20, and 23. For each of these questions, the proportion of correct answers *decreases* overall as the students' scores *decrease*. Question 19, on the other hand, exhibits a correct response pattern that makes no sense. According to the numbers, a greater proportion of middle scoring students than high scoring students answered the question correctly!

Desirable response patterns for incorrect answers are exhibited by Question 5, option C; Question 9, options B and C; Question 19, options B and C; Question 20, options B and C; and Question 23, options A and B. For each of these response options, the proportion of incorrect answers *increases* overall as the students' scores *decrease*. The remaining distracters exhibit response patterns that are either unchanging, or changing in ways that may be difficult to interpret. For example, Question 5, options A and B, and Question 20, option D, exhibit response patterns that are virtually unchanging across score groups. In general, response patterns that display no orderly relation between the distracter and students' test scores should be improved or replaced because they are not working as intended.

Question 19, option A, on the other hand, exhibits a response pattern that more closely resembles one desired for a correct answer than for an incorrect answer. Indeed, more students chose option A than the correct answer, option D.

Question 19 and its answer choices appear below:

- 19) Which of the following statements about recursion is always true?
- a) A function that implements a recursive algorithm consists of two parts: a base case and a recursive step.
 - b) Recursion, unlike iteration, can never occur infinitely.
 - c) Any programming problem that can be solved either recursively or iteratively can be solved more efficiently using recursion.
 - d) Any programming problem that can be solved recursively can also be solved iteratively.

One possible explanation for this behavior relates to the topic the question addresses and an example frequently used to illustrate it. The topic is recursion; the frequently-used example is the mathematical function *factorial*. Implemented recursively, *factorial* has two parts: a base case that it solves by returning “1,” and a more complex case that it solves by calling itself until it is called to solve the base case. Question 19 asks “Which of the following statements about recursion is always true?” Option A states “A function that implements a recursive algorithm consists of two parts: a base case and a recursive step.” It is not surprising that students whose experience with recursion is limited to simple examples such as *factorial* selected this response. In retrospect, the question turned out to be too difficult for the students, as subsequent analyses will show.

The above tabular techniques provide a glimpse into desirable and undesirable test question response patterns. However, in order to formally evaluate test question performance, statistical techniques are required.

8.5.3 Item Discrimination Analysis

The next step is to carry out item (question) discrimination analysis to formally evaluate the overall performance of the computer concepts assessment tool as well as the performance of the individual questions of which it is composed (Ebel and Frisbie, 1991; Elvin, 2004; Haladyna, 2004). Unlike the above analyses, this process requires using the spreadsheets for the pre-test and post-test in which the students' letter responses have been converted to numbers (as described in the scoring procedure). The steps are as follows:

- 1) Sort the students' records in descending order by score.
- 2) Compute the proportion of students who answered each question correctly. As mentioned earlier, this value is the *item facility*, or *p value*.
- 3) Identify upper and lower groups separately. The upper group is the highest scoring 27% of the whole group. The lower group is the lowest scoring 27% of the whole group.
- 4) For each question, count the number of students in the upper group who chose the correct response. Divide this number by the size of the group. This value is the *item facility* for the upper group.
- 5) Compute the *item facility* for the lower group by following steps similar to those in step 4.
- 6) Subtract the *item facility* for the lower group from the *item facility* for the upper group. This value is the *item discrimination* (also referred to as *item-discrimination index* or *index of discrimination*).

- 7) For each question, add the counts for the correct number of responses for the upper and lower groups. Divide this sum by the total number of students in the two groups. This value is the *estimated index of item difficulty*.
- 8) Compute the *average score* and *standard deviation* of the test. These values can be easily calculated using functions built into spreadsheet programs such as Microsoft Excel.
- 9) Determine the *reliability* of the test. Reliability is calculated here using one of two widely accepted formulas developed by Kuder and Richardson (1937), K-R21.

Formula K-R21 is:

$$r = \frac{k}{k-1} \left[1 - \frac{\bar{X}(k-\bar{X})}{ks^2} \right]$$

where:

- r = reliability coefficient
 k = number of test items
 \bar{X} = mean test score
 s = standard deviation of test scores

K-R21 provides a conservative estimate of a test's reliability. K-R20, Kuder and Richardson's more complex formula for estimating reliability, cannot be used in this case because it is not applicable to tests that employ weighted scoring.

10) Estimate the *standard error of measurement* of the test. Standard error of measurement is calculated using the following formula:

$$s_E = s_X \sqrt{1-r}$$

where:

s_E = *standard error of measurement*

s_X = *standard deviation of test scores*

r = *reliability coefficient*

The results of the item discrimination analysis for the pre-test and post-test appear in Appendices F and G, respectively. Only the scores for the 61 students who completed both parts of the study were used in the calculations. Figure 8-4 below displays the results for a subset of questions that, in general, performed reasonably well on both the pre-test and post-test. The item facility (total group, top 27%, bottom 27%), item discrimination, and estimated index of item difficulty are clearly labeled.

It is important to note that while index of item difficulty is sometimes measured by calculating the proportion of test-takers who answer a question *incorrectly*, this researcher adopts the approach of Ebel and Frisbie (1991) who measure item difficulty by calculating the proportion of test-takers who answer a question *correctly*. As a result, higher values indicate easier questions and vice versa. Because the index of item difficulty reflects the ability of a particular group of test-takers, and not just the content of the question, it is more appropriately referred to as *estimated* index of item difficulty.

Pre-test

Question Number	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10	Q 11
Item Facility Total	0.607	0.803	0.738	0.295	0.361	0.459	0.672	0.607	0.492	0.689	0.393
Item Facility Top 27%	0.875	0.938	0.938	0.688	0.625	0.750	0.938	0.875	0.813	0.750	0.563
Item Facility Bottom 27%	0.313	0.750	0.500	0.000	0.188	0.063	0.438	0.438	0.188	0.563	0.125
Item Discrimination	0.563	0.188	0.438	0.688	0.438	0.688	0.500	0.438	0.625	0.188	0.438
Estimated Index of Item Difficulty	0.594	0.844	0.719	0.344	0.406	0.406	0.688	0.656	0.500	0.656	0.344

Post-test

Question Number	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10	Q 11	
Item Facility Total	0.689	0.689	0.705	0.295	0.344	0.393	0.590	0.705	0.508	0.557	0.475	
Item Facility Top 27%	0.875	1.000	0.938	0.688	0.625	0.875	0.938	0.938	0.875	0.813	0.750	
Item Facility Bottom 27%	0.438	0.500	0.500	0.063	0.063	0.063	0.375	0.375	0.313	0.500	0.188	
Item Discrimination	0.438	0.500	0.438	0.625	0.563	0.813	0.563	0.563	0.563	0.313	0.563	
Estimated Index of Item Difficulty	0.656	0.750	0.719	0.375	0.344	0.469	0.656	0.656	0.594	0.656	0.469	
Question Number	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10	Q 11	
Question Category	Basics			Logical expressions			Conditionals			For loops		

Index of Discrimination Scale:	
Very good	0.40 and up
Reasonably good	0.30 to 0.39
Marginal	0.20 to 0.29
Poor	Below 0.19

Figure 8-4. Session one (pre-test) and session two (post-test) item discrimination analysis for questions 1 through 11.

The most important values to attend to here are those for item discrimination. These values indicate how well questions discriminate between high and low scoring students. Since the goal of an effectively functioning test question is to discriminate between high and low scorers, it should go without saying that the higher these values are the better. The chart at the bottom shows a scale that is normally suitable for evaluating test question effectiveness based on item-discrimination indices (Ebel and Frisbie, 1991).

Figure 8-4 shows nine questions that exhibit very good item-discrimination indices for both the pre-test and the post-test: 1, 3 through 9, and 11. Questions 2 and 10, on the other hand, performed poorly on the pre-test, but respectably on the post-test. Questions that consistently perform poorly must be rejected or revised. Question 10's reasonably good performance on the post-test indicates that it may still be subject to improvement.

Figure 8-5 shows ten questions that exhibit a variety of changes in the item-discrimination indices from the pre-test to the post-test. The item-discrimination indices for Questions 13, 14, 15, and 24 are very good for the pre-test. Question 14, however, exhibits only marginal performance for the post-test. The item-discrimination indices for Questions 16, 21, and 22 are reasonably good for the pre-test and very good for the post-test. The remaining three questions show significant improvement in performance between the pre- and post-tests. Questions 18, 20, and 23 all performed poorly on the pre-test. On the post-test, however, Question 18's performance was reasonably good, while that of Questions 20 and 23 was very good.

Pre-test

Question Number	Q 13	Q 14	Q 15	Q 16	Q 18	Q 20	Q 21	Q 22	Q 23	Q 24
Item Facility Total	0.426	0.328	0.311	0.148	0.230	0.525	0.393	0.295	0.377	0.459
Item Facility Top 27%	0.688	0.563	0.625	0.438	0.250	0.688	0.563	0.438	0.375	0.750
Item Facility Bottom 27%	0.125	0.125	0.125	0.063	0.188	0.625	0.250	0.125	0.313	0.125
Item Discrimination	0.563	0.438	0.500	0.375	0.063	0.063	0.313	0.113	0.063	0.625
Estimated Index of Item Difficulty	0.406	0.344	0.375	0.250	0.219	0.656	0.406	0.281	0.344	0.438

Post-test

Question Number	Q 13	Q 14	Q 15	Q 16	Q 18	Q 20	Q 21	Q 22	Q 23	Q 24
Item Facility Total	0.377	0.311	0.361	0.262	0.279	0.410	0.377	0.328	0.393	0.393
Item Facility Top 27%	0.625	0.438	0.625	0.750	0.438	0.813	0.750	0.813	0.813	0.875
Item Facility Bottom 27%	0.188	0.188	0.188	0.063	0.063	0.125	0.250	0.063	0.063	0.125
Item Discrimination	0.438	0.250	0.438	0.688	0.375	0.688	0.500	0.750	0.750	0.750
Estimated Index of Item Difficulty	0.406	0.313	0.406	0.406	0.250	0.469	0.500	0.438	0.438	0.500
Question Number	Q 13	Q 14	Q 15	Q 16	Q 18	Q 20	Q 21	Q 22	Q 23	Q 24
Question Category	While loops		Functions		Software Engineering	Recursion		Classes and objects		

Index of Discrimination Scale:





Very good		0.40 and up
Reasonably good		0.30 to 0.39
Marginal		0.20 to 0.29
Poor		Below 0.19

Figure 8-5. Session one (pre-test) and session two (post-test) item discrimination analysis for questions 13 through 16, 18, and 20 through 24.

Figure 8-6 shows the results of the item discrimination analysis for the remaining three questions in the computer concepts survey. Basically, Questions 12, 17, and 19 all performed poorly on both the pre-test and the post-test. However, Question 17's item-discrimination index increased from 0.00 on the pre-test to 0.19 on the post-test, placing it very close to the minimum cutoff for marginal performance (0.20). I would therefore be inclined to include an improved version of this question in future versions of the test. Questions 12 and 19, on the other hand, would have to be revised significantly or

dropped. A negative item-discrimination index (e.g., Pre-test, Question 12) is particularly bad because it indicates that more low scoring students got the question right than high scoring students. A zero item-discrimination index (e.g., Post-test, Question 12) is almost as bad because it indicates that the question does not discriminate between high and low scoring students.

Pre-test

Question Number	Q 12	Q 17	Q 19
Item Facility Total	0.230	0.361	0.180
Item Facility Top 27%	0.188	0.375	0.313
Item Facility Bottom 27%	0.313	0.375	0.125
Item Discrimination	-0.125	0.000	0.188
Estimated Index of Item Difficulty	0.250	0.375	0.219
Question Number	Q 12	Q 17	Q 19
Question Category	Array w/o loop	Functions	Recursion

Post-test

Question Number	Q 12	Q 17	Q 19
Item Facility Total	0.197	0.262	0.197
Item Facility Top 27%	0.125	0.375	0.188
Item Facility Bottom 27%	0.125	0.188	0.063
Item Discrimination	0.000	0.188	0.125
Estimated Index of Item Difficulty	0.125	0.281	0.125
Question Number	Q 12	Q 17	Q 19
Question Category	Array w/o loop	Functions	Recursion





Index of Discrimination Scale:		
Very good		0.40 and up
Reasonably good		0.30 to 0.39
Marginal		0.20 to 0.29
Poor		Below 0.19

Figure 8-6. Session one (pre-test) and session two (post-test) item discrimination analysis for questions 12, 17, and 19.

To explain the variation in the item-discrimination indices, the way they are calculated will be reviewed. Specifically, item-discrimination index is computed by subtracting the item facility for the lower group from the item facility for the upper group. (Recall that item facility is the proportion of students in a particular group who answered each question correctly.) So, for example, if the performance of the upper group on a specific question improved, while that of the lower group declined or stayed the same, the item-discrimination index would increase. If, on the other hand, the opposite occurred, the item-discrimination index would decrease. Factors influencing the performance of the individual students who make up the different groups may include learning, motivation, stress, fatigue, and so on.

In the event that the above discussion has led the reader to believe that a question must perform well every time a test is administered, that is not the case. The purpose of all the comparisons was to show how (and why) question performance can change for better or worse from one administration of a test to the next. If, after repeated administrations, a question performs reasonably well most of the time, it should be retained. If, on the other hand, its performance is poor or inconsistent, it should be revised or replaced. The whole point of item discrimination analysis is to determine whether or not a test and the questions of which it is composed provide us with useful information about student performance. A question that performs poorly or inconsistently is useless because it provides us with no information.

Figures 8-7 and 8-8 below report reliability, average score, standard deviation, and standard error of measurement statistics for the pre-test and post-test. A reliability

coefficient of 0.60 or higher is considered sufficient for exploratory research; 0.70 or higher is considered adequate; and 0.80 or higher is considered good (Garson, 2010). Reliability and standard deviation are higher for the post-test, while average score and standard error of measurement are lower. A decrease in average score accompanied by an increase in reliability may seem “odd” until one considers the relationship between reliability, variance, and item-discrimination index. Score variance (standard deviation squared) is directly proportional to the square of the sum of the item-discrimination indices (Ebel and Frisbie, 1991). The reader will recall from the above discussion that the item-discrimination indices increased overall from the pre-test to the post-test. Consequently, variance also increased. It is generally true that the greater the score variance, the higher the reliability (Ebel and Frisbie, 1991).

Reliability	0.568
Average Score	11.402
Standard Deviation	3.626
Standard Error of Measurement	2.382

Figure 8-7. Session one (pre-test) reliability, average score, standard deviation, and standard error of measurement statistics for questions 1 through 24. Reliability is calculated using K-R21.

Reliability	0.749
Average Score	10.730
Standard Deviation	4.586
Standard Error of Measurement	2.297

Reliability	0.786
Average Score	10.230
Standard Deviation	4.682
Standard Error of Measurement	2.166

Figure 8-8. Session two (post-test) reliability, average score, standard deviation, and standard error of measurement statistics for questions 1 through 24 (left); and questions 1 through 24, minus questions 12 and 19 (right). Reliability is calculated using K-R21.

Figure 8-8 merits some additional discussion. As the caption indicates, the chart on the left reports the reliability, average score, standard deviation, and standard error of measurement values obtained when all 24 questions in the computer concepts survey are included in the calculations. The chart on the right, however, reports the values obtained when Questions 12 and 19 are excluded from the calculations. This was done to see what impact dropping two of the worst performing questions on the survey would have on the overall reliability of the test. Since dropping these questions also resulted in the two lowest item discrimination values being excluded from the calculations, reliability rose, as the discussion in the preceding paragraph suggested it would.

A student's observed score on a test consists of two components, a true score and an error score (Ebel and Frisbie, 1991; Elvin, 2004; Garson, 2010). Since these scores are not known, a method has been devised to estimate the standard deviation of the hypothetical error scores. This estimated value is called the standard error of measurement. It is calculated from the standard deviation and reliability coefficient of the observed scores using the formula provided in step 10 of the procedure for item discrimination analysis. The values for standard error of measurement and reliability are inversely related: as standard error of measurement goes down, reliability goes up, and vice versa. This is because reliability is the ratio of the true score to the observed (true + error) score. Therefore, the smaller the error score, the closer the observed score will be to the true score.

The values displayed in Figures 8-7 and 8-8 exemplify the inverse relationship of reliability and standard error of measurement. In all cases, an increase in reliability was accompanied by a decrease in standard error of measurement.

As a second measure of the assessment tool's reliability, *Cronbach's alpha* was computed using PASW Statistics 18 (2010). Unlike K-R20, Cronbach's alpha can be used to estimate reliability for tests that employ weighted scoring (Ebel and Frisbie, 1991). The formula is:

$$\alpha = \frac{k}{k-1} \left[1 - \frac{\sum s_i^2}{s^2} \right]$$

where:

α = alpha coefficient

k = number of test items

s_i^2 = variance of a single test item

$\sum s_i^2$ = sum of variances for all test items

s^2 = variance of test scores

The figure below displays the results of the computations.

Reliability Statistics			
Cronbach's Alpha	Cronbach's Alpha Based on Standardized Items	N of Items	Comment
0.652	0.646	24	Session 1 values: Questions 1 – 24
0.787	0.779	24	Session 2 values: Questions 1 – 24
0.816	0.815	22	Session 2 values: Questions 1 – 24, minus questions 12 and 19

Figure 8-9. Cronbach's alpha values for session one (pre-test) and session two (post-test).

Interestingly, the Cronbach's alpha values are all higher than the K-R21 reliability values. K-R21, however, may underestimate the K-R20 reliability coefficient depending on the average scores of the individual test items. Since the formulas for Cronbach's alpha and K-R20 are similar, the reliability estimates generated by each will also be similar.

Section 4.5.4 discussed steps already taken to demonstrate the "content validity" of the computer concepts assessment tool (Sections 8.5.2, 8.5.3), as well as steps in progress to demonstrate its "construct validity." Several different types of evidence provide support for content validity: intrinsic characteristics, study of question responses, expert review. Content validity is intrinsic, since the tool is based on the core concepts of the *Programming Fundamentals* knowledge area defined by the 2008 ACM/IEEE curricular guidelines (CS2008 Review Taskforce, 2008).

Haladyna (2004) makes a case for the importance of collecting evidence to validate both test scores and the question responses aggregated to form them. Section 8.5.2 analyzed the patterns for correct responses and incorrect responses (distracters) to gather evidence in support of question response validity. Analysis showed that all possible responses, correct as well as incorrect, were selected at least once, the desired result. Next, student records were arranged in descending order by score and divided into quartiles to examine the relationship between answer choice and overall test score. Ideally, the proportion of students choosing correct answers should be directly related to their scores, while the proportion choosing distracters should be inversely related. The desired relationship between correct responses and scores held for about 67% of the questions for the pre-test and 75% of the questions for the post-test. Although I am aware of no "magic number," this would seem to be a respectable showing. Between the

pre-test and the post-test, at most 50% of the distracters exhibited the desired inverse relationship between incorrect responses and scores, indicating that many of the distracters need to be improved or replaced.

Section 8.5.3 analyzed the overall performance of the computer concepts assessment tool as well as the performance of its individual questions to gather additional evidence in support of question response validity. Analysis showed that the assessment tool performed well overall at both the pre- and post-test administrations based on the item (question) discrimination values for the individual questions. Specifically, item discrimination ratings were either “very good” or “reasonably good” for 21 out of 24 questions.

Finally, I have obtained feedback on the computer concepts assessment tool from several of the computer science faculty whose students participated in the study. Faculty were asked to complete a review form (Appendix P) that asked them to: 1) answer the question; 2) decide whether the question reflects fundamental programming concepts that students should know after completing an introductory course in object-oriented programming; 3) rate the quality of the question on a scale of “do not use again,” “use again with major changes,” “use again with minor changes,” or “use again as is.” Reviewers were encouraged to indicate how questions should be changed or why they should not be used again, if appropriate. Feedback has been largely positive. One reviewer indicated that 19 out of 24 questions reflected basic concepts and should be used again “as is” or with “minor changes.” A second reviewer responded similarly, but specified different questions that should be changed or omitted.

Construct validity is more difficult to demonstrate than content validity.

Exploratory factor analysis is a statistical technique that can be used to determine, based on students' responses, whether the questions that make up an assessment tool group into constructs the tool is intended to measure (Hoegh and Moskal, 2009). Preliminary factor analysis extracted nine components. Since the last two components consisted of only one question each, the factor analysis was re-run without those two questions. Seven components were extracted, three of which were readily comprehensible. Questions 16, 20, 23, and 24 loaded on a factor I labeled "methods and functions." Questions 2, 3, and 5 loaded on a factor I labeled "mathematical and logical expressions." Questions 7, 9, and 11 loaded on a factor I labeled "control structures." Questions 4, 6, and 8 possibly load on a "questions with conditions" factor, but that would overlap with the "control structures" factor. Such overlap is not surprising, however, since control structures contain conditions. Figure 8-10 shows the factors and their question loadings.

Question	Factor			
	Methods and functions	Mathematical and logical expressions	Questions with conditions	Control structures
Q 16	.733			
Q 24	.722			
Q 23	.721			
Q 20	.632			
Q 2		.850		
Q 3		.769		
Q 5		.593		
Q 8			.814	
Q 4			.542	
Q 6			.353	
Q 11				.769
Q 7				.674
Q 9				.511

Figure 8-10. Factors representing the construct “understanding of fundamental programming concepts,” extracted through principal component analysis.

Ideally, *all* assessment tool questions should group into clearly defined factors that represent the construct “understanding of fundamental programming concepts,” but that will have to be a matter for future work.

8.6 Student Performance

8.6.1 Score Distribution Analysis

The figure below displays the score distributions for the 61 students who completed both parts of the study. Both resemble normal distributions. The post-test curve, however, is somewhat flatter than the pre-test curve. This is because the post-test mean is slightly lower than the pre-test mean, and the post-test scores are more widely distributed than the pre-test scores. Although inspection of the numeric data suggested a normal spread, it seemed prudent to confirm this visually since many statistical tests assume that data is normally distributed.

