Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable

Sebastian Elbaum, Suzette Person, Jon Dokulil, Matt Jorde
Computer Science and Engineering Department,
University of Nebraska-Lincoln,
Lincoln, Nebraska, USA,
{elbaum,sperson,jdokulil,majorde}@cse.unl.edu

Abstract

Software testing efforts account for a large part of software development costs. However, as educators, we struggle to properly prepare students to perform software testing activities. This struggle is caused by multiple factors: 1) it is challenging to effectively incorporate software testing into an already over-packed curriculum, 2) ad-hoc efforts to teach testing generally happen too late in the students' career, after bad habits have already been developed, and 3) these efforts lack the necessary institutional consistency and support to be effective. To address these challenges we created ***Bug Hunt****, a web-based tutorial to engage students in learning software testing strategies. In this paper we describe the most interesting aspects of the tutorial including the lessons and feedback mechanisms, and the facilities for instructors to configure the tutorial and obtain automatic student assessment. We also present the lessons learned after two years of deployment.*

Categories and Subject Descriptors: D.2.5: **Software** Software Engineering, Testing and Debugging; K.3.2: **Computer Education** Computer and Information Science Education.

General Terms: Verification.

Keywords: Software Testing Education, Web-based Tutorial.

1 Introduction

> "... we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing." Bill Gates [7]

Early integration of software engineering principles and techniques into the undergraduate CS curriculum creates several benefits for students. First, it helps instill good software development practices as soon as the students begin to tackle their first programming assignments [10]. Second, it makes the students' software development experiences more realistic [1, 18]. Third, it reduces the tendency to develop hard-to-break, poor software development habits [1, 10].

Software testing principles and techniques have been identified as one of the areas that should be integrated early in the curriculum, for example, in the CS1/CS2 sequence [4, 9, 12, 17, 19]. Testing is relevant because it is likely to be an important part of the professional life of most graduates (a trend that seems likely to continue and accelerate in the future [16]). Furthermore, testing is a vehicle for demonstrating the importance of other software artifacts that students may find difficult to appreciate within a "classroom" scale development environment (e.g., the value of strong (or weak) requirements documents can be easily appreciated as the basis for the generation of black box tests).

Incorporating testing earlier into the curriculum, however, has proven to be challenging. Several educators leading this integration process [8, 12, 13, 14, 19] have identified several general challenges: 1) lack of properly trained instructors, 2) lack of physical resources such as lab space, 3) diversity of students' goals and skills levels, and 4) large amount of material already present in the CS1/CS2 course sequence.

Several potential solutions have been proposed to address these challenges. For example, Patterson et al. suggested integrating testing tools into programming environments. A prototype of this approach joins the JUnit framework with BlueJ, a Java IDE to create an easy-to-use interface that supports structured unit testing [15]. Jones suggests several testing-related activities that can be incorporated into early CS courses, and has explored the integration of testing into the core introductory CS courses through a structured testing lab and different forms of courseware [13]. Edwards proposes that, from the very first program-

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

ming activities in CS1, students submit test cases with their code for the purpose of demonstrating program correctness [5, 6]. Goldwasser suggests requiring students to submit a test set with each programming assignment. The test sets are then used to test all the programs submitted and students are graded on how well their programs perform on other students' test sets as well as how their test set performs in uncovering flaws in others' programs [8]. Marrero et al. push this idea further by providing extra credits for students submitting test suites that find faults in other students' implementations [14]. More recently, we are witnessing emerging efforts to develop specific learning modules and methodologies targeting testing [2, 11].

Although supportive of the idea of including testing concepts in the CS1/CS2 sequence, our Department has struggled for several years to make this effort effective and sustainable. In addition to the previously identified challenges, we have also observed problems with the students' level of interest and engagement regarding their testing assignments and labs. This problem has also been perceived by others [3, 14, 15], but not addressed in the available solutions. Faculty support for including testing topics in the CS1/CS2 sequence has also been inconsistent. We have found that while some instructors were hesitant to reuse courseware because of the effort required to adapt the materials, others wanted fully packaged courseware that would minimize the amount of time required for preparation and grading.

In an effort to promote early integration of software testing concepts into the CS curriculum in a manner that engages students, while making it amenable for wide-spread adoption among instructors, we have developed a hands-on, web-based tutorial named *Bug Hunt*. *Bug Hunt* has several features that make it attractive to both instructors and students:

- It incorporates challenges in each lesson and provides immediate feedback to promote engagement while students practice the application of fundamental testing techniques.
- It is self-paced so students can spend as much time as they feel necessary to complete the material.
- It provides an "out of the box" solution, and it is also configurable to accommodate the instructors's requirements.
- It provides a complete and automatic assessment of students' performance to reduce the instructor's load.

In the next section, we discuss the organizational structure of *Bug Hunt*. Section 3 discusses *Bug Hunt's* features including lessons and challenges, feedback mechanisms, degree of configurability, and automated student assessment. In Section 4, we summarize the lessons learned after two years of deployment.

