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

WANDA M. KUNKLE, Penn State Harrisburg
ROBERT B. ALLEN, Yonsei University

Learning to program, especially in the object-oriented paradigm, is a difficult undertaking for many students. 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, determining the effectiveness of these interventions poses a problem. The research presented here developed an instrument to assess student learning of fundamental and object-oriented programming concepts, then used that instrument to investigate the impact of different teaching approaches and languages on university students' ability to learn those concepts. Extensive data analysis showed that the instrument performed well overall. Reliability of the assessment tool was statistically satisfactory and content validity was supported by intrinsic characteristics, question response analysis, and expert review. Preliminary support for construct validity was provided through exploratory factor analysis. Three components that at least partly represented the construct "understanding of fundamental programming concepts" were identified: methods and functions, mathematical and logical expressions, and control structures. Analysis revealed significant differences in student performance based on instructional language and approach. The analyses showed differences on the overall score and questions involving assignment, mathematical and logical expressions, and code completion. Instructional language and approach did not appear to affect student performance on questions addressing object-oriented concepts.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques; K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms: Languages

Additional Key Words and Phrases: Education

ACM Reference Format:
Wanda M. Kunkle and Robert B. Allen. 2016. The impact of different teaching approaches and languages on student learning of introductory programming concepts. ACM Trans. Comput. Educ. 16, 1, Article 3 (January 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2785807

1. INTRODUCTION

Many students experience difficulties when learning to program. They find learning to program in the object-oriented paradigm particularly challenging. In an effort to help students learn complex object-oriented concepts, 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

Authors' addresses: W. M. Kunkle, Department of Computer Science and Mathematical Sciences, Penn State Harrisburg, 777 West Harrisburg Pike, Middletown, PA 17057; R. B. Allen, Department of Library and Information Science, Yonsei University, 50 Yonsei-ro, Seodaemun-gu, Seoul 120-749, Korea.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 1946-6226/2016/01-ART3 $15.00
DOI: http://dx.doi.org/10.1145/2785807

low-level approaches that focus on writing source code. They have also experimented with different programming languages. Some of the languages that have been used to introduce students to object-oriented programming include C++, Java, Python, and Visual Basic. However, having tried these different instructional techniques, computing educators are faced with yet another dilemma: how to tell if any of these interventions actually work.

The research presented here was motivated by an interest in improving practices in computer science education in general and improving our own practices as computer science educators 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.

The article begins by examining current approaches to teaching introductory object-oriented programming and reviewing the novice programmer literature. The next section discusses assessment tools for computing and related fields. The article continues by presenting the research questions and study design. Finally, results of the study are discussed with an eye toward implications for future work.

2. APPROACHES FOR TEACHING OBJECT-ORIENTED PROGRAMMING

At the time of the research reported in this work, the participating universities were teaching introductory object-oriented programming in Alice, Java, JavaScript, Python, and Visual Basic. Three examples demonstrate the diversity of approaches. Two focus on creating and manipulating classes and objects, whereas the third focuses on writing source code. The first represents an "objects-first" approach supported by the BlueJ environment. The second represents an "objects-early" approach supported by the Alice environment. The third represents a more traditional "imperative-first" approach using Python.

2.1. Objects-First Approach Using BlueJ

The objects-first approach introduces students to complex object-oriented concepts at the start of a course in object-oriented programming. Students become acquainted with objects and their methods by visualizing and interacting with them. Only after they have gained sufficient experience creating and manipulating objects are students introduced to the source code used to create them. Fundamental programming concepts such as control structures are introduced within the context of object-oriented concepts.

BlueJ is an easy-to-use integrated development environment (IDE) that supports the objects-first approach [BlueJ 2005; Barnes and Kölling 2005; Kölling et al. 2003]. It was designed specifically for teaching and learning introductory object-oriented programming in Java. Classes and objects are represented using UML-like diagrams. Students interact with the diagrams using dialogue boxes and popup menus. Templates support code generation.

2.2. Objects-Early Approach Using Alice

The objects-first and objects-early approaches differ in the amount of time that elapses before students are introduced to source code. However, the terms *objects-first* and *objects-early* are not clearly defined and are used differently by different people. In the context of this work, the reader may assume that both approaches involve introducing students to objects in the first few weeks of a one-term course. Objects-early is defined here to mean introducing them to the code for objects sooner than for objects-first.

Alice is a three-dimensional (3D) graphics programming environment that supports the objects-early approach [Alice v3.0 2010; Dann et al. 2006]. It was designed to enable students with no 3D graphics or programming experience to build virtual worlds.

Students create programs, or *worlds*, as they are called in Alice, by dragging and dropping objects, their associated methods, and standard control structures into the appropriate editor. Code is generated, not typed.

2.3. Imperative-First Approach Using Python

The imperative-first approach introduces students to imperative (procedural) aspects of object-oriented languages first and object-oriented aspects of object-oriented languages last. Imperative features include logical expressions, control structures, and functions. Object-oriented features include classes, objects, and methods. The imperative-first approach focuses on learning to write source code, which is the traditional approach to teaching programming.

Python is one of a variety of languages used to teach object-oriented programming using the imperative-first approach (Microsoft Visual Basic and C++ are others) [Python 2007; Grandell et al. 2006]. Features that make Python particularly suitable for beginning programmers are its simple syntax and interactive mode for experimenting with code. Supporting environments typically consist of a text editor and built-in tools for compiling, debugging, and running programs.

3. REVIEW OF NOVICE PROGRAMMER STUDIES

Many studies have been conducted for the purpose of exploring why students have difficulty when learning to program. The studies reviewed here address programming in different paradigms and languages, learning fundamental concepts such as iteration and recursion, learning object-oriented concepts, and identifying student misconceptions of programming. They were chosen specifically to demonstrate the wide range of novice programmer studies that have been carried out over the years. The papers are organized by category, except for the first one, which is itself a review.

Robins et al. [2003] reviewed the literature on studies of programming from a psychological/educational perspective. The primary focus of the review was on learning and teaching introductory programming in procedural and object-oriented styles. Topics related to program generation were especially emphasized. Of particular relevance to instructors of follow-up courses is the authors' observation that "many students make very little progress in a first programming course." In the closing section, the authors stressed the importance of spending more time helping novices develop strategies for creating programs and less time demonstrating and explaining completed sample programs.

3.1. Procedural and Functional Languages

Early studies focused on novices learning to program in procedural and functional languages.

Soloway et al. [1989] explored the relationship between the strategies that 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 ith element on the ith pass through the loop, as opposed to a strategy that processes the ith element and reads the $(i + 1)^{st}$ element on the ith pass through the loop. The authors 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.

3.2. Object-Oriented Languages

More recent studies focused on novices learning to program in object-oriented languages.

Cooper et al. [2003a] discussed their new approach to teaching objects-first in introductory programming courses using Alice [2010] and presented some preliminary findings. The researchers 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 et al. 2003b]. Finally, course exit evaluations indicated that students exhibited confidence in their programming abilities.

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 versus object, instantiation and constructors, simple versus composed classes, and program flow. A total of 58 conceptions and difficulties were identified.

3.3. Object-Oriented Versus Procedural Languages

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.

In a more recent study, Vilner et al. [2007] examined the impact of shifting from the procedural to the object-oriented paradigm in their introductory programming course (CS1) on students' perception of fundamental concepts. They compared the performance of students enrolled in the procedural course, taught using C++, to that of students enrolled in the object-oriented course, taught using Karel J Robot [Bergin et al. 2005] and Java. The object-oriented course used Karel J Robot to introduce students to object-oriented programming, then switched to BlueJ [BlueJ 2005; Barnes and Kölling 2005; Kölling et al. 2003] to teach them Java. The researchers found no significant difference in the overall performance of the two groups of students on their respective final exams, which were comparable. However, they did find a difference in the students' perceptions of specific concepts. Whereas the procedural and object-oriented students performed similarly on the two questions addressing recursion and efficiency of algorithms, the object-oriented students outperformed the procedural students on the question addressing software design. These results suggest that the object-oriented students' semester-long experience with designing classes better prepared them to design a solution for a larger problem.

