

Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable

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

Abstract

*Software testing efforts account for a large part of the software development costs. We still struggle, however, 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 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 the learning of software testing strategies. In this paper we describe the most interesting aspects of the tutorial including the lessons designed to engage students, and the facilities for instructors to configure the tutorial and obtain automatic student assessment. We also present the lessons learned after the first year 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 [6]

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 [9]. Second, it makes the students' software development experiences more realistic [1, 16]. Third, it reduces the tendency to develop hard-to-break, poor software development habits [1, 9].

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 [3, 8, 10, 15, 17]. 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 [14]) and because it is a vehicle to demonstrate the role of other software artifacts which may be difficult for students to appreciate is isolation during small scale development (e.g., requirements documents are valuable to derive test cases and to check the program's behavior).

Incorporating testing earlier into the curriculum, however, has proven to be challenging. Several educators leading this integration process [7, 10, 11, 17] have identified three general challenges: 1) lack of properly trained instructors, 2) lack of physical resources such as lab space, and 3) the large amount of material already present in the CS1/CS2 curriculum.

Several potential solutions have been proposed to address these challenges. For example, Patterson et al. suggested to integrate testing tools into programming environments. A prototype of this approach joins the JUnit framework with BlueJ, a Java IDE to create a easy-to-use interface that supports structured unit testing [13]. Jones has explored the integration of testing into the core introductory CS courses through a structured testing lab and different forms of courseware [11]. Edwards proposes that, from the very first programming activities in CS1, students submit test cases with their code for the purpose of demonstrating program correctness [4, 5]. Similarly, Goldwasser suggests requiring students to submit a test set with each programming assignment. The test sets are then used to test all of the programs submitted and students are graded on both how well their programs perform on other students' test sets as well as how their test set performs in uncovering flaws in others' programs [7].

Although supportive of the idea of including testing concepts in the CS1/CS2 sequence through some of the enumerated solutions, 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 (a problem also perceived by others [2, 13]) but not addressed in the available solutions. Faculty support for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Submitted to the Educational Track of ICSE 2006, Shanghai, China
Copyright 2006 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

including testing topics in the CS1/CS2 sequence has also been inconsistent. We 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 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 basic 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. Section 3.4 reviews *Bug Hunt*’s automatic assessment feature. Last, in Section 4, we summarize the lessons learned in our first year of deployment and sketch a plan to further evaluate the effectiveness of the tutorial.

2. *Bug Hunt* Organizational Structure

Figure 1 presents the tutorial high-level structure. A student begins by registering in a tutorial course, and login. A student logged-in for the first time is provided with a set of tutorial guidelines, while a student that has already performed part of the tutorial is taken to the same lesson number and test suite contents at which she was working when exited the tutorial.

Once logged, a student can progress through a series of lessons, applying various testing strategies to find program faults. As students work through each lesson, individual 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 the tutorial, all lessons include a set of objectives, an exercise, and the results for the exercise. Once the tutorial is completed, the student receives an overall performance summary.

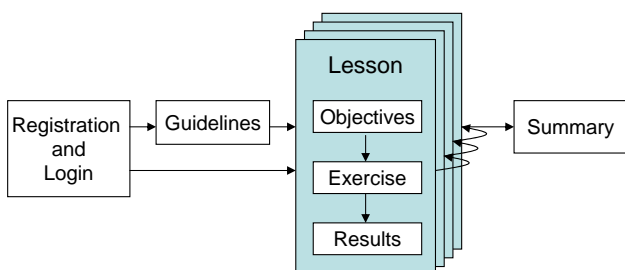


Figure 1. *Bug Hunt* Lesson-based Structure.

The *Bug Hunt* tutorial attempts 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 encouraging exploration of the input space and systematic test case development. The final lesson ties everything together by introducing students to automated testing. (We discuss the capabilities to tailor the set of lessons in Section 3.3). Test cases created in each lesson carry forward to subsequent lessons, emphasizing the complementary nature of the presented testing strategies. Test cases are also maintained across tutorial sessions allowing students to exit and re-enter the *Bug Hunt* tutorial without restrictions.

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

Instructions. A brief set of instructions describes the lesson’s challenge and testing activity. These instructions are listed on the main lesson page for easy reference while the student works through the testing activity.

Artifacts. Each lesson draws from a set of artifacts (e.g., requirements document, source code listing) to provide the information appropriate for the lesson. This information is presented at the top of the exercise for the student to consult while applying the testing strategy.

Widget Each lesson has a unique display widget that provides visual feedback to the student after each test case execution in relation to the lesson’s challenge.

Tests. During each lesson, students devise and submit test cases for execution 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* system 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...?”

3. Features

This section describes the features of *Bug Hunt* aiming to engage students while making it amenable for wide adoption among CS instructors.

3.1 Lessons and Challenges

Each lesson constitutes a learning-object, with a particular set of objectives, a unique challenge, and a distinct widget to encourage the exploration of different fault findings strategies. Table 1 summarizes the challenge and widget of each lesson, and the following paragraphs provide more details about each one of them.

