Helping Students Test Programs That Have Graphical User Interfaces

Matthew Thornton, Stephen H. Edwards,
and Roy Patrick Tan
Dept. of Computer Science, Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
+1 540 231 5723

{ thorntom, edwards, rtan}@cs.vt.edu

ABSTRACT

Within computer science education, many educators are incorporating software testing activities into regular programming assignments. Tools like JUnit and its relatives make software testing tasks much easier, bringing them into the realm of even introductory students. At the same time, many introductory programming courses are now including graphical interfaces as part of student assignments to improve student interest and engagement. Unfortunately, writing software tests for programs that have significant graphical user interfaces is beyond the skills of typical students (and many educators). This paper presents initial work at combining educationally oriented and open-source tools to create an infrastructure for writing tests for Java programs that have graphical user interfaces. Critically, these tools are intended to be appropriate for introductory (CS1/CS2) student use, and to dovetail with current teaching approaches that incorporate software testing in programming assignments. We also include in our findings our proposed approach to evaluating our techniques.

Keywords: on-line education, computer science, test-driven development, test-first coding, GUI, objectdraw, JUnit

1. INTRODUCTION

With the loss of productivity and system downtime caused by code defects, it is important for information technology students to learn how to test software. At the same time, however, lack of educational support for testing and quality assessment in university computer science curricula can result in students that are ill prepared for producing commercial-quality code. As a result, it is becoming more common for computer science educators to include software testing activities across multiple courses, often by adding software testing requirements to traditional programming assignments. Approaches include using explicit instructor-provided tests in assignment specifications, using instructor-provided tests for automatic grading, requiring students to write test plans and test cases, and even requiring students to practice test-driven development (TDD). Recent studies using test-driven development (TDD) in the classroom show that students produce higher quality code when they write their own tests, with a 28% reduction in the number of bugs per thousand lines of student-written code (KSLOC), on average [14]. In fact, when students were required to write their own tests and were graded on how well they did this using our techniques, the top 20% of students in our most recent experimental evaluation achieved defect rates of approximately 4 defects per KSLOC or better, which is comparable to most commercial-quality software written in the United States. Of the students in the control group who were not required to turn in their own tests and were not evaluated on their own testing behavior, none achieved this level of performance, with the best score reaching only 30 defects/KSLOC [14, 16]. Consequently, it has been demonstrated that test-driven development has some impact on the quality of student-written code.

Testing frameworks, such as JUnit for Java [3, 22] and similar XUnit frameworks for languages such as C++ [7] are a critical enabling factor in developing a curriculum around test driven development. Many educators have found that JUnit makes writing tests easy, even for introductory-level students. Most modern interactive development environments for Java, including those targeted at educational communities, offer student-friendly support for JUnit. The spread of JUnit as the de facto standard for writing unit-level tests in Java has provided a useful educational advantage in this regard.

At the same time, however, it is also becoming more common for introductory programming courses to include graphical user interface (GUI) aspects in assignments. GUIs are a common metaphor used in discussing object-oriented programming techniques [10]. GUIs also aid in explaining basic programming concepts, because activities such as implementing the "what happens next" response to a mouse click or what happens when you drag and drop an item into a bin can be quickly understood by beginning programmers. This, in addition to the proliferation of GUI frameworks available such as Swing [5] and objectdraw [11], makes teaching students to program GUIs a very inviting prospect for instructors. However, while there are a number of level-appropriate educational GUI frameworks to simplify teaching tasks, there is no level-appropriate support for *testing* GUI applications. This dilemma is illustrated in Figure 1. Consequently, it is necessary to develop a framework that allows introductory computer science students to develop test cases for their GUI-based programming assignments if one wishes to include software testing activities.

To address this problem, we are adapting an introductory GUI package called objectdraw [11], together with the Abbot GUI testing library [1, 13] (based on JUnit), to develop a student-friendly testing framework for GUI assignments. These tools can be extended to make student testing easier and can be inte-

