Impact of Using Tools in an Undergraduate Software Testing Course Supported by WReSTT

PETER J. CLARKE, DEBRA L. DAVIS, and RAYMOND CHANG-LAU,
Florida International University
TARIQ M. KING, Ultimate Software Group Inc.

Software continues to affect a major part of our daily lives, including the way we use our phones, home appliances, medical devices, and cars. The pervasiveness of software has led to a growing demand for software developers over the next decade. To ensure the high quality of software developed in industry, students being trained in software engineering also need to be trained on how to use testing techniques and supporting tools effectively at all levels of development.

In this article, we investigate how testing tools are used in the software project of an undergraduate testing course. We also investigate how a cyberlearning environment—the Web-Based Repository of Software Testing Tutorials (WReSTT)—is used to supplement the learning materials presented in class, particularly the tutorials on different software testing tools. The results of a study spanning three semesters of the undergraduate course suggest that (1) the use of code coverage tools motivates students to improve their test suites; (2) the number of bugs found when using coverage tools slightly increased, which is similar to the results found in the research literature; and (3) students find WReSTT to be a useful resource for learning about software testing techniques and the use of code coverage tools.

CCS Concepts: • **Social and professional topics** → **Software engineering education**; • **Software and its engineering** → *Software testing and debugging*; • **Applied computing** → *Collaborative learning*;

Additional Key Words and Phrases: Testing tools, code coverage, cyberlearning

ACM Reference format:
Peter J. Clarke, Debra L. Davis, Raymond Chang-Lau, and Tariq M. King. 2017. Impact of Using Tools in an Undergraduate Software Testing Course Supported by WReSTT. *ACM Trans. Comput. Educ.* 17, 4, Article 18 (August 2017), 28 pages.
https://doi.org/10.1145/3068324

1 INTRODUCTION

During the past decade, software has become an integral part of everyday life. Software has been incorporated in many activities, from those used for leisure, such as gaming, to those used to

This work was supported by the National Science Foundation under grants DUE-0736833 and DUE-1225742. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
Authors' addresses: P. J. Clarke, D. L. Davis, and R. Chang-Lau, School of Computing and Information Sciences, Florida International University, 11200 SW 8th Street, Miami, Florida 33199; T. M. King, Ultimate Software Group Inc., 2000 Ultimate Way, Weston, FL 33326.

https://doi.org/10.1145/3068324

improve the quality of life for persons with life-threatening diseases, such as insulin pumps for diabetics.[1] As software becomes more ubiquitous, it is important that developers use techniques to reduce the number of bugs in software during the development process. Some industry reports state that the cost of software bugs to the global economy is in excess to $300 billion per year (Britton et al. 2013).

Testing continues to be the main validation and verification (V&V) technique used during the software development process. However, many software developers are not familiar with the types of testing techniques and tools that can be used to detect bugs in software. This lack of testing knowledge is in part due to the dearth of testing concepts and skills being taught in many academic institutions (ACM/IEEE-CS Interim Review Task Force 2008), although there has been noticeable improvement in the quantity and quality of software testing tools available for use in academia (Hackett 2013).

In this article, we present studies conducted during three semesters of an undergraduate software testing course at Florida International University (FIU). The studies investigate students' use of testing tools and their interaction with a cyberlearning environment—the Web-Based Repository of Software Testing Tutorials (WReSTT) (Clarke et al. 2014b). More specifically, the studies attempted to find answers to the following questions: (1) Does the use of code coverage tools motivate students to improve their test suites during testing? (2) Does an increase in the percentage of code covered during white-box testing improve bug detection? (3) Do students find WReSTT a useful learning resource for testing techniques and tools? (4) Do students find the features in WReSTT support collaborative learning?

This article extends the work of Clarke et al. (2014a)[2] in the following ways: (1) additional details are presented regarding the undergraduate testing course, thereby providing other instructors with the ability to replicate the study at their institutions; (2) the study has been expanded to include results from three semesters (Fall 2011, 2012, 2013), including the one presented in Clarke et al. (2014a) (Fall 2012); and (3) the study in this article reports on the number of bugs found during the testing of the student projects and tries to establish a relationship with increased code coverage, unlike the study in Clarke et al. (2014a).

There are few, if any, studies that report on how students use code coverage to improve their test suites during unit, subsystem, and system testing. Many of the studies focus on code coverage to improve unit testing either in the context of assessing student programs (Aaltonen et al. 2010; Edwards 2004) or in the context of evaluating students' skills in performing testing using a unit testing tool (Carver and Kraft 2011; Edwards and Shams 2014). This article also describes how theoretical concepts, such as the subsumes relation, are realized during the practical exercise of testing a software application using currently available testing tools. Students are usually exposed to the application of these theoretical concepts using trivial programs (e.g., class examples) but are seldom exposed to the complexities of testing "real" software applications developed by other programmers. In this study, we expose students to testing programs developed by students in other classes, providing them with experiences that more closely align with experiences they will encounter in industry.

The main contributions of this article are as follows:

(1) A description of an undergraduate software testing course that provides instructors the ability to design a testing project that uses unit, functional, and code coverage tools
(2) The results of a testing project for three semesters of an undergraduate testing course that illustrates how code coverage tools may be used to improve test suites during white-box testing.

[1] http://www.webmd.com/diabetes/insulin-pump.

[2] ©2014 American Society for Engineering Education, 2014 ASEE Annual Conference, Atlanta, GA.

(3) Results showing the impact a cyberlearning environment, WReSTT, has on student learning in an undergraduate software testing course, when students are provided with access to learning content and tutorials on testing tools.

The remainder of the article is organized as follows. Section 2 introduces software testing, testing tools, and test coverage criteria. Section 3 describes aspects of the undergraduate testing course used in our study, including the pedagogical approach, course structure, course project, and cyberlearning environment (WReSTT). Section 4 presents the study by identifying the objectives and describing the methods, results, and discussion. Section 5 describes related work, and we conclude in Section 6. Appendices A and B contain documentation for the testing course and a survey used in the study, respectively.

2 BACKGROUND

In this section, we present an overview of testing concepts and a brief description of the testing tools used in the course, and introduce the subsumes relation used in software testing.

2.1 Testing Concepts

Software testing is defined as the "dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior" (Bourque and Dupuis 2004). Key to this definition is that elements of the program being executed must be observable and there must be some notion of controllability with respect to the elements of the program.

The literature on testing is presented from several perspectives, including testing levels (unit, integration, system), the view of the component under test (white-box, black-box, grey-box), the coverage criteria used to determine the effectiveness of testing (function, control flow and data flow), and the objectives of testing (regression, acceptance, alpha), among others (Ammann and Offutt 2008; Binder 1999; Bourque and Dupuis 2004; Mathur 2008; McGregor and Sykes 2001).

Black-box testing techniques generate test cases from the formally or informally specified requirements of the component (Mathur 2008). Formal requirements specification of a component may be represented using decision tables, a set of algebraic axioms, and finite state machines, among others, or as informal specifications using structured natural language, such as use cases (Bruegge and Dutoit 2009). Test data is typically generated from informally specified requirements using techniques such as boundary value analysis, equivalence partitioning, and category partitioning.

White-box testing techniques guide the generation of test cases based on properties of the implementation (source code); these properties are usually derived from control flow and data flow analyses. The results of the analyses allow testers to determine the coverage obtained when a program is exercised using a given test suite. Control flow coverage is a ratio of the entities (related to control flow) covered versus the total number of entities in the program; these entities may include statements, branches, and multiple conditions, among others. The data flow coverage is a ratio representing how the data elements in the program are defined and used versus the total possible set of definitions and uses for a given test suite.

2.2 Testing Tools

During the past decade, there has been an increase in the number of testing tools available for use by academic institutions; these include both freely available tools and those made available via academic initiatives (Hackett 2013; Wikipedia 2013). Many of these tools exhibit similar characteristics to the tools used in industry to test software. Tools are usually categorized based on the

testing perspective used and the view of the component under test. In this article, we focus on the tools students used in the course projects described in Section 3.3, which include the following:

- *JUnit*, a free unit testing framework for the Java programming language (Gamma and Beck 2015)
- *Rational Functional Tester (RFT)*, a commercial automated functional and regression testing tool (IBM 2015a)
- *EclEmma*, a free Java code coverage tool for Web and desktop applications (Hoffmann et al. 2015), of which coverage metrics include instruction, statement, and branch, among others
- *eCobertura*, a free Java code coverage reporting tool for Eclipse (Hofer 2010), of which coverage metrics include statement and branch
- *CodeCover*, a free white-box testing tool for Eclipse developed in 2007 at the University of Stuttgart (CodeCover Team 2011), of which coverage metrics include statement, branch, and loop, among others.

Yang et al. (2007) provide an extensive survey of other coverage-based testing tools that could be used in a software testing course. This survey includes tools for languages other than Java.

2.3 Test Coverage Criteria

As stated previously, white-box testing guides the generation of test cases based on the coverage of certain properties of the program or implementation. The properties of the implementation used in white-box testing are classified as either control flow related (e.g., statement coverage, branch coverage, multiple condition coverage) or data flow related (e.g., all definitions of variables, all uses of variables, all definition-uses of variables). Several researchers have identified various relationships between the different control flow and data flow coverage criteria listed previously. This relationship is referred to as the *subsumes relation*. The relation states that criterion 1 subsumes criterion 2 if every test suite that satisfies criterion 1 also satisfies criterion 2 (Frankl and Weyuker 1993; Zhu et al. 1997).

It should be noted that test coverage criteria does not say anything about the fault-detecting ability of the test suites generated based on a given criteria (Frankl and Weyuker 1993); however, using test coverage criteria still provides the tester with some confidence in the quality of the test suite. This is particularly true when teaching testing techniques to novice testers.

The elements of the subsumes relations that we investigated in this research is the relation between statement coverage and branch coverage. The subsumes relations state that branch coverage subsumes statement coverage. We focus on these two criteria because the tools used in the testing course consistently provide data on statement and branch coverage. These tools include EclEmma (Hoffmann et al. 2015), eCobertura (Hofer 2010), and CodeCover (CodeCover Team 2011). In addition, these are two of the coverage criteria taught in the software testing class and are based on the notion of a flow graph.

A *flow graph* is a directed graph that consists of a set of nodes and edges. The set of nodes include three special nodes: the *start*, *end*, and *basic block* nodes (Mathur 2008). A complete path represents a list of edges beginning with the *start* node and finishing with the *end* node. Statement coverage is defined in terms of the ratio of nodes covered against the total number of nodes (excluding unreachable nodes) for a set of complete paths. Similarly, branch coverage is defined in terms of the ratio of edges covered against the total number of edges (excluding unreachable edges) for a set of complete paths. Using toy programs (methods), students are able to basically understand the concept of the subsumes relation by identifying test cases that cover the nodes (statements) and the edges (branches) of a flow graph.