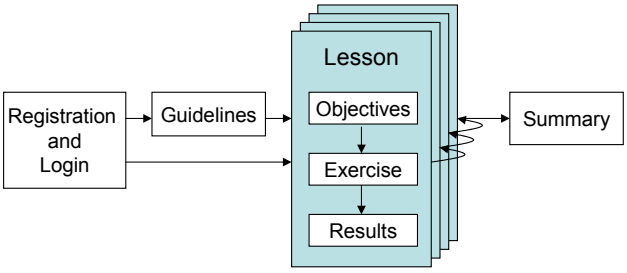


Figure 1. Bug Hunt Lesson-based Structure.

2 *Bug Hunt* Organizational Structure

Bug Hunt is a web application built with Java 2 Enterprise Edition (J2EE) technologies using a standard model-view-controller architecture that we now briefly describe. Persistent data such as the exercises and the students' performance are stored in a relational database. The view consists of a series of JSP pages with access restricted based on the user's account type (student or instructor) that generate the appropriate user page. The execution logic is divided into modules corresponding to low-level tasks such as database access and test case execution. The controller uses these modules to access the model, perform necessary actions, and update the user view.

Users interact with *Bug Hunt* through a web browser. For example, a student utilizes a web browser for registering, logging into, and completing the lessons in *Bug Hunt*. Students logging into *Bug Hunt* for the first time, are provided with a set of general guidelines including the objectives of the tutorial and a description of how to use the tutorial, while a student who has already begun the lessons is taken to the same lesson number and test suite contents at which she was working when the tutorial was exited. Once logged in, a student progresses through a series of lessons, using a specific testing strategy in each lesson to find program faults. As a student works through each lesson, progress is measured by the number of faults detected by the student's test suite and by how the student performs relative to the other participating classmates. To provide a uniform content presentation throughout *Bug Hunt*, each lesson includes a set of objectives, an exercise, and the results for the exercise. Once the tutorial is completed, the student receives an overall performance summary. Figure 1 summarizes the high-level structure of *Bug Hunt* from the student's perspective.

Bug Hunt's lesson-based structure is meant to incrementally build the student's understanding of software testing practices. The first lesson introduces basic software testing concepts and terminology. Subsequent lessons build on these concepts, giving students hands-on practice with black-box and white-box testing techniques, while encour-

aging exploration of the input space and systematic test case development. The final lesson ties everything together by introducing students to automated testing. Test cases created in each lesson carry forward to subsequent lessons, emphasizing the complementary nature of the various testing strategies. Test cases are also maintained across tutorial sessions allowing students to exit and re-enter the *Bug Hunt* tutorial without losing their work.

Each *Bug Hunt* lesson is comprised of the following components:

Objectives. This page provides a brief description of the testing concepts a student should be familiar with at the end of the lesson and identifies the testing strategy the student will practice during the lesson.

Exercise. An exercise consists of several sub-components.

Instructions. A brief set of instructions describes the lesson's challenge and testing activity. This information is presented at the top of the exercise for the student to consult while applying the testing strategy.

Artifacts. Each lesson draws from a set of artifacts to provide the information appropriate for the lesson. The artifacts are listed on the main lesson page for easy reference while the student works through the testing activity. In Figure 2 for example, both the requirements and the source code listing are provided to help the student complete Lesson 1.

Distinctive Feedback Each lesson has a unique display widget tailored to the particular type of testing being practiced by the student. For example, in Figure 4, where the student is practicing white-box testing, the display widget indicates which parts of the code are covered by the test suite, and the degree to which the code is covered (i.e., how many test cases cover a given section of code). These widgets provide visual feedback that is updated each time a student submits a test case for execution.

Tests. During each lesson, students devise and submit test cases for execution, one at a time, through the *Test Case Input* area. Each test case consists of a set of program inputs and the associated expected output value. The *Bug Hunt* application server executes each test case as it is submitted and provides immediate feedback through various devices such as the *bug jar*, the *Test Execution Log*, the *Display Widget*, and several location-sensitive tool-tips (more details on the feedback mechanisms are presented in Section 3.2).

Assistance. Throughout the tutorial, students have access to a set of lesson "hints" presented in the form of "Frequently Asked Questions." This information covers topics such as "Where do I start?" and "How do I...?"

Results. This page summarizes the student's performance in terms of fault detection and provides additional information about the faults found. The Test Execution Log is also displayed for the student to review before proceeding to the next lesson. Figure 6 shows the results for a student who has completed Lesson 3.

3 Features

This section describes the main features of *Bug Hunt* that are intended to engage students while making it amenable for widespread adoption among CS instructors.

3.1 Lessons and Challenges

Each lesson contains a specific set of objectives, a unique challenge, and a distinctive feedback widget to encourage the exploration of different fault findings strategies. Table 1 summarizes the challenges and feedback widgets by lesson, and the following paragraphs provide more details about each.