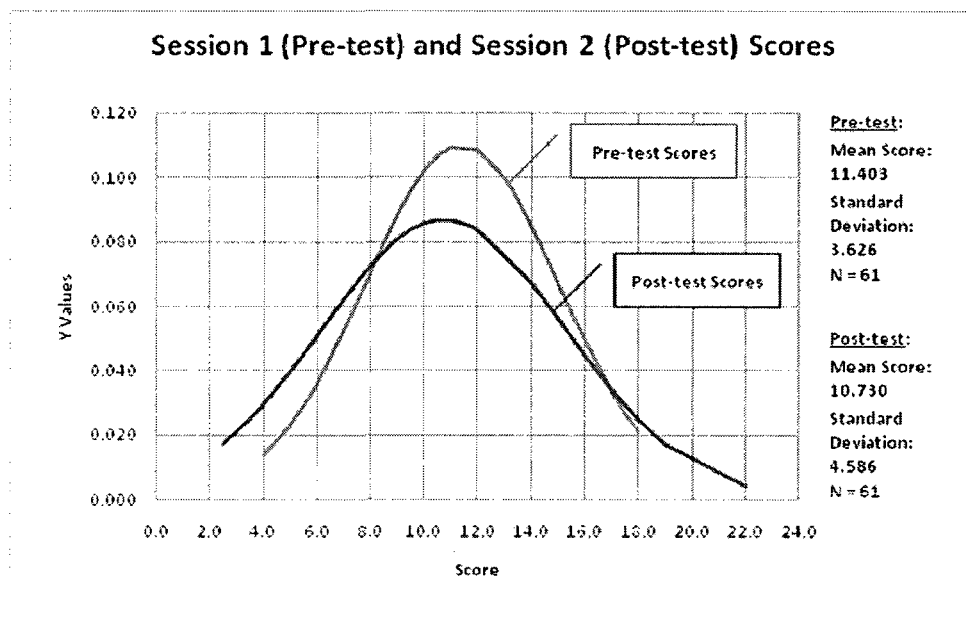


Figure 8-11. Session one (pre-test) and session two (post-test) scores for the 61 students who completed both parts of the study.

It seemed wise to determine definitively whether or not the difference between the pre-test and post-test means was significant. A within-subjects paired samples t-test *failed* to reveal a statistically reliable difference between the mean pre-test ($M = 11.40$, $s = 3.63$) and mean post-test ($M = 10.73$, $s = 4.59$) scores of the students: $t(60) = 1.65$, $p = .11$, $\alpha = .05$. The importance of this finding will become apparent in subsequent sections which examine student performance by language group.

One of the goals of this research is to determine the possible impact of different teaching languages (Java, Visual Basic, etc.) on students' learning of fundamental programming concepts. Therefore, the next step is to analyze their performance based on the language they were programming in when they participated in this study. The programming languages of interest are Java, Visual Basic, and C++.

8.6.2 Mean Scores Analysis by Language Group

This section begins an exploration into the research question:

- Q 2: What effect does the teaching approach and/or programming language have on students' ability to grasp fundamental programming concepts (e.g., assignment, iteration, functional decomposition, program flow, etc.)?

and its accompanying hypothesis:

- H 2: Students taught to program using different approaches and/or languages will exhibit different strengths and weaknesses with respect to their performance on questions that test their knowledge of fundamental programming concepts.

Students were divided into groups based on the language they were studying in the programming course they were taking when they participated in the study. The students had been recruited from eight different programming classes: two teaching Java,

two teaching Visual Basic, and four teaching C++. It seemed sensible to begin by looking at overall performance before focusing attention on student performance with respect to different programming concepts (e.g., conditionals, functions, etc.). The first step was therefore to calculate and compare the mean scores of the three different language groups (Java, Visual Basic, C++) on the pre-test and post-test.

Figure 8-11 displays the pre-test and post-test means broken down by language category. The Java students and the C++ students appear to have performed similarly on both the pre- and post-tests. The Visual Basic students, on the other hand, consistently performed more poorly than the other two groups of students.

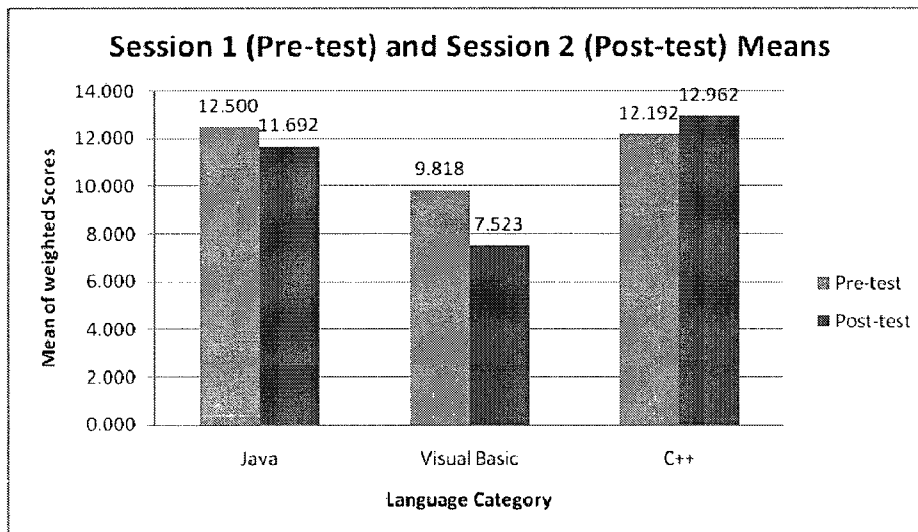


Figure 8-12. Mean scores for session one (pre-test) and session two (post-test) by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

To further investigate what Figure 8-11 suggests, statistical analysis was carried out to test whether or not language of instruction affects student performance. A one-factor, within-subjects analysis of variance (ANOVA) was performed to test if any of the language category means were significantly different. A conservative post hoc test, the Tukey test, was also used to make pair-wise comparisons among the means.

The tests of between-subjects effects for the one-factor, within-subjects analysis of variance (ANOVA) revealed a reliable effect of language category on the repeated measure, score: $F(2, 58) = 8.58, p < 0.01, MS_{error} = 23.23, \alpha = 0.05$. Partial eta squared = 0.23, which is considered a large effect (Kinnear and Gray, 2010). The Tukey test was used next to determine which pairs of means were significantly different.

The Tukey test revealed significant differences between two of the three pairs of means at the 0.05 α -level. The test confirmed differences between the Visual Basic mean and the means for Java and C++; the difference between the Java and C++ means is insignificant. The p -values for the differences between the Visual Basic mean and those for Java and C++ are 0.02 and less than 0.01, respectively; the p -value for the difference between the Java and C++ means is 1.00. The differences between the Visual Basic mean and those for Java and C++ are 3.43 and 3.91, respectively; the difference between the Java and C++ means is 0.48.

The results of the analysis presented above confirm that language of instruction does indeed affect overall student performance in introductory programming courses. It may be that highly structured languages like Java and C++ promote better overall learning than less structured languages like Visual Basic. It may also be that the interface design aspect of Visual Basic programming detracts from student learning of

fundamental concepts. I have learned through personal experience teaching introductory programming using Visual Basic that students seem to like the graphical aspect. However, it may be that the time spent learning to create graphical user interfaces (GUIs) would be better spent learning the fundamental programming constructs needed to implement them.

Figure 8-11 also showed a decrease in mean scores for the Java and Visual Basic students and an increase in mean scores for the C++ students from the pre-test to the post-test. Although not all changes appeared to be significant, a test to determine the difference (if any) between the pre- and post-test means was indicated. A within-subjects paired samples *t*-test was performed to test if the pre-test and post-test means within each of the different language groups (Java, Visual Basic, C++) were significantly different.

The paired samples *t*-test revealed a significant difference between the pre-test and post-test means for only one of the three language groups. The *t*-test confirmed a difference between the pre- and post-test means for the Visual Basic students; the difference between the pre- and post-test means for the Java and C++ students is insignificant. Specifically, a paired samples *t*-test revealed a statistically reliable difference between the mean pre-test ($M = 9.82$, $s = 3.06$) and post-test ($M = 7.52$, $s = 2.12$) scores of the Visual Basic students: $t(21) = 3.29$, $p < 0.01$, $\alpha = 0.05$. However, the paired samples test failed to reveal a statistically reliable difference between the mean pre-test ($M = 12.50$, $s = 4.03$) and post-test ($M = 11.69$, $s = 5.62$) scores of the Java students: $t(12) = 0.92$, $p = 0.37$, $\alpha = 0.05$. The test similarly failed to reveal a statistically

reliable difference between the mean pre-test ($M = 12.19$, $s = 3.52$) and post-test ($M = 12.96$, $s = 4.10$) scores of the C++ students: $t(25) = -1.59$, $p = 0.12$, $\alpha = 0.05$.

As the above analysis confirmed, the performance of the Java and C++ students remained fairly constant over the course of the research study, while the performance of the Visual Basic students declined. One possible explanation for this result is the approach used to teach the course. The prerequisite course introduced the students to Visual Basic programming using the typical lecture/lab approach in which they were assessed at regular intervals to reinforce their learning. The follow-up course – the course the students were taking when they participated in this study – used a project-based approach. According to the course syllabus, 90% of the student's grade was based on the design and implementation of a small-scale enterprise system. Although the project was intended to cover all basic and object-oriented programming concepts, it may be that a semester-long project was less effective at reinforcing student learning of fundamental concepts than exams, quizzes, and lab assignments given regularly throughout the course.

Examination of how students responded to different questions on the attitude survey, particularly those addressing the usefulness of programming in their future work, may provide some insight into this result.

8.6.3 *Topic Area Analysis by Language Group*

To explore the impact of language of instruction on student performance with respect to different programming concepts, questions were grouped into the categories used during the item discrimination analysis (basics, logical expressions, conditionals, and so forth). For each student, a score was computed for each question category by adding the scores of the questions that made up the category. (The maximum score for each category is equal to the number of questions in the category: for example, “3” for “basics” and “3” for “conditionals.”) The mean scores for the question categories were calculated for the pre-test and the post-test. One-factor, within-subjects analysis of variance (ANOVA) tests were then performed to determine if language of instruction affected student performance on the fundamental programming concepts represented by the different question categories.

Figures 8-12 and 8-13 display the pre- and post-test means for the two question categories on which language of instruction had a significant effect, “basics” and “logical expressions.”

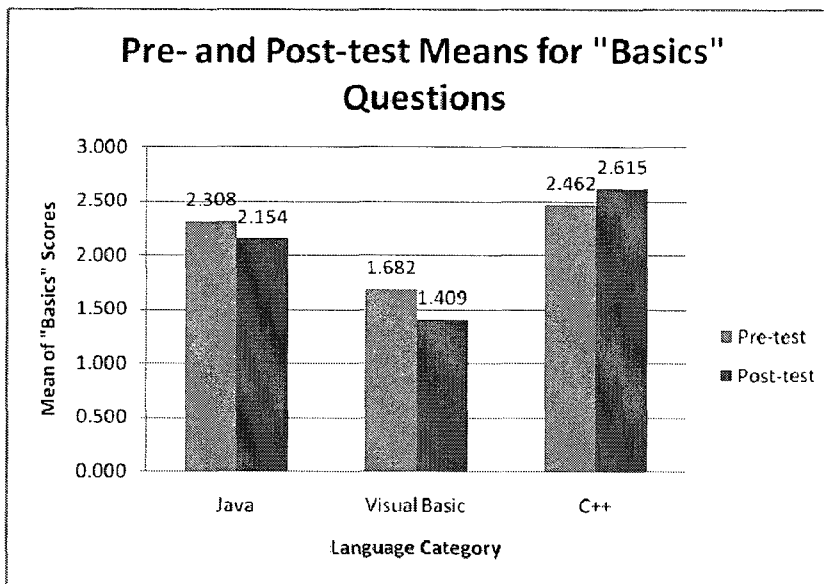


Figure 8-13. Mean "basics" scores for pre-test and post-test by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

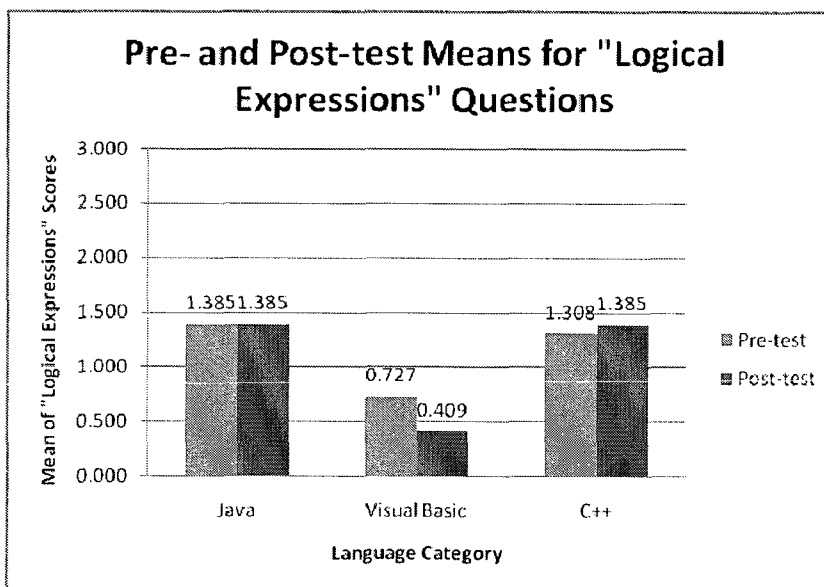


Figure 8-14. Mean "logical expressions" scores for pre-test and post-test by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

The tests of between-subjects effects for the one-factor, within-subjects analysis of variance (ANOVA) revealed a reliable effect of language category on the repeated measure, “basics” score: $F(2, 58) = 16.75, p < 0.01, MS_{error} = 0.72, \alpha = 0.05$. Partial eta squared = 0.37, which is considered a large effect (Kinnear and Gray, 2010). The Tukey test revealed significant differences between the Visual Basic mean and the means for Java and C++ at the 0.05 α -level; the difference between the Java and C++ means is insignificant. The p -values for the differences between the Visual Basic mean and those for Java and C++ are less than 0.01 and less than 0.01, respectively; the p -value for the difference between the Java and C++ means is 0.14. The differences between the Visual Basic mean and those for Java and C++ are 0.69 and 0.99, respectively; the difference between the Java and C++ means is 0.31.

The tests of between-subjects effects for the one-factor, within-subjects analysis of variance (ANOVA) also revealed a reliable effect of language category on the repeated measure, “logical expressions” score: $F(2, 58) = 5.31, p = 0.01, MS_{error} = 1.66, \alpha = 0.05$. Partial eta squared = 0.16, which is considered a large effect (Kinnear and Gray, 2010). The Tukey test once again revealed significant differences between the Visual Basic mean and the means for Java and C++ at the 0.05 α -level; the difference between the Java and C++ means is insignificant. The p -values for the differences between the Visual Basic mean and those for Java and C++ are 0.01 and 0.01, respectively; the p -value for the difference between the Java and C++ means is 0.90. The differences between the Visual Basic mean and those for Java and C++ are 0.82 and 0.78, respectively; the difference between the Java and C++ means is 0.04.

These results are particularly disturbing, since assignment, mathematical operations, and logical expressions are fundamental constructs upon which all other programming constructs are based. Without a firm grasp of the basics, a student cannot begin to understand conditionals, loops, functions, classes, and so forth. Unfortunately, it is almost impossible to build a computer program that does anything useful without employing these more advanced structures.

The remainder of this section explores the research question:

- Q 3: What effect does the teaching approach and/or programming language have on students' ability to grasp specific object-oriented programming concepts (e.g., class creation, method invocation, object instantiation, etc.)?

and its accompanying hypothesis:

- H 3: Students taught to program using the “objects-first/objects-early” approach will perform better on problems that test their knowledge of object orientation than students taught to program using the “programming-first” approach.

Figure 8-14 displays the pre- and post-test means for the “classes and objects” question category.

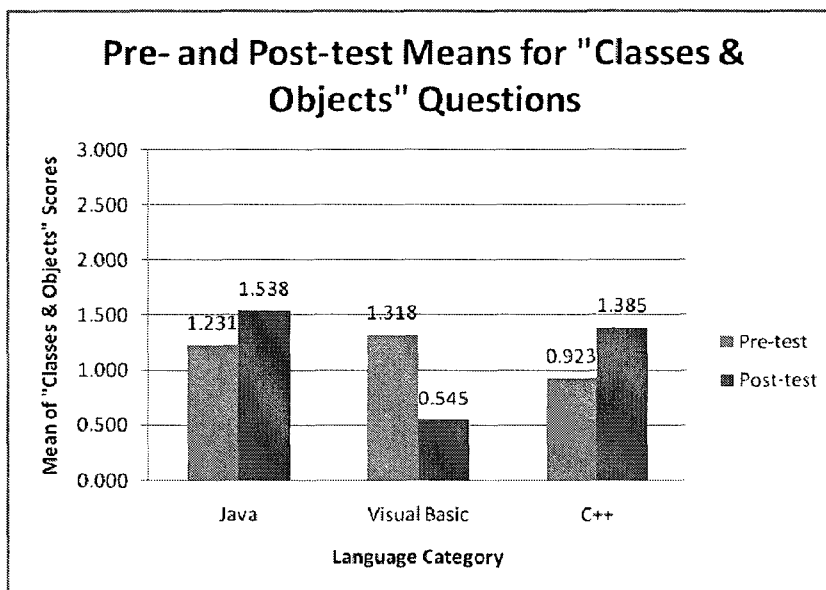


Figure 8-15. Mean “classes and objects” scores for pre-test and post-test by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

A one-factor, within-subjects analysis of variance (ANOVA) was performed to determine if language of instruction affected student performance on the “classes and objects” questions. The tests of between-subjects effects for the one-factor, within-subjects ANOVA *failed* to reveal a reliable effect of language category on the repeated measure, “classes and objects” score: $F(2, 58) = 0.98$, $p = 0.38$, $MS_{error} = 1.76$, $\alpha = 0.05$.

Figure 8-14 showed a decrease in the mean “classes and objects” scores for the Visual Basic students and an increase in the mean “classes and objects” scores for the Java and C++ students. To determine if the pre- and post-test “classes and objects” means within any of the language groups were significantly different, within-subjects paired samples *t*-tests were performed. The results were only significant for the C++ and Visual Basic students:

- C++:
 - A paired samples *t*-test revealed a statistically reliable difference between the mean pre-test ($M = 0.92$, $s = 1.13$) and post-test ($M = 1.39$, $s = 1.30$) “classes and objects” scores of the C++ students: $t(25) = -2.74$, $p = 0.01$, $\alpha = 0.05$.
- Visual Basic:
 - A paired samples *t*-test revealed a statistically reliable difference between the mean pre-test ($M = 1.32$, $s = 1.17$) and post-test ($M = 0.55$, $s = 0.60$) “classes and objects” scores of the Visual Basic students: $t(21) = 2.94$, $p = 0.01$, $\alpha = 0.05$.

The one-factor, within-subjects ANOVA failed to show that language of instruction affects student performance on questions addressing the object-oriented concepts “classes and objects”. However, the *t*-tests demonstrated that there were performance differences *within* two of the three language groups. Although nothing definitive can be said about the impact of teaching approach and/or programming language on students’ ability to grasp object-oriented concepts, it is worth noting that, of the three groups of students, the Java students performed the most consistently on the “classes and objects” questions throughout the study. (The mean pre-test “classes and objects” scores for the Java, Visual Basic, and C++ students are 1.23, 1.32, and 0.92, respectively; the mean post-test scores are 1.54, 0.55, and 1.39, respectively.) Members of the Rowan Computer Science faculty use BlueJ, a novice programming environment that supports the “objects-first” approach, to introduce students to Java programming. It may be that being introduced to Java using this approach has affected the students’ performance with respect to the “classes and objects” questions.

8.6.4 Code-Completion Analysis by Language Group

This final section explores the research question:

- Q 4: What effect does the teaching approach and/or programming language have on students' ability to select the correct code to complete the solution to a programming problem when provided with multiple options?

and its accompanying hypothesis:

- H 4: Students taught to program using highly structured languages will perform better on code-completion problems than students taught to program using less structured languages.

Figure 8-15 displays the pre- and post-test means for the “code-completion” question category.

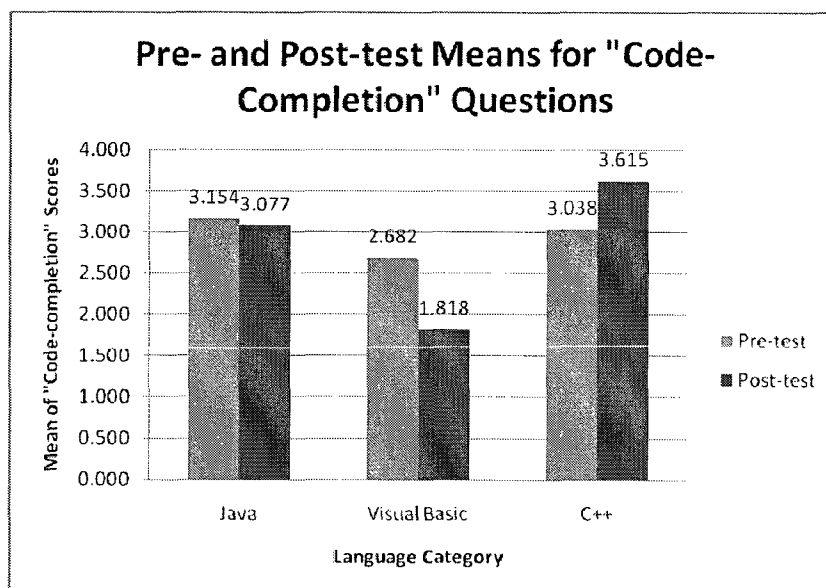


Figure 8-16. Mean “code-completion” scores for pre-test and post-test by language group where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

A one-factor, within-subjects analysis of variance (ANOVA) was performed to determine if language of instruction affected student performance on the “code-completion” questions. The tests of between-subjects effects for the one-factor, within-subjects ANOVA revealed a reliable effect of language category on the repeated measure, “code-completion” score: $F(2, 58) = 3.60, p = 0.03, MS_{error} = 4.07, \alpha = 0.05$. Partial eta squared = 0.11, which is considered a medium effect (Kinnear and Gray, 2010). The Tukey test revealed a significant difference between the Visual Basic mean and the C++ mean at the 0.05 α -level; the difference between the Java mean and the means for Visual Basic and C++ is insignificant. The p -value for the difference between the Visual Basic and C++ means is 0.01; the p -values for the differences between the Java mean and those for Visual Basic and C++ are 0.09 and 0.66, respectively. The difference between the Visual Basic and C++ means is 1.08; the differences between the Java mean and those for Visual Basic and C++ are 0.87 and 0.21, respectively.

Poor performance on “code-completion” questions could be indicative of poor performance on “code-writing” questions. Although the purpose of a tool such as the computer concepts inventory is to provide an *objective* measure of student performance, that does not preclude the use of *subjective* measurement tools when deemed necessary. In order to determine if students can program effectively in their language of study, they must be asked to write (or type) code. The “forgiving” nature of Visual Basic in combination with its emphasis on interface design may detract from student learning of basic programming skills.

CHAPTER 9: DISCUSSION

9.1 Importance of Computer Science Education Research

Numerous authors have written about the difficulties novices experience in learning to program. Theories have been put forth as to why this is so and solutions proposed to remedy the situation. Several decades of research have yielded no definitive answers, and the number of students choosing to go into computer and information science has continued to drop. According to the Computing Research Association (CRA), however, enrollment in American computer science programs during the 2007-2008 academic year increased for the first time in six years (Zweben, 2009). These findings suggest that it would be in the best interests of the computer and information science community to actively pursue research that encourages this trend.