3 UNDERGRADUATE TESTING COURSE

The study presented in this article was performed in the CEN4072 Fundamentals of Software Testing course taught at FIU. The course is offered during the fall and summer semesters, and it is one of the more popular elective courses since several local companies have shown interest in hiring software testers and developers with testing knowledge. During the course, we expose students to WReSTT (Clarke et al. 2012, 2014b), a cyberlearning environment with learning content that supplements their instruction on software testing concepts and testing tools.

3.1 Pedagogical Approach

In the testing course, we use a combination of pedagogical approaches to improve students' learning of testing concepts and skills. These approaches include *collaborative learning* (Smith and MacGregor 1992), *problem-based learning* (PBL) (Shim et al. 2009), and *gamification* (Deterding et al. 2011; Malone 1980). WReSTT supports the pedagogical approaches used in the course (Clarke et al. 2014b).

Collaborative learning is done mainly through the team project, where students are required to test the software developed by students in a previous Software Engineering I course or a Senior Project course. We provide more details about the project in subsequent sections. Collaborative learning is encouraged both at the project team level and the class level. Members of a team usually have different views, abilities, and personalities, yet they have to work together to achieve success in the team project. Teams are randomly selected, which usually results in teams with teammates who have never worked together. These characteristics of the team force members to assist each other, learn how to resolve their differences, and build consensus, all key tenets of collaborative learning (Smith and MacGregor 1992). Sometimes the instructor needs to intervene when the differences in the team impede productivity.

Collaborative learning at the class level occurs when students share resources using the WReSTT forum. These resources may include tutorials on how to set up and use new testing tools, tutorials on how to install plug-ins required to get the software application running (i.e., the application to be tested), and answers to questions posted by other students in the class regarding the course project. These resources posted to the forum are very helpful to students having a hard time getting the software applications running. Extra credit is usually awarded to those students who posted resources to the forum and are considered helpful by other students in the class.

Students working on team projects incorporate both PBL and collaborative learning. The PBL strategy is employed by getting students to work on solving real-world problems. In our case, the real-world problem is testing a software system written by students in a different class. Although the system is not very large, it challenges students in the testing class in several ways, including trying to understand incomplete documentation, a software design that is inconsistent with good design principles (low coupling and high cohesion), and an incomplete implementation of the requirements.

Gamification is incorporated in the course mainly through students' use of WReSTT. Gamification uses game design elements and game mechanics to engage students and improve their user experience (Malone 1980). WReSTT provides students with the opportunity to obtain virtual points by participating in various activities, as well as leader boards to keep students engaged. In addition to the leader boards, we also use activity streams and active discussion boards, elements from social networking features (Liccardi et al. 2007), to further keep students involved in the activities in WReSTT. Additional details on how WReSTT is used in the course is provided in Section 3.4.

3.2 Course Structure

The CEN4072 course topics as stated in the FIU catalog description are test plan creation, test case generation, program inspections, black-box testing, white-box testing, GUI testing, and the use of testing tools. The prerequisite for CEN4072 is the data structures course. Students in the course are evaluated based on three exams, a team-based project, and class participation. The course is structured as a lecture section with no lab component, and students are expected to work on the project outside of class on their own time. Since the inception of the course in Fall 2010, it has been taught by only one instructor. Details on the course project are provided in the next section.

The specific topics taught in the CEN4072 course are centered around black-box and white-box testing techniques (Ammann and Offutt 2008; Mathur 2008). The black-box testing techniques, sometimes referred to as *specification-based* testing techniques, presented in the course include random, equivalence partition, boundary-value analysis, and state-based testing. *Random* testing selects random independent inputs from an input domain. *Equivalence partition* testing selects inputs from each equivalent class that partitions the input domain. *Boundary-value analysis* testing selects values on and around the boundaries of the equivalent classes. *State-based* testing is a formal technique that selects inputs to traverse the various transitions in a state machine that represents the component under test.

The white-box techniques, sometimes referred to as *implementation-based* testing techniques, taught in the CEN4072 course include control flow coverage (statement, branch, multiple condition, and basis path) and data flow coverage (all definitions, all uses, and all definition-use paths). Each of the coverage criteria listed previously represents a ratio of the entity (*statements*) exercised in the program over the total number of that entity (*statements*) in the program minus the infeasible number of that entity (*unreachable statements*). During white-box testing, the black-box test suite is used to exercise the program; based on the results of the specific coverage criteria, additional test cases are developed specifically to cover those parts of the program not covered by the original test suite. Additional details for the other testing techniques can be found in the literature (Ammann and Offutt 2008; Bourque and Dupuis 2004; Mathur 2008).

3.3 Course Project

As previously stated, one of the pedagogical approaches used in the CEN4072 course is problem-based learning (PBL). In PBL, providing students with a significant problem to be solved is essential. This provides them the opportunity to learn outside the course material presented in class. One of the requirements of the CEN4072 course is a team-based semester-long project that involves all students in the class. The objective of the project is to provide students with experience analyzing and validating software, writing test cases, creating test documents, and working in teams. Students are also required to use testing tools to automate various aspects of the testing process. The software to be tested is a project from a previous Software Engineering I course or a Senior Project course. The software artifacts for the project include the requirements document, design document, implementation document, UML diagrams, and source code.

The project component of the CEN4072 course is evaluated based on two deliverables and two in-class formal presentations. Each in-class presentation includes a slide presentation and a demonstration of the tools used to test the software for that deliverable. The project consists of two major phases: the first phase focuses on black-box testing of the software application at the unit level, subsystem level, and system level, and the second phase focuses on the white-box testing of the software application, using the same levels are the first phase.

The first phase ends with the submission of the first deliverable, *Specification-Based Test Deliverable* (SBTD), and the in-class presentation. The second phase ends with the submission of the

second (final) deliverable, *Implementation-Based Test Deliverable* (IBTD), and the final in-class presentation. The IBTD also includes the test cases created during the first phase of the project and shows the increase in coverage after updating the test suites using the various white-box testing techniques. The format of the test document for the IBTD is shown in Appendix A. The structure of the SBTD document is similar to that of the IBTD document, except there is no Section 6.2.

The first phase of the project requires students to use tools to automate the black-box testing of the system's functionality at the unit level, subsystem level, and system level. The tools used during this phase include JUnit (Gamma and Beck 2015), a unit testing tool for both unit-level and subsystem-level testing, and IBM RFT (IBM 2015a), a functional testing tool with capture playback capabilities for GUIs for system-level testing.

In the second phase of the project, at least two code coverage tools are required to determine the coverage achieved for the black-box test cases from the first phase. Then, based on the coverage recorded for the black-box test suites, the test suites are updated to increase code coverage during white-box testing in the second phase. The tools used in the second phase of the project are selected from EclEmma (Hoffmann et al. 2015), eCobertura (Hofer 2010), and CodeCover (CodeCover Team 2011).

To make the project manageable in one semester, the unit testing consists of testing two classes from the same package, whereas the subsystem testing focuses on testing only one subsystem package, the package containing the two classes used in the unit testing. The entire system is also tested based on the implemented requirements. The typical software project has about 20 Java classes and 2,000 source lines of code (SLOC). A table is presented in Section 4.2 that provides the reader with additional details of the software projects that were tested, including the actual size of the projects and type of application. The size of the each student team is on average four to six students depending on the size of the class. Due to the logistics of the class, including topics to be covered and holidays, we can allocate at most four class sessions for the two in-class presentations, which results in at most six teams, with three teams presenting in each session of 75 minutes.

3.4 Cyberlearning Environment

During the last three offerings of the CEN4072 course, students were exposed to WReSTT (WReSTT Team 2015). WReSTT is a cyberlearning learning environment that contains tutorials on software testing concepts and testing tools, and provides students the ability to post resources to the forums to help other students in the class. The details of WReSTT have been reported in other publications (Clarke et al. 2012, 2014b) and will not be presented in this article. We encourage the interested reader to review these publications. WReSTT was developed as a learning resource for those courses that do not focus on software testing (software engineering (SE), data structures, etc.). During the past 5 years WReSTT has evolved from a simple repository of testing tool tutorials to a cyberlearning environment that integrates the pedagogical approaches described in Section 3.1.

WReSTT is used mainly as a learning resource in the CEN4072 course for students to access the tutorials on testing tools. This is done since there is little or no time during the semester for in-class instruction on how to use the various testing tools. In addition to using WReSTT as a learning resource for testing tools, students use WReSTT to access tutorials and quizzes on various testing concepts. These tutorials reinforce concepts taught during the class. Obtaining a good score on the quizzes provides students an opportunity to obtain virtual points. The access to virtual points when using the resources in WReSTT is part of the gamification approach described in Section 3.1.

The opportunity to obtain virtual points in WReSTT is based on students' interactions with various learning resources. Virtual points are awarded to students on an individual basis, as well as on a team basis. Individual points are awarded for obtaining a score of 80% or higher on the

quizzes, uploading a picture to their profile, and posting to the forum in WReSTT. Team points are awarded based on when the entire team completes the quiz and each team member scoring higher than 80% on each quiz. Note that it is up to the course instructor to decide whether the WReSTT virtual points will be converted into extra credit for the course. Anecdotal evidence has suggested that those teams that acquire bonus virtual points also do well on the team project.

Instructors can monitor how students are using the resources in WReSTT by viewing a report for each student in the class. This feature allows instructors to monitor which students have accessed which tutorials and their performance on the quizzes associated with the various tutorials. WReSTT also provides instructors other features, including the ability to upload class rolls; assign teams; make tutorials and quizzes available to students; and, more recently, administer a pre/posttest on software testing concepts and tools. Instructors may access WReSTT to use in their classes by sending an email to any of the authors of this article or visiting the WReSTT Web site[3] (WReSTT Team 2015).

The tool tutorials in WReSTT are kept up to date by allowing students to create new tutorials for extra credit. In the Fall 2011 CEN4072 software testing course, one of the students (who also works in industry as a software test engineer) created a high-quality video for the Selenium testing tool (Selenium Development Team 2014). The creation of tool tutorials also has some pedagogical benefit to students since it is reported in the literature that preparing tutorials on a subject improves student learning (AAAS and NSF 2013).

4 EMPIRICAL STUDY

In this section, we report on a study that shows how testing tools were used in three offerings of the CEN4072 course in the Fall 2011, Fall 2012, and Fall 2013 semesters. The focus of the study is to determine how the increased use of testing tools improves the quality of software testing and the impact of WReSTT on student learning. In the next section, we present the objectives of the study, followed by the methods employed during the study, the results and analysis, and finally a discussion of study results.