The first lesson of the *Bug Hunt* tutorial, Lesson 1, helps familiarize students with basic software test suite terminology (e.g, test case, test case execution, expected outcomes, passing and failing test cases) and test activity organization (e.g., test suite, test execution log). The Lesson 1 challenge is to find a particular fault using the clues provided (e.g., highlighted requirement) as they create their test cases. Figure 2 provides a screenshot of Lesson 1.

Lesson 2 introduces the concept of black-box testing where students have access only to the program requirements, but not to the source code. Beyond finding new bugs, the challenge in this *Bug Hunt* lesson is to develop tests that expose the potential program outputs. To achieve this exposure, students must explore the relationships between inputs and program outputs, thereby discovering various input categories in the process. Figure 3 shows a screenshot of the main lesson page for Lesson 2 (note the distinctive feedback utilized to measure output exposure).

In Lesson 3, students learn about white-box testing. They no longer have access to the program requirements, but instead use the program code as the primary source of information to find faults. *Bug Hunt* generates an annotated version of the source code after every test case execution to indicate the number of times a particular line of code is executed by the student's test suite. The Lesson 3 challenge is to build a test suite that achieves complete statement coverage. Figure 4 shows a screenshot of the exercise page for Lesson 3, and Figure 6 presents a Lesson Summary page for the this lesson. (the same summary format is presented at the end of each lesson).

The final *Bug Hunt* lesson, Lesson 4, builds on the three previous lessons by introducing the concepts of automation

Figure 2. Lesson 1: Concepts and Terminology.

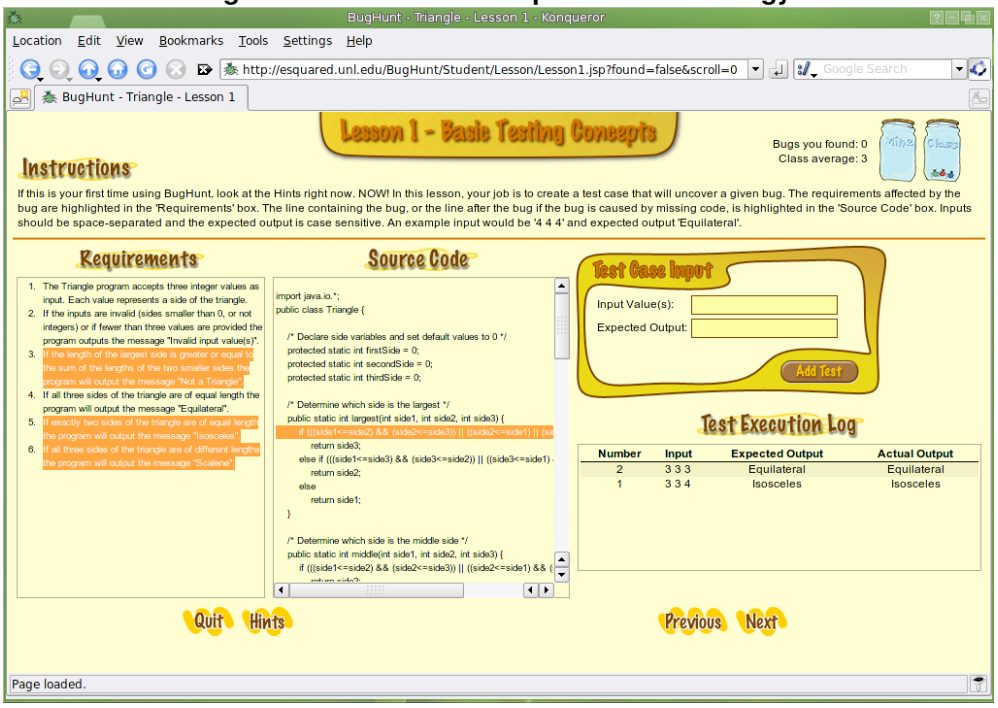
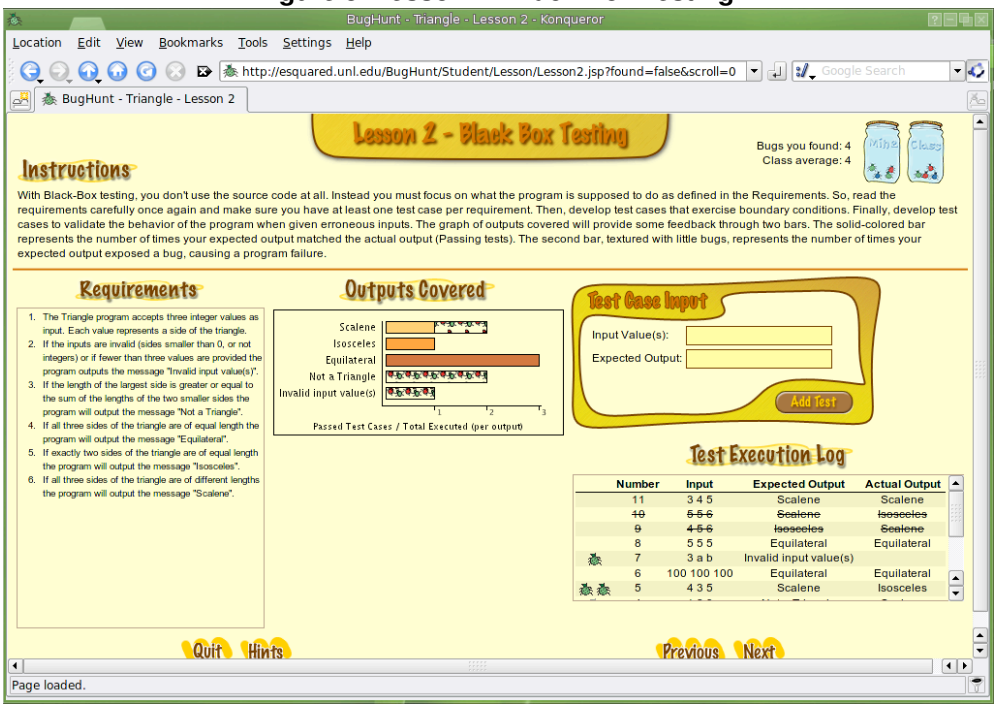