The research presented here was motivated by an interest in improving practices in computer science education in general and improving my own practices as a computer science educator in particular. Its purpose was to develop an instrument to assess student learning of fundamental and object-oriented programming concepts, and to use that instrument to investigate the impact of different languages and teaching approaches on students' ability to learn those concepts.

This discussion proceeds by examining the performance of the assessment instrument itself (Research Question 1), followed by explorations into the findings the instrument yielded (Research Questions 2, 3, and 4). The remaining sections discuss implications of the findings for computer science education and directions for future work.

9.2 Assessment Instrument Performance

- *Research Question 1:* In a field like computer science where there are educational guidelines but no actual standards, is it possible to create a tool to assess student knowledge of fundamental and object-oriented programming (OO) concepts that is sufficiently “generic” to be adopted for general use?

The computer concepts assessment tool performed better in its first trial than I had dared to hope. The tool was field-tested at Drexel and Rowan Universities during the 2009-2010 academic year. Sixty-one students enrolled in a variety of freshman level programming courses participated in the two-part study.

Analysis of the data gathered showed that the assessment tool performed reasonably well at both the pre-test and post-test administrations. Reliability estimates calculated using K-R21 (Kuder and Richardson, 1937) and Cronbach’s Alpha (Ebel and Frisbie, 1991) yielded values ranging from 0.57 (pre-test) to 0.75 (post-test), and 0.65 (pre-test) to 0.79 (post-test), respectively. A reliability coefficient of 0.60 or higher is sufficient for exploratory research; 0.70 or higher is adequate; and 0.80 or higher is good (Garson, 2010). Reliability was estimated separately for the pre-test and post-test because of the diverse backgrounds of the student participants. Analysis of the demographic surveys indicated that some of the students had not been exposed to all of the basic and OO programming concepts before taking the pre-test. By the time they took the post-test, they would have been introduced to all of the concepts.

Reliability is a necessary but not sufficient condition to ensure the validity of an assessment tool. Consequently, evidence was gathered to demonstrate the two types of validity most appropriate for this research, content validity and construct validity.

Different types of evidence provided support for content validity: intrinsic characteristics, study of question responses, expert review. Content validity is intrinsic, since the tool is based on the core concepts of the *Programming Fundamentals* knowledge area defined by the 2008 ACM/IEEE curricular guidelines (CS2008 Review Taskforce, 2008).

Haladyna (2004) stresses the importance of collecting evidence to validate both test scores and the question responses aggregated to form them. Section 8.5.2 analyzed the patterns for correct responses and incorrect responses (distracters) to gather evidence in support of question response validity. Analysis showed that all possible responses, correct as well as incorrect, were selected at least once, the desired result. Next, student records were arranged in descending order by score and divided into quartiles to examine the relationship between answer choice and overall test score. Ideally, the proportion of students choosing correct answers should be directly related to their scores, while the proportion choosing distracters should be inversely related. The desired relationship between correct responses and scores held for 67% of the questions for the pre-test and 75% of the questions for the post-test, which is a respectable result. Between the pre-test and the post-test, at most 50% of the distracters exhibited the desired inverse relationship between incorrect responses and scores, indicating that many of the distracters need to be improved or replaced.

Section 8.5.3 analyzed the overall performance of the computer concepts assessment tool as well as the performance of its individual questions to gather additional evidence in support of question response validity. Analysis showed that the assessment tool performed well overall at both the pre- and post-test administrations based on the

item (question) discrimination values for the individual questions. Specifically, item discrimination ratings were either “very good” or “reasonably good” for 21 out of 24 questions. The 3 questions that performed poorly will have to be replaced or revised.

Feedback from Computer Science (CS) faculty who have reviewed the assessment tool provide additional support for its validity. Several CS faculty whose students participated in the study were asked to complete a review form that asked them to: 1) answer the question; 2) decide whether the question reflects fundamental programming concepts that students should know after completing an introductory course in object-oriented programming; 3) rate the quality of the question on a scale of “do not use again,” “use again with major changes,” “use again with minor changes,” or “use again as is.” Reviewers were encouraged to indicate how questions should be changed or why they should not be used again, if appropriate. Feedback has been largely positive. One reviewer indicated that 19 out of 24 questions reflected basic concepts and should be used again “as is” or with “minor changes.” A second reviewer responded similarly, but specified different questions that should be changed or omitted.

Evidence providing support for construct validity is still being gathered. Exploratory factor analysis is a statistical technique that can be used to determine, based on students’ responses, whether the questions that make up an assessment tool group into constructs the tool is intended to measure (Hoegh and Moskal, 2009). Preliminary factor analysis extracted seven components, three of which were readily comprehensible. These three factors were labeled “methods and functions,” “mathematical and logical expressions,” and “control structures.” A possible fourth factor that overlapped with the “control structures” factor was labeled “questions with conditions.” Such overlap is not

surprising, however, since control structures contain conditions. Ideally, *all* assessment tool questions should group into clearly defined factors that represent the construct “understanding of fundamental programming concepts,” but that is a matter for future work.

9.3 Student Performance

- *Research Question 2:* What effect does the teaching approach and/or programming language have on students’ ability to grasp fundamental programming concepts (e.g., assignment, iteration, functional decomposition, program flow, etc.)?
- *Research Question 3:* What effect does the teaching approach and/or programming language have on students’ ability to grasp specific object-oriented programming concepts (e.g., class creation, method invocation, object instantiation, etc.)?
- *Research Question 4:* What effect does the teaching approach and/or programming language have on students’ ability to select the correct code to complete the solution to a programming problem when provided with multiple options?

When students are given a test such as the computer concepts inventory at the beginning of a course and again at the end of the course, the scores are generally expected to go up. When they do not, one needs to ask why? Overall, the mean post-test score ($M = 10.73$, $s = 4.59$) was slightly lower than the mean pre-test score ($M = 11.40$, $s = 3.63$), but the difference was not significant at the .05 α -level. However, when the students were divided into groups based on the language they were studying in the programming course they were taking, the situation got more interesting.

The students had been recruited from eight different programming classes: two instructing Java, two instructing Visual Basic, and four instructing C++. Students were

divided into a Java group, a Visual Basic group, and a C++ group. Pre-test and post-test means were calculated for each of the language groups and examined graphically. The Java students and the C++ students appeared to have performed similarly on both the pre- and post-tests. The Visual Basic students, however, performed worse than the Java and C++ students on both tests. Statistical analysis was therefore carried out to test whether or not language of instruction affects student performance. The one-factor, within-subjects analysis of variance (ANOVA) revealed a reliable effect of language category on the repeated measure, score: $F(2, 58) = 8.58, p < 0.01, MS_{error} = 23.23, \alpha = 0.05$. Partial eta squared = 0.23, which is considered a large effect (Kinnear and Gray, 2010). These results suggest that language of instruction does indeed affect overall student performance in introductory programming courses.

To explore the impact of language of instruction on student performance with respect to specific programming concepts and code-completion problems, questions were grouped into categories (basics, logical expressions, conditionals, code-completion, and so forth). Individual students' scores were computed for each category for the pre-test and the post-test. The mean scores for each category were calculated for the pre- and post-tests. One-factor, within-subjects analysis of variance (ANOVA) tests were then performed to determine if language of instruction affected student performance on questions addressing specific programming concepts and code-completion.

Statistical analyses revealed an effect of language of instruction on student performance with respect to questions involving basics, logical expressions, and code-completion. Specifically, the one-factor, within-subjects ANOVAs revealed a significant effect of language of instruction on the following repeated measures:

- “basics” score: $F(2, 58) = 16.75, p < 0.01, MS_{error} = 0.72, \alpha = 0.05$
- “logical expressions” score: $F(2, 58) = 5.31, p = 0.01, MS_{error} = 1.66, \alpha = 0.05$
- “code-completion” score: $F(2, 58) = 3.60, p = 0.03, MS_{error} = 4.07, \alpha = 0.05$

For “basics” and “logical expressions,” the Tukey test revealed significant differences between the Visual Basic mean and the means for Java and C++ at the 0.05 α -level; the difference between the Java and C++ means is insignificant. For “code-completion,” the Tukey test revealed a significant difference between the Visual Basic mean and the C++ mean at the 0.05 α -level; the difference between the Java mean and the means for Visual Basic and C++ is insignificant.

Language of instruction did not appear to affect student performance on questions addressing object-oriented concepts. However, differences were noted *within* language groups. Within-subjects paired samples *t*-tests showed significant differences between the pre- and post-test “classes and objects” means for the C++ and Visual Basic students. For the C++ students, a paired samples *t*-test revealed a statistically reliable difference between the mean pre-test ($M = 0.92, s = 1.13$) and post-test ($M = 1.39, s = 1.30$) “classes and objects” scores: $t(25) = -2.74, p = 0.01, \alpha = 0.05$. For the Visual Basic students, a paired samples *t*-test revealed a statistically reliable difference between the mean pre-test ($M = 1.32, s = 1.17$) and post-test ($M = 0.55, s = 0.60$) “classes and objects” scores: $t(21) = 2.94, p = 0.01, \alpha = 0.05$.

Overall student performance with respect to programming language can be summarized as follows:

- The performance of the Java students was consistent.
- The performance of the C++ students improved.
- The performance of the Visual Basic students declined.

The first two scenarios are acceptable; the third is not.

The performance of the Visual Basic (VB) students was consistently poorer than that of the Java and C++ students and declined overall. These observations lead to the following questions:

- 1) Why was the performance of the VB students poorer than that of the Java and C++ students?

Several possible reasons present themselves. It may be that highly structured languages like Java and C++ promote better overall learning than less structured languages like Visual Basic. For example, Visual Basic allows programmers to use variables without first declaring them. Visual Basic also allows certain types of methods to be invoked using more than one syntactic expression. Neither of these practices is allowed in C++ or Java.

It may also be that the interface design aspect of Visual Basic programming detracts from student learning of fundamental concepts. I have learned through personal experience teaching introductory programming using Visual Basic that students seem to like the graphical aspect. However, it may be that the time spent learning to create graphical user interfaces (GUIs) would be better spent learning the fundamental programming constructs needed to implement them.

2) Why did the performance of the VB students decline?

One possible explanation for this result has to do with the approach used to teach the course. The prerequisite course introduced the students to Visual Basic programming using the typical lecture/lab approach in which they were assessed at regular intervals to reinforce their learning. The follow-up course – the course the students were taking when they participated in this study – used a project-based approach. According to the course syllabus, 90% of the student's grade was based on the design and implementation of a small-scale enterprise system. The project was intended to cover all basic and object-oriented programming concepts, but emphasized interface design and event-driven programming. It may be that a tenuous grasp of fundamental concepts in combination with the emphasis on designing interfaces and handling events caused the students to forget what they had learned in the previous course. It may also be that a semester-long project was less effective at reinforcing student learning of fundamental concepts than exams, quizzes, and lab assignments given regularly throughout the course.

3) Why is this result a reason for concern in CS education?

As discussed in Section 9.4, students who take introductory programming courses taught in Visual Basic may very well go on to take additional programming courses taught in other languages. The results of this research suggest that these students may be at a disadvantage when they find themselves in programming classes with students who have learned to program in languages such as C++ or Java. A firm grasp of basic concepts and skills is necessary to succeed in any programming course, but particularly

so in courses that use complex, highly structured object-oriented languages like C++ and Java.

9.4 Implications for Computer Science Education

I recently asked a Rowan colleague why we (meaning the Computer Science Department) switched from teaching Scheme to teaching Visual Basic in the Introduction to Programming course back in 1996. He gave two reasons: 1) Because the Introduction to Programming course was a “terminal” programming course for many students, the faculty thought it was better to introduce them to programming using a RAD (Rapid Application Development) tool such as Visual Basic; 2) Microsoft Office applications support programming with Visual Basic for Applications (VBA), a programming language derived from classic Visual Basic. However, many students who take the Introduction to Programming course and other introductory programming courses taught in Visual Basic (Introduction to Scientific Programming, Enterprise Computing I) go on to take additional programming courses in languages other than Visual Basic. The results of this research suggest that these students may be at a disadvantage when they find themselves in programming classes with students who have learned to program in languages such as C++ or Java.

In general, the results of this research suggest that diverse perspectives must be considered when choosing languages in which to introduce students to programming. A language that may seem to be the ideal choice when viewed from one perspective may turn out to be the worst possible choice when viewed from another perspective. Programming classes that students may take in the future should be considered when selecting introductory languages.

9.5 Future Work

Future work will entail improving the computer concepts assessment tool and continuing my investigations with regard to the impact of different languages and teaching approaches on student learning of introductory programming concepts. In addition, analysis of the computer concepts survey data and the attitude survey data will continue.

Although the computer concepts assessment tool performed reasonably well at its first trial, the discussion in Section 9.2 suggests areas for improvement. First, the three questions that performed poorly must either be revised or replaced with different questions addressing similar topics. Second, about half of the distracters did not exhibit the desired inverse relationship between percentage chosen and student score. As a result, they must be modified or replaced with different distracters. Another problem in conjunction with a correct response and an incorrect response in one of the “classes and objects” questions, Question 24, was identified by one of the faculty reviewers. He noted that although the pseudocode for the two responses was syntactically different, either the correct or the incorrect response would work in Visual Basic because of the language’s “forgiving” nature. (Only the correct response would work in C++ or Java.) The improperly performing distracter will have to be replaced with one that is incorrect in *all* languages. Finally, work must continue to fully validate the tool after the indicated changes have been made.

Because the students who participated in this study were not randomly chosen, it may have been more appropriate to analyze the computer concepts survey data using analysis of covariance (ANCOVA) instead of analysis of variance (ANOVA). The

ANCOVA can be used to control for factors which cannot be randomized. Specifically, it reduces the error term, thereby increasing the power of the F test (Garson, 2010; Kinnear and Gray, 2010).

The original goal of this research was to explore the impact of different languages and teaching approaches on student learning of introductory programming concepts. In retrospect, it was an overly ambitious goal. The backgrounds of the students I was able to recruit allowed me to investigate the impact of different programming languages on student learning of introductory concepts, but only to conjecture about the impact of different teaching approaches. Investigating the impact of different teaching approaches most likely requires the involvement of multiple instructors at multiple institutions. For example, members of the Rowan Computer Science faculty introduce students to Java programming using BlueJ, a novice programming environment that supports the “objects-first” approach. Comparing the impact of “objects-first” using BlueJ to the impact of “imperative-first” using jGRASP (for example) on student learning of Java concepts would necessitate involving instructors at universities who teach Java using the “imperative-first” approach. In order to properly investigate the impact of different teaching approaches, the programming language must be the same. Apart from Java, all other programming courses at Rowan are taught using the “imperative-first” approach (as far as I know, anyway).

Finally, analysis of the attitude survey data is still in progress. I was able to obtain some sense of the students’ overall responses during the process of entering the data into the computer. However, that is hardly sufficient. Properly exploring the relationship between students’ attitudes toward computer science and programming and

their performance on the computer concepts survey requires statistical analysis. I am particularly interested in examining the correlation between students' responses to the questions addressing the usefulness of programming in their future work and their scores on the concepts survey.

LIST OF REFERENCES

1. Aberson, C. L. (2010). *Applied Power analysis for the Behavioral Sciences*. New York, NY: Routledge.
2. Adams, J. C. (2007). Alice, middle schoolers & the imaginary worlds camps. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 307-311.
3. Agarwal, K. K., and Agarwal, A. (2005). Python for CS1, CS2 and beyond. *Journal of Computing Sciences in Colleges*, 20(4), 262-270.
4. Agarwal, K. K., and Agarwal, A. (2006). Simply Python for CS0. *Journal of Computing Sciences in Colleges*, 21(4), 162-170.
5. Algebra End-of-Course Assessment. (2009). Educational Testing Service (ETS). Retrieved 29 December, 2009, from <http://www.ets.org/portal/site/ets/menuitem.435c0b5cc7bd0ae7015d9510c3921509/?vgnextoid=4d7de3b5f64f4010VgnVCM10000022f95190RCRD>.
6. Alice v3.0. (2010). Pittsburgh: Carnegie Mellon University. Retrieved 13 August, 2010, from <http://www.alice.org/>.
7. Allen, E., Cartwright, R., and Stoler, B. (2002). DrJava: A lightweight pedagogic environment for Java. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, New York: ACM Press, 137-141.
8. Allen, K. (2007). Concept inventory central. Retrieved 29 December, 2009, from <https://engineering.purdue.edu/SCI/workshop/tools.html>.
9. American Educational Research Association, American Psychological Association & National Council on Measurement in Education (1999). *Standards for Educational and Psychological Testing*. Washington, DC: American Educational Research Association.
10. Anderson, J. R. (1983). *The Architecture of Cognition*. Cambridge, MA: Harvard University Press.
11. Ausubel, D. P. (1968). *Educational Psychology: A Cognitive View*. New York: Holt, Rinehart, and Winston.
12. Ausubel, D. P. (1960). The use of advance organizers in the learning and retention of meaningful verbal material. *Journal of Educational Psychology*, 51(5), 267-272.

13. Babbie, E. (2004). *The Practice of Social Research*. Belmont, CA: Wadsworth/Thomson Learning.
14. Bailie, F., Blank, G., Murray, K., and Rajaravivarma, R. (2002). Java visualization using BlueJ. *Journal of Computing Sciences in Colleges*, 18(3), 175-176.
15. Barnes, D. J., and Kölling, M. (2005). *Objects First With Java: A Practical Introduction Using BlueJ*. London: Pearson Prentice Hall.
16. Barnett, S. M., and Ceci, S. J. (2002). When and where do we apply what we learn? A taxonomy for far transfer. *Psychological Bulletin*, 128(4), 612-637.
17. Ben-Ari, M. (1998). Constructivism in computer science education. In *SIGCSE '98: Proceedings of the 29th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 257-261.
18. BlueJ. Retrieved 14 December, 2005, from <http://www.bluej.org/>.
19. Bonar, J., and Soloway, E. (1989). Preprogramming knowledge: A major source of misconceptions in novice programmers. In E. Soloway and J. Spohrer (Eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, 325-353.
20. Bonar, J., and Soloway, E. (1983). Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York: ACM Press, 10-13.
21. Borland JBuilder 2006. (2006). Retrieved 31 March, 2006, from <http://www.borland.com/us/products/jbuilder/index.html>.
22. Bransford, J. D., Brown, A. L., and Cocking, R. R. (Eds.). (1999). *How People Learn: Brain, Mind, Experience, and School*. Washington, DC: National Academy Press.
23. Clancy, M. (2004). Misconceptions and attitudes that interfere with learning to program. In S. Fincher and M. Petre (Eds.), *Computer Science Education Research*. London, UK: Taylor and Francis Group, 85-100.
24. Cobb, P., and Steffe, L. P. (1983). The constructivist researcher as teacher and model builder. *Journal for Research in Mathematics Education*, 14(2), 83-94.
25. Cobb, P., Wood, T., Yackel, E., Nicholls, J., Wheatley, G., Trigatti, B., and Perlwitz, M. (1991). Assessment of a problem-centered second-grade mathematics project. *Journal for Research in Mathematics Education*, 22(1), 3-29.
26. CodeLab. (2004). Retrieved 18 April, 2007, from <http://turingscraft.com/>.

27. Computer Science Teachers Association (CSTA). (2005). Retrieved 5 November, 2009 from <http://www.csta.acm.org/>.
28. Concept Inventories for Computer Science. Retrieved 12 January, 2010, from <http://www-sal.cs.uiuc.edu/~zilles/csci.html>.
29. Constructivism as a Paradigm for Teaching and Learning. (2004). Educational Broadcasting Corporation. Retrieved 14 May, 2007, from <http://www.thirteen.org/edonline/concept2class/constructivism/index.html>.
30. Conway, M., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K., Deline, R., et al. (2000). Alice: Lessons learned from building a 3D system for novices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, New York: ACM Press, 486-493.
31. Cooper, S., Dann, W., and Pausch, R. (2000). Alice: A 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5), 107-116.
32. Cooper, S., Dann, W., and Pausch, R. (2003a). Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 191-195.
33. Cooper, S., Dann, W., and Pausch, R. (2003b). Using animated 3D graphics to prepare novices for CS1. *Computer Science Education*, 13(1), 3-30.
34. CS2008 Review Taskforce. (2008). Computer Science Curriculum 2008: An Interim Revision of CS 2001. Association for Computing Machinery (ACM)/IEEE Computer Society. Retrieved 6 April, 2010, from <http://www.acm.org/education/curricula/ComputerScience2008.pdf>.
35. Dann, W., Cooper, S., and Pausch, R. (2006). *Learning to Program with Alice*. Upper Saddle River: Pearson Prentice Hall.
36. Dann, W., Cooper, S., and Pausch, R. (2000). Making the connection: Programming with animated small world. *ACM SIGCSE Bulletin*, 32(3), 41-44.
37. Dann, W., Cooper, S., and Pausch, R. (2003). Objects: Visualization of behavior and state. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, New York: ACM Press, 84-88.
38. DeAyala, R. J., Plake, B. S., and Impara, J. C. (2001). The impact of omitted responses on the accuracy of ability estimation in item response theory. *Journal of Educational Measurement*, 38(3), 213-234.
39. Dougherty, J. P. (2007). Concept visualization in CS0 using Alice. *Journal of Computing Sciences in Colleges*, 22(3), 145-152.

40. DrJava. Retrieved 31 March, 2006, from <http://drjava.org/>.
41. DuBoulay, B. (1989). Some difficulties of learning to program. In E. Soloway and J. Spohrer (Eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, 283-299.
42. Ebel, R. L., and Frisbie, D. A. (1991). *Essentials of Educational Measurement*. Englewood Cliffs, NJ: Prentice-Hall.
43. Eclipse SDK 3.1.2. (2006). Retrieved 31 March, 2006, from <http://www.eclipse.org/>.
44. Elvers, G. C. (2006). Using SPSS for One Way Analysis of Variance. Retrieved 27 July, 2010, from <http://academic.udayton.edu/gregelvers/psy216/SPSS/1wayanova.htm>.
45. Elvin, C. (2004). Test Item Analysis Using Microsoft Excel Spreadsheet Program. Retrieved 30 March, 2010, from <http://www.eflclub.com/elvin/publications/2003/itemanalysis.html>.
46. Evans, D. L., Gray, G. L., Krause, S., Martin, J., Midkiff, C., Notaros, B. M., et al. (2003). Progress on concept inventory assessment tools. In *Proceedings of the 33rd ASEE/IEEE Frontiers in Education Conference*, New York: IEEE, T4G1-T4G8.
47. Even, R. (2005). Using assessment to inform instructional decisions: How hard can it be? *Mathematics Education Research Journal*, 17(3), 45-61.
48. Fincher, S., and Petre, M. (Eds.). (2004). *Computer Science Education Research*. London, UK: Taylor and Francis Group.
49. Fleury, A. (2000). Programming in Java: Student-constructed rules. *ACM SIGCSE Bulletin*, 32(1), 197-201.
50. Garson, G. D. (2010). Reliability Analysis, from *Statnotes: Topics in Multivariate Analysis*. Retrieved 20 July, 2010, from <http://faculty.chass.ncsu.edu/garson/pa765/statnote.htm>.
51. Gersting, J. L. (2007). *Mathematical Structures for Computer Science*. New York, NY: W. H. Freeman and Company.
52. Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., and Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a Delphi process. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on computer science education*, New York: ACM Press.