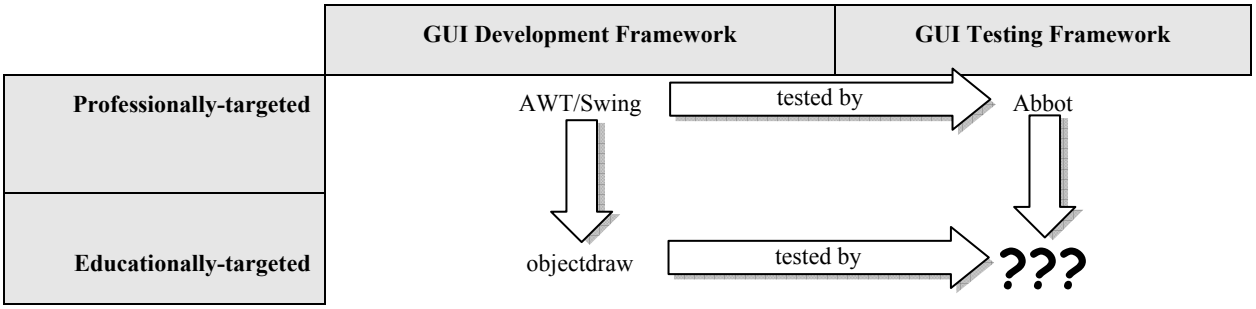

Figure 1: The goal is to create level-appropriate educational support for testing objectdraw-based applications, using Abbot as the underlying technology.

grated into existing software development and testing tools, such as BlueJ, Eclipse, and Web-CAT. Once students have been given a complete suite of frameworks and tools, we can then evaluate student performance at GUI-based software testing.

The rest of this paper discusses the work that is being developed in creating student-testable GUIs. Some of the previous work in the area of student software testing, GUI testing, and GUI frameworks will be discussed in the following section, followed by a discussion of how we are adapting those tools for student testing of GUIs in Section 3. We will follow this discussion with a brief summary of our initial observations from students and a proposal for formally evaluating these techniques.

2. RELATED WORK

Over the past 5 years, the idea of including software testing in student programming assignments across many courses has grown in popularity with many different results and observations being documented [14, 17, 18, 19]. Automated software evaluation tools such as Web-CAT have been widely documented in the literature as an approach to evaluating student performance in programming assignments and closed labs [14, 15, 16, 17, 18, 19]. However, prior to this work, Web-CAT has been unable to run tests on GUI-based programming assignments.

Kim Bruce's work on the objectdraw library [11, 10] is a way of simply abstracting the Swing library for Java to make writing GUIs easier on students. This package will be extended to provide students with a way to assert properties about particular shapes in their program. This technology will be the source for developing student assignments using GUIs and because it is based on Swing, it can make use of many additional packages that can test GUIs designed from Swing components.

Abbot is a mature, professional-level testing framework based on JUnit. It allows one to test partially-developed GUI code as well as entire GUI-driven applications [1, 13]. It is not necessary to have a complete, runnable program in order to write or execute Abbot-based tests. Abbot provides especially strong support for students who are using test-driven development [8, 9, 14, 15], where one rapidly cycles between adding new test cases and incrementally extending code. Abbot also supports comprehensive record-and-playback functionality, including a script editor called Costello for hand constructing or modifying recorded event sequences. However, it is different from many other GUI testing frameworks in its support for simple, clean, hand-written test cases targeted at code under development instead of complete applications.

Among educationally oriented interactive development environments (IDEs) for students learning to program, BlueJ is widely used by those learning Java. BlueJ provides particularly strong support for JUnit-based testing, because it allows students to directly create and interact with raw Java objects using only the mouse, and also interactively "record" these actions as a test case, even before one knows how to write JUnit-style test cases explicitly [2, 21]. Because BlueJ is so widely used in the educational community, we have chosen to explore supporting its interactive test case recording mechanism for use with GUI test cases, although the general GUI framework we describe here is applicable in all IDEs.

3. TOOL DEVELOPMENT FOR GUI TESTING

Two significant issues make it difficult to apply existing GUI testing tools in introductory computer science courses. First, existing GUI tools require extensive programming knowledge that introductory students do not possess. Second, these tools typically require a lot of work to setup and use. Consequently, developing GUI tests with such tools appears to students to be time-consuming busywork—that is, overhead in addition to actually completing the assignment—rather than a value-adding activity that makes assignments easier to complete. Consequently, changes and enhancements must be made at multiple levels of the student's development process to guarantee that these roadblocks can be removed.

3.1 The Objectdraw Library

The objectdraw library was originally designed to present a simplified, streamlined application programming interface (API) to students, making it easy for them to write simple but expressive GUI programs with just a few lines of code and no excess clutter. Objectdraw was not designed to support writing GUI-based tests, however. Most critically, it offers no support for writing *assertions* about the state of a program's graphical interface. Assertions are simple statements or claims about the state of an object or a collection of objects, and most test cases use some form of assertion to express the expected behavior or

intended effect of a sequence of actions. If you want to express claims about the content of or changes in a program's GUI, you must be able to "talk about" the various pieces that make up the GUI. To this end, objectdraw must be extended to support assertions that are appropriate to a student's level of knowledge and performance. These assertions must be at a sufficiently introductory level while, at the same time, providing sufficient test cases that the students are able to create tests that are worthwhile to run.