Figure 3. Lesson 2: Black Box Testing.



29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

Figure 4. Lesson 3: White Box Testing.

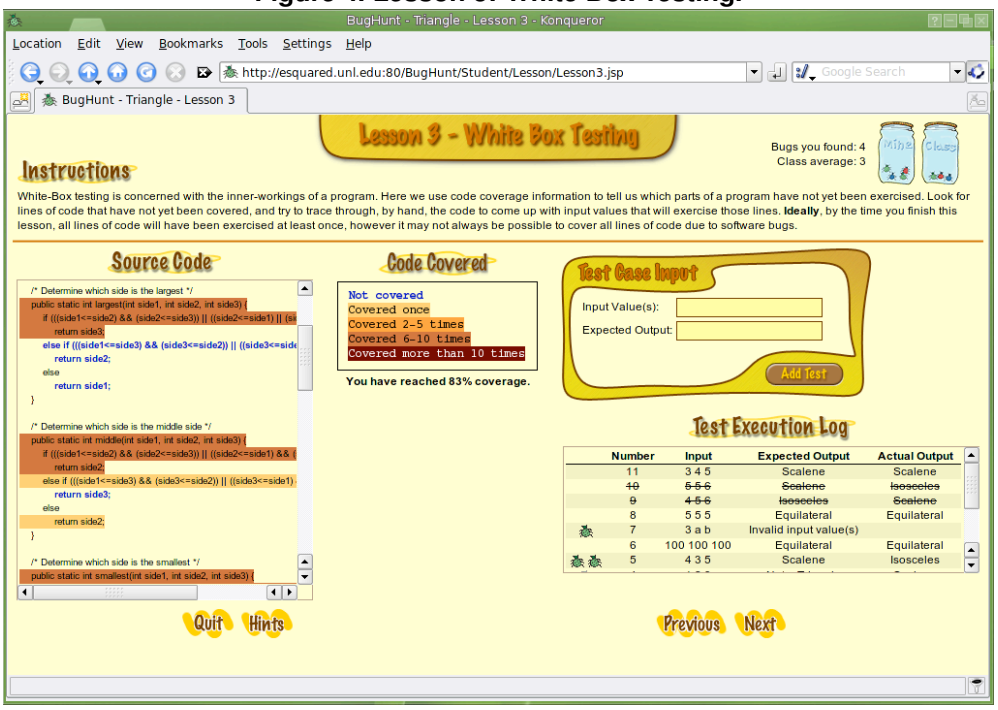


Figure 5. Lesson 4: Testing Automation and Efficiency.