53. Grandell, L., Peltomäki, M., Back R., and Salakoski, T. (2006). Why complicate things?: Introducing programming in high school using Python. In *Proceedings of the 8th Australian conference on Computing education*, Darlinghurst, Australia: Australian Computer Society, Inc., 71-80.
54. Gray, K. E., and Flatt, M. (2003). ProfessorJ: a gradual introduction to Java through language levels. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York: ACM Press, 170-177.
55. Greeno, J. G. (2006). Authoritative, accountable positioning and connected, general knowing: Progressive themes in understanding transfer. *The Journal of the Learning Sciences*, 15(4), 537-547.
56. Gross, P., and Powers, K. (2005). Evaluating assessments of novice programming environments. In *Proceedings of the 2005 international workshop on Computing education research*, New York: ACM Press, 99-110.
57. Hadjerrouit, S. (1999). A constructivist approach to object-oriented design and programming. In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, New York: ACM Press, 171-174.
58. Haladyna, T. M. (2004). *Developing and Validating Multiple-Choice Test Items*. Mahwah, NJ: Lawrence Erlbaum Associates.
59. Hartley, S. J. (2009). Introduction to scientific programming syllabus. Retrieved 15 November, 2009 from <http://elvis.rowan.edu/~hartley/Courses/IntroSciProg/2009/Fall/syllabus.html>.
60. Hendrix, T. D., Cross, II, J. H., and Barowski, L. A. (2004). An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 387-391.
61. Hergenhahn, B. R., and Olson, M. H. (2001). *An Introduction to Theories of Learning*. Upper Saddle River: Pearson Prentice Hall.
62. Herman, G. L., Loui, M. C., and Zilles, C. (2010). Creating the digital logic concept inventory. In *SIGCSE '10: Proceedings of the 41st SIGCSE technical symposium on computer science education*, New York: ACM Press.
63. Hestenes, D., and Halloun, I. (1995). Interpreting the Force Concept Inventory. *The Physics Teacher*, 33(8), 502-506.

64. Hestenes, D., Wells, M., and Swackhamer, G. (1992). Force Concept Inventory. *The Physics Teacher*, 30(3), 141-151.
65. Hoegh, A., and Moskal, B. (2009). Examining science and engineering students' attitudes toward computer science. In *Proceedings of the 39th ASEE/IEEE Frontiers in Education Conference*, New York: IEEE, W1G-1-W1G-6.
66. Hsia, J. I., Simpson, E., Smith, D., and Cartwright, R. (2005). Taming Java for the classroom. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 327-331.
67. Hundhausen, C. D., Farley, S., and Brown, J. L. (In press). Can direct manipulation lower the barriers to programming and promote positive transfer to textual programming? An experimental study. To appear in *Proceedings of the IEEE 2006 Symposium on Visual Languages and Human-Centric Computing*.
68. International Technology Education Association (ITEA). (2007). *Standards for Technological Literacy: Content for the Study of Technology*. Reston, VA: ITEA.
69. Jackson, S. L., Krajcik, J., and Soloway, E. (1998). The design of guided learner-adaptable scaffolding in interactive learning environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, New York: ACM Press, 187-194.
70. Java Platform, Standard Edition (J2SE). (2005). Santa Clara: Sun Microsystems, Inc. Retrieved 25 December, 2005, from <http://java.sun.com/j2se/downloads/index.html>.
71. JavaScript. (2007). Wikipedia, the Free Encyclopedia. Retrieved 1 May, 2007, from <http://en.wikipedia.org/wiki/JavaScript>.
72. Jeliot 3. Retrieved 31 March, 2006, from <http://cs.joensuu.fi/jeliot/index.php>.
73. jGRASP 1.8.4. (2006). Retrieved 20 October, 2006, from <http://jgrasp.org/>.
74. Joint Task Force on Computing Curricula. (2001). Computing Curricula 2001 Computer Science. IEEE Computer Society/Association for Computing Machinery. Retrieved 7 December, 2009, from http://www.computer.org/portal/cms_docs_ieeeecs/ieeeecs/education/cc2001/cc2001.pdf.
75. JUnit. (2004). Retrieved 9 April, 2006 from <http://www.junit.org/index.htm>.
76. Kaczmarczyk, L. C., Petrick, E. R., East, J., P., and Herman, G. L. (2010). Identifying student misconceptions of programming. In *SIGCSE '10: Proceedings of the 41st SIGCSE technical symposium on computer science education*, New York: ACM Press.

77. Kelleher, C., and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2), 83-137.
78. Kelleher, C., and Pausch, R. (2005). Stencils-based tutorials: Design and evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, New York: ACM Press, 541-550.
79. Kelley, L. (2002). Course Embedded Assessment Process. California Assessment Institute. Retrieved 20 May, 2007, from <http://cai.cc.ca.us/Resources/>.
80. Kessler, C. M., and Anderson, J. R. (1989). Learning flow of control: Recursive and iterative procedures. In E. Soloway and J. Spohrer (Eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, 229-260.
81. Kinnear, P. R., and Gray, C. D. (2010). *PASW Statistics 17 Made Simple*. New York, NY: Psychology Press.
82. Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Computer Science Education*, 13(4), 249-268.
83. Kuder, G. F., and Richardson, M. W. (1937). The theory of the estimation of test reliability. *Psychometrika*, 2(3), 151-160.
84. Kurland, D. M., and Pea, R. D. (1989). Children's mental models of recursive Logo programs. In E. Soloway and J. Spohrer (Eds.), *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, 315-323.
85. Kyd, C. (2006a). An Excel Tutorial: An Introduction to Excel's Normal Distribution Functions. ExcelUser, Inc. Retrieved 21 July, 2010, from <http://www.exceluser.com/explore/statsnormal.htm>.
86. Kyd, C. (2006b). An Excel Tutorial: How to Create Normal Curves in Excel, with Shaded Areas. ExcelUser, Inc. Retrieved 21 July, 2010, from <http://www.exceluser.com/explore/normalcurve.htm>.
87. Lobato, J. (2006). Alternative perspectives on the transfer of learning: History, issues, and challenges for future research. *The Journal of the Learning Sciences*, 15(4), 431-449.
88. Lobato, J. (2003). How design experiments can inform a rethinking of transfer and vice versa. *Educational Researcher*, 32(1), 17-20.

89. Major Field Test for Computer Science. (2009). Educational Testing Service (ETS). Retrieved 29 December, 2009, from <http://www.ets.org/portal/site/ets/menuitem.1488512ecfd5b8849a77b13bc3921509/?vgnextoid=fd29af5e44df4010VgnVCM10000022f95190RCRD&vgnextchannel=eddc144e50bd2110VgnVCM10000022f95190RCRD>.
90. Mayer, R. E. (1979). Can advance organizers influence meaningful learning? *Review of Educational Research*, 49(2), 371-383.
91. Mayer, R. E. (1975). Different problem-solving competencies established in learning computer programming with and without meaningful models. *Journal of Educational Psychology*, 67(6), 725-734.
92. Mayer, R. E. (1975). Information processing variables in learning to solve problems. *Review of Educational Research*, 45(4), 525-541.
93. Mayer, R. E. (1989). The psychology of how novices learn computer programming. In E. Soloway and J. Spohrer (Eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, 129-159.
94. Mayer, R. E. (1976). Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order. *Journal of Educational Psychology*, 68(2), 143-150.
95. Mayer, R. E., and Bromage, B. K. (1980). Different recall protocols for technical texts due to advance organizers. *Journal of Educational Psychology*, 72(2), 209-225.
96. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4), 125-140.
97. McIver, L. (2002). Evaluating languages and environments for novice programmers. In J. Kuljis, L. Baldwin, and R. Scoble (Eds.), *Proceedings of the 14th Annual Workshop of the Psychology of Programming Interest Group*, Brunel University, London, UK, 100-110.
98. Medlock, A. A. (2008). CS 130: Computer programming concepts with 3-D animation. Retrieved 15 November, 2009 from <http://www.cs.drexel.edu/~aalban/CS131/>.
99. Mercer, R. (1995). *Computing Fundamentals with C++*. Wilsonville, OR: Franklin, Beedle and Associates.
100. *Merriam-Webster's Collegiate Dictionary (11th ed.)*. (2003). Springfield, MA: Merriam-Webster, Inc.

101. Microsoft Visual J++ 6.0. (2006). Retrieved 31 March, 2006, from <http://msdn.microsoft.com/vjsharp/productinfo/visualj/visualj6/>.
102. Microsoft Visual Studio 2010. (2010). Retrieved 19 August, 2010, from <http://www.microsoft.com/visualstudio/en-us>.
103. Mitchell, J. W., and Martin, J. K. (2007). Use of thermodynamics, fluid mechanics and heat transfer concept inventories to assess student learning in core courses. Retrieved 29 December, 2009 from <https://engineering.purdue.edu/SCI/workshop/summaries/Fluids%20Heat%20Thermo%20Long%20Summary.doc>.
104. Moreno, A., and Myller, N. (2003). Producing an educationally effective and usable tool for learning, the case of the Jeliot family. In *Proceedings of the International Conference on Networked e-learning for European Universities*, Granada, Spain.
105. Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, New York: ACM Press, 373-376.
106. Moskal, B., Lurie, D., and Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 75-79.
107. National Educational Technology Standards (NETS•S) and Performance Indicators for Students. (2007). International Society for Technology in Education (ISTE). Retrieved 11 August, 2010, from http://www.iste.org/Content/NavigationMenu/NETS/ForStudents/2007Standards/NETS_for_Students_2007_Standards.pdf.
108. NetBeans IDE 5.0. Retrieved 22 July, 2006, from <http://www.netbeans.org/>.
109. Nourie, D. (2002). Teaching Java technology with BlueJ. Santa Clara: Sun Microsystems, Inc. Retrieved 23 December, 2005, from <http://java.sun.com/features/2002/07/bluej.html>.
110. Nugent, G., Soh, L., Samal, A., and Lang, J. (2006). A placement test for computer science: Design, implementation, and analysis. *Computer Science Education*, 16(1), 19-36.
111. Olan, M. (2004). Dr. J vs. the bird: Java IDE's one-on-one. *Journal of Computing Sciences in Colleges*, 19(5), 44-52.
112. Oldham, J. D. (2005). What happens after Python in CS1? *Journal of Computing Sciences in Colleges*, 20(6), 7-13.

113. Packer, M. (2001). The problem of transfer, and the sociocultural critique of schooling. *The Journal of the Learning Sciences*, 10(4), 493-514.
114. Papert, S. (1993). *Mindstorms: Children, Computers, and Powerful Ideas*. Cambridge, MA: Perseus Publishing.
115. PASW Statistics 18. (2010). Retrieved 22 July, 2010, from <http://www.spss.com/>.
116. Patterson, A., Kölling, M., and Rosenberg, J. (2003). Introducing unit testing with BlueJ. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, New York: ACM Press, 11-15.
117. Pattis, R. (1981). *Karel the Robot*. New York: John Wiley and Sons.
118. Pea, R. D., Soloway, E., and Spohrer, J. (1987). The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics*, 9(1), 5-30.
119. Perkins, D. N., and Salomon, G. The science and art of transfer. Retrieved 4 February, 2007, from <http://learnweb.harvard.edu/alps/thinking/docs/trancost.htm>.
120. Pirolli, P., and Recker, M. (1994). Learning strategies and transfer in the domain of programming. *Cognition and Instruction*, 12(3), 235-275.
121. Polson, P. G., Bovair, S., and Kieras, D. (1987). Transfer between text editors. In *CHI and GI 1987 Conference Proceedings: Human Factors in Computing Systems and Graphics Interface*, New York, NY: ACM Press, 27-32.
122. Polson, P. G., and Kieras, D. E. (1985). A quantitative model of the learning and performance of text-editing knowledge. In *CHI 1985 Conference Proceedings: Human Factors in Computing Systems and Graphics Interface*, New York, NY: ACM Press, 207-212.
123. Polson, P. G., Muncher, E., and Engelbeck, G. (1986). A test of a common elements theory of transfer. In *CHI 1986 Conference Proceedings: Human Factors in Computing Systems and Graphics Interface*, New York, NY: ACM Press, 78-83.
124. Popyack, J. L. (2009). CS 171/SE 102: Computer Programming I. Retrieved 15 November, 2009 from <http://www.cs.drexel.edu/~mcs171/Wi09>.
125. Powers, K., Ecott, S., and Hirshfield, L. M. (2007). Through the looking glass: Teaching CS0 with Alice. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 213-217.

126. Preece, J., Rogers, Y., and Sharp, H. (2002). *Interaction Design: Beyond Human-Computer Interaction*. New York: John Wiley & Sons.
127. ProfessorJ. Retrieved 31 March, 2006, from <http://www.cs.utah.edu/~kathyg/profj/>.
128. Provine, D. (2009). Introduction to programming syllabus. Retrieved 15 November, 2009 from <http://elvis.rowan.edu/~kilroy/class/intro/?syllabus>.
129. Puntambekar, S., and Hübscher, R. (2005). Tools for scaffolding students in a complex learning environment: What have we gained and what have we missed? *Educational Psychologist*, 40(1), 1-12.
130. Putnam, R. T., Sleeman, D., Baxter, J. A., and Kuspa, L. K. (1989). A summary of misconceptions of high-school BASIC programmers. In E. Soloway and J. Spohrer (Eds.), *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, 301-314.
131. Python. (2007). Python Software Foundation. Retrieved 22 April, 2007, from <http://python.org/>.
132. Radenski, A. (2006). "Python first": A lab-based digital introduction to computer science. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, New York: ACM Press, 197-201.
133. Ragonis, N., and Ben-Ari, M. (2005a). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, 15(3), 203-221.
134. Ragonis, N., and Ben-Ari, M. (2005b). On understanding the statics and dynamics of object-oriented programs. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 226-230.
135. Reis, C., and Cartwright, R. (2004). Taming a professional IDE for the classroom. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 156-160.
136. Rist, R. S. (1986). Plans in programming: Definition, demonstration and development. In E. Soloway and R. Iyengar (Eds.), *Empirical Studies of Programmers*. Norwood, NJ: Ablex Publishing Corporation, 28-47.
137. Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, 13, 389-414.
138. Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137-172.

139. Savitch, W. (2005). *Java: An Introduction to Problem Solving & Programming*. Upper Saddle River: Pearson Prentice Hall.
140. Savitch, W. (2003). *Problem Solving with C++: The Object of Programming*. Boston, MA: Addison-Wesley.
141. Scholtz, J., and Wiedenbeck, S. (1993). An analysis of novice programmers learning a second language. In C. R. Cook, J. C. Scholtz, and J. C. Spohrer (Eds.), *Empirical Studies of Programmers: Fifth Workshop*. Norwood, NJ: Ablex Publishing Corporation, 187-205.
142. Scholtz, J., and Wiedenbeck, S. (1990a). Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction*, 2(1), 51-72.
143. Scholtz, J., and Wiedenbeck, S. (1990b). Learning to program in another language. In *Proceedings of INTERACT '90*, Amsterdam: Elsevier Science, 925-930.
144. Sebesta, R. W. (2002). *Concepts of Programming Languages*. Boston, MA: Addison-Wesley.
145. Shannon, C. (2003). Another breadth-first approach to CS I using Python. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, New York: ACM Press, 248-251.
146. Singley, M. K., and Anderson, J. R. (1988). A keystroke analysis of learning and transfer in text editing. *Human-Computer Interaction*, 3(3), 223-274.
147. Singley, M. K., and Anderson, J. R. (1989). *The Transfer of Cognitive Skill*. Cambridge, MA: Harvard University Press.
148. Singley, M. K., and Anderson, J. R. (1985). The transfer of text-editing skill. *International Journal of Man-Machine Studies*, 22, 403-423.
149. Sivula, K. (2005). *A qualitative case study on the use of Jeliot 3*. Unpublished master's thesis, University of Joensuu, Joensuu, Finland.
150. Soloway, E., Bonar, J., and Ehrlich, K. (1989). Cognitive strategies and looping constructs: An empirical study. In E. Soloway and J. Spohrer (Eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 191-207.
151. Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. (1984). What do novices know about programming? In A. Badre and B. Shneiderman (Eds.), *Directions in Human-Computer Interaction*. Norwood, NJ: Ablex Publishing Corporation, pp. 27-54.

152. Soloway, E., and Spohrer, J. C. (Eds.). (1989). *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates.
153. Spohrer, J. C., Soloway, E., and Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. In E. Soloway and J. Spohrer (Eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, 355-399.
154. Spohrer, J. C., and Soloway, E. (1989). Novice mistakes: Are the folk wisdoms correct? In E. Soloway and J. Spohrer (Eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, 401-416.
155. Stamouli, I., and Huggard, M. (2006). Object oriented programming and program correctness: The students' perspective. In *Proceedings of the 2006 international workshop on Computing education research*, New York: ACM Press, 109-118.
156. Tadepalli, P., and Cunningham, H. C. (2004). JavaCHIME: Java class hierarchy inspector and method executer. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, New York: ACM Press, 152-157.
157. Test Link. (2009). Educational Testing Service (ETS). Retrieved 29 December, 2009, from <http://www.ets.org/portal/site/ets/menuitem.1488512ecfd5b8849a77b13bc3921509/?vgnextoid=ed462d3631df4010VgnVCM10000022f95190RCRD&vgnextchannel=85af197a484f4010VgnVCM10000022f95190RCRD>.
158. Tew, A. E., and Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. In *SIGCSE '10: Proceedings of the 41st SIGCSE technical symposium on computer science education*, New York: ACM Press.
159. Thorndike, E. L. (1906). *Principles of Teaching*. New York, NY: A. G. Seiler.
160. Van Haaster, K. (2003). *Introductory programming in an OO environment: An evaluation of a visual tool*. Unpublished honors thesis, Monash University, Caulfield East, Victoria 3145, Australia.
161. Van Haaster, K., and Hagan, D. (2004). Teaching and learning with BlueJ: An evaluation of a pedagogical tool. In *Proceedings of the Information Science and Information Technology Education Joint Conference*, Rockhampton, QLD, Australia.
162. Von Glasersfeld, E. (1990). An exposition of Constructivism: Why some like it radical. In R. B. Davis, C. A. Maher, and N. Noddings (Eds.), *Monographs of the Journal for Research in Mathematics Education*. Reston, VA: National Council of Teachers of Mathematics, pp. 19-29.

163. Wiedenbeck, S., Ramalingam, V., Sarasamma, S., and Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11, 255-282.
164. Wiedenbeck, S., and Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International journal of Human-Computer Studies*, 51, 71-87.
165. Wood, D., Bruner, J. S., and Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry and Allied Disciplines*, 17(2), 89-100.
166. Wu, Q., and Anderson, J. R. (1991). Knowledge transfer among programming languages. In *Proceedings of the 13th Conference of the Cognitive Science Society*, Hillsdale, NJ: Lawrence Erlbaum Associates, 376-381.
167. Wu, Q., and Anderson, J. R. (1990). Problem-solving transfer among programming languages. For submission to: *Human-Computer Interaction*.
168. Zweben, S. (2009). Computing degree and enrollment trends. Computing Research Association. Retrieved 5 December, 2009, from http://www.cra.org/govaffairs/blog/archives/CRA_TaulbeeReport-StudentEnrollment-07-08.pdf.

APPENDIX A: DEMOGRAPHIC SURVEY DATA FOR PILOT STUDY

Subject		Demographic Survey		Programming Language Experience															
ID Number	Level	Academic Code	Major	SAT Math Score	Gender	Alice													
						Basic	C	C++	Java	JavaScript	Maple	Pascal	PHP	Python	Ruby	Visual Basic			
2495	4	Junior	Physics	740	Male	2													
2496	3	Pre-junior	Physics		Female														
2497	4	Junior	Math	740	Female	3													
			Software																
2498	1	Freshman	Engineering	590	Male														
2499	2	Sophomore	Digital Media	700	Male	3													
2500	1	Freshman	Physics	780	Male														
2501	1	Freshman	Physics	690	Female														
			Computer																
2502	1	Freshman	Science	720	Male														
			Information																
2503	2	Sophomore	Technology	500	Female														
2504	1	Freshman	Physics	650	Female														
			Software																
2505	1	Freshman	Engineering	680	Male														
			Computer																
2506	1	Freshman	Science		Male														
2507	2	Sophomore	Digital Media	540	Male	3	1												
2508	1	Freshman	Math	800	Male														

Figure A-1. Demographic survey data for pilot study.

APPENDIX B: PRE-TEST RESULTS FOR PILOT STUDY

Subject		Computer Concepts Survey												
ID Number	Scores (by topic area)										Scores (by question type)			
	Basics	Logical	Loops (w/o arrays)	Loops (w/ arrays)	Functions (parameters)	Functions (return values)	Arrays (only)	Objects	Overall	Definition	Trace	Write		
2495	3.000	-0.333	3.000	1.667	2.000	0.000	-0.333	1.000	1.000	11.001	4.001	4.000	3.000	
2496	1.667	0.667	1.667	3.000	2.000	-0.666	-0.333	1.000	-0.333	8.669	2.335	3.334	3.000	
2497	3.000	2.000	1.667	1.667	-0.666	-0.666	-0.333	1.000	-0.333	7.336	5.001	2.001	0.334	
2498	1.667	0.667	0.334	1.667	0.667	0.667	-0.333	1.000	1.667	8.003	4.668	2.001	1.334	
2499	0.334	2.000	3.000	1.667	2	-0.666	-0.333	-0.333	3.000	10.669	3.335	3.334	4.000	
2500	1.667	0.667	0.334	1.667	0.667	0.667	1.000	1.000	3.000	10.669	4.668	4.667	1.334	
2501	2.000	1.000	1.667	1.667	2.000	-0.666	-0.333	1.000	3.000	11.335	7.334	2.334	1.667	
2502	3.000	0.667	1.667	1.667	0.667	0.667	-0.333	1.000	3.000	12.002	7.334	0.668	4.000	
2503	0.667	-0.666	0.334	0.334	0.667	-0.666	-0.333	0.000	-0.999	-0.662	1.002	-0.332	-1.332	
2504	1.667	0.667	0.334	0.334	0.667	-0.333	-0.333	-0.333	1.667	4.337	0.669	2.334	1.334	
2505	0.334	-0.666	1.667	-0.999	-0.666	-0.666	-0.333	-0.333	1.667	0.005	0.669	-0.665	0.001	
2506	3.000	0.667	1.667	1.667	2.000	0.667	-0.333	1.000	3.000	13.335	7.334	2.001	4.000	
2507	0.334	0.667	3.000	1.667	0.667	0.667	-0.333	-0.333	3.000	9.336	3.335	3.334	2.667	
2508	1.667	2.000	3.000	0.667	-0.666	-0.666	-0.333	1.000	1.667	8.336	4.668	2.001	1.667	
Overall Mean	1.715	0.715	1.667	1.310	0.857	-0.119	-0.238	0.548	1.715	8.169	4.025	2.215	1.929	
Alice Mean	1.334	1.667	2.667	1.417	0.334	-0.333	-0.333	0.334	1.834	8.319	4.085	2.668	2.167	
Python Mean	1.534	0.467	0.867	1.400	1.200	-0.333	-0.066	0.533	1.267	6.370	3.202	2.467	1.201	

Primary Language Key	Alice	
	Python	
	Other	

Figure B-1. Pre-test results for pilot study.