3.4. Different Programming Languages

Tew et al. [2005] compared how students instructed in different languages in a first programming course performed in a second course taught in the same language. One of the introductory courses was designed primarily for computer science majors and was taught in Python. The other introductory course was designed for engineering majors and was taught in MATLAB and Java. Both courses used an objects-early approach and covered simple data structures and the design and analysis of algorithms. The second programming course was an object-oriented course taught in Java. It covered more complex data structures, as well as introductory topics in program design and software engineering. A pre-test administered at the start of the second course revealed significant differences in students' understanding of introductory concepts such as conditionals, arrays, and sorting. A comparable post-test administered at the end of the course showed no significant differences in students' understanding of those same concepts.

3.5. Reading and Comprehension Skills

Whalley et al. [2006] investigated the reading and comprehension skills of novice programmers at three New Zealand higher education institutions. An important contribution of their work was the development of a theoretical framework for evaluating the performance of novices in computer programming tasks. The researchers used a revised version of Bloom's taxonomy [Anderson et al. 2001] to generate a set of 11 mostly multiple-choice questions that required students to perform the increasingly complex cognitive tasks of understanding, applying, and analyzing in the programming domain. They used both the Bloom taxonomy and the SOLO taxonomy [Biggs and Collis 1982] to analyze students' responses to the questions. Four SOLO categories, ranging from low to high (prestructural, unistructural, multistructural, relational), were developed to analyze responses to a single short-answer question with respect to knowledge of programming constructs and width (amount of code considered) and depth of understanding.

Data analysis revealed that 60% or more of the students answered the multiple-choice questions at the two lower levels of cognitive processing (understand, apply) correctly. Fewer than 40% answered the multiple-choice questions at the analyze level correctly. For the short-answer question that required students to explain what a piece of code does, about 55% provided a line-by-line description (multistructural response), but only 30% summarized the code's purpose (relational response). This finding is significant because it is the authors' contention that to write code, a student needs to be able to read code and describe it relationally.

3.6. Student Performance and Teacher Expectations

Utting et al. [2013] examined students' performance on a practical task and survey and compared it to their teachers' expectations of their performance. A two-part assessment consisting of a CS1 concept assessment and a programming skill assessment was devised. The FCS1 Assessment Instrument [Tew and Guzdial 2011] was used for the concept assessment. A simple programming task that required students to complete the implementation of a Time class representing a 24-hour clock was used for the skill assessment. To compare students' performance to their teachers' expectations, teachers were asked to predict how their students would perform on each part of the assessment. Once the results of the assessment were known, the teachers were asked to reflect on how well their predictions matched their students' actual performance.

Analysis of the data gathered by the researchers showed a strong, positive correlation between students' scores on the concept assessment and the skill assessment. Students

answered an average of 42% of the 27 questions on the FCS1 Assessment Instrument correctly. Results of the clock task were analyzed by group, based on whether or not students had been provided a test harness (main method with test cases). Those with the test harness completed an average of 3.26 of the 4 required methods. Those without the test harness completed an average of 0.83 of the 4 methods. Taken together, these results yielded a clock task completion rate of 2.72 out of 4 methods.

With respect to students' overall performance compared to their teachers' expectations, Utting et al. [2013] found that two-thirds of the teachers felt that their students had performed as they had predicted they would.

3.7. Programming Misconceptions

Clancy [2004] and Soloway and Spohrer [1989] are excellent resources for learning about the misconceptions that beginning programmers commonly hold and the kinds of errors they typically make. They provide many examples of research addressing misconceptions related to procedural and object-oriented programming. Summaries of several of these studies follow.

Bonar and Soloway [1989] and Putnam et al. [1989] identified misconceptions related specifically to procedural programming. 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).

Fleury [2000] 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 Javaclasses 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.

A review of research on programming misconceptions more recent than Clancy [2004] can be found in the Ph.D. dissertation by Sorva [2012]. He provides a short review in Section 3.4 of the comprehensive list of misconceptions catalogued in Appendix A of the dissertation. Two studies dealing with mental models are of particular interest. Vainio [2006] interviewed students to elicit their mental models of fundamental programming concepts. He found that some of their misconceptions had to do with types, such as the idea that value types can change on the fly in Java. Sajaniemi et al. [2008] had students record their perceptions of the state of Java programs at specific stages of execution to elicit their mental models of program state. They identified a number of misconceptions related to objects and method calls.

Finally, a recent study by Chen et al. [2012] unveiled both misconceptions and missing conceptions of novice Java programmers. (The authors define missing conceptions as "knowledge that students have either not retained or never learned.") One misconception was failing to recognize that a class is just a "big type" that can be used to

define objects. One missing conception was not understanding how the for-each and for-loop constructs execute internally.

4. ASSESSMENT IN MATHEMATICS, SCIENCE, ENGINEERING, AND COMPUTING

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 pre-college 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 et al. 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 [Concept Inventories 2008]. Assessment tools are also under development in biology, calculus, and chemistry, to name but a few [Assessment Instruments 2016].

4.2. Computing

Concept inventories 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 although 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. Two organizations that have taken steps toward standardization at the pre-college level are the International Technology and Engineering Educators Association (ITEEA) and the International Society for Technology in Education (ISTE). (The ITEEA is the former International Technology Education Association (ITEA), which was renamed in March 2010.) In April 2000, the ITEA and its Technology for All Americans Project (TfAAP) published *Standards for Technological Literacy: Content for the Study of Technology* [ITEA 2007]. This document defines technological literacy and articulates the knowledge and abilities needed by pre-college students to become technologically literate. It does not provide a curriculum, however. ISTE publishes the two-page ISTE Standards for Students [ISTE 2007], which outlines six areas in which pre-college students should possess technological competency. These include understanding technology concepts, systems, and operations, among others.

Despite a lack of concept inventories, assessment tools for computing do exist. The Test Collection at the Educational Testing Service (ETS), a database of more than 25,000 tests and other measurement devices created by authors from the United States, Canada, the United Kingdom, and Australia, contains more than 100 tests with the search term comput* in the title field [Test Link 2013]. However, based on their descriptions, none seems to be appropriate for assessing fundamental programming knowledge.

Until fairly recently, little or no work had been done for the purpose of developing concept inventories for computer science. Goldman et al. and Tew and Guzdial have undertaken research to remedy the situation [Concept Inventories 2010; Goldman et al. 2008; Tew and Guzdial 2010].

Zilles leads the multi-institution project reported on in Goldman et al. to develop concept inventories for three introductory computer science subjects: discrete math,

digital logic design, and programming fundamentals [Concept Inventories 2010]. The development process for each inventory consists of four steps:

—*Step 1*: Use a Delphi process to identify important and difficult concepts.
—*Step 2*: Use student interviews to identify common misconceptions about the concepts identified in Step 1.
—*Step 3*: Use the misconceptions identified in Step 2 to design questions for the inventory.
—*Step 4*: Use field tests to gather data to validate the inventory.

Herman et al. [2010] described 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. Kaczmarczyk et al. [2010] reported the results of the second phase in the development of the Programming Fundamentals Concept Inventory: identifying student misconceptions. Student interviews revealed four distinct categories of misconceptions involving memory usage, while loops, objects, and code tracing. The paper details three specific misconceptions within the memory usage category. Results of the first phase in the development of the Discrete Math Concept Inventory were reported by Goldman et al. [2008].

In March 2010, Tew and Guzdial [2010] reported on their progress in creating a language-independent CS1 assessment instrument. They stated in the closing section of the paper that they had completed the test specification (conceptual content, question format, scoring procedures), developed the bank of questions, and were ready to proceed with pilot studies. In her dissertation, Tew [2010] reported that she had succeeded in constructing and validating a tool for assessing university-level students' understanding of introductory computing concepts specifically for procedurally based introductory computing courses taught in Java, MATLAB, or Python. The tool was recently used in the study by Utting et al. [2013], which is reviewed in Section 3.6.