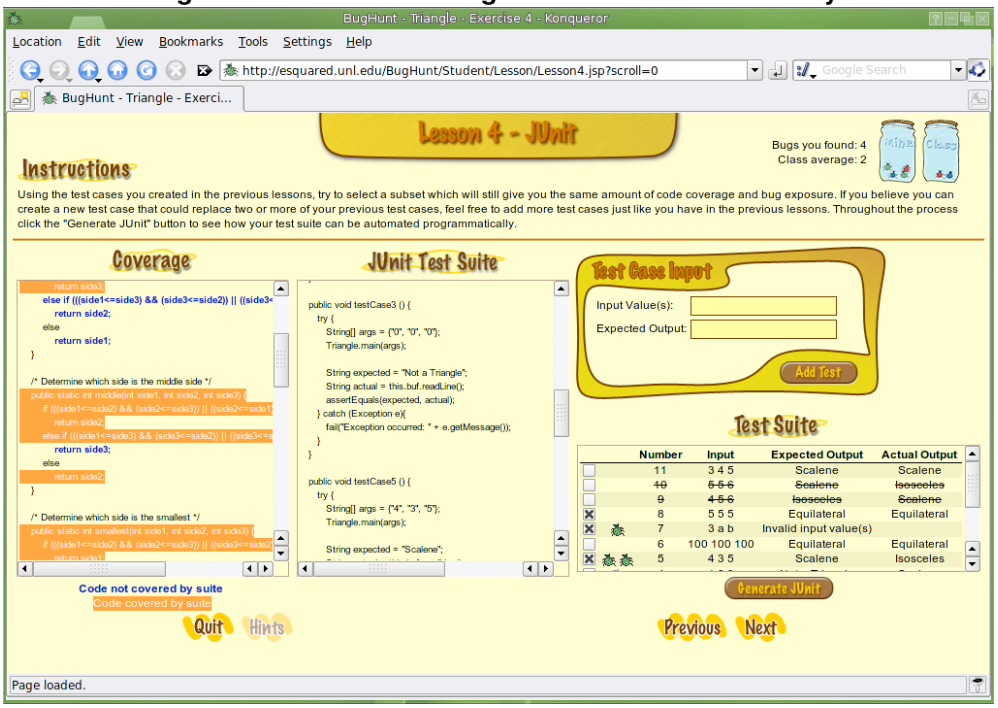


Figure 6. Results for a lesson.

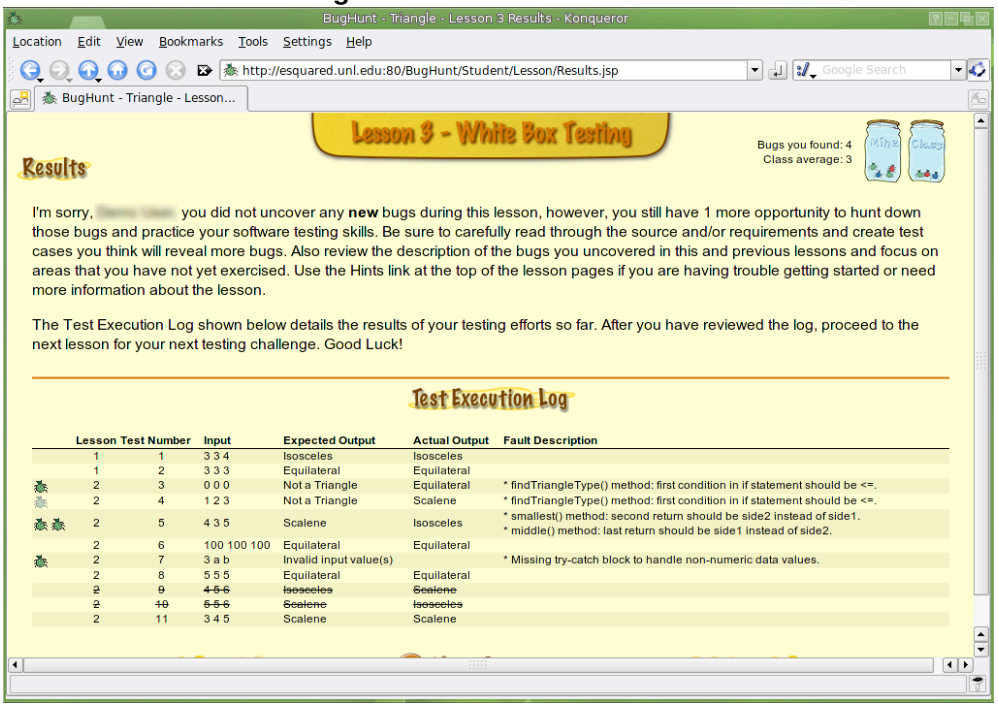
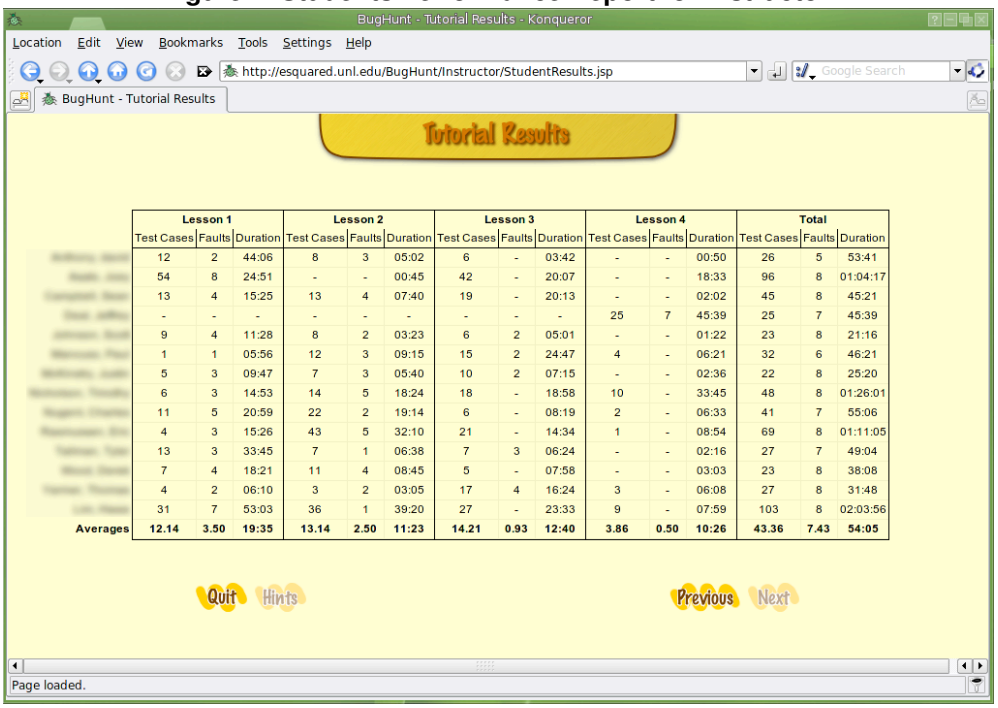


Figure 7. Students Performance Report for Instructor.