APPENDIX C: POST-TEST RESULTS FOR PILOT STUDY

Subject	Computer Concepts Survey														
	Scores (by topic area)					Scores (by question type)									
	Logical	Basics	Operators	If	Arrays	Loops (w/o arrays)	Loops (w/ arrays)	Functions (parameters)	Functions (return values)	Arrays	Objects	Overall	Definition	Trace	Write
2495	3.000	2.000	3.000	3.000	2.000	3.000	2.000	0.667	-0.333	1.000	1.000	15.334	7.667	4.667	3.000
2496	1.667	2.000	0.334	1.667	0.667	1.667	0.667	-0.666	-0.333	1.000	-0.333	6.003	1.002	2.001	3.000
2497	1.667	0.667	3.000	3.000	0.667	3.000	0.667	0.667	-0.333	1.000	-0.333	10.002	6.334	0.668	3.000
2498	1.667	2.000	3.000	0.334	0.667	0.667	0.667	2.000	-0.333	1.000	3.000	13.335	4.668	4.667	4.000
2499	3.000	2.000	3.000	3.000	0.667	0.667	0.667	0.667	1.000	-0.333	1.667	14.668	7.334	3.334	4.000
2500	0.334	0.667	1.667	0.334	2.000	2.000	2.000	-0.666	-0.333	1.000	1.667	6.670	2.002	4.667	0.001
2501	3.000	0.667	3.000	1.667	-0.666	0.667	0.667	0.667	-0.333	1.000	3.000	12.002	8.667	0.668	2.667
2502	1.667	-0.666	1.667	1.667	0.667	0.667	0.667	0.667	-0.333	1.000	3.000	9.336	6.001	2.001	1.334
2503	-0.999	0.667	1.667	-0.999	0.667	0.667	0.667	0.667	-0.333	1.000	1.667	4.004	0.669	3.334	0.001
2504	1.667	0.667	1.667	1.667	0.667	0.667	0.667	-0.666	-0.333	1.000	3.000	9.336	4.668	3.334	1.334
2505	3.000	0.667	0.334	1.667	0.667	0.667	0.667	-0.666	-0.333	1.000	3.000	9.336	6.001	0.668	2.667
2506	1.667	0.667	1.667	3.000	2.000	2.000	2.000	0.667	-0.333	1.000	1.667	12.002	6.001	4.667	1.334
2507	0.334	-0.666	-0.999	1.667	0.667	0.667	0.667	2.000	-0.333	1.000	1.667	5.337	2.002	0.668	2.667
2508	3.000	-0.666	3.000	3.000	-0.666	0.667	0.667	0.667	-0.333	-0.333	1.667	9.336	3.335	2.001	4.000
Overall Mean	1.762	0.762	1.857	1.762	0.762	0.762	0.762	0.477	-0.238	0.810	1.810	9.764	4.739	2.668	2.358
Alice Mean	2.000	0.334	2.000	2.667	0.334	0.334	0.334	1.000	0.000	0.334	1.167	9.836	4.751	1.668	3.417
Python Mean	1.134	0.934	1.667	0.857	0.667	0.667	0.667	-0.133	-0.333	1.000	1.800	7.603	3.402	2.801	1.401

Primary Language Key	Alice
	Python
	Other

Figure C-1. Post-test results for pilot study.

APPENDIX D: PRE-TEST QUESTION RESPONSES BY QUARTILE FOR MAIN STUDY

Response %	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10	Q 11	Q 12
A	0.21	0.00	0.00	0.18	0.02	0.02	0.00	0.02	0.21	0.02	0.15	0.10
	0.15	0.02	0.00	0.08	0.03	0.03	0.00	0.02	0.15	0.02	0.13	0.05
	0.16	0.00	0.00	0.03	0.03	0.03	0.05	0.08	0.08	0.08	0.08	0.07
	0.08	0.00	0.02	0.00	0.02	0.11	0.02	0.05	0.05	0.07	0.03	0.03
B	0.02	0.00	0.23	0.00	0.03	0.05	0.00	0.23	0.03	0.03	0.02	0.05
	0.03	0.02	0.20	0.00	0.03	0.02	0.00	0.13	0.00	0.02	0.00	0.02
	0.02	0.02	0.18	0.00	0.03	0.11	0.00	0.13	0.03	0.00	0.07	0.08
	0.03	0.02	0.13	0.05	0.03	0.13	0.03	0.11	0.08	0.00	0.07	0.08
C	0.02	0.23	0.02	0.02	0.03	0.00	0.23	0.00	0.00	0.00	0.07	0.05
	0.07	0.20	0.05	0.03	0.08	0.00	0.16	0.03	0.02	0.00	0.05	0.08
	0.05	0.18	0.07	0.03	0.10	0.02	0.16	0.02	0.05	0.00	0.03	0.07
	0.05	0.20	0.07	0.00	0.15	0.00	0.11	0.05	0.07	0.03	0.13	0.11
D	0.00	0.02	0.00	0.05	0.15	0.18	0.02	0.00	0.00	0.18	0.00	0.02
	0.00	0.02	0.00	0.11	0.10	0.18	0.08	0.00	0.03	0.20	0.00	0.05
	0.02	0.05	0.00	0.18	0.07	0.08	0.05	0.02	0.05	0.16	0.02	0.02
	0.08	0.05	0.03	0.20	0.05	0.02	0.08	0.05	0.03	0.15	0.02	0.03
Total	0.98	1.00	0.98	0.97	0.95	0.98	0.98	0.90	0.89	0.95	0.85	0.90

Correct Response:

Figure D-1. Proportion of pre-test students who chose each response for questions 1 through 12, divided into quartiles by score (high to low).

Response %	Q 13	Q 14	Q 15	Q 16	Q 17	Q 18	Q 19	Q 20	Q 21	Q 22	Q 23	Q 24
A	0.08	0.07	0.16	0.00	0.10	0.11	0.11	0.18	0.05	0.02	0.03	0.00
	0.10	0.07	0.08	0.05	0.03	0.05	0.05	0.11	0.03	0.03	0.05	0.03
	0.07	0.13	0.03	0.08	0.02	0.02	0.03	0.07	0.02	0.00	0.03	0.02
	0.15	0.13	0.03	0.05	0.07	0.10	0.08	0.16	0.07	0.07	0.03	0.03
B	0.00	0.00	0.05	0.03	0.10	0.03	0.02	0.00	0.15	0.03	0.05	0.02
	0.02	0.02	0.03	0.07	0.08	0.02	0.02	0.05	0.08	0.05	0.07	0.02
	0.03	0.00	0.03	0.08	0.08	0.00	0.07	0.07	0.10	0.08	0.05	0.07
	0.05	0.03	0.02	0.13	0.10	0.02	0.02	0.07	0.07	0.08	0.08	0.10
C	0.00	0.15	0.00	0.11	0.02	0.02	0.03	0.00	0.02	0.05	0.10	0.20
	0.00	0.08	0.03	0.02	0.08	0.05	0.10	0.00	0.05	0.02	0.08	0.15
	0.02	0.07	0.07	0.00	0.11	0.16	0.03	0.03	0.03	0.07	0.11	0.08
	0.03	0.03	0.13	0.02	0.07	0.08	0.08	0.00	0.08	0.08	0.08	0.03
D	0.16	0.00	0.00	0.07	0.02	0.07	0.08	0.07	0.00	0.11	0.03	0.00
	0.11	0.03	0.07	0.05	0.02	0.07	0.02	0.05	0.02	0.08	0.00	0.00
	0.11	0.02	0.11	0.08	0.02	0.05	0.05	0.03	0.02	0.07	0.02	0.05
	0.03	0.05	0.07	0.05	0.02	0.05	0.03	0.03	0.03	0.03	0.07	0.08
Total	0.97	0.87	0.92	0.89	0.92	0.89	0.82	0.92	0.80	0.87	0.89	0.87

Correct Response:

Figure D-2. Proportion of pre-test students who chose each response for questions 13 through 24, divided into quartiles by score (high to low).

APPENDIX E: POST-TEST QUESTION RESPONSES BY QUARTILE FOR MAIN STUDY

Response %	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10	Q 11	Q 12
A	0.21	0.00	0.00	0.18	0.03	0.02	0.00	0.00	0.21	0.02	0.18	0.08
	0.18	0.00	0.00	0.05	0.02	0.05	0.02	0.03	0.15	0.05	0.11	0.13
	0.18	0.02	0.03	0.05	0.03	0.10	0.03	0.03	0.07	0.05	0.13	0.05
	0.11	0.03	0.07	0.02	0.03	0.07	0.05	0.03	0.08	0.03	0.05	0.07
B	0.03	0.00	0.23	0.00	0.05	0.02	0.00	0.25	0.00	0.00	0.00	0.03
	0.03	0.03	0.23	0.00	0.05	0.05	0.03	0.18	0.03	0.03	0.03	0.05
	0.02	0.07	0.11	0.03	0.07	0.05	0.00	0.18	0.05	0.02	0.05	0.08
	0.07	0.07	0.13	0.02	0.07	0.13	0.03	0.10	0.05	0.02	0.10	0.03
C	0.00	0.25	0.00	0.02	0.00	0.00	0.23	0.00	0.00	0.02	0.07	0.11
	0.03	0.21	0.00	0.05	0.07	0.03	0.15	0.03	0.05	0.05	0.08	0.03
	0.03	0.10	0.10	0.08	0.08	0.05	0.11	0.02	0.05	0.03	0.05	0.05
	0.02	0.13	0.03	0.07	0.15	0.03	0.10	0.10	0.10	0.08	0.08	0.10
D	0.00	0.00	0.02	0.05	0.16	0.21	0.02	0.00	0.02	0.21	0.00	0.02
	0.00	0.00	0.00	0.15	0.11	0.11	0.05	0.00	0.00	0.10	0.00	0.02
	0.00	0.07	0.00	0.08	0.05	0.05	0.10	0.02	0.05	0.11	0.00	0.03
	0.03	0.03	0.03	0.16	0.02	0.02	0.08	0.03	0.03	0.13	0.02	0.05
Total	0.95	1.00	0.98	1.00	0.98	0.98	1.00	1.00	0.93	0.95	0.95	0.93

Correct Response:

Figure E-1. Proportion of post-test students who chose each response for questions 1 through 12, divided into quartiles by score (high to low).

Response %	Q 13	Q 14	Q 15	Q 16	Q 17	Q 18	Q 19	Q 20	Q 21	Q 22	Q 23	Q 24
A	0.10	0.15	0.16	0.00	0.08	0.05	0.16	0.20	0.05	0.00	0.00	0.00
	0.13	0.08	0.11	0.07	0.07	0.03	0.10	0.11	0.05	0.05	0.05	0.02
	0.07	0.13	0.03	0.08	0.05	0.05	0.03	0.07	0.08	0.07	0.07	0.00
	0.15	0.13	0.05	0.10	0.03	0.10	0.08	0.03	0.05	0.07	0.10	0.02
B	0.00	0.00	0.00	0.03	0.10	0.05	0.02	0.00	0.18	0.03	0.02	0.00
	0.02	0.00	0.05	0.08	0.05	0.03	0.02	0.02	0.10	0.03	0.08	0.07
	0.07	0.03	0.03	0.08	0.07	0.10	0.03	0.08	0.03	0.07	0.07	0.08
	0.03	0.00	0.08	0.11	0.05	0.11	0.05	0.10	0.07	0.03	0.10	0.08
C	0.00	0.10	0.03	0.20	0.02	0.03	0.02	0.00	0.00	0.02	0.21	0.23
	0.00	0.11	0.02	0.00	0.08	0.05	0.03	0.03	0.02	0.07	0.08	0.07
	0.02	0.05	0.10	0.05	0.11	0.03	0.05	0.03	0.02	0.07	0.08	0.07
	0.03	0.05	0.08	0.02	0.13	0.02	0.07	0.07	0.08	0.15	0.02	0.03
D	0.15	0.00	0.03	0.00	0.05	0.10	0.03	0.05	0.00	0.20	0.00	0.02
	0.10	0.02	0.05	0.08	0.03	0.11	0.07	0.03	0.03	0.08	0.02	0.07
	0.08	0.02	0.05	0.02	0.00	0.05	0.08	0.03	0.08	0.03	0.02	0.05
	0.05	0.05	0.05	0.02	0.05	0.02	0.02	0.05	0.03	0.02	0.03	0.08
Total	0.98	0.92	0.93	0.93	0.97	0.93	0.85	0.90	0.87	0.97	0.93	0.87

Correct Response:

Figure E-2. Proportion of post-test students who chose each response for questions 13 through 24, divided into quartiles by score (high to low).

APPENDIX F: ITEM DISCRIMINATION ANALYSIS FOR PRE-TEST FOR MAIN STUDY

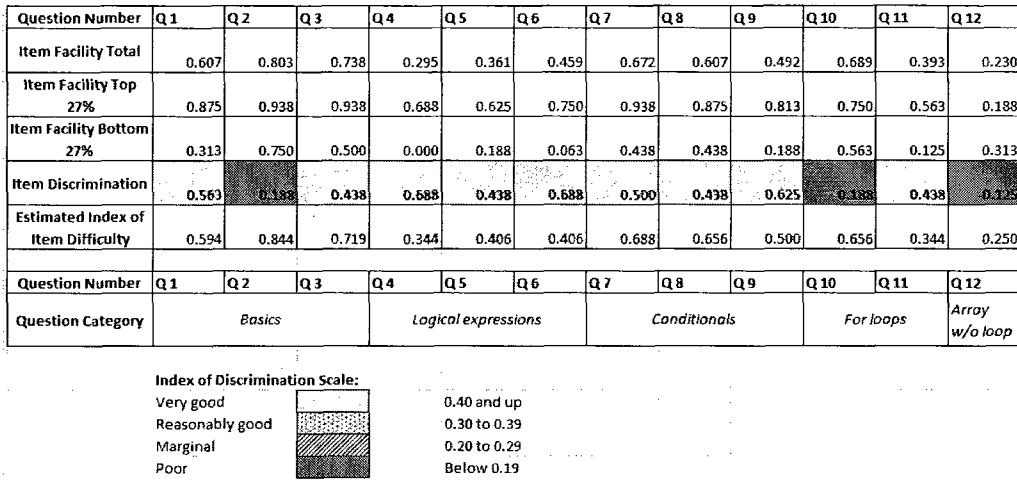


Figure F-1. Pre-test item discrimination analysis for questions 1 through 12.

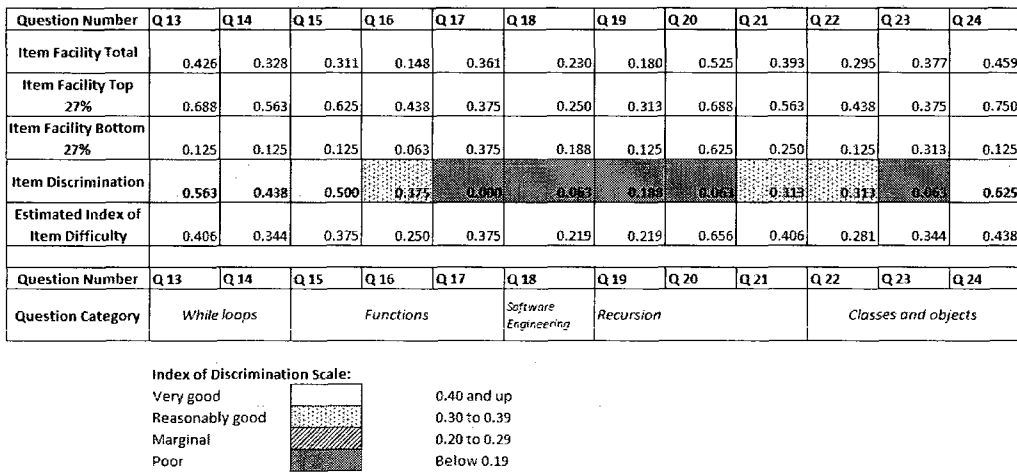


Figure F-2. Pre-test item discrimination analysis for questions 13 through 24.

**APPENDIX F: ITEM DISCRIMINATION ANALYSIS FOR PRE-TEST FOR
MAIN STUDY**

Reliability	0.568
Average Score	11.402
Standard Deviation	3.626
Standard Error of Measurement	2.382

Figure F-3. Reliability, average score, standard deviation, and standard error of measurement statistics for pre-test.

APPENDIX G: ITEM DISCRIMINATION ANALYSIS FOR POST-TEST FOR MAIN STUDY

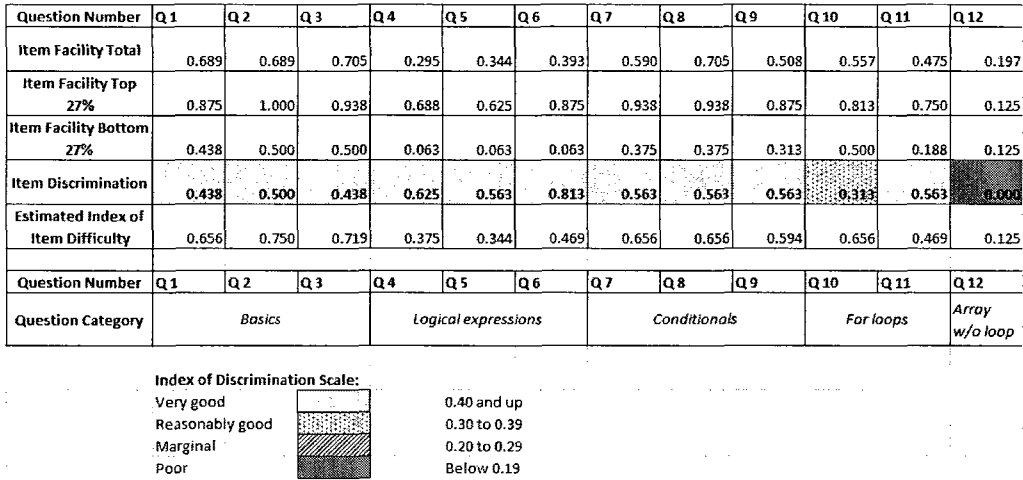


Figure G-1. Post-test item discrimination analysis for questions 1 through 12.

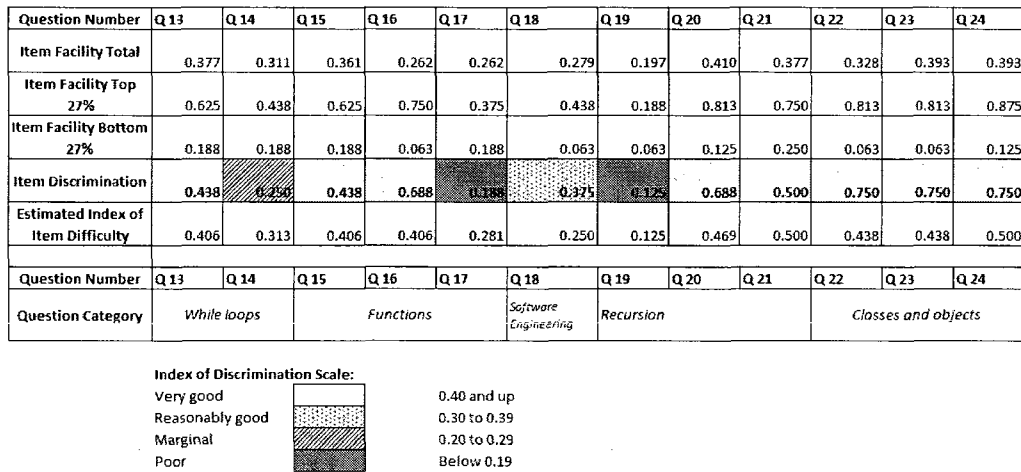


Figure G-2. Post-test item discrimination analysis for questions 13 through 24.

**APPENDIX G: ITEM DISCRIMINATION ANALYSIS FOR POST-TEST FOR
MAIN STUDY**

Reliability	0.749
Average Score	10.730
Standard Deviation	4.586
Standard Error of Measurement	2.297

Figure G-3.Reliability, average score, standard deviation, and standard error of measurement statistics for post-test.

APPENDIX H: CORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR PRE-TEST FOR MAIN STUDY

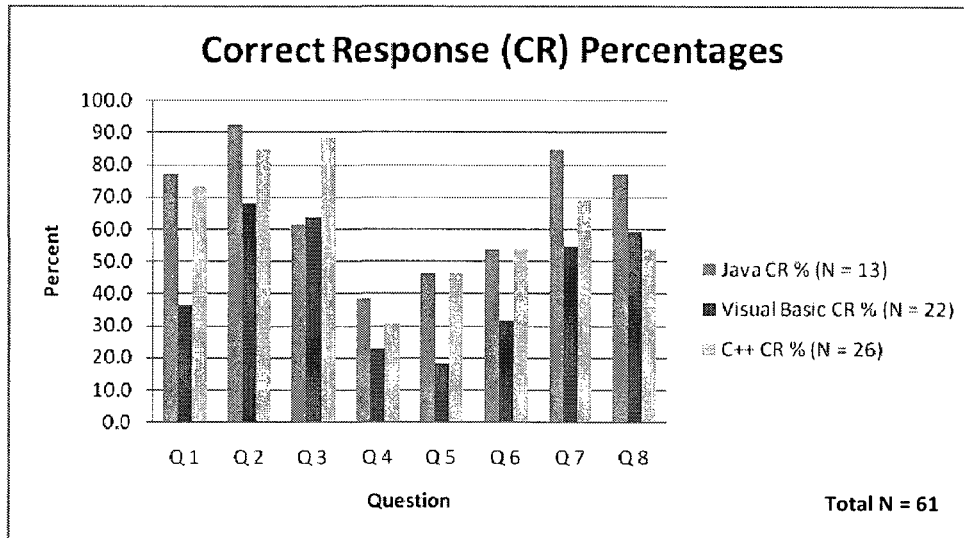


Figure H-1.Correct response percentages for questions 1 through 8 of pre-test.