Finally, Sheard et al. [2011] investigated the characteristics of a sampling of examination instruments used for summative assessment of students in introductory programming courses. A project working team developed and refined a scheme that classifies questions that make up an introductory programming exam based on weight of question, topics covered, skill required to answer question (e.g., write code), style of question (e.g., multiple choice), number of possible answers (e.g., one vs. many), cultural references, and degree of difficulty (low, medium, or high). They then piloted the scheme on 11 exams from eight universities in five countries. Results of the pilot analysis showed considerable variation in topics covered as well as in overall exam characteristics. Such disparities point out the inherent difficulties in creating computing concepts inventories for general use.

5. RESEARCH QUESTIONS

Explorations into the different teaching approaches and programming languages currently being used to teach introductory object-oriented programming led to questions regarding the impact of these approaches and languages on students' ability to learn both fundamental and object-oriented programming concepts. One type of tool that can be used to assess these abilities is a computer concepts inventory. Since no such tool was available at the time of this research, one was developed. The following research questions were investigated:

—*RQ 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?

—*RQ 2*: What effect does the teaching approach and/or programming language have on students' ability to grasp fundamental programming concepts (assignment, iteration, functional decomposition, program flow, etc.)?
—*RQ 3*: What effect does the teaching approach and/or programming language have on students' ability to grasp specific object-oriented programming concepts (class creation, method invocation, object instantiation, etc.)?
—*RQ 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. EXPERIMENTAL STUDY

6.1. Purpose

This research study examined the performance of university students enrolled in first-year programming courses for the purpose of investigating the research questions listed earlier. It was specifically designed to explore the impact of different teaching approaches and/or languages on students' grasp of fundamental and object-oriented programming concepts using the assessment tool developed for the study.

6.2. Design

The study employed a one-factor within-subjects design with three levels. The factor was programming language/teaching approach. The three levels were three different programming languages: C++, Java, and Visual Basic. The subjects were 61 university students enrolled in first-year programming courses teaching these three languages. We recruited the subjects with the help of their instructors, as will be explained in the next section.

6.3. Participants

The 61 participants were recruited from two different universities in the Mid-Atlantic region during the 2009–2010 academic year. Students in particular courses were targeted because of the level of the course and the programming language in which it was taught. Specifically, we sought to recruit students taking first-year programming courses that employed a variety of languages and teaching approaches. We also sought to recruit entire classes whenever possible to ensure a wide range of abilities and a diversity of majors among student participants. To realize these goals, we contacted the instructors of the courses of interest, who then advertised the study on our behalf or who graciously invited the first author into their classes so that their students, if willing, could complete our surveys.

The courses from which the students were recruited either required or recommended previous programming experience. Although the prerequisites were not always enforced, it was clear from student responses to the demographic survey that most had programmed before. Languages previously studied included Alice, Basic, C, C++, Java, JavaScript, Python, and Visual Basic [Kunkle 2010].

The specific courses were two different Java courses, three different C++ courses, and one Visual Basic course (two sections). The topics covered in the courses were comparable, namely fundamentals (basics, conditionals, loops, functions, etc.) and object-oriented concepts, but the teaching approaches were different. Both Java courses were taught using the objects-first approach supported by the BlueJ IDE. The course from which most of the C++ students were recruited was taught using the imperative-first approach. The remaining C++ students were taught using objects-early. The supporting environment for the C++ courses and the Visual Basic course was Microsoft Visual Studio. The Visual Basic course was the second in a two-course sequence. It began

with a review of the fundamentals covered in the preceding course, then continued with object-oriented concepts and advanced features of Visual Basic, such as graphical and event-driven programming. Since a primary goal of the course was to have each student develop a small-scale enterprise system, the course was taught using a project-based approach. Assigning students to work on a term-long project was a significantly different approach than that used in the Java and C++ courses where lab projects were assigned regularly to reinforce concepts introduced in lecture. The authors wish to inform the reader, however, that the Visual Basic course that preceded the one currently being discussed did employ the more typical lecture/lab project approach.

Students who participated in the study represented a variety of majors. Fourteen were computer science majors. Twenty-one were management information systems (MIS) majors; two of the MIS majors had second majors (one in marketing and one in radio, television, and film). Nine were electrical and computer engineering majors. Ten were mechanical engineering majors. Three were mathematics majors; two of the math majors had second majors (one in physics and one in economics). Of the four remaining students, one was a software engineering major, one was a physics major, one was a liberal arts major, and one was nonmatriculated.

6.4. Task

The study consisted of two sessions. The first session was scheduled near the beginning of the term, and the second session was scheduled near the end of the term. During the first session, the students were asked to complete a consent form and three surveys:

—A demographic survey that asked them to provide information about their academic level, major, gender, ethnicity, age, and programming language experience, among others.
—An attitude survey that asked them to respond to 40 statements designed to gain insight into their attitudes toward computer science and programming.
—A computer concepts survey that asked them to answer 24 multiple-choice questions designed to assess their knowledge of computer programming concepts.

During the second session, the students were asked once again to complete the attitude and computer concepts surveys for purposes of comparison. Each student received a $15.00 stipend for completing the study.

7. ASSESSMENT TOOL

7.1. Development

As noted earlier, assessment tools for computing are sorely lacking. To investigate the impact of different teaching approaches and languages on student learning, we developed our own tool by modifying an experimental tool developed at another university. A list of guidelines for the tool's development was constructed by drawing on the authors' personal experiences teaching computer science and programming, the literature on computer concepts inventories under development [Goldman et al. 2008; Tew and Guzdial 2010], and the literature on novice programmers' errors and misconceptions [Clancy 2004; Soloway and Spohrer 1989]. The resulting tool was based on the following:

—Topics generally taught in introductory programming courses
—Topics that experts agree should be taught in introductory programming courses
—Novice errors and misconceptions
—Language-independent pseudocode.

The final version consisted of 24 equally weighted questions that focused on assessing students' grasp of the following fundamental and object-oriented concepts: basics, logical expressions, conditionals, loops, arrays, functions, recursion, classes, and objects. The assessment tool is only applicable to imperative/object-oriented programming.

7.2. Performance

Extensive analysis of the data gathered showed that the assessment tool performed reasonably well at both the pre- and post-test administrations. Reliability was estimated using internal consistency measures. Internal consistency measures use information internal to the test. This method was chosen because it has been used by researchers developing similar tools at other universities [Herman et al. 2010; Nugent et al. 2006]. Two different formulas were used:

—*K-R21*: This formula uses number of test items, test mean, and variance of test scores to estimate reliability. It 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*: This formula uses number of test items, variance of each item, and variance of test scores to estimate reliability. It generally yields reliability estimates that are a little higher than those obtained using K-R21.

Separate reliability ratings were obtained for the pre-test and the post-test instead of obtaining a single, overall reliability rating. This was because 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. Table I displays the reliability statistics.

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

Content validity is supported by several different types of evidence: intrinsic characteristics, question response analysis, and expert review. The instrument is intrinsically valid since its questions are 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 produce them. For that reason, we 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, which was 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, whereas 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 would seem to be 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.

Table I. Reliability Statistics for the Pre-Test and Post-Test

	K-R21	Cronbach's Alpha
Pre-Test	0.57	0.65
Post-Test	0.75	0.79

Note: 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].

To collect additional evidence in support of question response validity, we examined the performance of the individual questions for both the pre-test and the post-test. Analysis of the item discrimination values for each question indicated that performance was good overall. 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 faculty who reviewed the assessment tool provides additional support for its validity. Several computer science 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; and (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 was 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 seven components, three of which were readily comprehensible. These three factors were labeled "methods and functions," "mathematical and logical expressions," and "control structures." The factors and their question loadings appear in Table II. Their identification suggests that the assessment tool has the potential to effectively represent the construct "understanding of fundamental programming concepts," but that more work needs to be done to achieve that goal.