Objectdraw provides a primary class from which main programs descend: the *FrameWindowController* class [4]. The FrameWindowController class manages what a student "sees" in the graphical user interface. It also includes the mouse events that can occur in a graphical user interface, including mouse clicks, mouse movement, leaving the window area, entering the window area, and pressing and releasing the mouse button. Many student assignments that use the objectdraw library [10] revolve around implementing an extension to the *FrameWindowController* class. Consequently, as a typical course progresses, students become very familiar with the behavior of this class.

In order to make assertions about the properties of shapes that are rendered on the *FrameWindowController*, we enhanced the *FrameWindowController* by creating a new subclass. The *TestableWindowController* class is a subclass of *FrameWindowController* that includes several primitive assertion operations. These primitive assertions allow one to express claims about the existence of the various kinds of 1- or 2-dimmensional shapes that can be created in the objectdraw library. There are also primitives that allow asserting that low-level shapes have specific properties (color, size, etc.). While these methods are simple, they provide a basic platform for writing assertions about the structure or content of a GUI.

At the same time, however, the primitive assertions in *TestableWindowController* are more detailed and more complex than we expect beginning students to use, especially when they first start out. As a result, we have added an additional layer of abstraction by creating a *StudentTestableWindowController* class. This class is a subclass of the *TestableWindowController* class and includes methods that are significantly simplified, including methods such as assertCanvasEmpty(), assertFramedRectangleExists(), and others. The methods in this second subclass require few or no parameters. This allows very basic assertions to be used by students, even if method parameters have not been discussed in the course at that point. Further, these methods dovetail nicely with the test recording apparatus provided by BlueJ, for example, so that students can interactively play out their test cases in a meaningful way, even before they have mastered programming.

The assertions that are included in both *TestableWindowController* and *StudentTestableWindowController* behave similarly to the standard assertions provided by JUnit, and are fully compatible with that testing framework.

The assertions that come in the *StudentTestableWindowController* class come in two varieties. One gorup of assertion methods in this class allows one to check the basic properties of an object at a particular screen location. Two parameters are included in the list of formal parameters that represent the x and y coordinates in the drawable area of the window. The student enters the coordinates and the assert method looks for an object at that location to see if an object exists at that location with the desired properties. The second group of assert methods instead take a reference to a specific GUI object, instead of the object's (x, y) coordinates. This can be especially useful when instructors wish to simulate real-world behavior by introducing randomness to an assignment or if a particular test requires that a very specific object on the canvas have a certain property. This is an opportunity for instructors to introduce accessor methods to the students and provides a mechanism for asserting behavior about the objects that are returned by those accessor methods.

```
public void testInitialCondition()
{
    x.onMouseMove(new Location(100, 100));

    x.assertFramedRectExistsAt(20, 100);
    x.assertFramedRectExistsAt(100, 100);
    x.assertFramedRectExistsAt(180, 100);
    x.assertTextExistsAt(35, 120);
    x.assertTextExistsAt(115, 120);
    x.assertTextExistsAt(195, 200);
    x.assertFilledRectExistsAt(60, 20);
    x.assertFramedRectExistsAt(60, 20);
    x.assertTextExistsAt(20, 200);
    x.assertTextExistsAt(100, 200);
    x.assertTextAt(35, 120, "whites");
    x.assertTextAt(115, 120, "darks");
    x.assertTextAt(195, 120, "colors");
    x.assertTextAt(20, 200, "correct = 0");
    x.assertTextAt(100, 200, "incorrect = 0");
    x.assertColorAt(60, 20, Color.white);
}
```

Figure 2: An example test case testing the initial state of a laundry sorting program. The x variable is an instance of the `LaundrySorter` class.

Figure 2 provides an example of a test case that students could write about the behavior of a program in its initial state. The assertions are provided simply by making assertion calls like you would make any other method call. The assertions provided here indicate information about the initial state of a "laundry sorter" application where a swatch is dropped in one of 3 different bins.