4.1 Study Objectives

Determining how much testing to perform on a piece of software is a hard problem to solve when the input domain is large. Dijkstra (1970) states that testing can show the presence but not the absence of bugs. Since testing cannot show the absence of bugs, is it possible to provide a client some confidence in the software application after it has been tested? It is important for students in a software testing class to answer this question, as many students do not grasp the difficulty involved in removing all bugs from a piece of software.

In the recent offerings of the CEN4072 course at FIU, students were exposed to both black-box and white-box testing techniques with the objective of showing how these techniques complement each other. In addition, students were able to show how improving a test suite using various white-box testing techniques, based on statement and branch coverage criteria, can improve one's confidence in the quality of the software application after testing. After completing the project, students were also able to determine if the theoretical subsumes relation between branch coverage and statement coverage holds (Frankl and Weyuker 1993).

Finally, the study allowed us to determine the effectiveness of WReSTT in the context of a software testing class. Based on previous studies (Clarke et al. 2012, 2014b), WReSTT was considered a useful learning resource in an SE class, where testing is only a small component of the course.

[3] http://wrestt.cis.fiu.edu/.

Table 1. Samples Used in the Study

Semester	Enrollment (#)	Teams (#)	WReSTT Survey	
			Participation (#)	(%)
Fall 2011	26	5	24	92.3%
Fall 2012	35	6	33	94.3%
Fall 2013	30	6	27	90.0%

Table 2. Summary of Software Projects Used in the Study

Semester(s)	Project Name	Technology Used	Packages	.java Files	.class Files	Methods	SLOC
Fall 2011 & Fall 2013	Professor Schedule Manager (PSM)	3-Tier, Java Swing, MySQL	3	14	64	356	2,555
Fall 2012	PantherLot Interactive	Client-Server, Java Swing, MySQL	5	25	46	269	1,538

The specific objectives of the study conducted during the Fall 2011, Fall 2012, and Fall 2013 CEN4072 classes were to determine the following:

(1) If the availability and knowledge of the use of code coverage tools positively impacts and increases students' propensity to improve the quality of their black-box test suites
(2) If an increase in code coverage during white-box testing results in an increase in the number of bugs students find during testing
(3) If students find WReSTT a useful learning resource for testing techniques and tools
(4) If students find that WReSTT supports collaborative learning.

4.2 Methods

Sample. Students participating in the study were from the Fall 2011, Fall 2012, and Fall 2013 CEN4072 Fundamentals of Software Testing classes at FIU. Table 1 shows the data for each class, including the number of students enrolled, the number of project teams, and number of students participating in the WReSTT survey along with the percentage participation.

Data collection. Data for the study was collected from several sources. These sources were the two project deliverables (SBTD and IBTD) whose artifacts included documents, presentation slides, and source code (test harnesses: drivers and stubs); observation of the in-class team presentations; and a survey instrument consisting of 12 questions (see Appendix B). Observing the in-class presentations provided evidence that students used the testing tools and were able to describe how they worked.

The WReSTT survey shown in Appendix B contains three sections. The first section contains one question to determine if students use other learning repositories in the class. The second section contains six questions (2 through 7) related to the quality and usefulness of the learning material in WReSTT. The third section contains five questions (8 through 12) that are related to the collaborative learning aspect of WReSTT.

The project teams in each class were assigned the same SE project to be tested. Table 2 contains a summary of the metrics for the two projects used in the study. The metrics were obtained using Dependency Finder (Tesser 2010). The project tested in the Fall 2011 and Fall 2013 classes was the

Professor Schedule Manager (PSM). PSM is a desktop application that helps professors keep track of the time left in a class. Prior to 2008, all classes at FIU were 1 hour and 15 minutes in length. At the beginning of the 2008–2009 academic year, classes on Monday, Wednesday, and Friday were scheduled to be 50 minutes in length, and classes on Tuesday and Thursday continued to be 1 hour and 15 minutes in length. As a result, professors were having trouble adapting to the change, so students in the SE class developed the PSM application. PSM uses a three-tier architecture consisting of three packages, 14 Java files, 64 classes, 356 methods, and 2,555 SLOC. The PSM application was built using Java Swing technology and a MySQL database (see the first row of Table 2).

In Fall 2012, the application to be tested was PantherLot Interactive, a garage parking system that is expected to reduce the time for members of the FIU community to find an available parking spot. The parking system uses scanners and sensors to determine if there are any free parking spots before drivers enter a parking lot. The system serves four types of users: students, faculty, administrators, and guests. All hardware-specific devices, such as scanners and sensors, were simulated by software modules. PantherLot Interactive uses a client-server architecture consisting of five packages, 25 Java files, 46 classes, 269 methods, and 1,538 SLOC. The PantherLot application was built using Java Swing technology and a MySQL database (see the second row of Table 2).

Design. In each of the three classes used in the study, students were given access to the software projects (to be tested) during the 2nd week of the semester. Students were expected to have the software running by the 4th week of the semester. The SBTD was due in the 8th week of the semester and the IBTD in the 15th week. Students were informed of the study at the beginning of the semester and were given the opportunity to opt out of the study if they wanted to, without penalty. During the 2nd week of the semester, students were enrolled in WReSTT and provided access to the learning materials, including the tool tutorials. The WReSTT survey was administered during the last week of the semester. The study presented in this article is covered under IRB Exemption numbers 062307-00 and 062112-00 approved by the Office of Research and Integrity at FIU.

As stated previously, the first testing deliverable, SBTD, focused on testing the software using black-box testing techniques. Although students were required to use the black-box testing techniques presented in class (random, equivalence partitioning, boundary value analysis, and state based), the system test cases were developed from the use cases. In other words, for each implemented use case, students were required to create at least three sunny day scenario test cases and three rainy day scenario test cases. By sunny day scenario, we mean that the instance of the use case should follow the normal transaction flow, and for the rainy day scenario, the transaction should include one alternative path or an exception. Based on the system test cases, students were required to develop subsystem and unit test cases by tracing the requirements from the system level to the class level and using the appropriate black-box testing techniques.

The second deliverable, IBTD, required the use of white-box testing to enhance the test suites developed in the first deliverable (SBTD). Students were required to use the coverage testing tools specified to obtain the code coverage for the black-box test suites, then create new test cases to increase the coverage. In other words, based on the percentage statement and branch coverage obtained when running the original (black-box) test suite, new test cases were added to improve code coverage.

At the end of each class in the study, we engaged in formative assessment of student use of the tools in the course and the tutorials in WReSTT. The formative assessment included a review of the project documentation for each of the deliverables to determine if results reported in the presentations were accurate. This allowed us to determine whether students were in fact using the tools and tutorials as expected, as well as whether additional resources were needed. For example,

students were required to include screen shots in the document reflecting how the tools were used, which led us to believe that some students were reading the incorrect coverage results. In addition, for the same coverage criteria, the tools were reporting different coverage results. The coverage tools used during Fall 2011 were eCobertura (Hofer 2010) and EclEmma (Hoffmann et al. 2015). The results of our formative assessments led to the inclusion of additional resources in WReSTT and the addition of the code coverage tool CodeCover (CodeCover Team 2011).

4.3 Results and Analysis

In this section, we present the results found during the study, which will be the basis for deciding if the claims stated in the objectives can be supported or not. The results for Objectives 1 and 2 are grouped together, as they are both related to code coverage.

Objectives 1 and 2. To determine if the availability and knowledge of the use of code coverage tools positively impacts and increases students' propensity to improve the quality of their black-box test suites, we collected the data that is summarized and shown in Tables 3, 4, and 5. The data shown in these tables was collected during three semesters Fall 2011, Fall 2012, and Fall 2013. It should be noted that students reported the results in the tables during their in-class presentations. These results were confirmed during student in-class demonstrations by running the tests and by reviewing the submitted deliverable documents. However, the results were not independently checked by rerunning the applications.

Table 3 shows data for the following categories: the number of test cases created by each team during black-box testing (columns 2 through 4), the total number of test cases after white-box testing (columns 5 through 7), and the number of test cases added during white-box testing (columns 8 through 10). The data for each of these categories is presented for the three levels of testing (unit, subsystem, and system).

Table 3 is separated into four sections: one for each of the three semesters in which the study was conducted and a final row showing the averages for all semesters. The averages for the number of black-box test cases created, total number of test cases after white-box testing, and the number of test cases added during white-box testing (and percentage increase) are shown in the last row for each semester. For the three semesters shown in Table 3, the unit and subsystem testing were done using JUnit (Gamma and Beck 2015) and the system testing was done using IBM RFT (IBM 2015a).

In general, all teams except one added new test cases or kept unchanged the number of test cases during the white-box testing phase of the project (see Table 3). The exception was Team 3 in Fall 2011, where there was a decrease in the number of test cases during white-box testing for unit and subsystem testing. The numbers of unit and subsystem test cases created during black-box testing by Team 3 were already high compared to the other teams in the Fall 2011 semester.

Table 4 shows the averages for the number of bugs found during each of the semesters for unit, subsystem, and system testing. Columns 2 through 4 show the average number of bugs found during black-box testing for the teams in each of the semesters in the study. Columns 5 through 7 show the total number of bugs found after white-box testing, and columns 8 and 9 show the increased number of bugs found (and percentages) during white-box testing. The last row of the table shows the averages for all semester in the study. The average number of test cases added for all teams in the Fall 2011, Fall 2012, and Fall 2013 semesters, shown in the last row of the Table 3, are 8.1 (unit), 18.6 (subsystem), and 4.8 (system), whereas the number of new bugs found, shown in Table 4, are 1.0 (unit), 2.2 (subsystem), and 0.8 (system).