8. STUDENT PERFORMANCE

8.1. Score Distributions

Figure 1 displays the pre- and post-test score distributions for the 61 students who participated in the study [Kunkle 2010]. Although the difference between the means did not appear to be significant, it was important to determine this definitively. 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: $t(60) = 1.65$, $p = 0.11$, $\alpha = .05$. The importance of this finding will become apparent in subsequent sections that examine student performance by teaching language and approach.

8.2. Impact of Instructional Language and Approach on Student Learning in General

One of the goals of this research was to determine the possible impact of different teaching languages (Java, C++, etc.) and approaches (objects-first, imperative-first,

Table II. Factors Identified through Principal Component Analysis

Question	Factor		
	Methods and Functions	*Mathematical and Logical Expressions*	*Control Structures*
Q 16	.733	0.65	
Q 24	.722	0.79	
Q 23	.721		
Q 20	.632		
Q 2		.850	
Q 3		.769	
Q 5		.593	
Q 11			.769
Q 7			.674
Q 9			.511

Note: The question numbers refer to the questions in the actual computer concepts survey, not the sample questions included in the appendix.

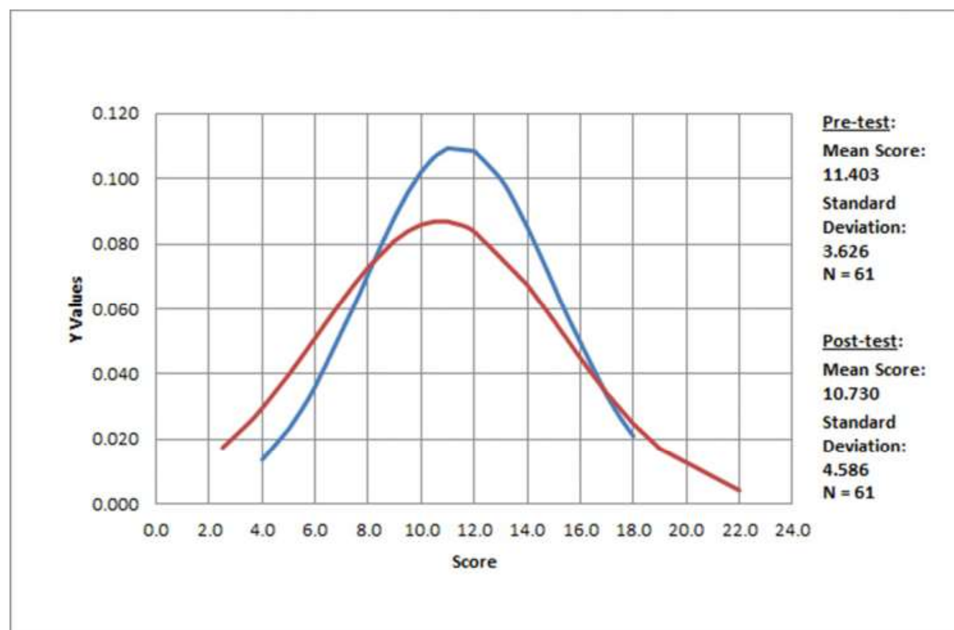

Fig. 1 Session 1 (pre-test) and Session 2 (post-test) scores for the 61 students who completed both parts of the study.

etc.) on students' overall learning of object-oriented programming. The next step was to analyze students' performance based on the type of programming course they were taking when they participated in this study. Since all of the Java students were taught using an objects-first approach, all of the Visual Basic students were taught using a project-based approach, and all but a handful of the C++ students were taught using an imperative-first approach, it seemed appropriate to divide the students into three groups based on language of instruction. Our decision to keep a particular language paired with the approach used to teach it was not motivated by a conviction that the two should be treated as a unit (i.e., a single construct), but rather by our inability to neatly separate the two. Thus, for the purpose of discussing the study results, each of

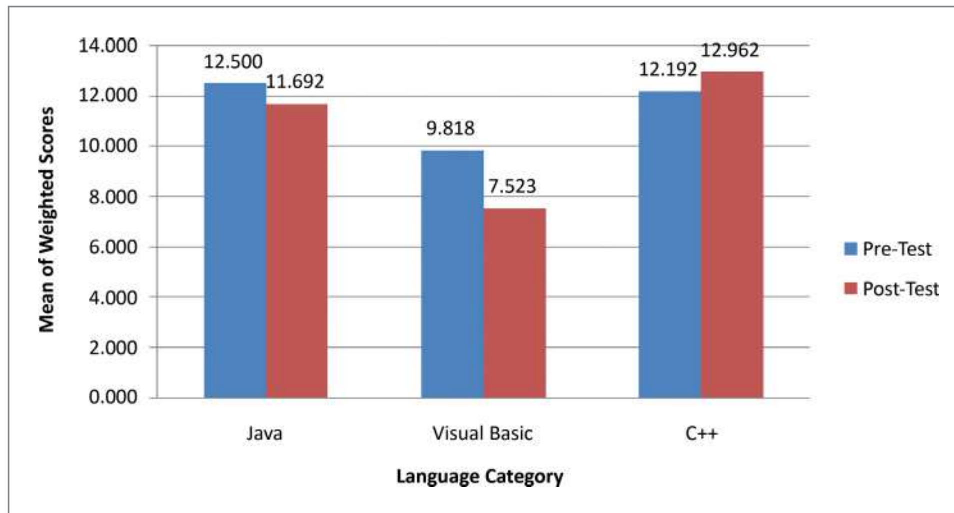

Fig. 2. Mean scores for Session 1 (pre-test) and Session 2 (post-test) by language group where Java $N = 13$, Visual Basic $N = 22$, C++ $N = 26$, and total $N = 61$.

the three groups will henceforth be referred to as the Java students, the Visual Basic students, and the C++ students.

The mean scores for each group were calculated and compared. Figure 2 displays the pre-test and post-test means broken down by language category. The Java students and the C++ students performed similarly on both the pre- and post-tests. The Visual Basic students, onthe other hand, consistently performed more poorly than the other two groups of students.

A one-factor, within-subjects analysis of variance (ANOVA) revealed a significant effect of instructional language and approach on the repeated measure, score: $F(2, 58) = 8.58$, $p < 0.01$, $MS_{error} = 23.23$, $\alpha = 0.05$. The Tukey test confirmed 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 was not significant.

At this point, some readers may be tempted to argue that the Visual Basic course was not directly comparable to the Java and C++ courses because of its project-based approach and its emphasis on topics that supported its goal of enabling each student to develop a small-scale enterprise system that incorporated graphical user interfaces and event handling. However, the ability to implement such advanced features requires mastery of both fundamental and object-oriented concepts, making these findings all the more disturbing.

8.3. Impact of Instructional Language and Approach on Student Learning of Specific Topics

Another goal of this research was to determine the possible impact of different teaching languages and approaches on students' learning of specific topics. Research Questions 2, 3, and 4 addressed this goal. Required steps included the following:

—Group questions into categories (basics, logical expressions, conditionals, code-completion, etc.).
—Compute individual students' scores for each category for the pre- and post-tests.
—Compute the mean scores for each category for the pre- and post-tests.
—Perform one-factor, within-subjects ANOVA tests to determine if teaching language and approach affected student performance on the concepts represented by the different categories.