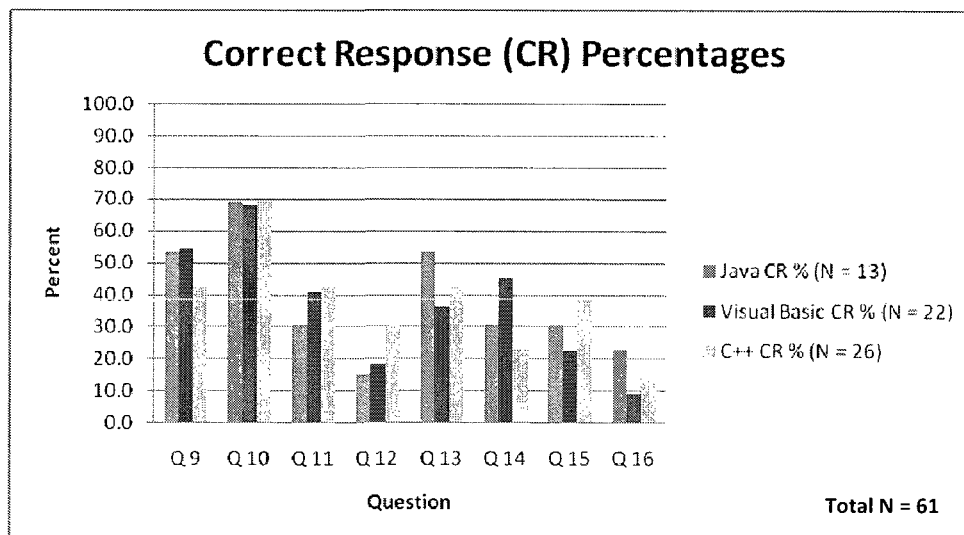


Figure H-2.Correct response percentages for questions 9 through 16 of pre-test.

APPENDIX H: CORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR PRE-TEST FOR MAIN STUDY

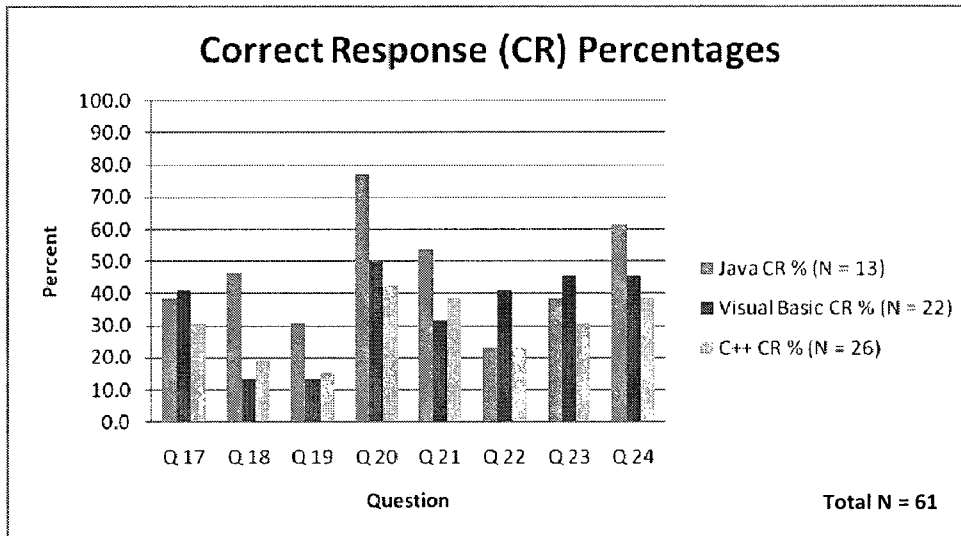


Figure H-3.Correct response percentages for questions 17 through 24 of pre-test.

APPENDIX I: CORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR POST-TEST FOR MAIN STUDY

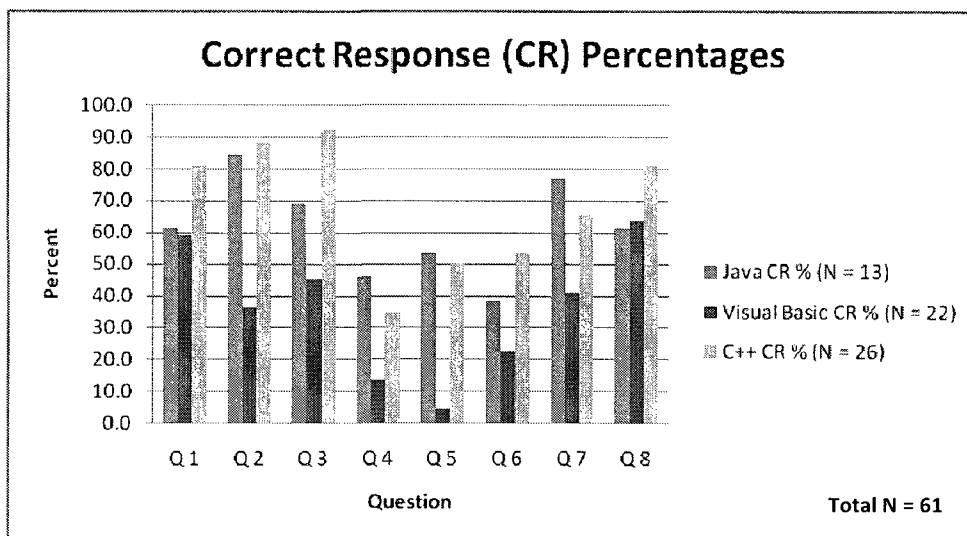


Figure I-1. Correct response percentages for questions 1 through 8 of post-test.

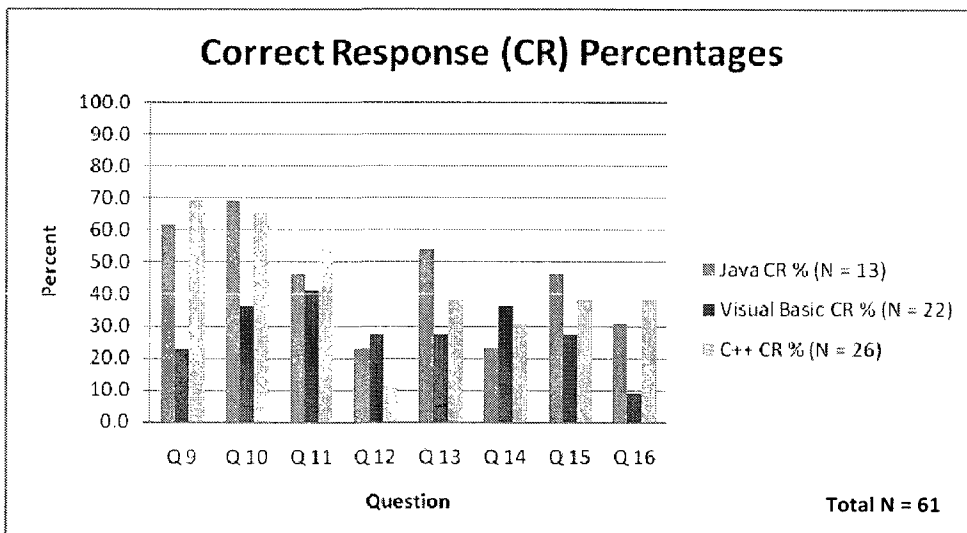


Figure I-2. Correct response percentages for questions 9 through 16 of post-test.

APPENDIX I: CORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR POST-TEST FOR MAIN STUDY

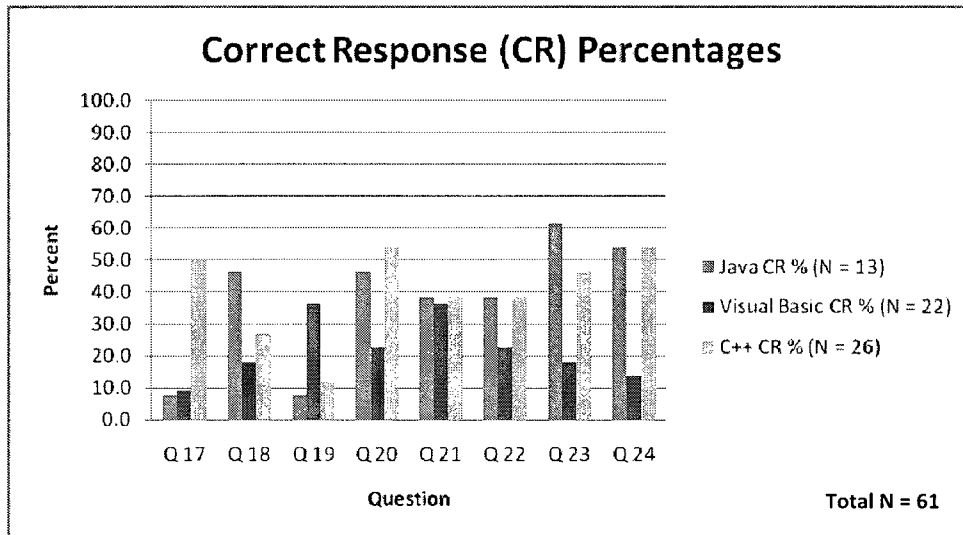


Figure I-3. Correct response percentages for questions 17 through 24 of post-test.

APPENDIX J: INCORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR PRE-TEST FOR MAIN STUDY

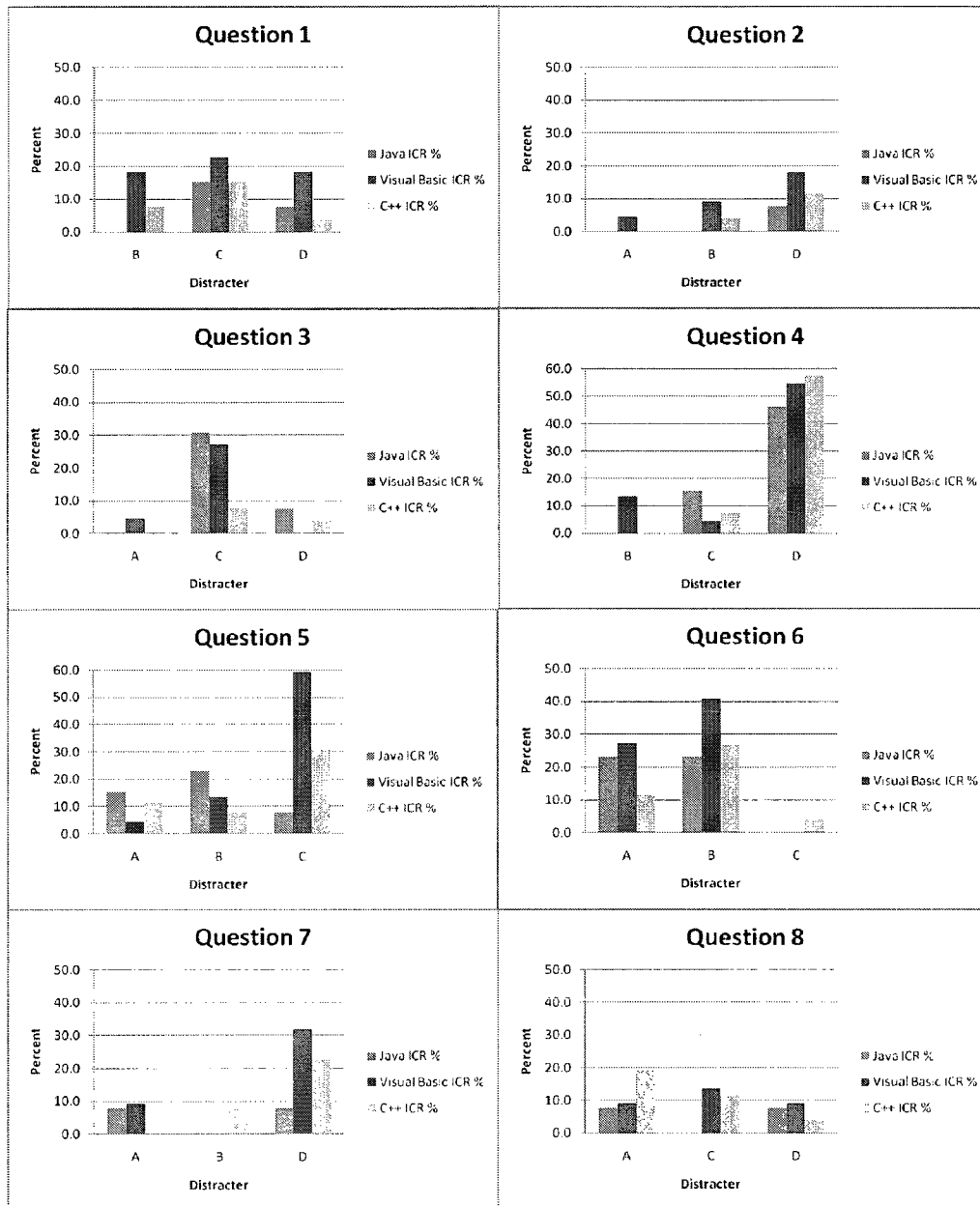


Figure J-1. Incorrect response percentages for questions 1 through 8 of pre-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

APPENDIX J: INCORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR PRE-TEST FOR MAIN STUDY

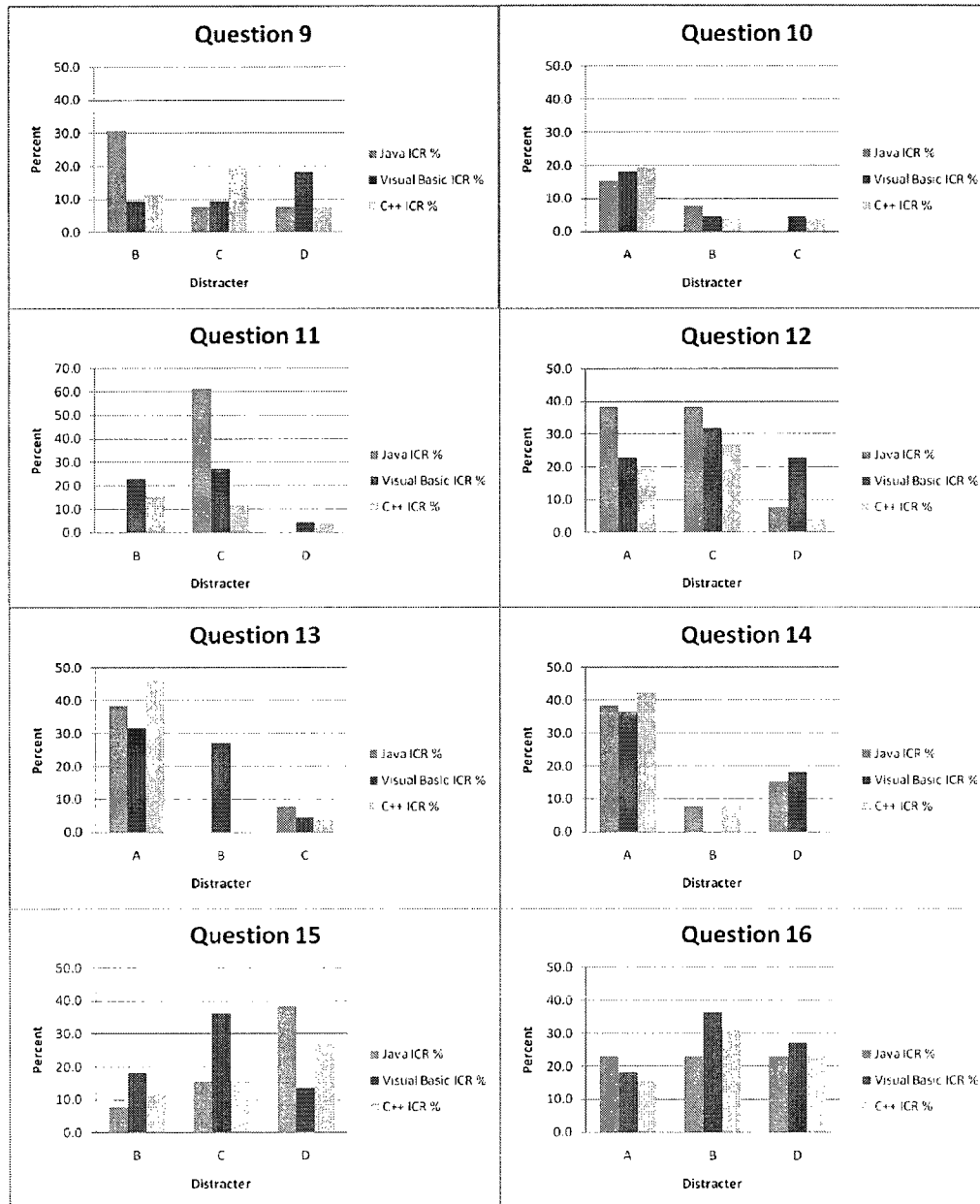


Figure J-2. Incorrect response percentages for questions 9 through 16 of pre-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

APPENDIX J: INCORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR PRE-TEST FOR MAIN STUDY

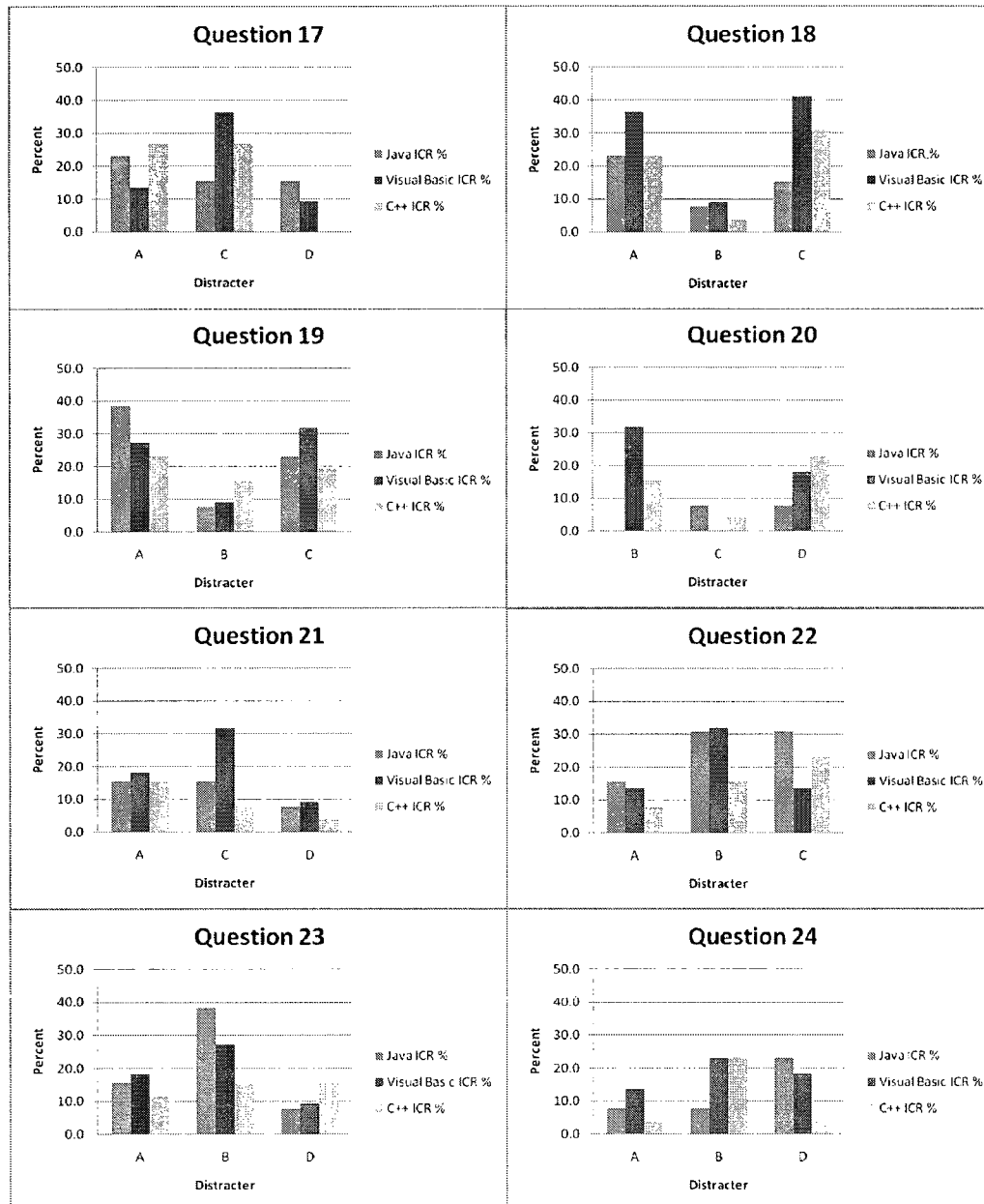


Figure J-3. Incorrect response percentages for questions 17 through 24 of pre-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

APPENDIX K: INCORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR POST-TEST FOR MAIN STUDY

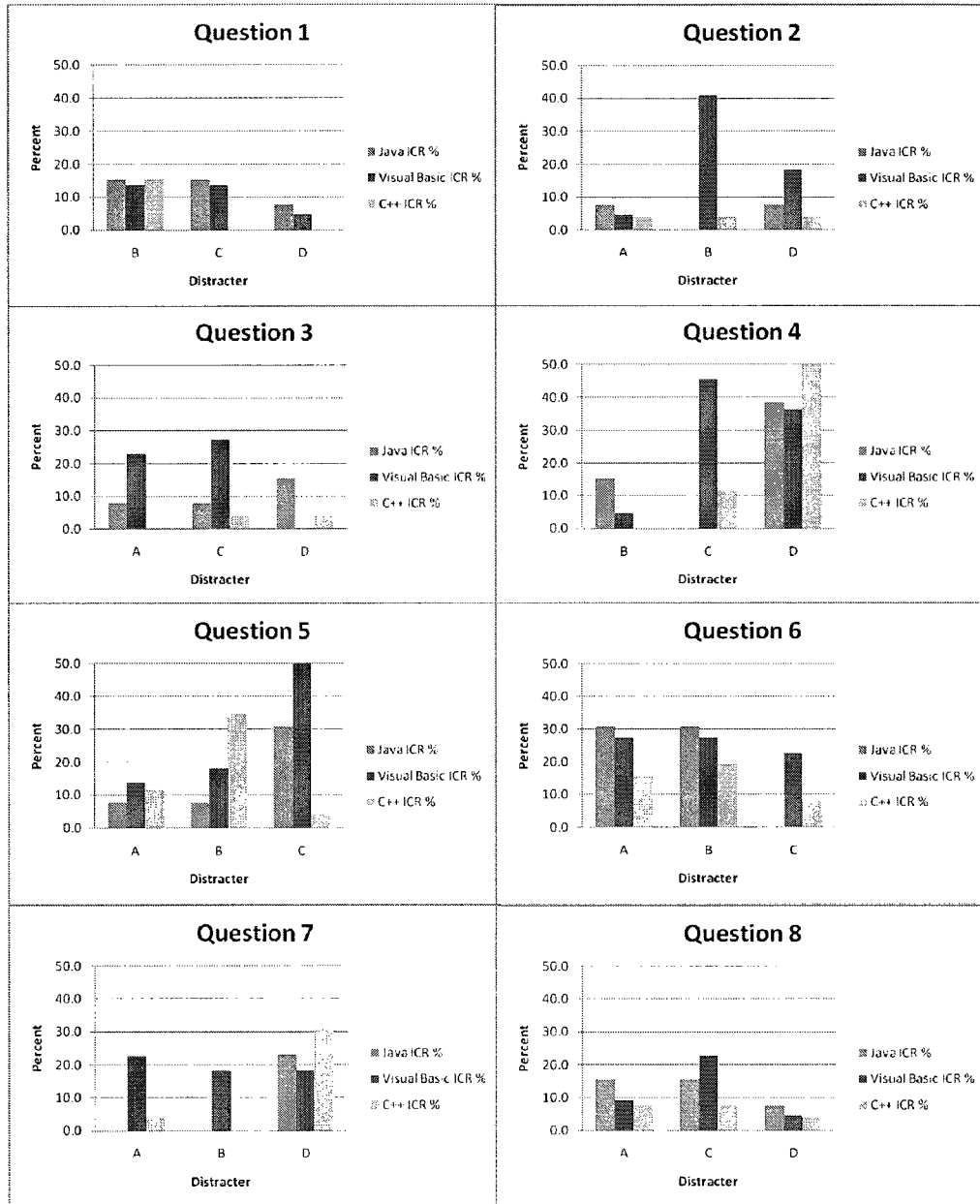


Figure K-1. Incorrect response percentages for questions 1 through 8 of post-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