Table 5 shows the percentage of code coverage obtained for the black-box and white-box test suites using the various tools. The columns of the table are partitioned into three sections. Column 1 shows the team number, columns 2 through 7 show the statement and branch coverage for the

Table 3. Number of Test Cases Created During Black-Box Testing, the Total Number of Test Cases, and the Increase in the Number of Test Cases Added During White-Box Testing

	Initial Test Suite (Black-Box Testing)			Final Test Suite (After White-Box Testing)			Test Cases Added (During White-Box Testing)		
Team	Unit	Subsys	System	Unit	Subsys	System	Unit	Subsys	System
Fall 2011:									
1	17	20	21	32	54	28	15	37	7
2	3	7	24	11	8	25	8	1	1
3	68	68	4	35	42	4	−33	−26	0
4	5	5	3	22	129	5	17	124	2
5	15	10	20	31	31	29	16	21	9
Avg.	**21.6**	**22.0**	**14.4**	**26.2**	**52.8**	**18.2**	**4.6 (21%)**	**30.8 (140%)**	**3.8 (26%)**
Fall 2012:									
1	51	5	65	57	12	65	6	7	0
2	10	31	18	14	32	24	4	1	6
3	12	43	24	21	61	29	9	18	5
4	26	16	27	32	25	34	6	9	7
5	17	20	36	24	31	48	7	11	12
6	14	36	33	27	39	38	13	3	5
Avg.	**21.7**	**25.2**	**33.8**	**29.2**	**33.3**	**39.7**	**7.5 (35%)**	**8.2 (32%)**	**5.8 (17%)**
Fall 2013:									
1	26	24	48	42	28	52	16	4	4
2	8	3	48	26	56	51	18	53	3
3	26	29	48	35	37	51	9	8	3
4	16	13	48	35	31	61	19	18	13
5	10	19	48	11	30	51	1	11	3
6	19	14	43	30	21	46	11	7	3
Avg.	**17.5**	**17.0**	**47.2**	**29.8**	**33.8**	**52.0**	**12.3 (70%)**	**16.8 (99%)**	**4.8 (10%)**
All Semesters:									
Avg.	**20.3**	**21.4**	**31.8**	**28.4**	**40.0**	**36.6**	**8.1 (40%)**	**18.6 (87%)**	**4.8 (15%)**

various testing levels for the black-box test suite, and columns 8 through 13 show the statement and branch coverage for the various testing levels for the white-box test suite.

Table 5 shows that Team 1 in Fall 2013 obtained the following code coverage averages using three coverage tools. For the black-box test suite, 71.1% statement and 23.1% branch coverage for unit testing, 82.4% statement and 60.8% branch coverage for subsystem testing, and 81.0% statement and 44.7% branch coverage for system testing. After white-box testing, the updated test suite resulted in 95.9% statement and 94.8% branch coverage for unit testing, 96.5% statement and 82.2% branch coverage for subsystem testing, and 87.3% statement and 45.7% branch coverage for system testing. Note that from the results in Table 3, Fall 2013, Team 1, 16 unit test cases, four subsystem test cases, and four system testing cases were added during white-box testing to achieved the increased coverage.

Table 4. Number of Bugs Found During Black-Box Testing, the Total Number of Bugs Found, and the Number of Bugs Found During White-Box Testing

Team	Initial Bugs Found (Black-Box Testing)			Total Number of Bugs (After White-Box Testing)			Bugs Added (During White-Box Testing)		
	Unit	Subsys	System	Unit	Subsys	System	Unit	Subsys	System
Fall 2011:									
Avg.	1.4	1	5.6	2	2.2	5.4	**0.6 (43%)**	**1.2 (120%)**	**− 0.2 (−4%)**
Fall 2012:									
Avg.	0.2	4.2	3.5	0.3	5.5	4.5	**0.2 (100%)**	**1.3 (32%)**	**1.0 (29%)**
Fall 2013:									
Avg.	1.3	2.5	22.5	3.5	7.2	24.2	**2.2 (163%)**	**4.2 (188%)**	**1.7 (7%)**
All Semesters:									
Avg.	**1.0**	**2.6**	**10.5**	**1.9**	**5.0**	**11.4**	**1.0 (101%)**	**2.4 (94%)**	**0.8 (8%)**

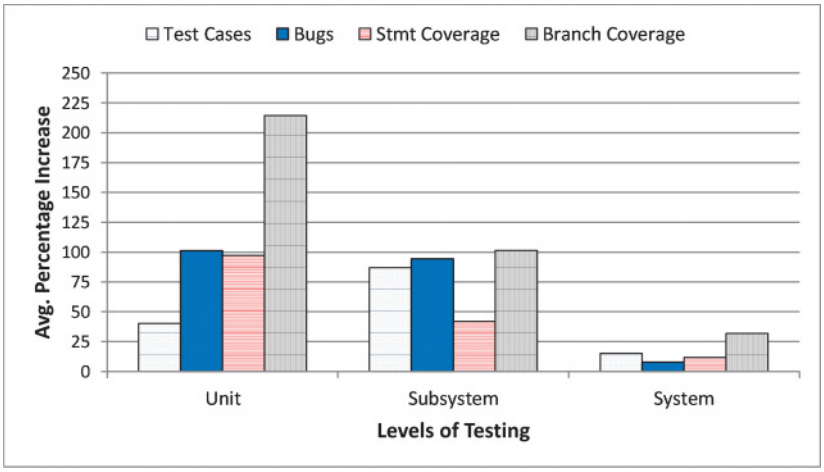

Fig. 1. Average percentage increase in the number of test cases, bugs, statement coverage, and branch coverage for each testing level for all teams in the Fall 2011, Fall 2012, and Fall 2013 semesters.

For each of the semesters shown in Table 5, the last two rows in that section show the average and standard deviation for the results obtained by the teams. For example, the average statement coverage obtained during black-box testing for the teams in Fall 2013 is 56.3% and the standard deviation is 18.1. The standard deviation is high for black-box testing since there is no black-box testing technique that guides testers on how to select a minimal subset of test cases from a usually infinite input space. The standard deviation is smaller for the white-box testing since code coverage helps to guide the selection of test cases.

Figure 1 shows the average percentage increase in the number of test cases, bugs, statement coverage, and branch coverage for each testing level between the black-box testing phase and the white-box testing phase of the project. This figure shows the average for all teams for all semesters,

Table 5. Percentage Coverage for Black-Box and White-Box Test Suites

Team	Initial Test Suite (Percentage Coverage)						Updated Test Suite (Percentage Coverage)					
	Unit		Subsystem		System*		Unit		Subsystem		System*	
	Stmt	Br	Stmt	Br	Stmt	Br	Stmt	Br	Stmt	Br	Stmt	Br
Fall 2011:												
1	—	—	64.1	18.8	71.2	14.4	—	—	91.5	69.4	86.3	58.9
2	24.9	—	83.6	—	69.2	—	86.3	—	86.0	73.7	69.7	42.4
3	30.2	—	54.7	—	81.8	—	82.8	—	87.6	—	81.8	—
4	13.7	2.9	58.8	—	—	—	87.1	52.9	91.8	54.1	81.0	—
5	33.9	5.9	—	—	70.8	47.9	99.8	95.8	—	—	87.1	71.4
Avg.	**25.7**	**4.4**	**65.3**	**18.8**	**73.3**	**31.2**	**89.0**	**74.3**	**89.2**	**65.7**	**81.2**	**57.6**
SD.	**8.8**	**2.1**	**12.8**	**—**	**5.8**	**23.6**	**7.4**	**30.4**	**2.9**	**10.3**	**6.9**	**14.6**
Fall 2012:												
1	54.0	59.9	81.9	71.4	85.0	78.0	90.2	86.4	85.6	77.4	85.0	78.0
2	48.4	32.7	77.6	70.5	43.0	39.0	91.0	76.0	85.1	74.6	77.0	73.0
3	49.7	50.0	70.8	61.9	77.0	71.0	100.0	100.0	98.7	96.1	89.0	85.0
4	66.3	45.7	66.3	55.5	79.0	67.0	84.8	79.2	86.1	80.3	89.0	83.0
5	58.0	41.7	39.4	51.2	83.0	78.0	89.5	83.4	88.9	81.8	84.0	80.0
6	47.7	35.0	39.1	32.8	73.5	78.0	87.0	84.0	72.6	66.7	75.0	81.0
Avg.	**54.0**	**44.2**	**62.5**	**57.2**	**73.4**	**68.5**	**90.4**	**84.8**	**86.2**	**79.5**	**83.2**	**80.0**
SD.	**7.1**	**10.0**	**18.8**	**14.4**	**15.5**	**15.2**	**5.2**	**8.3**	**8.4**	**9.7**	**5.9**	**4.2**
Fall 2013:												
1	71.1	23.1	82.4	60.8	81.0	44.7	95.9	94.8	96.5	82.2	87.3	45.7
2	26.5	0.4	26.9	15.5	87.2	63.0	84.2	75.9	65.5	59.3	87.2	63.0
3	77.8	58.0	77.4	58.4	84.0	76.0	92.6	85.2	82.0	82.6	85.0	94.0
4	52.8	11.3	52.0	14.0	49.3	28.0	91.6	79.4	89.4	86.3	49.3	28.0
5	59.3	38.9	58.0	39.5	87.0	63.0	80.6	48.6	100.0	87.7	90.2	74.0
6	50.0	25.6	42.5	20.4	50.1	24.3	87.2	71.5	83.3	69.1	88.4	53.2
Avg.	**56.3**	**26.2**	**56.5**	**34.8**	**73.1**	**49.8**	**88.7**	**75.9**	**86.1**	**77.9**	**81.2**	**59.7**
SD.	**18.1**	**20.3**	**21.0**	**21.3**	**18.3**	**20.9**	**5.7**	**15.6**	**12.3**	**11.2**	**15.7**	**23.0**
All Semesters:												
Avg.	**45.3**	**24.9**	**61.4**	**36.9**	**73.3**	**49.8**	**89.4 (97%)**	**78.4 (214%)**	**87.2 (42%)**	**74.3 (101%)**	**81.9 (12%)**	**65.7 (32%)**

Note: The dash (—) represents that a value was not reported. The asterisk (*) denotes that teams were only able to use at most two coverage tools.

except the unit branch coverage for Fall 2011. The leftmost group of bars represents unit testing, the group in the center subsystem testing, and the rightmost group system testing. Each group consists of four bars: from left to right, the bars represent the average percentage increase of test cases, bugs, statement coverage, and branch coverage.

The values for the unit branch coverage for Fall 2011 were removed since the reported values skewed the results; based on the results shown in Table 4, the teams seemed to have not reported the coverage results accurately. Fall 2011 was the second time the course was taught (the first time involving a study), and the teaching methods for code coverage were still being developed. As a result, the code coverage data for Fall 2011 is very sparse, as shown by the dashed (—) entries in

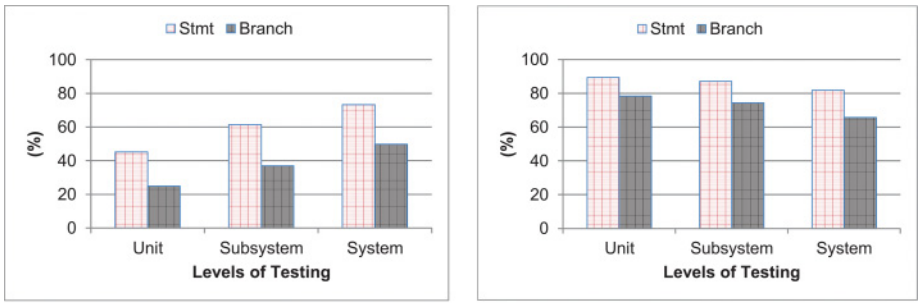

Fig. 2. Averages of statement and branch coverage for unit, subsystem, and system testing for all teams in the Fall 2011, Fall 2012, and Fall 2013 semesters. (a) Data for the black-box test cases. (b) Data for the white-box test cases.

the top part of Table 5. We determined that it would therefore be inappropriate to include the Fall 2011 data in our analyses looking at code coverage. Thus, we have focused our analyses on Fall 2012 and Fall 2013. We provide additional discussion for the sparseness of the data in Section 4.4.