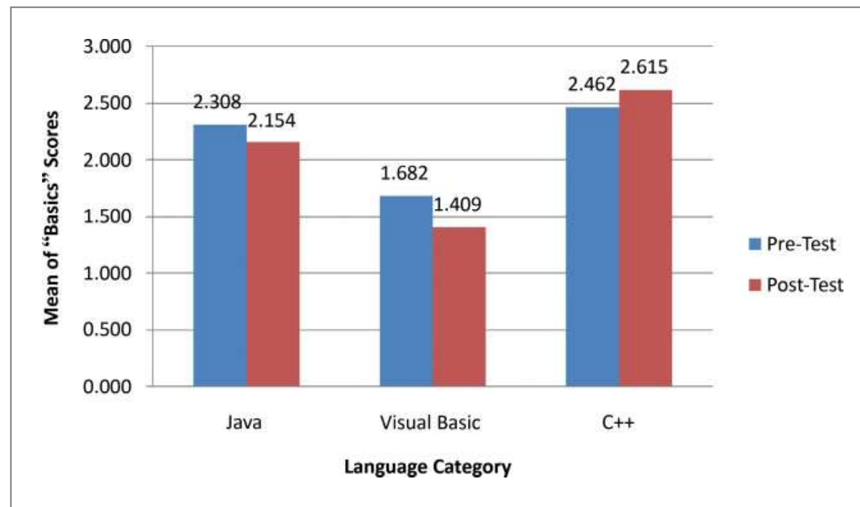

Fig. 3 Mean "basics" scores for pre-test and post-test by language group where Java $N = 13$, Visual Basic $N = 22$, C++ $N = 26$, and total $N = 61$.

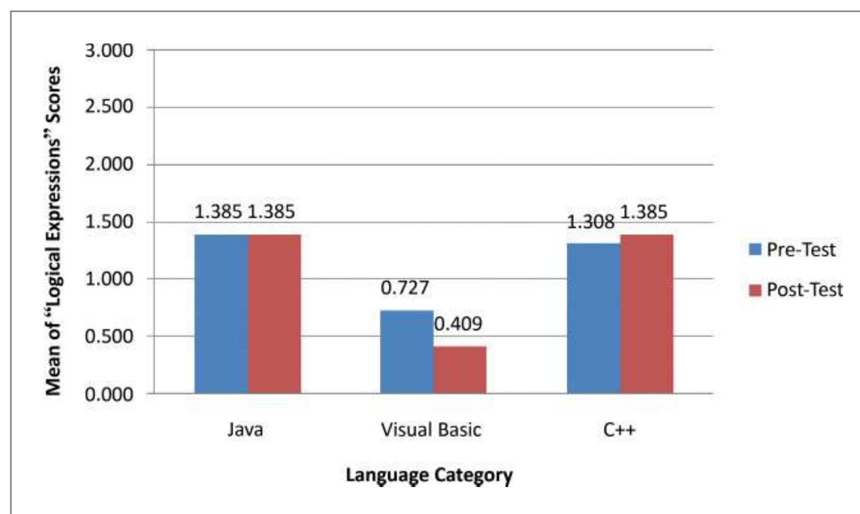

Fig. 4. Mean "logical expressions" scores for pre-test and post-test by language group where Java $N = 13$, Visual Basic $N = 22$, C++ $N = 26$, and total $N = 61$.

The second research question investigated the impact of teaching language and approach on students' ability to grasp fundamental programming concepts. Figures 3 and 4 display the pre- and post-test means for two question categories on which instructional language and approach had a significant effect: "basics" and "logical expressions."

As noted earlier, instructional language and approach had a significant effect on the repeated measure, "basics" score. A one-factor, within-subjects ANOVA yielded the following values: $F(2, 58) = 16.75$, $p < 0.01$, $MS_{error} = 0.72$, $\alpha = 0.05$. Differences at the 0.05 α-level between the Visual Basic mean and the means for Java and C++ were confirmed by the Tukey test; the difference between the Java and C++ means was not significant.

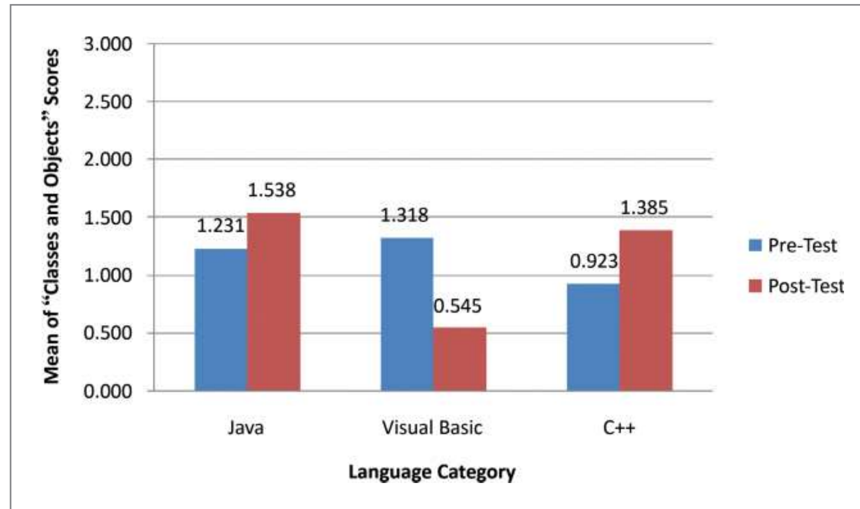

Fig. 5 Mean "classes and objects" scores for pre-test and post-test by language group where Java $N = 13$, Visual Basic $N = 22$, C++ $N = 26$, and total $N = 61$.

Instructional language and approach also had a significant effect on the repeated measure, "logical expressions" score. The values produced by a one-factor, within-subjects ANOVA were $F(2, 58) = 5.31$, $p = 0.01$, $MS_{error} = 1.66$, $\alpha = 0.05$. The Tukey test confirmed differences between the Visual Basic mean and the means for Java and C++ at the same significance level as before; the difference between the Java and C++ means was not significant.

The third and fourth research questions explored the impact of teaching language and approach on students' ability to grasp specific object-oriented programming concepts and to solve "code-completion" problems. A one-factor, within-subjects ANOVA failed to reveal a significant effect of language and approach on the repeated measure, "classes and objects" score: $F(2, 58) = 0.98$, $p = 0.38$, $MS_{error} = 1.76$, $\alpha = 0.05$. This is interesting, as the Visual Basic students and most of the C++ students had not been introduced to classes and objects until later in their respective courses (but prior to completing the post-test). Further examination of the pre- and post-test means 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. Paired samples t-tests revealed that the differences were only significant for the C++ and Visual Basic students. Figure 5 displays the pre- and post-test means for the "classes and objects" category.

Finally, the ANOVA revealed an effect of teaching language and approach for the repeated measure, "code-completion" score: $F(2, 58) = 3.60$, $p = 0.03$, $MS_{error} = 4.07$, $\alpha = 0.05$. For the latter measure, a significant difference between the Visual Basic mean and the C++ mean at the 0.05 α-level was confirmed by the Tukey test; the difference between the Java mean and the means for Visual Basic and C++ was not significant. Figure 6 displays the pre- and post-test means for the "code-completion" category.

9. DISCUSSION

Numerous authors have written about the difficulties that novices experience in learning to program. Theories have been put forth as to why this is so and solutions proposed

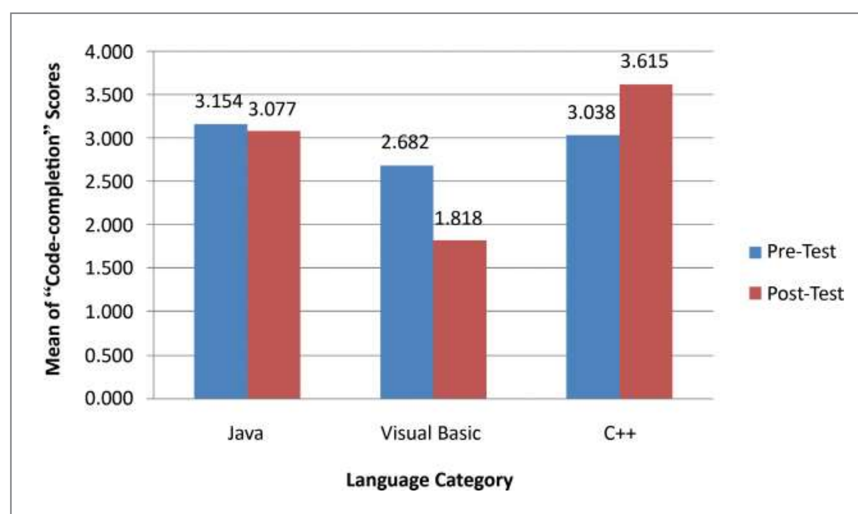

Fig. 6 Mean "code-completion" scores for pre-test and post-test by language group where Java $N = 13$, Visual Basic $N = 22$, C++ $N = 26$, total $N = 61$.

to remedy the situation. Several decades of research yielded no definitive answers, and the number of students choosing to go into computer and information science dropped steadily from 2000 to 2007. 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 6 years [Zweben 2009] and continues to rise. The 2012 Taulbee survey reported that undergraduate enrollments in U.S. computing majors rose for the fifth straight year, about 29% since the last survey [Zweben 2013]. These findings, while encouraging, suggest that it is in the best interests of the computer and information science community to actively pursue research that continues this trend.