APPENDIX K: INCORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR POST-TEST FOR MAIN STUDY

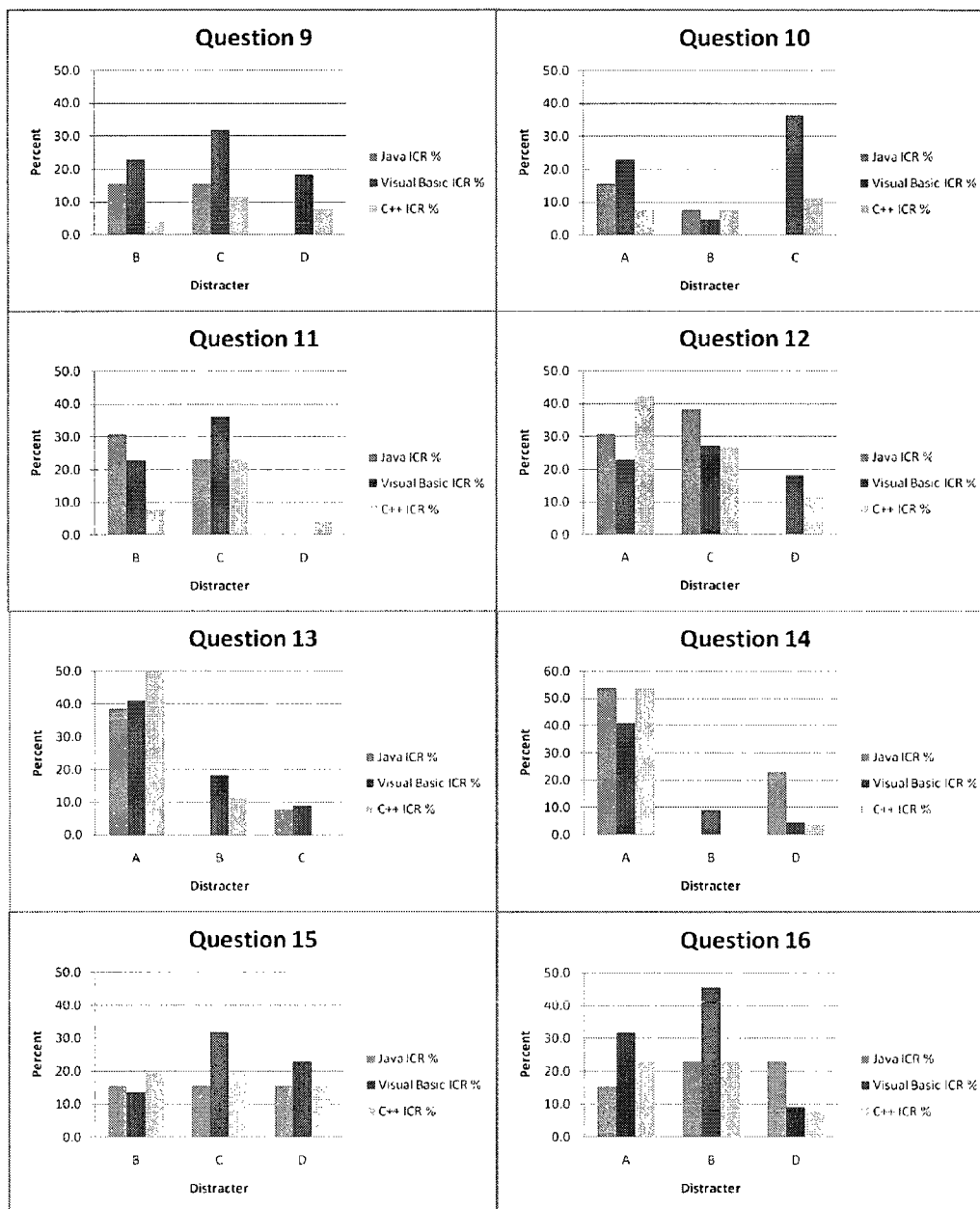


Figure K-2. Incorrect response percentages for questions 9 through 16 of post-test where:
Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

APPENDIX K: INCORRECT RESPONSE PERCENTAGES BY LANGUAGE GROUP FOR POST-TEST FOR MAIN STUDY

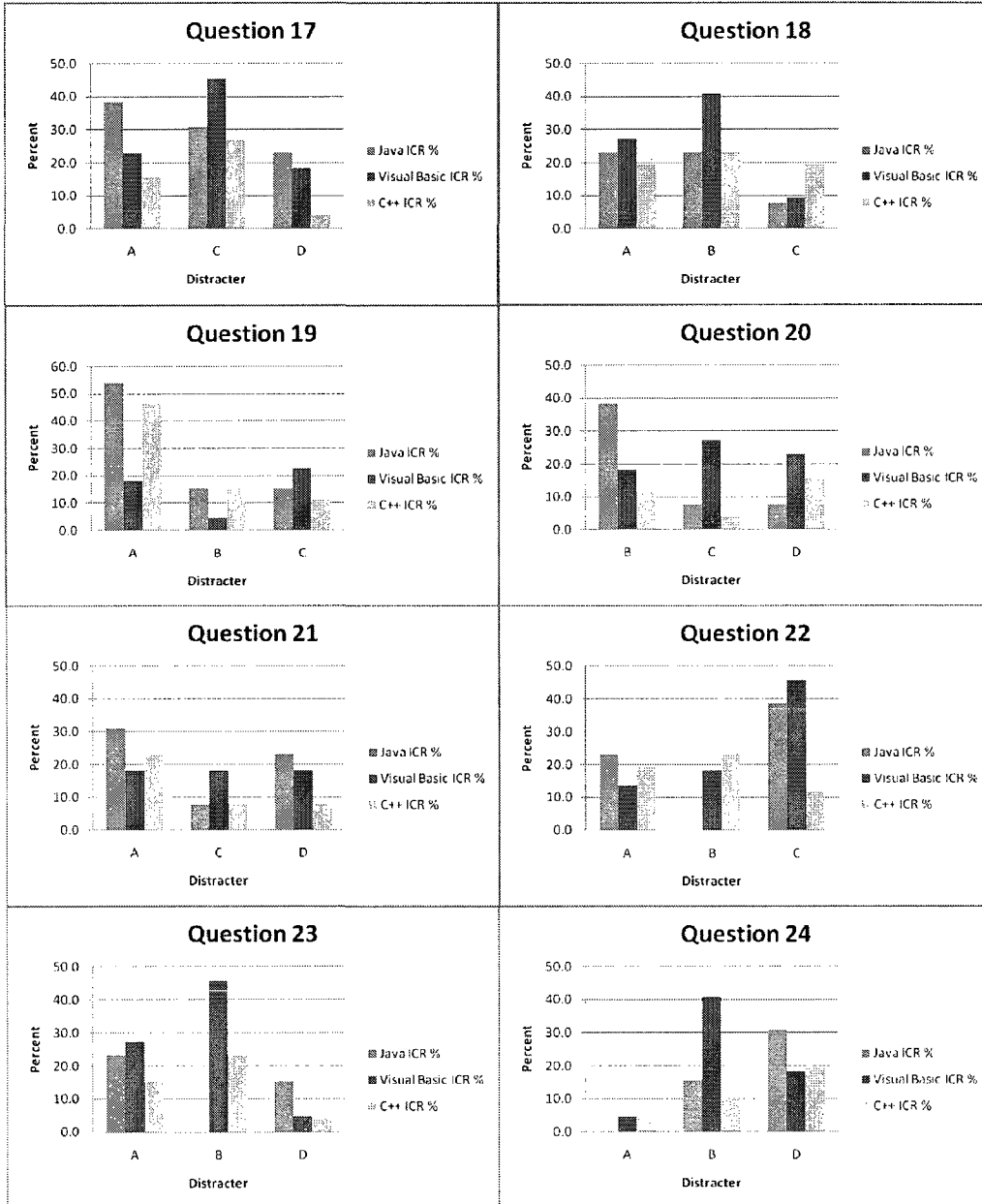


Figure K-3. Incorrect response percentages for questions 17 through 24 of post-test where: Java N = 13, Visual Basic N = 22, C++ N = 26, Total N = 61.

APPENDIX L: DEMOGRAPHIC SURVEY DATA FOR MAIN STUDY

ID_Number	Completed_Study	Language_Class	Academic_Level	Major	Gender	Ethnicity	Age_Range	Immediate_College_Entry	SAT_Math_Score	ACT_Math_Score
CS04113-5	1	1	Transfer	CS	M	Other	20	Y	999	99
CS04113-2	1	1	Freshman	CS	M	Caucasian	22-30	N	999	99
CS04113-1	1	1	Junior	CS	M	Asian (non-Indian)	20	Y	999	99
CS04113-3	1	1	Sophomore	CS	M	Caucasian	22-30	Y	500	99
CS04113-6	1	1	Transfer	CS	M	Caucasian	22-30	Y	999	99
CS04113-4	1	1	Other	N/A	M	Caucasian	30+	Y	680	99
CS04113-10	1	1	Junior	CS	M	Caucasian	21	Y	999	99
CS04113-8	0	1	Junior	CS	M	Caucasian	21	Y	999	99
CS04113-9	1	1	Sophomore	CS	M	Caucasian	20	Y	660	99
CS04113-7	1	1	Transfer	CS	M	Caucasian	22-30	Y	999	99
CS04114-17	1	1	Freshman	CS	M	Caucasian	18	Y	640	99
CS04114-18	1	1	Freshman	CS	M	Caucasian	19	Y	550	99
CS04114-24	1	1	Freshman	CS	M	Caucasian	19	Y	660	31 (Overall)
CS04114-23	1	1	Freshman	CS	M	Caucasian	18	Y	700	99

ID_Number	Alice	Basic	C	C_PlusPlus	C_Sharp	Cobol	Html	IDL	Java	JavaScript	Pascal	Perl	PHP	Python	SQL	Verilog_HDL	Visual_Basic
CS04113-5	0	0	3	3	0	0	0	0	3	0	0	0	0	0	0	0	3
CS04113-2	0	0	1	0	0	0	0	0	2	2	0	0	0	0	0	0	0
CS04113-1	0	0	0	3	0	0	2	0	2	0	0	0	0	0	0	0	0
CS04113-3	3	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	3
CS04113-6	0	0	1	3	0	0	0	0	2	0	0	0	0	0	0	0	3
CS04113-4	0	0	2	0	0	5	0	0	2	0	0	0	0	0	5	0	2
CS04113-10	3	0	0	0	0	0	3	0	3	0	0	0	0	0	0	0	1
CS04113-8	0	0	0	3	0	0	0	0	2	0	0	0	0	0	0	0	3
CS04113-9	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0
CS04113-7	0	2	3	3	1	0	0	0	5	2	0	2	0	1	0	0	3
CS04114-17	0	0	0	3	0	0	0	0	4	0	0	0	0	0	0	0	0
CS04114-18	0	0	0	0	0	0	0	0	2	2	0	0	0	0	0	0	0
CS04114-24	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	3
CS04114-23	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	3

Figure L-1. Demographic information for the 14 Java students; the highlighted records represent students who only completed the first part of the study.

APPENDIX L: DEMOGRAPHIC SURVEY DATA FOR MAIN STUDY

ID_Number	Completed_Study	Language_Class	Academic_Level	Major	Gender	Ethnicity	Age_Range	Immediate_College_Entry	SAT_Math_Score	ACT_Math_Score
CS04141-22	1	2 Junior	MIS	F	African American	20	Y	999	99	
CS04141-25	1	2 Freshman	MIS	M	Caucasian	19	Y	660	99	
CS04141-26	1	2 Senior	MIS	M	Caucasian	22-30	Y	999	99	
CS04141-28	0	2 Junior	MIS	M	Caucasian	21	Y	550	99	
CS04141-31	0	2 Senior	MIS	M	Caucasian	22-30	Y	680	99	
CS04141-27	1	2 Senior	MKT, MIS	F	Caucasian	22-30	Y	999	99	
CS04141-24	0	2 Senior	MIS	M	Caucasian	20	Y	999	99	
CS04141-23	0	2 Sophomore	MIS	M	Other	19	Y	999	99	
CS04141-29	1	2 Senior	MIS	M	Other	22-30	Y	1150	99	
CS04141-18	0	2 Junior	MIS	M	Hispanic	20	Y	999	99	
CS04141-17	1	2 Sophomore	MIS	M	Caucasian	20	N	999	99	
CS04141-21	0	2 Junior	MIS	M	Caucasian	22-30	N	999	99	
CS04141-20	1	2 Junior	MIS	M	Caucasian	22-30	N	999	99	
CS04141-16	1	2 Junior	MIS	M	Caucasian	20	Y	999	99	
CS04141-15	1	2 Junior	MIS	M	Caucasian	20	Y	999	99	
CS04141-10	1	2 Junior	MIS	M	Other	21	Y	999	16	
CS04141-4	1	2 Freshman	MIS	M	Caucasian	18	Y	600	99	
CS04141-1	1	2 Transfer	MIS	M	Caucasian	21	Y	999	99	
CS04141-13	1	2 Freshman	MIS	M	Caucasian	19	Y	660	99	
CS04141-14	1	2 Freshman	MIS	M	Caucasian	18	Y	610	99	
CS04141-12	1	2 Sophomore	MIS	M	Asian (non-Indian)	20	Y	999	99	
CS04141-3	1	2 Junior	MIS, RTF	M	Caucasian	21	Y	999	99	
CS04141-5	1	2 Freshman	MIS	F	Caucasian	18	Y	600	99	
CS04141-8	0	2 Junior	MIS	M	Hispanic	20	Y	500	99	
CS04141-11	1	2 Junior	MIS	M	Caucasian	21	Y	999	99	
CS04141-2	1	2 Transfer	LA	M	Caucasian	21	Y	999	99	
CS04141-6	1	2 Junior	MIS	M	Hispanic	21	Y	610	99	
CS04141-7	1	2 Freshman	MIS	M	Asian (non-Indian)	18	Y	999	99	
CS04141-9	1	2 Senior	MIS	M	Caucasian	21	Y	999	99	

ID_Number	Alice	Basic	C	C_PlusPlus	C_Sharp	Cobol	Html	IDL	Java	JavaScript	Pascal	Perl	PHP	Python	SQL	Verilog_HDL	Visual_Basic
CS04141-22	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
CS04141-25	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	3
CS04141-26	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	3
CS04141-28	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
CS04141-27	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-23	0	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-29	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-18	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
CS04141-17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-21	0	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0	4
CS04141-20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-16	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
CS04141-15	3	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	3
CS04141-10	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-1	0	0	0	3	0	0	0	0	3	1	0	0	0	0	0	0	3
CS04141-13	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	3
CS04141-14	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	3
CS04141-12	0	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-3	0	0	0	0	0	0	0	0	3	0	0	2	0	0	0	0	3
CS04141-5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
CS04141-8	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-11	0	0	0	3	0	0	0	0	0	1	0	0	0	0	0	0	3
CS04141-2	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	2
CS04141-6	0	0	0	3	0	0	0	0	0	1	0	0	0	0	0	0	4
CS04141-7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04141-9	0	0	2	3	0	0	0	0	3	2	0	0	0	0	0	0	3

Figure L-2. Demographic information for the 29 Visual Basic students; the highlighted records represent students who only completed the first part of the study.

APPENDIX L: DEMOGRAPHIC SURVEY DATA FOR MAIN STUDY

ID_Number	Completed_Study	Language_Class	Academic_Level	Major	Gender	Ethnicity	Age_Range	Immediate_College_Entry	SAT_Math_Score	ACT_Math_Score
CS04103-2	1	3	Freshman	ME	F	Caucasian	19	Y	999	99
CS04103-21	1	3	Freshman	ECE	M	Caucasian	19	Y	670	99
CS04103-15	1	3	Freshman	ME	M	Caucasian	19	Y	710	99
CS04103-6	1	3	Freshman	ECE	M	Caucasian	18	Y	620	99
CS04103-5	1	3	Freshman	ECE	M	Caucasian	18	Y	560	99
CS04103-7	1	3	Freshman	ME	M	Caucasian	19	Y	999	99
CS04103-8	1	3	Freshman	ME	M	Caucasian	19	Y	780	99
CS04103-3	1	3	Freshman	ME	M	Caucasian	18	Y	720	99
CS04103-23	1	3	Freshman	ECE	M	Caucasian	18	Y	680	99
CS04103-4	1	3	Freshman	ECE	M	Caucasian	19	Y	630	99
CS04103-19	1	3	Freshman	ME	M	Caucasian	18	Y	700	99
CS04103-9	1	3	Freshman	ME	M	Caucasian	19	Y	710	99
CS04103-1	1	3	Freshman	ECE	M	Caucasian	18	Y	550	99
CS04103-14	1	3	Freshman	ME	M	Caucasian	19	Y	690	99
CS04103-11	1	3	Freshman	ME	F	Caucasian	19	Y	680	99
CS04103-10	1	3	Freshman	ECE	F	Caucasian	19	Y	670	99
CS04103-17	1	3	Senior	MATH, PHYS	F	Caucasian	22-30	Y	690	99
CS04103-12	1	3	Freshman	ME	M	Caucasian	18	Y	690	99
CS04103-16	1	3	Senior	PHYS	M	African American	21	Y	999	99
CS04103-22	1	3	Freshman	ECE	M	Caucasian	19	Y	680	99
CS04103-20	1	3	Freshman	ECE	M	Caucasian	18	Y	999	99
CS172-1	1	3	Freshman	CS	M	Asian (non-Indian)	19	Y	640	99
CS172-2	1	3	Freshman	CS	M	Caucasian	18	Y	999	99
SE103-1	1	3	Freshman	SE	M	Caucasian	18	Y	740	99
CS132-1	1	3	Senior	MATH	M	Asian (Indian)	22-30	Y	710	34
CS132-2	1	3	Freshman	MATH, ECON	M	Caucasian	18	Y	999	99

ID_Number	Alice	Basic	C	C_PlusPlus	C_Sharp	Cobol	Html	IDL	Java	JavaScript	Pascal	Perl	PHP	Python	SQL	Verilog_HDL	Visual_Basic
CS04103-2	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-21	0	2	0	3	0	0	0	0	0	2	0	0	0	0	2	0	0
CS04103-15	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-6	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-5	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-7	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-8	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-3	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-23	0	0	0	3	0	0	3	0	0	0	0	0	0	0	0	2	0
CS04103-4	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-19	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-9	0	2	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-1	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-14	0	1	0	3	0	0	0	0	2	2	0	0	0	0	0	0	1
CS04103-11	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04103-10	0	0	0	3	0	0	0	0	3	0	0	0	0	0	0	0	0
CS04103-17	0	0	0	2	0	0	0	5	0	0	0	0	0	0	0	0	3
CS04103-12	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-16	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	3
CS04103-22	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
CS04103-20	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
CS172-1	0	0	0	3	0	0	0	0	0	3	0	0	0	0	0	0	0
CS172-2	0	0	0	3	0	0	0	0	0	1	0	0	0	0	0	0	0
SE103-1	0	0	0	3	0	0	0	0	0	2	0	0	0	0	0	0	0
CS132-1	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	2
CS132-2	0	0	0	3	0	0	0	0	3	0	0	0	0	0	0	0	2

Figure L-3. Demographic information for the 26 C++ students; all of the C++ students completed both parts of the study.

APPENDIX M: DEMOGRAPHIC SURVEY FOR MAIN STUDY

Project Title: An Investigation of the Impact of Disparate Teaching Approaches on Student Learning of Introductory Programming Concepts
 Project Investigators: Robert B. Allen, PI, Watida M. Kunkle, Co-PI

Demographic Survey

This survey asks you to provide background information as part of the study in which you have agreed to participate. Please answer the questions as honestly and as accurately as possible. Your answers will be kept confidential.

1) Enter the identification number (ID) that you have been given:

2) What is your academic level?
 Freshman Sophomore Junior Senior Transfer
 Other (please specify)

3) What is your major area of study?

4) What is your gender?
 Male Female

5) What is your ethnicity?
 Asian (not from Indian subcontinent) Asian (from Indian subcontinent)
 Caucasian African American
 Hispanic Other or mixed

6) What is your age (range)?
 17 18 19 20 21
 22-30 30+

7) Did you enter college right after completing high school?
 Yes No

8) From the languages listed below, please indicate those with which you have programming experience (please check all that apply).

<input type="checkbox"/> Alice	<input type="checkbox"/> Basic	<input type="checkbox"/> C	<input type="checkbox"/> C++	<input type="checkbox"/> Java
<input type="checkbox"/> JavaScript	<input type="checkbox"/> Pascal	<input type="checkbox"/> Perl	<input type="checkbox"/> Python	<input type="checkbox"/> Visual Basic
<input type="checkbox"/> Other language (please specify) <input type="text"/>				

9) This question refers back to question #8. For those languages with which you indicated you have programming experience, please specify how much using the scale that appears below.

1	Slight (can read code; know a bit about the language)
2	Some (have tried to write programs in this language)
3	Good (have had programming courses in this language)
4	AP (took and passed AP-level programming courses in this language)
5	Better (worked - e.g., summers - as a programmer using this language)

Language	Scale	Language	Scale	Language	Scale
Alice	Basic	C	C++	Java	Visual Basic
JavaScript	Pascal	Perl	Python		
Other language (please specify) <input type="text"/>					

10) If you took the SAT, what was your SAT Math score? (Note: If you do not recall, or do not wish to answer, you may leave this question blank.)