3.2 Testing GUI Applications with Abbot

Abbot is a professional-quality tool for testing Java graphical user interfaces. Unfortunately, it has several features that make it difficult for beginning students to understand, and that also make it difficult for beginning students to write GUI tests. First, creating a tester in Abbot requires more knowledge than most beginning computer science students have. Abbot makes heavy use of anonymous classes and listeners, and is geared toward developers using the full power of the Swing library rather than a simplified, educationally-oriented library. Second, there is a mismatch between the ways that certain low-level information is represented between objectdraw and Abbot, including the representation for locations within a GUI window.

Objectdraw provides a *Location* class that represents a location on the canvas. It is much like Java's *Point* class with additional

methods included. Abbot uses native Java *Point* objects to specify locations, and then uses a "robot" to serve as an automated tester to interact with the program being tested. Crossing from *Location*s and *Point*s and back, especially with the high degree of similarity, imposes an unnecessary cognitive load on students, making them feel taxed and that they are just going through "administrative overhead."

To solve these problems, we have implemented a façade to the Abbot robot tester to perform all of the tasks that an introductory student would want to do. The *VTControllerTester* class is a wrapper class that encapsulates a lot of the functionality of a basic Abbot tester, but does so from the objectdraw perspective. The *VTControllerTester* class includes a constructor whereby instead of specifying a static name for the class like you do in the *ComponentTester* class in the Abbot package, you pass the specific instance of the object to be tested. Furthermore, mouseClick(), mousePress(), mouseRelease(), mouseMove(), and mouseDrag() methods are implemented that take a *Location* parameter rather than a *Point* parameter. As a result, students will be writing tests that are familiar to them in terms of the behavior of objectdraw, rather than the behavior of the Abbot package. This dramatically reduces the overhead of students having to "wade through" the mountain of code that was written for the Abbot package and focuses their attention on one class.

One of the issues with testing using Abbot and objectdraw is wading through the multithreading issues that occur between the two. Objectdraw has a model for how all of the objects are loaded in a window and when the window itself is opened. On the other hand, the Abbot library does not know anything about the specific timing constraints imposed by objectdraw, and it may begin its actions before the window is completely set up. Another problem involves testing in a multiwindow environment, particularly in Windows XP/2000. To solve these problems, the *VTControllerTester* attempts to manage the thread behavior so that the robot does not begin until the window is completely set up, and then the robot performs actions only when expected.

Figure 3 is an example of a test case that a student might write

```
public void testDragMode()
{
  prepareNewNonOverlappingSquareGame(x);

  tester.actionMousePress(new Location(10, 10));
  tester.actionMouseMove(new Location(25, 25));

  x.assertVisible(x.getSmallBox());
  x.assertVisible(x.getMediumBox());
  x.assertVisible(x.getLargeBox());

  tester.actionMouseRelease();

  x.assertInvisible(x.getSmallBox());
  x.assertInvisible(x.getMediumBox());
  x.assertInvisible(x.getLargeBox());
}
```

Figure 3: A test for mouse dragging in the Invisible-Game programming assignment. In this case, the mouse is pressed and then moved to simulate dragging. The game's invisible boxes must be visible during dragging, but then disappear once dragging is complete.

to test mouse movement in a program. This example uses the *VTControllerTester* class for a programming assignment called *InvisibleGame*. Students who are writing their own *Invisible-Game* are creating a simple GUI application that places three invisible boxes on the screen. The user attempts to click these boxes, getting "hot" or "cold" feedback about how close they are and racking up points for successful hits. In the assignment, the student is also asked to provide a "cheat mode" (or debug mode) that makes the three boxes visible whenever the mouse is dragged. In this case, the student has explicitly depressed the mouse button at some location on the screen, moved the mouse, and then released the mouse button. This behavior is easy for students to understand, since they drag the mouse on the screen every day.

The test case in Figure 3 shows what a student might do to simulate the click-drag-release action they would perform manually with the mouse. It also shows how one would make claims about the state of the GUI interleaved with the mouse actions.

3.3 BlueJ Development Environment

The test cases shown in Figures 2 and 3 could be written using any IDE. However, because BlueJ [2] is so widely used, and because it provides such strong support for unit testing even before students know how to write test cases, we are interested in exploring how the objectdraw and Abbot extensions we are developing can be used within that environment.