The research presented in this article investigated the impact of different languages and teaching approaches on students' learning of fundamental and object-oriented programming concepts by developing an assessment tool expressly for that purpose. This discussion revisits the performance of the assessment instrument (RQ 1) and the findings that it yielded (RQs 2, 3, and 4). Finally, implications for computer science education and directions for future work are addressed.

9.1. Assessment Tool 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 concepts that is sufficiently "generic" to be adopted for general use?

The computer concepts assessment instrument performed better in its first trial than the authors had dared to hope. Extensive data analysis showed that it performed respectably at both the pre- 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, as discussed in Section 7.2.

Different types of evidence provided support for content validity: intrinsic characteristics, study of question responses, and expert review. At the time the research discussed in this article was carried out, the authors maintained that the assessment tool was intrinsically valid in a broad sense, as it is based on the core concepts of the programming fundamentals knowledge area defined by the 2008 ACM/IEEE curricular guidelines [CS2008 Review Taskforce 2008]. However, we have since been encouraged to acknowledge that our claim may be debatable given the lack of consistency in topics taught in introductory programming courses. Dale [2005] conducted a survey to investigate what topics were being taught and emphasized in a first course in computing. She found, for example, that although selection and repetition were covered by most of the respondents, only about half covered recursion. She also found significant differences in coverage of such object-oriented concepts as information hiding and encapsulation. Sheard et al. [2011] analyzed the content and nature of introductory programming exams. The six topics receiving the greatest coverage over all the exams were object-oriented concepts, loops, methods, arrays, program design, and selection. Recursion received no coverage at all.

As discussed in Section 7.2, question response analysis determined that on average, 71% of correct responses varied directly with the students' scores, and all possible responses were selected at least once. The problem with the distracters needs to be resolved, as at most 50% exhibited the desired inverse relationship between incorrect responses and scores. Twenty-one out of 24 questions discriminated well between high- and low-scoring students. The 3 questions that did not addressed arrays, functions, and recursion, respectively. Each will have to be improved or replaced.

Feedback from computing educators who agreed to review the assessment instrument provided additional support for its validity. The general consensus was that 19 out of 24 questions reflected basic concepts and should be used again "as is" or with "minor changes." Reviewers indicated specific changes or otherwise why questions should not be reused, where appropriate.

Construct validity is harder to demonstrate than content validity, as we pointed out in Section 7.2. Exploratory factor analysis provided preliminary support for construct validity. Seven components were extracted. Of these, three tentatively represent the construct "understanding of fundamental programming concepts": "methods and functions," "mathematical and logical expressions," and "control structures." These results suggest that the tool under development has the potential to become a valid assessment instrument. However, the fact that it has not yet been fully validated must be kept in mind when interpreting the findings regarding student performance, which are discussed in the next section.

In Section 4.2, we briefly discussed the work done by Tew [2010] in developing and validating an assessment instrument for introductory computing concepts that is suitable for procedurally based introductory computing courses taught in Java, MATLAB, or Python. Our work is similar, inasmuch as we also developed a language independent tool for assessing student knowledge of fundamental programming concepts. However, our tool includes object-oriented concepts, and it was tested on students learning to program in Java, C++, and Visual Basic. Furthermore, our tool has yet to be validated. Based on descriptions of Tew's assessment instrument [Tew 2010; Tew and Guzdial 2010], it appears to be composed of multiple-choice questions that ask students to perform such tasks as trace code and complete code. Our tool is similarly composed, but it also includes a final question that asks students to rate the difficulty level of the tool as low, low to medium, medium, medium to high, or high.

9.2. Student Performance

Research question 2. What effect does the teaching approach and/or programming language have on students' ability to grasp fundamental programming concepts (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 (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?

Statistical analysis showed that teaching language and approach affected student performance overall and on questions addressing basics, logical expressions, and code completion (RQ 2, RQ 4). Specifically, the one-factor, within-subjects ANOVA revealed a significant effect of instructional language and approach on the following repeated measures:

—*Overall score*: $F(2, 58) = 8.58$, $p < 0.01$, $MS_{error} = 23.23$, $\alpha = 0.05$
—*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*: $F(2, 58) = 3.60$, $p = 0.03$, $MS_{error} = 4.07$, $\alpha = 0.05$.

The one-factor, within-subjects ANOVA failed to reveal an effect of teaching language and approach on student performance with respect to questions focusing on object-oriented concepts (RQ 3).

Overall, student performance with respect to instructional language and approach 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 less than ideal, but they are more tolerable than the third.

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

Why did the performance of the Visual Basic 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 course the students were taking when they participated in this study used a project-based approach in which each student was required to complete a semester-long project. The project was intended to cover all basic and object-oriented programming concepts but emphasized interface design and event-driven programming. A tenuous grasp of fundamental concepts in combination with the emphasis on designing interfaces and handling events might have caused the students to forget what they had learned in the previous course. Such "forgetting" could be the result of *fragile knowledge*, a term coined by Perkins and Martin [1986] to refer to knowledge that a novice programmer has forgotten, learned but not used, or learned but used inappropriately. It may also be that a semester-long project was less effective at reinforcing student learning of fundamental concepts (i.e., their fragile knowledge) than exams, quizzes, and lab assignments given regularly throughout the course.

Why was the performance of the Visual Basic students generally 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. It may also be that the interface design aspect of Visual Basic programming detracts from student learning of fundamental concepts. Although students seem to like the graphical component, the time spent learning to create graphical user interfaces might be better spent learning the fundamental programming constructs needed to implement them.

Factors such as major and motivation may have also contributed to the overall weak performance of the Visual Basic students compared to that of the Java and C++ students. All of the Java students but one were majoring in computer science. The majority of the C++ students were majoring in either electrical and computer engineering or mechanical engineering. All of the Visual Basic students but one were majoring in MIS. Some computer science educators would argue that students studying computer science and engineering typically perform better in programming courses than students studying management information systems and business. The first author has also observed that engineering majors outperform computer science majors because they seem more motivated to do well in their courses. This may explain the improved performance of the C++ students compared to the more uniform performance of the Java students.

Motivation may also partially explain the comparatively poorer performance of the Visual Basic students. As stated earlier, most of the Visual Basic students were MIS majors. MIS graduates need to know how to design, implement, and use information systems to best serve the needs of businesses and organizations. Those who aspire to become managers may be less motivated to learn programming because they intend to focus on the high-level skills of designing solutions to business problems rather than the low-level skills of implementing those solutions. Students enrolled in other majors with a computer programming requirement (i.e., mathematics) may also lack interest in programming because they doubt its usefulness in their future careers. The first author recently witnessed this behavior firsthand when she taught multiple sections of Visual Basic programming to a diverse population of non–computer science majors at another university in the Mid-Atlantic region. Informal comments made by some of the students, as well as the amount of effort those students put forth during class, clearly indicated that they did not see how taking a programming course would benefit them in their chosen field of study. Not surprisingly, many of those same students behaved as "stoppers," novice programmers who simply stop trying when they encounter programming problems that they do not know how to solve without help [Perkins et al. 1989].

Why are these findings a reason for concern in computer science education? 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.