Figure 2 shows charts comparing the statement and branch coverage for unit, system, and system testing. Part (a) of the figure shows the coverage results for black-box testing and part (b) for white-box testing. The data shown in Figure 2 is the average of all teams for the Fall 2011, Fall 2012, and Fall 2013 semesters. Note that for each group of two bars in the figure, the bar on the left representing statement coverage is always greater than the bar on the right representing branch coverage. This result supports the in-class exercises, using toy programs, where the number of test cases to achieve 100% branch coverage was usually greater than or equal to the number required for 100% statement coverage.

Statistical analysis. Mixed model repeated measures ANOVAs were conducted for unit, subsystem, and system testing to compare the effect of availability and knowledge of the use of code coverage tools (independent variable (IV)) on the number of test cases created (dependent variable (DV)) across the three semesters. A main effect of availability and knowledge of the use of code coverage tools was found for all three types of testing. Results found that the number of test cases created by each team significantly increased after students learned about using code coverage tools (training) in unit testing (no training: M = 20.27, SD = 16.57; after training: M = 28.40, SD = 11.47), F (1, 14) = 7.47, $p = .016$, $\eta^2 = .0806$ (medium effect size based on Cohen (Cohen 1988)), subsystem testing (no training: M = 21.39, SD = 16.56; after training: M = 39.96, SD = 27.11), F (1, 14) = 5.53, $p = .03$, $\eta^2 = .1667$ (large effect size), and system testing (no training: M = 31.82, SD = 17.20; after training: M = 36.62, SD = 17.7) F (1, 14) = 25.15, $p = .00$, $\eta^2 = .051$ (medium effect size).

Based on the results, students increased the number of test cases generated for unit, subsystem, and system testing. In addition, the effect sizes for these increases ranged from medium to large based on Cohen's guidelines (Cohen 1988). For unit testing, 8.1% of the variance (medium effect size) in the increase in test cases was accounted for by the (IV) knowledge of use of code coverage tools. For subsystem testing, 16.7% of the variance (large effect size) was accounted for by the IV, and for system testing, 5.1% of the variance (medium effect size) was accounted for by the IV.

There were no differences across semesters for unit testing or subsystem testing, and no significant interactions between semester and the use of code coverage tools. However, there was a significant difference between semesters for system testing. Specifically, teams from 2011 created

Table 6. Effect and Knowledge of Using Code Coverage Tools During Testing

Test Type	Initial Code Coverage	Final Code Coverage	F Value	Sig	Eta Squared
Unit testing statement coverage	M = 55.13, SD = 13.15	M = 89.55, SD = 5.30	F (1,10) = 79.89	$p = .00$	$\eta^2 = .76$
Unit testing branch coverage	M = 35.19, SD = 10.02	M = 80.37, SD = 12.81	F (1,10) = 60.84	$p = .010$	$\eta^2 = .74$
Subsystem testing statement coverage	M = 59.53, SD = 18.81	M = 86.42, SD = 10.05	F (1,10) = 30.99	$p = .00$	$\eta^2 = .70$
Subsystem testing branch coverage	M = 45.99, SD = 14.39	M = 78.68, SD = 10.05	F (1,10) = 47.71	$p = .00$	$\eta^2 = .56$
System testing statement Coverage	M = 73.26, SD = 16.14	M = 82.20, SD = 11.39	F (1,10) = 4.91	$p = .051$	$\eta^2 = .10$
System testing branch coverage	M = 59.17, SD = 19.96	M = 69.83, SD = 18.99	F (1,10) = 8.91	$p = .014$	$\eta^2 = .10$

significantly fewer test cases both prior to learning code coverage tools (M = 14.4, SD = 10.07) and after learning code coverage tools (M = 18.20, SD = 12.60) than the teams from 2012 (M = 33.82, SD = 16.56; M = 39.67, SD = 14.87, respectively) and 2013 (M = 47.17, SD = 2.04; M = 52.00, SD = 4.90, respectively). Follow-up analyses revealed that for 2012 ($t(5) = -3.69$, $p = .01$) and 2013 ($t(5) = -2.95$, $p = .03$), but not 2011, students created significantly more system test cases after learning how to use code coverage tools than prior to learning how to use code coverage tools. We believe that the reason for this is the result of greater experience of the instructional team in teaching the course. The course was first taught in 2010, and lessons learned were able to be applied to the subsequent years, providing for more effective teaching methods.

Mixed model repeated measures ANOVAs were conducted for both statement and branch coverage for unit, subsystem, and system testing to compare the effect of availability and knowledge of the use of code coverage tools (IV) on percentage of code coverage (DV) across the Fall 2012 and Fall 2013 semesters. A main effect of availability and knowledge of the use of code coverage tools was found for all types of testing, except for system testing–statement coverage. Results, shown in Table 6, found that the percentage of code coverage by each team significantly increased after learning about using code coverage tools. We only show a sample of the significant results here: (1) subsystem testing statement coverage (no training: M = 59.53, SD = 18.81; after training: M = 86.42, SD = 10.05), F (1, 10) = 30.99, $p = .00$, $\eta^2 = .70$ (very large effect size based on Cohen (Cohen 1988)), and (2) system testing–branch coverage (no training: M = 59.17, SD = 19.96; after training: M = 69.83, SD = 18.99) F (1, 10) = 8.91, $p = .014$, $\eta^2 = .10$ (medium effect size).

These results support that the availability and knowledge of the use of code coverage tools positively impacts and increases students' propensity to improve the quality of their black-box test suites (Objective 1) by exercising more of the code being tested. Students significantly increased the quality of black-box test suites as reflected in an increase in the percentage of code coverage for unit statement and branch coverage, subsystem branch coverage, and system statement and branch coverage testing. In addition, the effect sizes for these increases ranged from medium to very large based on Cohen's guidelines (Cohen 1988).

Consistent with prior findings in the literature, there were no significant differences in the number of bugs found prior to and after students learned about the use of code coverage tools in any of the testing (Objective 2). Interestingly, however, the number of bugs found prior to and after

Table 7. Students' Mean Scores (Standard Deviations) Measuring Perception of the Usefulness of Testing Tutorials in WReSTT

Q.	Testing-Related Questions	Fall 2011 M (SD)	Fall 2012 M (SD)	Fall 2013 M (SD)	Avg. M (SD)
2	The tutorials in WReSTT helped me to better understand testing concepts.	3.96 (1.00)	3.97 (0.85)	3.95 (2.59)	3.96 (1.48)
3	The tutorials in WReSTT helped me to better understand how to use unit testing tools.	4.08 (0.83)	4.06 (0.86)	3.83 (2.63)	3.99 (1.44)
4	The tutorials in WReSTT helped me to better understand how to use code coverage testing tools.	3.38 (1.10)	3.45 (1.35)	3.37 (2.43)	3.40 (1.63)
5	The tutorials in WReSTT helped me to better understand how to use functional testing tools.	3.17 (1.27)	3.24 (1.39)	3.63 (2.51)	3.35 (1.72)
6	The number of tutorials in WReSTT is adequate.	2.92 (1.14)	2.97 (1.16)	3.47 (2.36)	3.12 (1.55)
7	I would have used testing tools in my project if WReSTT did not exist.	3.67 (1.81)	4.03 (1.16)	3.95 (2.60)	3.88 (1.86)

students learned about the use of code coverage tools for subsystem testing–bugs found was close to significance (no training: M = 2.56, SD = 3.53; after training: M = 4.71, SD = 5.80) $F(1, 14) = 34.13$, $p = .051$, $\eta^2 = .05$ (with a near medium effect size). In addition, effect sizes for unit testing–bugs found ($\eta^2 = .11$) and system testing–bugs found ($\eta^2 = .06$) were also medium sized.

Due to these findings being insignificant, we normally would not discuss them further and generalizations of these results cannot be made. However, finding medium practical significance with the effect sizes given our sample size may suggest that it would be reasonable to further explore the number of bugs found when doing subsystem testing with a larger sample size or a more precise methodology that focuses on this aspect of the intervention. We further discuss this finding in Section 4.4.

Objective 3. Table 7 shows students' perceptions of the usefulness of the tutorials available in WReSTT. The table shows the results for Fall 2011, Fall 2012, and Fall 2013, and the average values over the three semesters. The size of the samples in each semester are as follows: 24 (Fall 2011), 33 (Fall 2012), and 27 (Fall 2013). These results were positive for questions 2 through 7 with all average scores above 3.10 out of 5 (where 5 is Strongly Agree). The complete survey instrument is shown in Appendix B. These questions focused on the effectiveness of WReSTT in helping students understand testing concepts and how to use the testing tools. The lowest value in the table is question 6 (M = 3.12, SD = 1.55), which focuses directly on students' perception of the usefulness of WReSTT with respect to the number of tutorials. The entry for question 6 has prompted the authors to improve the quality and number of tutorials in WReSTT.

The usefulness of WReSTT with respect to the availability of tutorials on tools is also reflected in the bar chart shown in Figure 3. The figure shows the average percentages for the number of tools used during white-box testing during the Fall 2011, Fall 2012, and Fall 2013 semesters. In Fall 2011, the five teams were only required to use two different code coverage tools, whereas in Fall 2012 and 2013, the six teams were required to use three different tools, respectively. The leftmost group of bars represents the tools used for statement coverage during unit testing: in Fall 2011, out

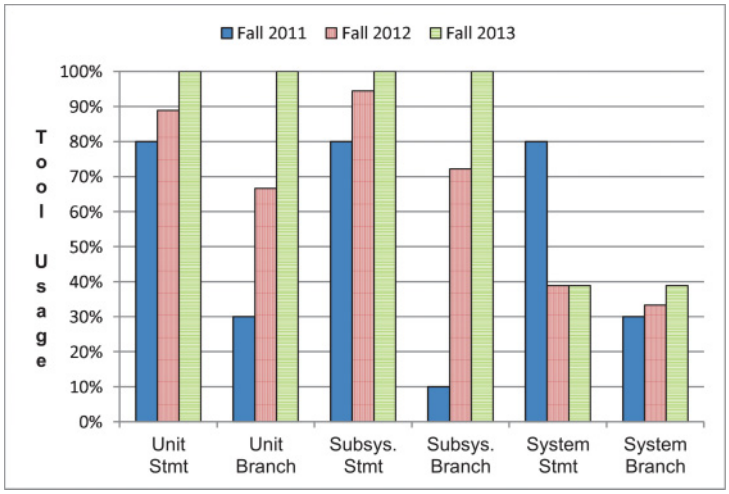

Fig. 3. Average percentage of tool usage during white-box testing for classes in the Fall 2011, Fall 2012, and Fall 2013 semesters.