Lesson	Challenge	Distinctive Feedback Widget
1	Find the specific fault described in the lesson instructions	Clued requirements and source code
2	Exercise all classes of program outputs	Bar graph of fault distribution according to exercised output
3	Cover all executable statements in the code	Annotated and colored code structure representing coverage
4	Select enough tests to maintain fault detection effectiveness	Test case selection device and Junit test suite

Table 1. Lessons, challenges, and mechanisms.

and efficiency in testing. Students are challenged to find as many program faults as possible and achieve as much statement coverage as possible by using the fewest number of test cases from their test suite. The level of code coverage is indicated using the same annotated source code display used in Lesson 3. For each test case selected, a JUnit test suite is also generated and the complete Java class containing all of the test cases is displayed on the main lesson page. Figure 5 shows a screenshot of Lesson 4.

3.2 Feedback Mechanisms

Providing students with meaningful and timely feedback is advocated as a good teaching practice to improve learning. *Bug Hunt* provides feedback to help students realize what they know, what they do not know, and how to improve their performance. In order to be more effective, *Bug Hunt* interleaves the feedback with the learning activities through several mechanisms.

Test Log. When a well specified test is provided in the text case input box, it is sent to the *Bug Hunt* server, where it is executed, and then posted in the student's Test Execution Log. When a test is not well specified (e.g., the expected value was set incorrectly), the test is returned by the server and marked as invalid in the Test Execution Log (see test #9 in Figure 4). This process assures the student that the test exercised the program as expected.

Widget. As tests are executing, the distinctive lesson feedback widget is updated. For example, in Lesson 3, code annotations and colors are modified as tests cover previously unexecuted code, and in Lesson 4 the generated Junit suite is updated when the selected tests change or a new test case is incorporated into the suite.

Faults. When a test case exposes a fault, the test log indicates that the test made the program fail by displaying a "bug" icon next to the test case. Newly exposed faults are distinguished from the ones already found by other tests through their hue. Students can obtain further instructive feedback on the fault found by hovering with the mouse over the "bug" icon to see a description of it and its associated code fragment. Each unique "bug" is inserted into the "bug jar" that appears at the top-right corner of each lesson. The display also contains a "bug jar" with the faults exposed by the class to serve as a benchmark.

Summaries. After each lesson, individual student results are presented in the form of a "Lesson Summary" (see Figure 6), which includes a Test Execution Log listing the cumulative test execution results and a brief personalized paragraph summarizing the student's testing efforts and encouraging him or her to continue the quest for finding "bugs." After a student has completed all of the lessons in *Bug Hunt*, a "Tutorial Summary" page is presented. The information contained in the Tutorial Summary includes the details of each test in the student's test suite (e.g., the input values, the expected output value, the actual output value, and the fault or faults exposed by the test when appropriate), the total number of faults discovered by the student's test suite, a description for each fault that was revealed, and the percentage of the program faults that were found.

Follow-up. Once the student has completed the tutorial, we encourage further practice and experimentation by emailing a package to the student containing the program source code, the JUnit test cases, and a JUnit test driver class. The program source code includes comments with detailed instructions on how to build and execute the program, and how to utilize the JUnit test suite.

3.3 Degree of Configurability

To set-up a new course in *Bug Hunt*, the instructor contacts the *Bug Hunt* administrator with the course name and the instructor's contact information. Once the course setup is complete, the instructor will have access to the roster management component where students' first and last names and their email addresses must be entered. Next, the instructor selects one of the pre-configured exercises through the course management menu (see Figure 8). Then, *Bug Hunt* emails the URL of the account activation page to the students enrolled in the roster. The account activation process requires each student to enter his or her last name. The student's email address is used to verify his or her identity. After choosing a login id and password the student is ready to begin the tutorial.

In *Bug Hunt* most of the lesson content and subcomponents are stored in a database. For example, all of the program requirements, source code, faults and tests, and the lesson instructions and objectives are part of the *Bug Hunt* database scheme. This data-oriented architecture allow us to support instructors interested in adding their own exercises to expose students to new challenges, to include

specific types of faults or more formal requirements, and to incorporate particular programming constructs that are more difficult to validate (e.g., polymorphic classes, concurrency). We are, however, still in the process of developing an interface for instructors to perform such activities directly without the assistance of a *Bug Hunt* administrator.