At this point, it seems appropriate to revisit a few other papers that addressed students learning to program, some of which were reviewed in Section 3. The observation by Robins et al. [2003] that "many students make very little progress in a first programming course" seems particularly relevant in light of the student performance results discussed earlier. Although the demographic data gathered at the start of the study

indicated that most of the participants had programmed before, only the C++ students showed an overall improvement in performance. “Fragile knowledge” [Perkins and Martin 1986] provides one possible explanation for the comparatively poorer performance of the other two groups of students.

One particular area in which we identified significant performance differences was with respect to code-completion questions. Based on the results of other comparable studies that used this type of multiple-choice question for assessment purposes, it would appear that our result is not unusual. Lister et al. [2004] observed that students found “fixed-code” questions (multiple-choice questions that asked them to determine values) easier than “skeleton-code” questions (multiple-choice questions that asked them to complete code). Whalley et al. [2006] classified code-completion questions at the highest level of cognitive processing (analyze). The results of their study confirmed the difficulty level of this problem type.

It is interesting to note that in both our study and the study by Utting et al. [2013], the average student score on the concept assessment was below 50%. The average score in our study was 46% (computed by averaging the mean pre- and post-test scores); the average score in theirs was 42%. We find these results disturbing but not surprising. We suspect that Robins et al. [2003] would feel similarly.

Our work, like that of the researchers whose studies we reviewed in Section 3, arose out of a desire to improve practices in computer science education, particularly with regard to teaching novices to program. Based on the results of even the most recent studies (e.g., Utting et al. [2013]), it seems that we still have a long way to go to truly help novices grasp the concepts and skills of computer programming more easily.

10. CONCLUSIONS

The results of this research give one pause to consider the suitability of different languages and approaches for teaching introductory programming. It also gives one pause to consider the basis for requiring students of less technical majors to study programming at all. (Please note that the authors do not consider MIS to be a “less technical” major.) The differences in performance among the students who participated in the study were significant and unexpected. Those learning to program in Java and C++ consistently performed better than those learning to program in Visual Basic. Although we maintain that programming language and/or teaching approach had an effect on student performance, we recognize the existence of other factors that could have contributed to the performance differences observed in this study. These include fragile knowledge, motivation, and major—all topics discussed in some detail in Section 9.2.

Another factor that could have affected student performance is the format of the assessment tool itself. We were not privy to information regarding the format of the exams administered to the study participants in the programming courses from which they were recruited (or in the prerequisite course, in the case of the Visual Basic students). Although standardized tests such as the SAT have ensured that most students have experience with multiple-choice exams in general, it is less likely that students have experience with multiple-choice programming exams. It is therefore possible that the multiple-choice question format of the assessment tool influenced how some of the students performed on it. Although only a few students inquired about the programming language in which the tool was written, the fact that it was written in language-independent pseudocode may also have impacted the performance of some.

We acknowledge that we cannot fully explain the performance differences among the students who participated in the study. Nevertheless, the results clearly warrant paying closer attention to programming courses that serve as stepping-stones to other programming courses, regardless of language or teaching approach.

Finally, accurately determining the impact of diverse teaching approaches and languages on student learning of computer science concepts is an unsolved problem.

Significant progress was made on the assessment tool presented in this research. However, the tool must be improved and generalized. Future work to build on and expand it would be better carried out by a team of computer science researchers working at multiple universities than by researchers working at a single university. For such a tool to be accepted and used widely, it must represent the viewpoints of a diversity of computing educators and be thoroughly tested throughout its development with a wide range of students. The authors believe that the work carried out to construct the tool can make a worthy contribution to the body of research on developing assessment instruments for computer science, but only if shared with other researchers engaged in similar pursuits.

APPENDIX

This appendix contains a selection of questions used in the assessment tool that discriminated well between high- and low-scoring students, along with a brief comment on each.

Question 1

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?

i. `y = a * x * x + b * x + c`
ii. `y = x * (a * x + b) + c`
iii. `y = a * (x * x) + b * (x + c)`

(a) iii only
(b) i and ii only
(c) i and iii only
(d) i, ii, and iii

Comment. A question that tests students' ability to code simple algebraic expressions.

Question 2

Given the expression:

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

Which of the following statements must always be true?

(a) The expression returns a value that represents true or false.
(b) The expression is equivalent to `(num1 >= num2) OR (num3 >= num2)`.
(c) The expression is only false if both parenthesized expressions are false.
(d) 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)`.

Comment. A question that tests students' ability to interpret logical expressions.

Question 3

The following algorithm prompts a user to re-enter a student's grade that is not valid (outside the range of 0 to 100):