The first lesson of the *Bug Hunt* tutorial, Lesson 1, helps students to familiarize with basic software test suite terminology (e.g.,

Lesson	Challenge	Specialized Widget
1	Find as many faults as possible in allocated time	Clock waning down
2	Exercise all program of 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 just enough tests to maintain fault detection effectiveness	Test case selection device and Junit test suite

Table 1. Lessons, challenges, and mechanisms.

test case, test case execution, expected outcomes, passing and failing test cases) and test activity organization (e.g., test suite, test execution log). During this lesson, students may reference the system requirements and the source code listing as they create their test cases. The Lesson 1 challenge is a race against the clock where each student selects the time frame (e.g., 1 minute, 2 minutes, 3 minutes...5 minutes) in which he or she attempts to find as many faults in the program as possible. When time is up, the student is prevented from entering additional test cases. Figure 2(a) 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. In order to achieve such exposure, students must explore what is the relationships between inputs and the program outcome, discovering various input categories in the process. Students are free to spend as much time as they need to complete this lesson. Figure 2(b) shows a screenshot of the main lesson page for Lesson 2 (note the widget 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. In *Bug Hunt*, an annotated version of the source code indicates code coverage including 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 will achieve complete statement coverage. Like Lesson 2, this lesson can be completed at the student's own pace. Figure 2(c) shows a screenshot of the exercise page for Lesson 3, and Figure 2(e) presents a Lesson Summary page for the this lesson. (the same summary format is presented at the end of each lesson to provide an opportunity for the student to obtain further feedback).

The final *Bug Hunt* lesson, Lesson 4, builds on the three previous lessons by introducing the concepts of automation 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 case is also generated and the complete Java class containing all of the test cases is displayed on the main lesson page. Figure 2(d) 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* continuously interleaves the feedback with the learning activity through several mechanisms.

For example, when a well specified test is provided, it is sent to the *Bug Hunt* server, it is executed, and then it is posted in the Test Execution Log. This feedback mechanism assures the students that the test exercised the program. When a test is not well specified (e.g., expected value was set incorrectly), the test is marked as invalid in the log. As test are executing, the specialized lesson widget is updated. For example, in Lesson 3 the code annotations and colors are modified, and in Lesson 4 the generated Junit suite is updated.

When an executed test exposes a fault, the test log indicates that the test made the program fail by displaying a "bug" next to the test case. Newly exposed "bugs" are distinguished from the ones already found by other tests through their hue. Students can obtain further instructive feedback on the fault found by navigating over the "bug" to see a description of it and its associated code fragment.

In addition to the log, a "bug" is inserted into the "bug jar" (appearing in the top-right of each lesson) which contains all the faults found (the display also contains a "bug jar" with the faults exposed by the class to serve as a benchmark). As previously mentioned, after each lesson, individual student results are presented in the form of a "Lesson Summary" (see Figure 2(e)), 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 the tutorial, a "Tutorial Summary" is generated and presented to the student. 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 test status), the total number of faults discovered by the student's test suite, and a description for each fault that was revealed.

Once the student has completed the tutorial, we encourage further practice and experimentation by emailing the source code for the program the student has been testing, the JUnit test cases, and a JUnit test driver class to the student. Comments included in the Java source code provide the student with detailed instructions on how to build and execute the program and JUnit test suite.

3.3 Degree of Configurability

We have designed *Bug Hunt* to offer multiple levels of configuration. Instructors who prefer an "out of the box" solution can use one of *Bug Hunt's* pre-configured exercises (target exercise program with faults and requirements). An online wizard guides the course set-up process. To set-up a new course in *Bug Hunt*, the instructor performs three configuration steps: 1) enter the course name and the instructor's contact information, 2) enter the course roster (students' first and last names and their email addresses), and 3) select one of the pre-configured exercises. The wizard then emails the URL of the account activation page to the students en-

rolled in step 2. 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.

Instructors interested in adding their own exercises to expose students to new challenges, or to include specific types of faults or more formal requirements, or to include particular programming constructs that are more difficult to validate (e.g., polymorphic classes, concurrency) can take advantage of *Bug Hunt's* tailoring capabilities. New exercises can be added and existing exercise components can be modified (e.g., add a new fault, remove a requirement) through an online interface. Instructors can even modify or add new "Hints" or "Objectives" to the lesson exercises. Last, *Bug Hunt* can support multiple courses and multiple exercises per course. This enables instructors to provide exercises at varying levels of difficulty during a given course, or to offer different exercises to different groups of students.

We also plan to increase the degree of configurability in terms of the lessons. The current infrastructure of self-contained lessons allows to add new lessons to cover a broader spectrum of strategies without impacting the existing ones. Furthermore, future releases will enable instructors to decide what lessons to include and the order in which they should be presented.