11) If you took the ACT, what was your ACT Math score? (Note: If you do not recall, or do not wish to answer, you may leave this question blank.)

Page 1 of 2

Figure M-1. Demographic survey, pages 1 and 2.

APPENDIX N: ATTITUDE SURVEY FOR MAIN STUDY

Project Title: An Investigation of the Impact of Disparate Teaching Approaches on Student Learning of Introductory Programming Concepts

Project Investigators: Robert B. Allen, Ph.D., Wanda M. Kunkle, Co-PI

Attitude Survey

This survey asks you to respond to a series of statements to gain insight into your attitude toward computer programming and computer science in general. Please note that your answers will be kept confidential.

Instructions
Enter the identification number (ID) that you have been given:

For each of the statements that follow:

- Read this statement.
- Consider the degree to which you agree or disagree with the statement.
- Place a checkmark in the box under the label that most closely reflects your agreement or disagreement with the statement.

Please keep in mind:

- There are no right or wrong answers, so don't be afraid to respond honestly.
- Move quickly through the survey.
- Complete all of the items.

	Strongly agree	Agree	Neutral	Disagree	Strongly disagree
1.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
6.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
7.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Project Title: An Investigation of the Impact of Disparate Teaching Approaches on Student Learning of Introductory Programming Concepts

Project Investigators: Robert B. Allen, Ph.D., Wanda M. Kunkle, Co-PI

Attitude Survey

This survey asks you to respond to a series of statements to gain insight into your attitude toward computer programming and computer science in general. Please note that your answers will be kept confidential.

Instructions
Enter the identification number (ID) that you have been given:

For each of the statements that follow:

- Read this statement.
- Consider the degree to which you agree or disagree with the statement.
- Place a checkmark in the box under the label that most closely reflects your agreement or disagreement with the statement.

Please keep in mind:

- There are no right or wrong answers, so don't be afraid to respond honestly.
- Move quickly through the survey.
- Complete all of the items.

	Strongly agree	Agree	Neutral	Disagree	Strongly disagree
8.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
9.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
11.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
12.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
13.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
14.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
16.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
17.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
18.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
19.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
20.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
21.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
22.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
23.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
25.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure N-1. Attitude survey, pages 1 and 2.

APPENDIX N: ATTITUDE SURVEY FOR MAIN STUDY

		Strongly agree	Agree	Neutral	Disagree	Strongly disagree
26.	Females are as good as males at programming.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
27.	Studying computer science is just as appropriate for women as for men.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
28.	I would trust a woman just as much as I would trust a man to figure out important programming problems.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
29.	Women actually are logical enough to do well in computer science.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
30.	It's hard to believe a female could be a genius in computer science.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
31.	It makes sense that there are more men than women in computer science.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
32.	I would have more faith in the answer for a programming problem solved by a man than a woman.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
33.	Women who enjoy studying computer science are a bit peculiar.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
34.	I'll need programming for my future work.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
35.	I study programming because I know how useful it is.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
36.	Knowing programming will help me earn a living.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
37.	Computer science is a worthwhile and necessary subject.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
38.	I'll need a firm mastery of programming for my future work.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
39.	I will use programming in many ways throughout my life.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
40.	Programming is of no relevance to my life.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure N-2. Attitude survey, page 3.

APPENDIX O: COMPUTER CONCEPTS SURVEY FOR MAIN STUDY

Project Title: An Investigation of the Impact of Disparate Teaching Approaches on Student Learning of Introductory Programming Concepts

Project Investigators: Robert B. Allen, PI; Wanda M. Kunkle, Co-PI

Computer Concepts Survey

The purpose of this survey is to assess your knowledge of computer programming concepts. Rest assured that your performance on it will have no impact on your grade in the computer programming course in which you are currently enrolled. Your answers will be kept confidential.

Instructions
Enter the identification number (ID) that you have been given:

For each of the questions that follow:

- Read the question and the accompanying answer choices carefully.
- Circle the letter that represents your best response.

- What is the meaning of the statement $y = x + 10$?
 - Retrieve this value of the variable x , add 10 to it, and then place the resulting sum into the variable y .
 - Retrieve the value of the variable x , add 10 to it, and then place the resulting sum into the variable y . y will always have that value.
 - Create a new variable y that will always get the value of the sum of 10 and whatever value is in the variable x . If x is 5, then y is 15. If x later becomes 10, then y updates to 20.
 - Retrieve the value of the variable x , add 10 to it, and then place the resulting sum into the variable y . If the next statement is $y = 23 - 5$, then x updates to 13 - 5.

- Given the statements:
 - a = 10
 - b = 7
 - c = 15
 - a = b
 - b = c
 - a = b + a
 - c = (a - b) / b

What are the values of b and c after this code executes?

 - b = 6
c = 15
 - b = 7
c = 9.
 - b = 15
c = 6
 - b = 105
c = 6

- In mathematics, the general form of a second-degree polynomial is $y = ax^2 + bx + c$. Which of the following assignment statements correctly represent(s) the general form?
 - $y = a * x + x + b * x + c$
 - $y = x * (a + x + b) + c$
 - $y = a * (x * x) + b * (x + c)$
 - iii only
 - i and ii only
 - i and iii only
 - i, ii, and iii

- Given the expression:


```
(num1 < num2) AND (num2 < num3)
```

Which of the following statements must always be true?

 - The expression returns a value that represents true or false.
 - The expression is equivalent to (num1 >= num2) OR (num3 >= num2).
 - The expression is only false if both parenthesized expressions are false.
 - The expression is basically the same as an algebraic expression (e.g., $1 < x$ and $x < 5$, or $1 < x < 5$), so it can alternately be coded as (num1 < num2 < num3).

Version Date: 02/07/10

Page 1 of 14

Figure O-1.Computer concepts survey, pages 1 and 2.

APPENDIX O: COMPUTER CONCEPTS SURVEY FOR MAIN STUDY

5) Given the statements:

```

response1 = (true AND false) OR true
response2 = false OR (true AND false)
response3 = NOT(response1 AND response2)
response4 = response3 AND NOT(response1 OR response2)

```

What are the values of response1, and response3 after this code executes?

a) response1 = false
response3 = false

b) response1 = true
response3 = true

c) response1 = true
response3 = false

d) response1 = false
response3 = true

6) The following algorithm prompts a user to re-enter a student's grade that is not valid (outside the range zero to one hundred):

While the grade entered by the user is outside the range zero to one hundred
 Display an error message to the user
 Prompt the user to re-enter the grade
 Input the grade

Which code fragment correctly implements the condition in the incomplete loop shown below?

```

WHILE (condition) DO
  WRITE("Invalid grade.")
  WRITE("Please enter a grade in the range 0 to 100. ")
  READ(grade)
END WHILE

```

a) condition = (grade < 0) AND (grade > 100)

b) condition = (grade >= 0) AND (grade <= 100)

c) condition = (grade >= 0) OR (grade <= 100)

d) condition = (grade < 0) OR (grade > 100)

7) Given the code:

```

IF (actualEnrollment < maxEnrollment) THEN
  WRITE("Seats available!")
ELSE
  WRITE("Sorry! NO seats available!")
END IF
WRITE("Please try again next term.")

```

Which of the following statements must always be true?

a) "Please try again next term." will only display when the value in the actualEnrollment variable is greater than the value in the maxEnrollment variable.

b) "Sorry! NO seats available!" will never display because there is no condition following the ELSE.

c) "Please try again next term." will display no matter what values the actualEnrollment and maxEnrollment variables hold.

d) "Sorry! NO seats available!" will only display when the value in the actualEnrollment variable is greater than the value in the maxEnrollment variable.

Version Date: 02/07/10

Page 3 of 14

Figure O-2.Computer concepts survey, pages 3 and 4.

APPENDIX O: COMPUTER CONCEPTS SURVEY FOR MAIN STUDY

8) Given the code:

```

x = 6
y = 5
z = 4
IF (y >= x) THEN
  IF (x < y) THEN
    x = x + x
  ELSE
    y = x + y
  END IF
END IF
IF (x >= y) THEN
  IF (y < x) THEN
    ELSE y = y + y
  ELSE x = y + x
  END IF
ELSE
  z = z * 2
END IF
IF (y >= x) THEN
  y = x + y
END IF
WRITE("x = ", x)
WRITE("y = ", y)

```

What is the output when this code executes?

- x = 11
y = 5
- x = 6
y = 31
- x = 6
y = 5
- x = 31
y = 6

9) Given the incomplete code:

```

homers = 2
runs = 7
singles = 15
IF (condition1) THEN
  IF (condition2) THEN
    WRITE("I love Baseball!")
  ELSE
    WRITE("Phillies are fantastic!")
  ELSE
    WRITE("2008 World Series champions!")
  END IF
ELSE
  homers = homers + 2
  IF (condition1) THEN
    WRITE("2008 World Series champions!")
  END IF
END IF

```

Which fragments complete the code to produce the output shown below?

2008 World Series champions!

- condition1 = homers <= runs
condition2 = runs > singles
condition3 = singles < homers
- condition1 = runs <= homers
condition2 = runs > singles
condition3 = singles < homers
- condition1 = homers <= runs
condition2 = singles > runs
condition3 = singles < homers
- condition1 = homers <= runs
condition2 = runs > singles
condition3 = homers < singles

Version Date: 02/07/10

Page 5 of 14

Version Date: 03/07/10

Page 6 of 14

Figure O-3. Computer concepts survey, pages 5 and 6.

APPENDIX O: COMPUTER CONCEPTS SURVEY FOR MAIN STUDY

10) Given the code:

```
n = 0
FOR (i = 1 to 10) DO
  n = n + 1
END FOR
```

What does this FOR statement do?

- It loops indefinitely.
- It adds the multiples of 3 in the range 1 through 10.
- It adds the odd numbers in the range 1 through 10.
- It adds the whole numbers in the range 1 through 10.

11) Given the incomplete code:

```
n = 4
i = 6
FOR (outerLoopHeader) DO
  FOR (innerLoopHeader) DO
    WRITE (newline)
  END FOR
END FOR
```

Which fragments complete the code to produce the output shown below?

```
****
****
****
****
****
```

- ```
outerLoopHeader = i = 1 to n
innerLoopHeader = j = 1 to m
loopBody = ""
```
- ```
outerLoopHeader = i = 1 to n
innerLoopHeader = j = m to n
loopBody = ""
```
- ```
outerLoopHeader = i = 1 to m
innerLoopHeader = j = 1 to n
loopBody = ""
```
- ```
outerLoopHeader = i = m to n
innerLoopHeader = j = 1 to m
loopBody = ""
```

12) Assume that integerArray is an integer array of size 10 and that its elements are numbered consecutively starting with 0 (zero). Given the code segment:

```
i = 7
j = 2
k = 4

integerArray(0) = 1
integerArray(1) = 5
integerArray(2) = integerArray(i) + integerArray(0)
integerArray(k) = integerArray(j) + integerArray(1) - integerArray(0)
k = integerArray(k)
```

Which of the following is true after the above code executes?

- The k^{th} element of the array integerArray has the value 10 in it.
- Element number k of the array integerArray is outside the bounds of the array.
- Element number k of the array integerArray has the value 10 in it.
- The k^{th} element of the array integerArray is outside the bounds of the array.

Version Date: 02/07/10

Page 7 of 14

Page 8 of 14

Figure O-4. Computer concepts survey, pages 7 and 8.

APPENDIX O: COMPUTER CONCEPTS SURVEY FOR MAIN STUDY

13) Given the code:

```

c = 0
i = 1
WHILE (i <= 10) DO
  n = n + i
END WHILE

```

What does this WHILE statement do?

- It adds the whole numbers in the range 1 through 10.
- It adds the odd numbers in the range 1 through 10.
- It adds the multiples of 3 in the range 1 through 10.
- It loops indefinitely.

14) Assume that squaresArray is an integer array of size 10 and that its elements are numbered consecutively starting with 0 (zero). Given the code:

```

squaresArray = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
arraySize = 10
limit = 59
count = 0
sum = 0

```

```

WHILE ((sum < limit) AND (count < arraySize)) DO
  sum = sum + squaresArray(count)
  count = count + 1
END WHILE

```

What is the value of the variable "sum" after this code executes?

- 55
- 6
- 91
- 30

15) Given the outline of the function:

```

computeInterest(p, r, t)
// p, r, and t are value parameters

```

```

RETURN (interest)

```

END function computeInterest

Assume we call the function:

```

computeInterest(p, r, t)

```

Which of the following statements must always be true about the function computeInterest?

- The initial value of *r* inside computeInterest will be the same as the value of *r*.
- If we call the function computeInterest without providing a value for *time*, the function will use a default value for *t*.
- Changing the value of *p* inside computeInterest will also change the value of principal.
- All the parameter values in computeInterest would be the same if we called computeInterest(*time*, *rate*, *principal*).

16) Given the completed function:

```

computeInterest(p, r, t)
// p, r, and t are value parameters

```

```

interest = p * r * t
RETURN (interest)
WRITE("Interest = ", interest)

```

END function computeInterest

Which of the following statements must always be true about the function computeInterest?

- The function will display the interest, and then return it.
- The function will return the interest, and then attempt to display it, generating an error.
- The function will not display any output.
- If a variable named interest already existed when computeInterest was called, that variable would be updated when computeInterest completed.

Version Date: 02/07/10

Page 9 of 14

Version Date: 02/07/10

Page 10 of 14

Figure O-5. Computer concepts survey, pages 9 and 10.

APPENDIX O: COMPUTER CONCEPTS SURVEY FOR MAIN STUDY

17) Given the program main and its function interchange:

```

interchange(firstNum, secondNum)
// firstNum and secondNum are value parameters
1
tempVal = firstNum
firstNum = secondNum
secondNum = tempVal
5
END function interchange

10
main()
15
firstNum = 8
secondNum = 11
interchange(firstNum, secondNum)
20
END program main

```

What are the values of the firstNum and secondNum variables when program execution reaches lines 5 and 15, respectively?

- line 5: firstNum = 11, secondNum = 8
line 15: firstNum = 11, secondNum = 8
- line 5: firstNum = 11, secondNum = 8
line 15: firstNum = 8, secondNum = 11
- line 5: firstNum = 8, secondNum = 11
line 15: firstNum = 11, secondNum = 8
- line 5: firstNum = 8, secondNum = 11
line 15: firstNum = 8, secondNum = 11

18) Assume that you have been asked to design and implement a function that takes an array as a parameter. Which of the following do software engineers recommend that you include in your test cases to insure that your function works as intended?

- Arrays that contain no values
- Arrays that contain anywhere from one to the maximum number of values
- Values in the first, middle, and last elements of arrays

- i and ii only
- iii only
- ii only
- i, ii, and iii

19) Which of the following statements about recursion is always true?

- A function that implements a recursive algorithm consists of two parts: a base case and a recursive step.
- Recursion, unlike iteration, can never occur infinitely.
- Any programming problem that can be solved either recursively or iteratively can be solved more efficiently using recursion.
- Any programming problem that can be solved recursively can also be solved iteratively.

20) Consider the following recursive function that accepts a single integer parameter:

```

Mystery(n)
// n is a value parameter
IF (n = 4) THEN
RETURN (2)
ELSE
RETURN (2 * Mystery(n - 1))
END IF
END function Mystery

```

What is the value returned by the function call Mystery(2)?

- 8
- 4
- 16
- 2

Version Date: 02/07/10

Page 11 of 14

Figure O-6.Computer concepts survey, pages 11 and 12.

APPENDIX O: COMPUTER CONCEPTS SURVEY FOR MAIN STUDY

- 21) The factorial of a nonnegative integer n , written $n!$, is the product $n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$, with $1!$ equal to 1, and $0!$ defined to be 1. Factorial can be defined recursively by observing that this is equivalent to $n! = n \cdot (n-1)!$ for $n >= 1$.

A partial implementation of recursive function `Factorial1` appears below. Which code fragment correctly completes the recursive step?

```
Factorial(n)
// n is a value parameter
IF (n <= 1) THEN
    RETURN 1
ELSE
    recursive step
END IF
END function Factorial

a) recursive step = n * Factorial(n - 1)
b) recursive step = RETURN (n * Factorial(n - 1))
c) recursive step = Factorial(n - 1)
d) recursive step = RETURN (Factorial(n - 1))
```

- 22) Assume that we have a class `Student` with private attributes `name`, `major`, and `GPA`, and public methods `getName`, `getMajor`, and `getGPA`. Also assume that we have a separate class `Professor` with private attributes `name` and `subject`.

Which of the following statements must always be true?

- Since `Student` has a method `getName`, `Professor` cannot have a method `getName`.
- Since `Student` and `Professor` both have an attribute `name`, the value of `name` for a `Student` object must always be different than the value of `name` for a `Professor` object.
- A `Professor` object can access a `Student` object's attributes.
- `Student` objects and `Professor` objects have different attributes and different methods.

- 23) Assume that a public method `display` that prints the values of the attributes has been added to class `Student` (question 22).

Which of the following statements must always be true?

- `display` can be called to print attribute values simply by typing its name.
- `display` will print the attribute values for all `Student` objects that exist when it is called.
- `display` will print the attribute values for the `Student` object that calls it.
- `display` had to be implemented from scratch because it could not use the other class methods to accomplish its task.

Version Date: 02/07/10

Page 13 of 14

- 24) Assume that we have two instances of class `Student` (question 22), `George` and `Jane`. Which code sequence will most likely produce the output shown below?

```
Student: George
Major: Biology
GPA: 3.0

a) WRITE("Student: ", name)
   WRITE("Major: ", major)
   WRITE("GPA: ", GPA)
b) WRITE("Student: ", George.getName)
   WRITE("Major: ", George.getMajor)
   WRITE("GPA: ", George.getGPA)
c) WRITE("Student: ", George.getName())
   WRITE("Major: ", George.getMajor())
   WRITE("GPA: ", George.getGPA())
d) WRITE("Student: ", getName())
   WRITE("Major: ", getMajor())
   WRITE("GPA: ", getGPA())
```

- 25) Please indicate your impression of the difficulty level of this computer concepts survey by making a selection from the categories shown below.

- High difficulty
- Medium to high difficulty
- Median difficulty
- Low to medium difficulty
- Low difficulty

Version Date: 02/07/10

Page 14 of 14

APPENDIX P: COMPUTER CONCEPTS SURVEY REVIEW FORM

Project Title: An Investigation of the Impact of Disparate Teaching Approaches on Student Learning of Introductory Programming Concepts

Project Investigators: Robert B. Allen, PI; Wanda M. Kunkle, Co-PI

Computer Concepts Survey Review Form

The purpose of this form is to obtain expert review of the computer concepts survey tool used in the study named above.

Instructions
Please enter your initials in the box below:

For each of the questions on the computer concepts survey:

1. Record your letter response to the question.
2. Decide whether the question reflects fundamental programming concepts that students should know after completing an introductory course in object-oriented programming. Indicate your response by placing an X in the appropriate box.
3. Rate the quality of the question using the scale provided. ("Quality" here refers to how well the question is written. For example: Is the content accurate? Is it clear what the question is asking? Is there anything about the pseudocode that you find especially confusing?) Indicate your response by placing an X in the appropriate box.

Question		Reflects basic concepts		Question quality				This column is for any optional comments you wish to make. For example, if you think a question should be changed, briefly indicate why and how. Comments (optional)
Number	Answer	Yes	No	Do not use again	Use again with major changes	Use again with minor changes	Use again as is	
1.								
2.								

Page 1 of 3

Figure P-1. Computer concepts survey review form, page 1.

APPENDIX P: COMPUTER CONCEPTS SURVEY REVIEW FORM

Question		Reflects basic concepts		Question quality				This column is for any optional comments you wish to make. For example, if you think a question should be changed, briefly indicate why and how. Comments (optional)
Number	Answer	Yes	No	Do not use again	Use again with major changes	Use again with minor changes	Use again as is	
3.								
4.								
5.								
6.								
7.								
8.								
9.								
10.								
11.								
12.								
13.								
14.								
15.								

Page 2 of 3

Figure P-2. Computer concepts survey review form, page 2.

APPENDIX P: COMPUTER CONCEPTS SURVEY REVIEW FORM

Question		Reflects basic concepts		Question quality				This column is for any optional comments you wish to make. For example, if you think a question should be changed, briefly indicate why and how. Comments (optional)
Number	Answer	Yes	No	Do not use again	Use again with major changes	Use again with minor changes	Use again as is	
16.								
17.								
18.								
19.								
20.								
21.								
22.								
23.								
24.								
25.		n/a	n/a					

Additional comments about the computer concepts tool you wish to add (optional):

Page 3 of 3

Figure P-3. Computer concepts survey review form, page 3.

VITA

WANDA M. KUNKLE

- EDUCATION:** Master of Science in Computer Science
West Chester University, West Chester, PA
May 1995
- Master of Music in Piano Pedagogy
West Chester University, West Chester, PA
December 1989
- Thesis:
An Analysis and Pedagogical Discussion of Selected Piano Pieces for the Left Hand Alone
- Bachelor of Music in Music Education/Mathematics (Secondary Education)
Marywood University, Magna Cum Laude, Scranton, PA
May 1979
- COLLEGE TEACHING EXPERIENCE:** COMPUTER SCIENCE INSTRUCTOR
Department of Computer Science, Rowan University, Glassboro, NJ -
September 1996 to the present
Teach Compiler Design, Programming Languages, Data Structures & Algorithms,
Java for Object-Oriented Programmers, Computer Science & Programming,
Introduction to Programming, Computer Organization, Computing
Environments, Computer Literacy.
- MATHEMATICS INSTRUCTOR
Department of Mathematics, Drexel University, Philadelphia, PA –
September 2000 to the present
Teach Introduction to Analysis I & II, Discrete Mathematics, Pre-Calculus.
- PRESENTATIONS:** “A Web-Based Integral Evaluator: A Demonstration of the Successful
Integration of WebEQ, Maple, and Java”
Poster/demonstration presented at the MathML International Conference
2002: MathML and Technologies for Math on the Web, June 2002
- HONORS:** Member of: Kappa Delta Pi (National Honor Society in Education),
Delta Epsilon Sigma (National Catholic Scholastic Honor Society),
Kappa Mu Epsilon (National Mathematics Honor Society),
Upsilon Pi Epsilon (International Honor Society in the Computing and
Information Disciplines),
Montclair Who’s Who in Collegiate Faculty (2006-2007)
- PROFESSIONAL AFFILIATIONS:** Member of: ACM,
ACM Special Interest Group on Computer Science Education (SIGCSE),
ACM Special Interest Group on Computer-Human Interaction (SIGCHI),
Computer Science Teachers Association (CSTA),
IEEE Computer Society