While the grade entered by the user is outside the range of 0 to 100,
   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 next?

```
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)`

Comment. A question that tests students' ability to code logical expressions.

Question 4

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.

Comment. A question that tests students' ability to interpret conditionals.

Question 5

Assume that we have a class `student` with private attributes `name`, `major`, and `GPA`, and public methods `getName`, `getMajor`, and `getGPA`. In addition, assume that we have a separate class `professor` with private attributes `name` and `subject`.

Which of the following statements must always be true?

(a) Since `student` has a method `getName`, `professor` cannot have a method `getName`.
(b) 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.
(c) A `professor` object can access a `student` object's attributes.
(d) `student` objects and `professor` objects have different attributes and different methods.

Comment. A question that tests students' understanding of classes and objects.

ACKNOWLEDGMENTS

The authors would like to thank Michael E. Atwood, Stephen C. Cooper, M. Carl Drott, and Jeffrey L. Popyack for their helpful remarks that led to the completion of the research and subsequent dissertation on which this article is based.

REFERENCES

Algebra End-of-Course Assessment. 2009. ETS Fast Facts. Retrieved December 17, 2015, from http://www.ets.org/about/fast_facts.

Alice v3.0. 2010. Alice Home Page. Retrieved December 17, 2015, from http://www.alice.org/.

Assessment Instruments and Tools. 2016. American Society for Biochemistry and Molecular Biology (ASBMB). Retrieved January 8, 2016, from https://www.asbmb.org/education/teachingstrategies/conceptinventory/.

L. W. Anderson, D. R. Krathwohl, P. W. Airasian, K. A. Cruikshank, R. E. Mayer, P. R. Pintrich, R. Raths, and M. C. Wittrock (Eds.). 2001. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Addison Wesley Longman, New York, NY.

D. J. Barnes and M. Kölling. 2005. *Objects First with Java: A Practical Introduction Using BlueJ*. Pearson Prentice Hall, London, UK.

J. Bergin, M. Stehlik, J. Roberts, and R. Pattis. 2005. *Karel J Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java*. Dream Songs Press.

J. B. Biggs and K. F. Collis. 1982. *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, New York, NY.

BlueJ. 2005. BlueJ Home Page. Retrieved December 17, 2015, from http://www.bluej.org/.

J. Bonar and E. Soloway. 1989. Preprogramming knowledge: A major source of misconceptions in novice programmers. In *Studying the Novice Programmer*, E. Soloway and J. Spohrer (Eds.). Lawrence Erlbaum Associates, Hillsdale, NJ, 325–353.

C.-L. Chen, S.-Y. Cheng, and J. M.-C. Lin. 2012. A study of misconceptions and missing conceptions of novice Java programmers. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS'12)*.

M. Clancy. 2004. Misconceptions and attitudes that interfere with learning to program. In *Computer Science Education Research*, S. Fincher and M. Petre (Eds.). Taylor & Francis Group, London, UK, 85–100.

Concept Inventories. 2008. Foundation Coalition (FC). Retrieved January 8, 2016, from http://www.foundationcoalition.org/home/keycomponents/concept/index.html.

Concept Inventories for Computer Science. 2010. Retrieved December 17, 2015, from http://zilles.cs.illinois.edu/csci.html.

S. Cooper, W. Dann, and R. Pausch. 2003a. Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'03)*. ACM, New York, NY.

S. Cooper, W. Dann, and R. Pausch. 2003b. Using animated 3D graphics to prepare novices for CS1. *Computer Science Education* 13, 1, 3–30.

CS2008 Review Taskforce. 2008. *Computer Science Curriculum 2008: An Interim Revision of CS 2001.* Association for Computing Machinery (ACM)/IEEE Computer Society. http://www.acm.org/education/curricula/ComputerScience2008.pdf.

N. Dale. 2005. Content and emphasis in CS1. *Inroads—The SIGCSE Bulletin* 37, 4, 69–73.

W. Dann, S. Cooper, and R. Pausch. 2006. *Learning to Program with Alice*. Pearson Prentice Hall, Upper Saddle River, NJ.

R. L. Ebel and D. A. Frisbie. 1991. *Essentials of Educational Measurement*. Prentice Hall, Englewood Cliffs, NJ.

A. Fleury. 2000. Programming in Java: Student-constructed rules. *ACM SIGCSE Bulletin* 32, 1, 197–201.

G. D. Garson. 2010. Statnotes: Topics in Multivariate Analysis. Retrieved December 17, 2015, from http://faculty.chass.ncsu.edu/garson/PA765/reliability.htm.

K. Goldman, P. Gross, C. Heeren, G. Herman, L. Kaczmarczyk, M. C. Loui, and C. Zilles. 2008. Identifying important and difficult concepts in introductory computing courses using a Delphi process. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'08)*. ACM, New York, NY.

L. Grandell, M. Peltomäki, R. Back, and T. Salakoski. 2006. Why complicate things? Introducing programming in high school using Python. In *Proceedings of the 8th Australian Conference on Computing Education*. 71–80.

T. M. Haladyna. 2004. *Developing and Validating Multiple-Choice Test Items*. Lawrence Erlbaum Associates, Mahwah, NJ.

G. L. Herman, M. C. Loui, and C. Zilles. 2010. Creating the Digital Logic Concept Inventory. In *Proceedings of the 41st SIGCSE Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, New York, NY.

D. Hestenes, M. Wells, and G. Swackhamer. 1992. Force Concept Inventory. *Physics Teacher* 30, 3, 141–151.

A. Hoegh and B. Moskal. 2009. Examining science and engineering students' attitudes toward computer science. In *Proceedings of the 39th ASEE/IEEE Frontiers in Education Conference*. IEEE, Los Alamitos, CA, W1G-1–W1G-6.

International Technology Education Association (ITEA). 2007. *Standards for Technological Literacy: Content for the Study of Technology*. ITEA, Reston, VA.

ISTE Standards for Students. 2007. International Society for Technology in Education (ISTE). Retrieved December 17, 2015, from http://www.iste.org/standards/iste-standards/standards-for-students.

Joint Task Force on Computing Curricula. 2001. *Computing Curricula 2001 Computer Science*. IEEE Computer Society/Association for Computing Machinery. http://www.acm.org/education/curric_vols/cc2001.pdf.

L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st SIGCSE Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, New York, NY.

C. M. Kessler and J. R. Anderson. 1989. Learning flow of control: Recursive and iterative procedures. In *Studying the Novice Programmer*, E. Soloway and J. Spohrer (Eds.). Lawrence Erlbaum Associates, Hillsdale, NJ, 229–260.

M. Kolling, B. Quig, A. Patterson, and J. Rosenberg. 2003. The BlueJ system and its pedagogy. *Computer Science Education* 13, 4, 249–268.

G. F. Kuder and M. W. Richardson. 1937. The theory of the estimation of test reliability. *Psychometrika* 2, 3, 151–160.

W. M. Kunkle. 2010. *The Impact of Different Teaching Approaches and Languages on Student Learning of Introductory Programming Concepts*. Ph.D. Dissertation. Drexel University, Philadelphia, PA.

R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4, 119–150.

G. Nugent, L. Soh, A. Samal, and J. Lang. 2006. A placement test for computer science: Design, implementation, and analysis. *Computer Science Education* 16, 1, 19–36.

D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons. 1989. Conditions of learning in novice programmers. In *Studying the Novice Programmer*, E. Soloway and J. Spohrer (Eds.). Lawrence Erlbaum Associates, Hillsdale, NJ, 261–279.

D. N. Perkins and F. Martin. 1986. Fragile knowledge and neglected strategies in novice programmers. In *Empirical Studies of Programmers, First Workshop*, E. Soloway and S. Iyengar (Eds.). Ablex, Norwood, NJ, 213–229.

R. T. Putnam, D. Sleeman, J. A. Baxter, and L. K. Kuspa. 1989. A summary of misconceptions of high-school BASIC programmers. In *Studying the Novice Programmer*, E. Soloway and J. Spohrer (Eds.). Lawrence Erlbaum Associates, Hillsdale, NJ, 301–314.

Python. 2007. Python Home Page. Retrieved December 17, 2015, from http://python.org/.

N. Ragonis and M. Ben-Ari. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education* 15, 3, 203–221.

A. Robins, J. Rountree, and N. Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education* 13, 2, 137–172.

J. Sajaniemi, M. Kuittinen, and T. Tikansalo. 2008. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. *Journal on Educational Resources in Computing* 7, 4, 1–31.

J. Sheard, Simon, A. Carbone, D. Chinn, M.-J. Laakso, T. Clear, M. De Raadt, D. D'Souza, J. Harland, R. Lister, A. Philpott, and G. Warburton. 2011. Exploring programming assessment instruments: A classification scheme for examination questions. In *Proceedings of the 2011 International Workshop on Computing Education Research*. ACM, New York, NY, 33–38.

E. Soloway, J. Bonar, and K. Ehrlich. 1989. Cognitive strategies and looping constructs: An empirical study. In *Studying the Novice Programmer*, E. Soloway and J. Spohrer (Eds.). Lawrence Erlbaum Associates, Hillsdale, NJ, 191–207.

E. Soloway and J. C. Spohrer (Eds.). 1989. *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, NJ.

J. Sorva. 2012. *Visual Program Simulation in Introductory Programming Education*. Ph.D. Dissertation. Aalto University, Helsinki, Finland.

Test Link. 2013. About the Test Collection at ETS. Retrieved December 17, 2015, from http://www.ets.org/test_link/about.

A. E. Tew. 2010. *Assessing Fundamental Introductory Computing Concept Knowledge in a Language Independent Manner*. Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA.

A. E. Tew and M. Guzdial. 2010. Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st SIGCSE Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, New York, NY.

A. E. Tew and M. Guzdial. 2011. The FCS1: A language independent assessment of CS1 knowledge. In *Proceedings of the 42nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'11)*. ACM, New York, NY.

A. E. Tew, W. M. McCracken, and M. Guzdial. 2005. Impact of alternative introductory courses on programming concept understanding. In *Proceedings of the 2005 International Workshop on Computing Education Research*. ACM, New York, NY, 25–35.

I. Utting, A. E. Tew, M. McCracken, L. Thomas, D. Bouvier, R. Frye, J. Paterson, M. Caspersen, Y. Ben-David Kolikant, J. Sorva, and T. Wilusz. 2013. A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education (ITiCSE-WGR'13)*. ACM, New York, NY.

V. Vainio. 2006. *Opiskelijoiden Mentaaliset Mallit Ohjelmien Suorituksesta Ohjelmoinnin Peruskurssilla*. Master's Thesis. University of Helsinki, Helsinki, Finland.

T. Vilner, E. Zur, and J. Gal-Ezer. 2007. Fundamental concepts of CS1: Procedural vs. object-oriented paradigm—a case study. In *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*. ACM, New York, NY.

J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. K. A. Kumar, and C. Prasad. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In *Proceedings of the 8th Australasian Computing Education Conference (ACE'06)*. 243–252.

S. Wiedenbeck and V. Ramalingam. 1999. Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies* 51, 71–87.

S. Zweben. 2009. Computing Degree and Enrollment Trends. Computing Research Association. Available at http://www.cra.org/govaffairs/blog/archives/CRATaulbeeReport-StudentEnrollment-07-08.pdf.

S. Zweben. 2013. Computing Degree and Enrollment Trends. Available at http://cra.org/govaffairs/blog/wp-content/uploads/2013/03/CRA_Taulbee_CS_Degrees_and_Enrollment_2011-12.pdf.

Received September 2012; revised February 2015; accepted May 2015