Table 8. Students' Mean Scores (Standard Deviations) Measuring Usefulness of the Collaborative Features in WReSTT

Q.	Collaborative Learning Related Questions	Fall 2011 M (SD)	Fall 2012 M (SD)	Fall 2013 M (SD)	Avg. M (SD)
8	The use of virtual points in WReSTT encouraged me to visit the website and complete the tasks.	4.58 (0.72)	4.27 (1.44)	3.89 (2.71)	4.25 (1.62)
9	The use of virtual points in WReSTT encouraged my team to visit the website and complete the tasks.	4.63 (0.65)	4.33 (1.36)	3.50 (2.54)	4.15 (1.52)
10	The event stream showing the activities of the other members in the class encouraged me to complete my tasks in WReSTT.	4.17 (0.96)	3.85 (1.28)	3.28 (2.38)	3.77 (1.54)
11	The event stream showing the activities of the other members in the class encouraged my team to complete our tasks in WReSTT.	4.04 (0.86)	3.79 (1.32)	3.33 (2.41)	3.72 (1.53)
12	Our team devised a plan to get the maximum number of points in WReSTT.	3.88 (1.33)	4.03 (1.02)	2.76 (2.29)	3.56 (1.55)

of 10 possible uses of the tools (five teams, two tools each), there were 8 uses (80%); in Fall 2012, out of 18 possible uses of the tools (six teams, three tools each), there were 16 uses (89%); and in Fall 2013, out of 18 possible uses of the tools, all 18 were used (100%).

Objective 4. Table 8 shows students' perceptions of the usefulness of the collaborative learning features in WReSTT. The table shows the results for questions 8 through 12, using a structure similar to that of Table 5. The results were all positive with scores above 3.56 out of 5. These questions

focused on how WReSTT motivated students to collaborate with team members to complete the various tasks in WReSTT.

As stated previously, students were awarded virtual points based on their participation in team activities, as well as their individual participation. It should be noted that question 8 (M = 4.25, SD = 1.62) was the highest value in the table, which showed that the assignment of virtual points encouraged the student to visit WReSTT and complete the assigned tasks. In this context, the results for question 8 reflect more the gamification (Malone 1980) aspect of WReSTT rather than on the collaborative learning. The next closest value was question 9 (M = 4.15, SD = 1.52), which showed that the use of virtual points in WReSTT encouraged a student's team to visit the Web site and complete the tasks. The results of question 9 focuses directly on the collaborative learning aspect of WReSTT.

4.4 Discussion

In this section, we discuss the results presented in the previous section in the context of the four research objectives stated in Section 4.1. The various threats to the validity of our results are also discussed, as well as the impact that these threats can have on the conclusions drawn.

Objective 1. Our results suggest that the availability and knowledge of the use of code coverage tools positively impacts and increases students' propensity to improve the quality of their black-box test suites. These results are shown in Tables 3, 4, and 5, and summarized in Figure 1. It should be noted by the reader that during the black-box testing phase of the project, no code coverage tools were used. Except for one team, Team 3 in Fall 2011 (Table 3), students added test cases or kept the number of test cases unchanged to achieve code coverage of 80% or higher. Note that although Team 3 decreased their number of test cases during unit and subsystem testing, there was an increase of 53% in statement coverage during white-box unit testing and an increase of 33% in statement coverage for the white-box subsystem testing (Table 5). These results would suggest that Team 3 removed redundant test cases from the black-box test suite and added new test cases to exercise different code segments, thereby improving the quality of the test suite.

As mentioned previously, students in the Fall 2011 class were not familiar with how to adequately use the coverage tools to measure branch coverage when testing the software, hence the sparse entries for branch coverage shown in Table 5. It was important to show these results since it supports the fact that using tools and not knowing how to exploit their full potential can reduce the impact the tool can have. Based on the results for the Fall 2011 class, it would have been difficult to draw any conclusions on the relationship between statement coverage and branch coverage.

Trying to improve the code coverage obtained for a given test suite, students are forced to inspect the code and get a better understanding of the logic of the program being tested. Since students were testing programs that they did not develop, performing code inspections allowed them to write better test cases to test the functionality that was sometimes not adequately defined. In addition, students realized that it was sometimes infeasible or difficult to achieve 100% code coverage for a given criterion. Many of the teams in the study documented the reasons for not achieving 100% code coverage in Section 6.2 of the IBTD test document (see Appendix A). The main reasons given by students were as follows: (1) not all requirements were fully implemented in the application, although the documentation stated that they were implemented, and (2) there were parts of the code that were unreachable, specifically some of the exception handlers.

One of the concepts taught in the testing class, as part of white-box testing topic, is the subsumes relation, which provides students some insight into how the various coverage criteria may be related. As mentioned in Section 2.3, students draw the flow graph for toy programs (methods) and then identify test cases to achieve both 100% statement and 100% branch coverage. The advantages

of performing both statement and branch coverage during the white-box testing part of the project are as follows: (1) students experience how the subsume relation may be applied in a practical context on real programs as compared to the toy programs used as examples in class; (2) students realize that branch coverage is one of the stronger white-box testing techniques currently available on a wide cross section of coverage tools; and (3) although there are many other coverage criteria described in the literature, such as all-definitions, all-uses, and all definition-use paths coverage criteria (Ammann and Offutt 2008; Mathur 2008), the stronger criteria are generally not available in coverage tools, specifically the open source coverage tools.

Objective 2. As a result of striving for higher code coverage, the data suggests that there was a slight increase in the number of bugs found, as shown in the three columns on the right side of Table 4. However, this increase was not statistically significant, as pointed out in Section 4.3, and is consistent with previous findings in the literature. This result was not surprising given the formative nature of our experimental design. The research literature states that test coverage criteria does not say anything about the fault-detecting ability of the test suites generated based on a given criteria (Edwards and Shams 2014; Frankl and Weyuker 1993; Inozemtseva and Holmes 2014; Zhu 1996). Note however, that anecdotal evidence shows that there are few, if any, better ways to get students in a software testing or programming class to improve their test suites other than focusing on code coverage provided by testing tools (Edwards 2004).

As stated earlier, the software applications assigned to the teams in the various testing classes during the study were developed by students in previous SE classes. Although testing these software applications provided a good experience for what students could expect in industry, the applications were not ideally suited to identifying bugs during black-box and white-box testing. The problems with the applications were twofold, the software requirements were not completely specified, and not all the requirements were implemented. Student relied on the use cases to identify system requirements; however, it was much more difficult to identify the requirements for the classes under test and the subsystem under test since the detailed design was also not complete. Although we cannot make any definite conclusions based on the results and analysis in Section 4.3, we plan to repeat the studies in the future using candidate software applications in the studies where the requirements are better specified and the implementation is complete.

Objective 3. The results in Table 7 show that students do find WReSTT a useful resource to support learning software testing techniques and how to use testing tools. This is reflected in the average scores shown in the rightmost column of Table 7. The highest average score is 3.99 (out of 5) for question 3, which asked students if the tutorials in WReSTT helped them understand how to use the unit testing tools. The lowest average score is 3.12 (out of a possible 5) for question 6, which asked students if the number of tutorials in WReSTT were adequate.

The results in Figure 3 support the findings that students find the tool tutorials in WReSTT useful. This is reflected in the increasing use of coverage tools from Fall 2011 to Fall 2013. In Fall 2012, one additional tutorial was added for the tool CodeCover (CodeCover Team 2011), the third tool to be used by students during white-box testing. In addition to the tutorials, students were also encouraged to add comments in the forums in WReSTT to help their fellow students use the tools. These comments were particularly helpful in using coverage tools during system testing with IBM RFT (IBM 2015a). There is one anomaly in Figure 3—the bar representing the tool usage for statement coverage during system testing in Fall 2011. Based on the class presentations, only one team was able to automate the process of using the code coverage tool during system testing. The other teams manually collected the coverage data while running the test suite with IBM RFT; however, they incorrectly reported that they used the coverage tools during system testing.

This part of the study shows that WReSTT is useful not only for SE classes, as reported by Clarke et al. (2014b), but also for software testing classes, although to a lesser extent. The main difference is that in the SE class, students found that the number of tutorials were more adequate than in the testing class. This finding can be attributed to the fact that unlike the software testing class where the focus is on software testing techniques and the use of tools, there is limited time in the SE class to cover the testing topics and tools. However, this is one area of WReSTT that is currently being remedied; the WReSTT team is currently developing more tutorials on testing concepts and encouraging students to create new tutorials on testing tools.

Although studies were conducted in both the SE and software testing classes at FIU, there was little overlap with students taking both classes. We gave pretests and posttests to all students who took both sets of studies; however, since we did not have a control group in the study with the software testing classes, we did not report the results. There was one question in the pretest that asked students if they know of any resource that provides resources on software testing. Based on the results of the pretest, there were three students in Fall 2011, two students in Fall 2012, and one student in Fall 2013 who had knowledge of WReSTT. Based on the small number of students who were exposed to WReSTT (7.1%), there is little impact on the results reported in this section. It should also be noted that students are required to use various testing tools during the software testing classes, unlike the SE classes.

Objective 4. The results in Table 8 show that students perceived WReSTT as supportive of collaborative learning since it encourages them to work collectively within their teams to complete tasks and compete against other teams for virtual points. The use of virtual points encouraged students to complete the tasks in WReSTT, resulting in an average score of 4.25 (out of 5) for question 8. The lowest average score recorded was 3.56 for question 12, which asked students if their team devised a plan to get maximum virtual points in WReSTT.

Students also felt positive about the event streaming feature showing how other classmates were doing (question 10 in Table 7). They also thought that the event streaming feature encouraged their teammates to complete their tasks in WReSTT (question 11). As compared to a previous study by Clarke et al. (2014b), students in the testing class were not as enthusiastic about the collaborative learning features in WReSTT as students in the SE class were. This could have been due to the fact that only part of the class project in the SE class focused on testing and students needed to learn most of the testing techniques and tools on their own.