As a student writes small increments of code, they can immediately begin to write (and run) simple test cases that cover the behavior they have just implemented. In BlueJ, the student can instead *interactively record* test cases by directly manipulating live objects. The objectdraw extensions described here allow students to record test cases for GUI-based classes in the same way that they create and record a test case for any other class in BlueJ. Mouse events can be recorded by right-clicking on the program in BlueJ's ObjectBench and invoking the corresponding event method. A student can record which actions they wish to execute by referring to the *VTControllerTester* actions to perform actions and then assert that the program has the correct behavior using the methods available in the *StudentTestableWindowController* class. Because of the way that methods can be referenced in the object workbench, the mouse events can also be explicitly called by the tester, rather than making Abbot calls. This allows students to continue to test their methods and is a substitute for writing test cases using Abbot. Instructors, therefore, have the option of including the Abbot test cases or simply having the students test the behavior of their program by calling the mouse event methods like they would any other method.

At the same time, however, we are also interested in supporting direct, live recording of mouse interactions. This would mean that the student need only point, click, drag, etc., right in the program's main window itself, rather than go through BlueJ's ObjectBench. While this has not been fully implemented yet, it remains as a key aspect of future work.

3.4 Automatic Grading with Web-CAT

Web-CAT is an automated testing environment that allows students and instructors to test student submissions [14]. The system grades student submissions based on the successfulness of a student's program against their *own* tests (as well as the amount of their program was covered in the tests) as well as the instructor's tests. Web-CAT supports assignments written in virtually any programming language, but it is most heavily used by instructors teaching in Java. Until now, automatically grading GUI-based programs on Web-CAT was not feasible, since there was no effective way to write executable tests for such programs in a way that students (or instructors) could manage. However, the extensions to the objectdraw framework described here now allow instructors to write tests for GUI-based programs as easily as students can. The result is that Web-CAT has now been successfully used to automatically execute and evaluate both student-written and instructor-provided tests for GUI-based programming assignments.

4. INITIAL DEPLOYMENT AND FEEDBACK

Virginia Tech has deployed the GUI testing framework for its CS1 course. The framework includes the BlueJ IDE and the objectdraw and Abbot extensions. Web-CAT has been updated to include a submission profile that allows instructors to create GUI-based programming assignments and the types of assignments that have been developed include the following:

- *Square Lab*: A student creates a square on the screen that disappears when the mouse button is pressed and reappears when the mouse is released. This is just to get students acquainted with the environment.
- *Squares Lab*: A student can create multiple squares by clicking at different locations on the canvas. The squares change from an initial red color to blue once a new square has been created. This lab exposes students to writing classes with private members.
- *Laundry Sorter Lab*: The laundry sorter is a simulator where students drag and drop a differently-colored swatch into one of 3 bins and is given credit for a correct or incorrect selection, exposing students to conditionals, program "states" and more complex mouse behavior.
- *Bullseye Lab*: In the bullseye lab, students draw a bullseye *recursively*. The bullseye can then be moved around the screen and its size changes as the mouse moves "faster" or slower.
- *Invisible Squares Program*: This programming assignment tests everything students have done up until the Laundry Sorter lab by giving students a game to implement where they try to find invisible boxes by clicking on the screen. The program provides feedback with regards to how close the student has come to clicking the box.

Initial observations have led to several improvements to the design of the *StudentTestableWindowController* class (For example, breaking the assertions into multiple subclasses would reduce the number of assertions that students have to wade through in searching for a particular assertion) as well as the amount of detail that is needed for lab and programming assignment instructions.

5. PROPOSED EVALUATION

In order to gauge the effectiveness of incorporating GUI testing into assignments, we have prepared to do a large amount of analysis. On submitting all of these assignments to Web-CAT, we will have a large number of student submissions to do analysis on. Web-CAT keeps a large amount of data on each submission and with additional research that's being done on creating reports based on that data [6], there will be a very practical system from which we can get a great deal of useful information out of the student's submission results. These can include things like:

- An aggregate comparison of student submissions and grades this semester versus previous semesters.
- A comparison of the success of student tests versus the instructor reference tests.
- An analysis of how many tests students wrote based on how much a particular lab was completed.
- An analysis of which types of test cases were most commonly passed/failed.

There are, obviously other reports that can be run, as well and each of these types of reports can gain us additional information about how students perceive the work they are doing.

Additionally, at the end of the semester, students will participate in a round-table discussion with the instructors and teaching assistants to give us their feelings on what went right and what went wrong in the class. We will also get their feedback on their views on test driven development and whether or not the use of GUI-based programming assignments helped or hindered the process. This will be repeated over several semesters, since we have no prior coursework that emphasized GUI testing to compare against our results from this semester.

6. FUTURE WORK AND CONCLUSIONS

There are additional tasks that would be desirable to put this system together. The immediate desire is to get the current version of the extensions and tools into a more refined state and then work on improving the usability of the objectdraw extensions and the Abbot wrapper. Furthermore, with the information gathered from our evaluation process, the assignments and tools will be revised to improve the quality of the work being done. Subsequently, testing of these techniques in additional CS1 courses will give us more data and show how the system has improved over time.