3.4 Automated Student Assessment

One of the key features for instructors utilizing *Bug Hunt* is the automatic student assessment feature which can generate a report such as the one presented in Figure 7. As a student creates and executes each test case, *Bug Hunt* tracks and records the results of the test case execution. Information collected includes the test case input values, expected output values, actual output values, the lesson number in which the student created the test case, and the results of the test case execution. When a test case fails, a reference to the fault discovered by the failing test case is also saved. Since students in a given course are linked together via a class identifier, individual student performance can also be measured relative to course performance. This feature eliminates the need for the instructor to grade, manage and report students' results.

4 Deployment and Lessons Learned

Since its deployment two years ago, over 400 students at several institutions have used *Bug Hunt*. For the students within our Department, we have included an anonymous questionnaire at the end of the tutorial to help us perform a preliminary assessment of *Bug Hunt*. The questionnaire includes 12 quantifiable questions measured in a likert scale, two open-ended questions, and a section for additional comments.

Some of the most interesting findings of the 204 responses collected follow:

- 77% of the students "agreed" or "strongly agreed" that *Bug Hunt* added significant value to the material presented in the lecture(s) on software testing (15% were neutral and 8% did not feel the tutorial added value).
- 56% of the students "agreed" or "strongly agreed" that *Bug Hunt* could totally replace the lectures on testing (21% were neutral, but 23% felt the lectures' material is still necessary). This seems to indicate that, for an important number of students, the tutorial is not enough on its own.
- 66% of the students "agreed" or "strongly agreed" that *Bug Hunt* taught them concepts that will be useful for their future assignments, and 60% felt the same way about the potential usefulness of this material for their future job. This may indicate that the students may not be fully aware of the need and impact of testing activities in real software development organization, and perhaps *Bug Hunt* may need to address this early on.
- 32% of the students found the white box testing lesson to be the most interesting and valuable. It was also surprising to see that 5% of the students thought that the JUnit and automated lesson was valuable but 27% thought was the most interesting. Clearly, the challenge and distinctive feedback utilized in each lesson may play a big role in these values. However, Lesson 3 is where the students found the greatest number of faults, raising an interesting general conjecture about whether the right challenge and widget may improve a tester's performance.

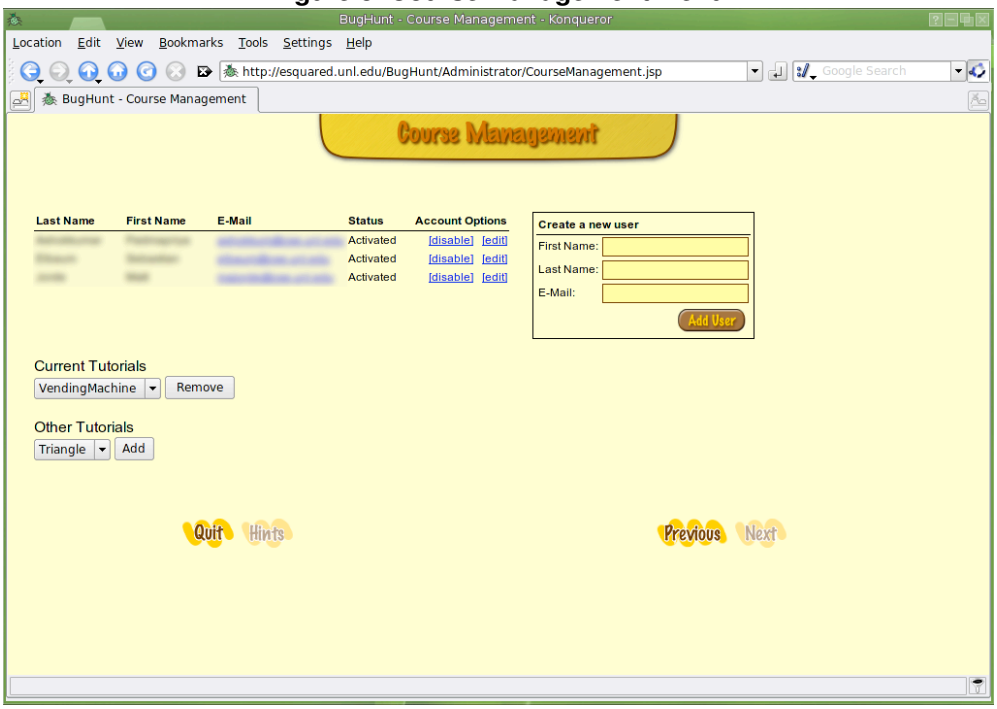
The section for additional comments was completed by over one hundred students. Many of these comments were of the type "I liked" and "I disliked" and were specific to certain lessons. For example, over 20% of the students commented on the value of the white-box testing lesson in making them aware of the association between tests and code, independently of whether they discovered a fault or not. Also, many students provided various suggestions to expedite the process of providing inputs such as "have a drop-down box with the inputs I [have already] used."