Threats to validity. The threats to validity for the empirical study are presented both in terms of threats to internal and external validity. The internal threats to validity focus on inferring the outcomes as stated in the objectives based on the results presented in this section.

The main internal threat was the gained experience of the instructor teaching the course for the Fall 2011, Fall 2012, and Fall 2013 semesters. Although the same instructor taught the software testing course to all classes in each of the semesters over the 3-year period, the instructor gained additional experience in using code coverage tools and using the collaborative environment in WReSTT. This threat is reflected in the coverage results shown in Table 5. The teams in the Fall 2011 class sporadically used the coverage tools to record branch coverage during unit, subsystem, and system testing during white-box testing. As a result, the instructor encouraged students to become more familiar with the coverage tools and provided extra credit to those students who posted tutorials that helped other students understand how to use the coverage tools. The extra credit was given based on how helpful the tutorial was to the class. As a result, the teams in Fall 2012 and Fall 2013 recorded branch coverage for all forms of testing (see Table 5).

Other internal threats not directly reflected in the results include the number of tools used to record the coverage results during the three semesters and students' ability to accurately document

the results during the testing process. In Fall 2011, two coverage tools were used to record the results, including EclEmma (Hoffmann et al. 2015) and eCobertura (Hofer 2010); in Fall 2012, three tools were used, including the two from Fall 2011 and CodeCover (CodeCover Team 2011). Students in Fall 2013 did the best job of performing code coverage—on average, all three tools were used for statement and branch coverage for unit and subsystem testing, and one tool was used for system testing, recording both statement and branch coverage. However, students in Fall 2011 used the two tools for code coverage at the unit and subsystem level and did not do a good job at the system level.

The other threat reflected in the results is that some teams did not provide enough information in the documentation to report a result or did not interpret the results correctly. The first instance of not appropriately recording data during the project is related to an entry in Table 3, Fall 2013, Team 2, in which 53 subsystem test cases were added (1,767% increase) during white-box testing, but no test results (Pass/Fail) were recorded. The second instance of not recording results is related to an entry in Table 5, Fall 2011, Team 1, where there is no record of statement coverage for unit testing for either the black-box test suite or the white-box test suite. It was clear that some teams could not interpret the results presented in the reports generated by the code coverage tools or did not take the time to record the results. For example, there were some inconsistencies with the screen shots of the tool reports that were available in the deliverable documents and the in-class presentations.

The main external threats to generalizing the impact of using code coverage testing tools in the class and the use of WReSTT in the class are the lack of experimentation conducted by other instructors and conducting the experiments in the same setting (e.g., courses at FIU). The instructor who conducted the studies has been doing software testing research for the past 10 years and may not be representative of instructors teaching a software testing course. Based on the experience of the instructor in software testing, the studies were tweaked in each subsequent semester to improve the use of tools and ensure that students capture the results for the study in their project documentation. FIU is classified as a Hispanic Serving Institution with around 70% Hispanic students, who were reflected in the classes participating in the study. In addition, only 20% of students in the class were female.

We plan to address the internal and external threats of the study by using other instructors to conduct the studies at FIU and perform large-scale studies at multiple academic institutions. We expect that a second instructor will be teaching the undergraduate software testing class in the near future, which will provide an opportunity to repeat the studies. The project supporting this research work includes three other academic institutions, and we are in the process of conducting the WReSTT component of the study at these institutions. These large-scale studies will address the threats related to the interaction of selection and treatment (Creswell 2004).

5 RELATED WORK

Carver and Kraft (2011) conducted an empirical study to determine whether students would be able to adequately test programs using complete test suites. The study involved two offerings of a senior-level computer science (CS) course and consisted of two steps: (1) the manual creation of a test suite and (2) the use of a code coverage tool—CodeCover (CodeCover Team 2011)—to improve the test suite. Three hypotheses were tested in the study that focused on the following: (1) students would be able to manually test a given program and achieve 100% coverage without the use of a tool, (2) students would obtain greater coverage when given a tool, and (3) students would not be able to use code coverage information to systematically improve their test suite to obtain a small yet complete set of tests.

The results in Carver and Kraft (2011) showed the following: (1) students were not able to achieve 100% code coverage on a small program without additional training in testing and the use of a tool; (2) provided with a tool, they were able to significantly increase branch and statement coverage; and (3) students were able to add test cases to increase code coverage without a deep understanding of which test cases would increase code coverage. Our work is similar to the work of Carver and Kraft (2011), except we are performing the study in a software testing class and are using a larger software project. Our project is so large that it is not expected that students identify a test suite to get a high percentage of coverage unless tools are used.

Garousi (2011) described an approach of incorporating real-world industrial testing projects into a graduate testing course. A description of the course is presented in the work and the testing projects are described. The topics in the course include white-box testing, black-box testing, and testing object-oriented systems. The course includes a project that requires students to work with several industrial software testing tools, including Cobertura (Hofer 2010), FitNesse (FitNesse Team 2015), IBM RFT (IBM 2015a), IBM Rational Test RealTime (IBM 2015b), JUnit (Gamma and Beck 2015), and NUnit (NUnit Team 2015), among others.

The work by Garousi (2011) identified several lessons learned from the project, including the need to proactively monitor student progress, the need to be realistic about the workload associated with such projects, and the need to ensure that students are focused not only on learning how to use the tools. In our work, we provided students the next best thing to working on a real-world project: testing an SE project developed by students in a previous SE class. The project is usually not complete (i.e., not all requirements are implemented), and it is usually difficult for students to identify the oracles for the test cases at the different levels of testing. In our studies, students usually complained about how difficult it was to run the software and understand the code.

There have been several educators and researchers leading efforts to integrate software testing into courses throughout the CS/IT curriculum (Dvornik et al. 2011; Edwards 2004; Frezza 2002; Janzen and Saiedian 2006; Jones 2000; Kaner and Padmanabhan 2007). Edwards (2004, 2015) created the Web-Based Center for Automated Testing (Web-CAT), which provides students access to code coverage tools to support testing of their programs. Web-CAT also supports grading programs by providing instructors the ability to generate student program reports that provide a code correctness score, a test completeness score with respect to code, and a test completeness and validity score with respect to the problem. Each student may also access his or her report in Web-CAT, which allows the student to improve the quality of the code. Based on the instructor settings of Web-CAT, students may submit their programs an unlimited number of times before the deadline.

The work of Edwards (2004, 2015) plays an important role of helping students to better test their programs by allowing them to easily access code coverage tools in the Web-CAT platform. Unlike the work by Edwards, we focus on providing students the opportunity to learn how to install and use the various code coverage tools for the three levels of testing. This allows students to experience the difficulties associated with instrumenting source code to then be used by a system testing tool such as IBM's RFT tool (IBM 2015a). When students are faced with these problems, the collaborative learning aspect of the class comes into play by posting solutions in the WReSTT environment to be used by other students.

Clarke et al. (2014b) described the WReSTT environment and a study conducted to determine the efficacy of using a minimally disruptive approach that integrates software testing into SE courses. WReSTT was initially designed to be a repository containing tutorials for software testing tools only, but it has since evolved to contain content of software testing concepts and techniques in the form of learning objects (Koohang 2004; Smith 2004). The results of the studies that focused on using WReSTT to support students' training in software testing in SE courses showed that (1) WReSTT can help students in an SE course better understand software testing techniques,

(2) WReSTT can help students in an SE course improve their knowledge and use of testing tools, (3) students find WReSTT engaging and enjoyable to use, and (4) students who use WReSTT are more likely to use software testing tools while working on the SE course project.

In this article, we repeated the aspect of the study by Clarke et al. (2014b) that is related to the usefulness of WReSTT as a learning resource in the context of a software testing class. The results of our study show that WReSTT is useful both as a learning resource by providing tutorials and as a collaborative learning environment by supporting team activities. However, students in the SE class found WReSTT to be more useful than in the software testing class, particularly with respect to the learning content on software testing concepts and unit testing tools. The other difference was that students in the SE class stated that they were less likely to use a testing tool in the project if the WReSTT were not available. This was expected, as students in the software testing class were expected to use testing tools.

6 CONCLUSIONS

In this article, we described our experiences of teaching three semesters of an undergraduate software testing class focusing on the use of testing tools and with the support of a cyberlearning environment (WReSTT). Our description of the software testing course included the pedagogical approach, the course structure, and the course project, and how WReSTT was used in the course. We conducted an empirical study that focused on the course project, including various testing activities, and the impact WReSTT had on students' ability to learn various testing techniques and tools.

The study, which was conducted over three semesters of the course, produced results that supported the following assertions: (1) the availability and knowledge of the use of code coverage tools positively impacts and increases students' propensity to improve the quality of their black-box test suites, (2) increasing the number of test cases required to improve the code coverage does not necessarily improve the bug-finding ability of the test suite, and (3) WReSTT is a useful resource for learning software testing in an undergraduate course. We plan to perform additional studies to better understand the impact of increasing code coverage on finding bugs, and explore how WReSTT can be more engaging for students and how it can be used effectively in a wider cross section of CS/IT courses.

APPENDIX

A FORMAT: IBTD TEST DOCUMENT

Cover Page
i. Abstract
ii. Table of Contents
1. Introduction
  1.1 Overview
  1.2 Requirements of the System
  1.3 Overall Testing Approach
  1.4 Terminology
  1.5 Document Organization
2. Implementation-Based Test Plan
  2.1 Team Organization
  2.2 Hardware and Software Requirements
  2.3 Test Reference Items
  2.4 Tested Features
  2.5 Features Not Tested
  2.6 Work Schedule

3. Unit Testing
Two classes were identified for unit test
3.1 Test Identification and Objective
3.2 Test Criteria and Procedures
3.3 Test Cases
3.4 Actual Test Results
4. Subsystem Testing
One subsystem was identified for subsystem test
Same structure as unit testing but focuses on a subsystem
5. System Testing
Same structure as unit testing but focuses on a system
6. Test Summary Report
6.1 Failure Identification and Explanation
6.2 Comparative Analyses of Code Coverage Testing Tools
7. Risk and Contingencies
8. Team Approval
9. Glossary
10. Appendix

B WRESTT SURVEY

This survey will NOT impact your course grade.