A long-term goal is to provide students with the tools to test objectdraw programming assignments with test scripts that are even easier to understand than the current ones, allowing students to create tests with as little knowledge of programming as possible, thereby reducing further the cognitive load on students and improving their results on assignments.

With software bugs and defects being the major roadblocks to software development and with the loss of productivity as a result [12, 20], making our students better testers before they enter their exacting field of industry, it's necessary to guarantee that they are in a better mindset about the benefits of thoroughly testing their code and introducing them to test driven development with graphical user interfaces is a way of combining good testing techniques with a common metaphor for discussing the most commonly-taught paradigm of the time, object-oriented programming.

7. ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation under Grant No. DUE-0618663. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] *Abbot Java GUI Test Framework Home Page*, Available at: http://abbot.sourceforge.net/

[2] *BlueJ-The Interactive Java Environment*, Available at: http://www.bluej.org/

[3] *JUnit Website*, Available at: http://www.junit.org/index.htm

[4] *Objectdraw API*, Available at: http://eventfuljava.cs.williams.edu/library/objectdrawJavadocV1.1.2/index.html

[5] *Swing (Java Foundation Classes)*, Available at: http://java.sun.com/javase/6/docs/technotes/guides/swing/index.html

[6] Allevato, T. and M. Thornton, *Web-CAT Reporting Engine*, 2006, Available at: http://web-cat.cs.vt.edu/CsEdWiki/WebCatReportingEngine

[7] Allowatt, A. and S. H. Edwards, *Designing an Adaptive Learning Module to Teach Software Testing*, 37th Technical Symposium on Computer Science Education, ACM Press, 2005, pp. 259-263.

[8] Beck, K., *Aim, Fire (Test-First Coding)*, IEEE Software, 18(5) (2001), pp. 87-89.

[9] Beck, K., *Test-Driven Development: By Example*, Addison-Wesley, Boston, MA, 2003.

[10] Bruce, K., A. Danyluk and T. Murtagh, *Java: An Eventful Approach*, Prentice-Hall, Upper Saddle River, NJ, 2005.

[11] Bruce, K., A. Danyluk and T. Murtagh, *A Library to Support a Graphics-Based Object-First Approach to CS1*, 32nd SIGCSE Technical Symposium on Computer Science Education, ACM, 2001, pp. 6-10.

[12] Cusumano, M. A., *Technology Strategy and Management: Who is Liable for Bugs and Security Flaws in Software*, Communications of the ACM, 47 (2004), pp. 25-27.

[13] Dutta, S., *Abbot--A Friendly JUnit Extension for GUI Testing*, Java Developer Journal, April 2003, pp. 8-12.

[14] Edwards, S. H., *Improving Student Performance by Evaluating How Well Students Test Their Own Programs*, Journal of Educational Resources in Computing, 3 (2003), pp. 1-24.

[15] Edwards, S. H., *Rethinking Computer Science Education from a Test-First Perspective*, Addendum to the 2003 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, 2003, pp. 148-155.

[16] Edwards, S. H., *Using Software testing to Move Students from Trial-and-Error to Reflection-in-Action*, 35th SIGCSE Technical Symposium on Computer Science Education, ACM, 2004, pp. 26-30.

[17] Goldwasser, M. H., *A Gimmick to Integrate Software Testing Throughout the Curriculum*, 33rd SIGCSE Technical Symposium on Computer Science Education, ACM, 2002, pp. 271-275.

[18] Jones, C. G., *Test-driven Development Goes to School*, Journal of Computing in Small Colleges, 20 (2004), pp. 220-231.

[19] Jones, E. L., *Software Testing in Computer Science Curriculum--A Holistic Approach*, Proceedings of the Autralasian Computing Education Conference, ACM Press, 2000, pp. 153-157.

[20] NIST, *The Economic Impacts of Inadequate Infrastructure for Software Testing--Planning Report 02-03*, 2002.

[21] Patterson, A., M. Kölling and J. Rosenberg, *Introducing Unit Testing With BlueJ*, Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education, ACM Press, Thessaloniki, Greece, 2003, pp. 11-15.

[22] Wick, M., D. Stevenson and P. Wagner, *Using Testing and JUnit Across the Curriculum*, 36th SIGCSE Technical Symposium on Computer Science Education, ACM, 2005, pp. 236-240.