The comments also provided some more general insights. For example, a third of the students emphasized the value of feedback, quick interaction, and an engaging interface with comments such as "I liked that the tutorial was interactive and you got results of your tests immediately.", "The tutorial was fun... it broke the monotony...", "I liked how it gave real experience instead of just theory" , or "Seeing the bugs appear in the jar made me feel I was accomplishing something".

Another common theme in the responses was the realization that different strategies provide distinct strengths. This was evident in comments such as "I liked black box because I had never before thought of looking for bugs without using source code", "I like to find bugs in different ways. I liked learning the different approaches", or "It showed me that there are a lot of different methods to debug a program and that some are more useful than others."

Clearly, this assessment process so far has been focused on pinpointing areas of the tutorial that are confusing, on identifying lessons that could be enhanced or made more attractive, and on determining the value of the lessons from a student's perspective. Still, we do not know whether the learning experience as a whole is effective when compared with alternative practices. The next assessment step is to perform a controlled experiment aimed at quantitatively determining the level of student understanding achieved through the tutorial.

Figure 8. Course Management Menu



In addition to the student assessments, discussions with the instructors have raised the need for further enhancements and even the potential for optional additional lessons to target more advanced courses. More importantly, instructors are expected to be the main contributors of exercises to enrich *Bug Hunt*[1]. Although the mechanism to share the exercises among instructors is already in place, we currently lack a way for the instructors to upload such exercises, so this will be a high-priority for the next version of *Bug Hunt* in order to enable community participation.

Acknowledgments

This work was supported in part by the Career Award 0347518 to the University of Nebraska-Lincoln, the Great Plains Software Technology Initiative, and the UCARE Project at the University of Nebraska-Lincoln. We would like to thank Nick Steinbaugh and Yu Lin for their help in implementing this tutorial, and Alex Baker and Andre Van Der Hoek from UC-Irvine for their feedback on the tutorial. Finally, we would also like to thank all of the students and instructors who participated in the preliminary assessment of *Bug Hunt*.

[1]Instructors are encouraged to explore and use *Bug Hunt* at: http://esquared.unl.edu/BugHunt "Try a demo" link

References

[1] Software Engineering 2004. Curriculum guidelines for undergraduate degree programs in software engineering. http://sites.computer.org/ccse, 2004.

[2] R. Agarwal, S.H. Edwards, and M.A. Perez-Quinones. Designing and adaptive learning module to teach software testing. In *Symposium on Computer Science Education*, pages 259–263, March 2006.

[3] E. Barriocanal, M. Urban, I. Cuevas, and P. Perez. An experience in integrating automated unit testing practices in an introductory programming course. *Inroads SIGCSE Bulletin*, 34(4):125–128, December 2002.

[4] CC2001-Task-Force. Computing curricula 2001. Final report, IEEE Computer Society and Association for Computing Machinery, 2001.

[5] S. Edwards. Rethinking computer science education from a test-first perspective. In *Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 148–155, October 2003.

[6] S. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Symposium on Computer Science Education*, pages 26–30, March 2004.

[7] Bill Gates. Q&A: Bill Gates on trustworthy computing. Information Week, May 2002.

[8] M. Goldwasser. A gimmick to integrate software testing thoughout the curriculum. In *Frontiers in Computer Science Education*, pages 271–175, March 2002.

[9] T. Hilburn and M. Townhidnejad. Software quality: a curriculum postscript? In *Symposium on Computer Science Education*, pages 167–171, 2000.

[10] U. Jackson, B. Manaris, and R. McCauley. Strategies for effective integration of software engineering concepts and techniques into the undergraduate computer science curriculum. In *Symposium on Computer Science Education*, pages 360–364, March 1997.

[11] D.S. Janzen and H. Saiedian. Test-driven learning: Intrinsic integration of testing into the cs/se curriculum. In *Symposium on Computer Science Education*, pages 254–258, March 2006.

[12] E. Jones. An experiential approach to incorporating software testing into the computer science curriculum. In *ASEE/IEEE Frontiers in Education Conference*, page F3D, October 2001.

[13] E. Jones. Integrating testing into the curriculum - arsenic in small doses. In *Symposium on Computer Science Education*, pages 337–341, February 2001.

[14] W. Marrero and A. Settle. Testing first: Emphasizing testing in early programming courses. In *Conference on Innovation and Technology in Computer Science and Education*, pages 4–8, 2005.

[15] A. Patterson, M. Kölling, and J. Rosenberg. Introducing unit testing with BlueJ. In *Conference on Innovation and Technology in Computer Science and Education*, pages 11–15, 2003.

[16] Program-Planning-Group. The economic impacts inadequate testing infrastructure. Planning Report 02-3, National Institute of Standards and Technology, May 2002.

[17] J. Roberge and C. Suriano. Using laboratories to teach software engineering principles in the introductory computer science curriculum. In *Symposium on Computer Science Education*, pages 106–110, February 1994.

[18] M. Shaw. Software engineering education: a roadmap. In *International Software Engineering Conference - Future of Software Engineering*, pages 371–380, May 2000.

[19] T. Shepard, M. Lamb, and D. Kelly. More testing should be taught. *Communications of the ACM*, 44(6):103–108, June 2001.