Panther ID Number (DO NOT use your name):

1. Have you ever used a learning resource other than WReSTT to learn about testing concepts or testing tools? Please circle your answer: Yes No
Please use the following scale and complete questions 2 through 7: *1 = Strongly Disagree, 2 = Disagree, 3 = Neither Agree nor Disagree, 4 = Agree, 5 = Strongly Agree, NA = Not Applicable.*

	Learning Material Related Questions	**Rating**
2	The tutorials in WReSTT helped me to better understand testing concepts.	
3	The tutorials in WReSTT helped me to better understand how to use unit testing tools.	
4	The tutorials in WReSTT helped me to better understand how to use code coverage testing tools.	
5	The tutorials in WReSTT helped me to better understand how to use functional testing tools.	
6	The number of tutorials in WReSTT is adequate.	
7	I would have used testing tools in my project if WReSTT did not exist.	
	Collaborative Learning Related Questions	
8	The use of virtual points in WReSTT encouraged me to visit the website and complete the tasks.	
9	The use of virtual points in WReSTT encouraged my team to visit the website and complete the tasks.	
10	The event stream showing the activities of the other members in the class encouraged me to complete my tasks in WReSTT.	
11	The event stream showing the activities of the other members in the class encouraged my team to complete our tasks in WReSTT.	
12	Our team devised a plan to get the maximum number of points in WReSTT.	

ACKNOWLEDGMENTS

The authors would like to thank the students in the CEN4072 Fundamentals of Software Testing classes at Florida International University that participated in the study. We also appreciate the insightful comments provided by the associate editor and reviewers on how to improve the article.

REFERENCES

AAAS and NSF. 2013. Transforming Undergraduate Education in STEM: Making and Measuring Impacts. In *Proceedings of the 2013 TUES Principal Investigators (Pls) Conference*. http://ccliconference.org/files/2013/01/2013-Tues-Conference-Program.pdf.

Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA'10)*. ACM, New York, NY, 153–160. DOI: http://dx.doi.org/10.1145/1869542.1869567

ACM/IEEE-CS Interim Review Task Force. 2008. Computer Science Curriculum 2008: An Interim Revision of CS 2001. Retrieved June 12, 2017, from http://www.acm.org/education/curricula/ComputerScience 2008.pdf.

Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press, New York, NY.

Robert V. Binder. 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley Longman, Boston, MA.

P. Bourque and R. Dupuis. 2004. *Guide to the Software Engineering Body of Knowledge 2004 Version*. IEEE, Los Alamitos, California.

Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. 2013. Reversible Debugging Software "Quantify the Time and Cost Saved Using Reversible Debuggers." Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.370.9611

Bernd Bruegge and Allen H. Dutoit. 2009. *Object-Oriented Software Engineering Using UML, Patterns, and Java (3rd ed.)*. Prentice Hall, Upper Saddle River, NJ.

J. C. Carver and N. A. Kraft. 2011. Evaluating the testing ability of senior-level computer science students. In *Proceedings of the 24th IEEE-CS Conference on CSEET*. IEEE, Los Alamitos, CA, 169–178. DOI: http://dx.doi.org/10.1109/CSEET.2011.5876084

Peter J. Clarke, Debra Davis, Raymond Chang-Lau, and Tariq King. 2014a. Observations on student use of tools in an undergraduate testing class. In *121st American Society for Engineering Education (ASEE)—Software Engineering Constituent Committee Division Track (SWECC)*. Paper No. 10123. ASEE, Washington DC.

Peter J. Clarke, Debra Davis, Tariq M. King, Jairo Pava, and Edward L. Jones. 2014b. Integrating testing into software engineering courses supported by a collaborative learning environment. *Transactions on Computing Education* 14, 3, Article No. 18. DOI: http://dx.doi.org/10.1145/2648787

Peter J. Clarke, Jairo Pava, Debra Davis, and Tariq M. King. 2012. Using WReSTT in SE courses: An empirical study. In *Proceedings of the 43rd SIGCSE Conference*. ACM, New York, NY, 307–312.

CodeCover Team. 2011. CodeCover Home Page. Retrieved June 12, 2017, from http://codecover.org/

Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences (2nd ed.)*. Lawrence Erlbaum, NJ.

John W. Creswell. 2004. *Educational Research: Planning, Conducting, and Evaluating Quantitative and Qualitative Research* (2nd ed.). Prentice Hall, Upper Saddle River, NJ.

Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart Nacke. 2011. From game design elements to gamefulness: Defining "gamification." In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments (MindTrek'11)*. ACM, New York, NY, 9–15. DOI: http://dx.doi.org/10.1145/2181037.2181040

E. W. Dijkstra. 1970. *Software Engineering Techniques*. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, October 1969.

T. Dvornik, D. S. Janzen, J. Clements, and O. Dekhtyar. 2011. Supporting introductory test-driven labs with WebIDE. In *Proceedings of the 24th IEEE-CS Software Engineering Education and Training Conference (CSEET'11)*. IEEE, Los Alamitos, CA, 51–60.

Stephen Edwards. 2015. Web-CAT: The Web-Based Center for Automated Testing. http://web-cat.org/.

Stephen H. Edwards. 2004. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04)*. ACM, New York, NY, 26–30. DOI: http://dx.doi.org/10.1145/971300.971312

Stephen H. Edwards and Zalia Shams. 2014. Comparing test quality measures for assessing student-written tests. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion'14)*. ACM, New York, NY, 354–363. DOI: http://dx.doi.org/10.1145/2591062.2591164

FitNesse Team. 2015. FitNesse Home Page. Retrieved June 12, 2017, from http://www.fitnesse.org/.

P. G. Frankl and E. J. Weyuker. 1993. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering* 19, 3, 202–213. http://dx.doi.org/10.1109/32.221133

S. Frezza. 2002. Integrating testing and design methods for undergraduates: Teaching software testing in the context of software design. In *Proceedings of the 32nd Annual Conference on Frontiers in Education (FIE'02)*. Vol. 3. IEEE, Los Alamitos, CA.

E. Gamma and K. Beck. 2015. JUnit Home Page. Retrieved June 12, 2017, from http://www.junit.org/.

V. Garousi. 2011. Incorporating real-world industrial testing projects in software testing courses: Opportunities, challenges, and lessons learned. In *Proceedings of the 24th IEEE-CS Conference on CSEET*. IEEE, Los Alamitos, CA, 396–400. DOI: http://dx.doi.org/10.1109/CSEET.2011.5876112

Michael Hackett. 2013. The changing landscape of software testing. *LogiGear Magazine* 7, 1, 7–15. http://www.logigear.com/magazine/wp-content/uploads/2013/02/LogiGear-Magazine-The-Rapidly-Changing-Testing-Landscape1.pdf.

Joachim Hofer. 2010. eCobertura Home Page. (August2010). Retrieved June 12, 2017, from http://ecobertura.johoop.de/.

Marc R. Hoffmann, Brock Janiczak, and Evgeny Mandrikov. 2015. EclEmma Home Page. Retrieved June 12, 2017, from http://www.eclemma.org/.

IBM. 2015a. Rational Functional Tester. Retrieved June 12, 2017, from http://www-01.ibm.com/software/awdtools/tester/functional/.

IBM. 2015b. Rational Test RealTime. Retrieved June 12, 2017, from http://www-03.ibm.com/software/products/en/realtime.

Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 435–445. DOI: http://dx.doi.org/10.1145/2568225.2568271

David S. Janzen and Hossein Saiedian. 2006. Test-driven learning: Intrinsic integration of testing into the CS/SE curriculum. *ACM SIGCSE Bulletin* 38, 1, 254–258. DOI: http://dx.doi.org/10.1145/1124706.1121419

Edward L. Jones. 2000. Software testing in the computer science curriculum—a holistic approach. In *Proceedings of the Australasian Conference on Computing Education (ACSE'00)*. ACM, New York, NY, 153–157.

C. Kaner and S. Padmanabhan. 2007. Practice and transfer of learning in the teaching of software testing. In *Proceedings of the 20th Conference on Software Engineering Education Training (CSEET'07)*. IEEE, Los Alamitos, CA, 157–166. DOI: http://dx.doi.org/10.1109/CSEET.2007.38

Alex Koohang. 2004. Creating learning objects in collaborative e-learning settings. *Issues in Information Systems* 4, 2, 584–590. *http://www.iacis.org/iis/2004/Koohang.pdf*

Ilaria Liccardi, Asma Ounnas, Reena Pau, Elizabeth Massey, Päivi Kinnunen, Sarah Lewthwaite, Marie-Anne Midy, and Chandan Sarkar. 2007. The role of social networks in students' learning experiences. *ACM SIGCSE Bulletin* 39, 4, 224–237. *http://doi.acm.org/10.1145/1345375.1345442*

Thomas W. Malone. 1980. What makes things fun to learn? Heuristics for designing instructional computer games. In *Proceedings of the 3rd ACM SIGSMALL Symposium and the 1st SIGPC Symposium on Small Systems (SIGSMALL'80)*. ACM, New York, NY, 162–169. DOI: http://dx.doi.org/10.1145/800088.802839

Aditya P. Mathur. 2008. *Foundations of Software Testing*. Pearson Education, Delhi, India.

John D. McGregor and David A. Sykes. 2001. *A Practical Guide to Testing Object-Oriented Software*. Addison Wesley Longman, Boston, MA.

NUnit Team. 2015. NUnit Home Page. Retrieved June 12, 2017, from http://www.nunit.org/.

Selenium Development Team. 2014. SeleniumHQ Home Page. Retrieved June 12, 2017, from http://www.seleniumhq.org/.

Charlie Y. Shim, Mina Choi, and Jung Yeop Kim. 2009. Promoting collaborative learning in software engineering by adapting the PBL strategy. In *Proceedings of the WASET International Conference on Computer and Information Technology (ICCIT'09)*. IEEE, Los Alamitos, CA, 1167–1170.

Barbara Leigh Smith and Jean T. MacGregor. 1992. What is collaborative learning? In *Collaborative Learning: A Sourcebook for Higher Education*, A. S. Goodsell, M. R. Maher, and V. Tinto (Eds.). National Center on Postsecondary Teaching, Learning, and Assessment, University Park, PA.

Rachel S. Smith. 2004. *Guidelines for Authors of Learning Objects. New Media Consortium.* Available at https://www.nmc.org/publication/guidelines-for-authors-of-learning-objects.

Jean Tesser. 2010. Dependency Finder. Retrieved June 12, 2017, from http://depfind.sourceforge.net.

Wikipedia. 2013. Category:Software testing tools. Retrieved June 12, 2017, from http://en.wikipedia.org/wiki/Category:Software_testing_tools.

WReSTT Team. 2015. WReSTT: Web-Based Repository for Software Testing Tutorials. Retrieved June 12, 2017, from http://wrestt.cis.fiu.edu/.

Qian Yang, J. Jenny Li, and David Weiss. 2007. A survey of coverage-based testing tools. *Computer Journal* 52, 5, 589–597.

Hong Zhu. 1996. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering* 22, 4, 248–255. DOI: http://dx.doi.org/10.1109/32.491648

Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys* 29, 366–427.

Received April 2015; revised August 2016; accepted December 2016