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 the one presented in Figure 2(f). 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 approach to immediate and automatic assessment eliminates the need for the instructor to tabulate and report results to each student. Instructors can also use the information collected by *Bug Hunt* to plan course changes and updates based on student performance on each of the lesson topics.

4. Deployment and Lessons Learned

During the Spring of 2004 we performed a beta deployment of the tutorial on our CS2 courses. We used informal feedback from the students and instructors in those courses to refine some functional and usability aspects (e.g., change default settings, refine problem statements, add tool-tips on faults) of the tutorial.

The second deployment phase was performed during the Fall 2004 and Spring 2005, when over 200 students at 6 institutions took the revised tutorial. All the students utilized *Bug Hunt* with an object oriented adaptation of the classic "triangle" exercise often used in testing courses [12].

As part of this deployment, we included a built-in anonymous questionnaire at the end of the tutorial. The questionnaire formalized and helped us to standardized the ad-hoc assessment we performed during the beta-deployment. It included 12 quantifiable

questions measured in a likert scale, two open-ended questions, and a section for additional comments.

The most interesting preliminary findings of the 133 responses we collected in our Department are:

- 79% of the students "agreed" or "strongly agreed" that *Bug Hunt* added significant value to the material presented in the lecture(s) on software testing (16% were neutral and only 5% did not feel the tutorial added any value).
- 61% of the students "agreed" or "strongly agreed" that *Bug Hunt* could totally replace the classes on testing (21% were neutral, but 18% felt the class material is still necessary). This seems to indicate that, for an important number of students, the tutorial is not enough on its own.
- 71% of the students "agreed" or "strongly agreed" that *Bug Hunt* taught them concepts that will be useful in their future assignments, but only 57% 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.
- 47% of the students found the white box testing lesson to be the most interesting and valuable, follow with 35% for the unit-testing and automation lesson. Clearly, the challenge and widget utilized in each lesson play a big role in these values. However, Lesson 3 is where the students found the greater 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 32 students. Many of these comments were of the type "I liked" and "I disliked" and were specific to certain lessons. For example, four students expressed frustration with having a timer in Lesson 1 and felt rushed by it. Six students reiterated on the value of the white-box testing lesson to make them aware of the association between tests and code, independently of whether they discovered a fault or not. Also, four 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, ten students emphasized the value of feedback, quick interaction, and practical usage 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 practical 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, the 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 aim at quantitatively determining the level of student understanding achieved through the tutorial.

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, the instructors are expected to be the main contributors of exercises to enrich *Bug Hunt*. Although the mechanism to share the exercises among instructors is already in place, we currently lack for 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.

4.1 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, Matt Jorde, 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*.

5. References

- [1] Software Engineering 2004. Curriculum guidelines for undergraduate degree programs in software engineering. <http://sites.computer.org/ccse>, 2004.
- [2] E. Barriocanal, M. Urban, I. Cuevas, and P. Perez. An Experience Integrating Automated Unit Testing Practices in an Introductory . *Inroads SIGCSE Bulletin*, 34(4):125–128, December 2002.
- [3] CC2001-Task-Force. Computing curricula 2001. Final report, IEEE Computer Society and Association for Computing Machinery, 2001.
- [4] S. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 148–155, October 2003.
- [5] S. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *SIGCSE Technical Symposium on Computer Science Education*, pages 26–30, March 2004.
- [6] Bill Gates. Q&a: Bill gates on trustworthy computing. Information Week, May 2002.
- [7] M. Goldwasser. A gimmick to integrate software testing throughout the curriculum. In *Frontiers in Computer Science Education*, pages 271–175, March 2002.
- [8] T. Hilburn and M. Townhidnejad. Software quality: a curriculum postscript? In *SIGCSE technical symposium on Computer science education*, pages 167–171, 2000.
- [9] U. Jackson, B. Manaris, and R. McCauley. Strategies for effective integration of software engineering concepts and techniques into the undergraduate computer science curriculum. In *SIGCSE Technical Symposium on Computer Science Education*, pages 360–364, March 1997.
- [10] 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.
- [11] E. Jones. Integrating testing into the curriculum - arsenic in small doses. In *SIGCSE Technical Symposium on Computer Science Education*, pages 337–341, February 2001.
- [12] Glenford Myers. *The Art of Software Testing*. John Wiley & Sons, 1 edition, 1979.
- [13] 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.
- [14] Program-Planning-Group. The economic impacts inadequate testing infrastructure. Planning Report 02-3, National Institute of Standards and Technology, May 2002.
- [15] J. Roberge and C. Suriano. Using laboratories to teach software engineering principles in the introductory computer science curriculum. In *SIGCSE Technical Symposium on Computer Science Education*, pages 106–110, February 1994.
- [16] M. Shaw. Software engineering education: a roadmap. In *International Software Engineering Conference - Future of Software Engineering*, pages 371–380, 2000.
- [17] T. Shepard, M. Lamb, and D. Kelly. More testing should be taught. *Communications of the ACM*, 44(6):103–108, June 2001.