Bug of the Day: Reinforcing the Importance of Testing

Daniel E. Krutz, Michael Lutz
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA

***Abstract*—Software engineering students typically dislike testing. In part this is due to the simplicity of the programming and design exercises in introductory computing courses - the payoff for thorough testing is not apparent. In addition, testing can be seen as tangential to what really matters: developing and documenting a design addressing the requirements, and constructing a system in conforming to the design.**

Such dismissive attitudes do not accord well with the realities of commercial software development. The cost of fixing, repairing and redistributing a flawed product can dwarf that of development.

The Software Engineering Department at the Rochester Institute of Technology (RIT) teaches (and requires) basic testing as part of its foundation courses in the first two years. In addition, it offers an upper division course on testing, giving an in-depth treatment of best-practice tools, techniques and processes. Recently we've incorporated a "Bug of the Day," which serves to broaden student awareness of the cost of software flaws. Class discussions focus on the cost of the bug, its root causes, and how it might have been discovered and repaired prior to product release.

Keywords—Software Testing, Software Engineering Education, Quality Assurance

I. INTRODUCTION

Creating software that is on time and within budget is difficult; identifying, locating and repairing bugs only adds to the difficulty. Defective software not only significantly increases the development time and cost of a system, if released, it can have catastrophic effects - up to and including death [1].

Students generally view software testing as a bland, unexciting and insignificant task [2, 3]. They feel that creating applications is fun, but reviewing the quality of an application is boring. Software testing may appear to be a time consuming, uneventful activity requiring long hours of work with few results. Furthermore, testing is not typically part of a students grade and there is a self fulfilling aspect to the view that testing is unimportant [4]. Finally, the current educational mindset is building, not breaking software [5].

By way of contrast, professional software engineers must balance software quality against product delivery on time and within budget, and testing typically accounts for over 50% of a software product's cost. Though most universities host one or more computing programs, few focus on issues related to developing high-quality, low-defect software. Testing, to the extent it is covered at all, is rarely an object of sustained focus. In particular, only a handful of programs offer a course devoted solely to software testing [6-8].

For curricula taking testing seriously, it is necessary to teach students contemporary principles, processes and practices. In addition, students need the opportunity to put what they learn into practice, using testing techniques and tools at all levels, from the individual unit to the system as a whole. However, such education is insufficient if students do not gain a deeper appreciation of the significance of testing. One way to address this concern is to incorporate material on significant software failures as part of formal coursework; at RIT we call this "The Bug of the Day" (BotD).

Prior to class, students read material describing a real world software bug and its impacts. Each class begins with a discussion of that day's bug, with the instructor serving mainly to start the discussion and keep it on track. During this time students are asked to reflect on the problem, its root causes, and how it might have been prevented. The goal is to ingrain a respect for the importance of software quality and the role of testing in any quality assurance process.

The remainder of this paper comprises five sections. Section 2 discusses the course context, including objectives supported by BotD. Section 3 describes the BotD activity in further detail, including some of the examples, guiding principles for BotD selection, some examples used in class, and associated discussion topics. Section 4 presents student feedback 4, while section 5 surveys related works. The final section summarizes our use of BotD.

II. THE TESTING COURSE CONEXT

The undergraduate software engineering program at RIT has offered an upper division software testing course since 1998. The sole prerequisite is a second year course, Introduction to Software Engineering; which introduces basic unit and functional testing in the context of a term-long, multi-iteration team project. In addition, the graduation requirement of a full year of co-operative education means most enrolled students have additional testing experience via their co-op jobs.

The course primarily serves to introduce principles and practices at the core of a disciplined testing process, and to give students experience applying contemporary testing tools and techniques. Much of this applied activity takes place in the context of a term-long testing project. With instructor guidance, teams of 4-5 students each select an open source software project, define test plans based on the available documentation and review of the code, and execute several testing iterations. At the end of the term, each group makes a presentation of their findings along with a detailed bug report. Teams are encouraged to forward their findings to the project development team.

A second objective is instilling both a sense of the importance of testing and the associated skeptical mindset that serves testers well. It is here that BotD, described in the next section, is most significant. The course meets twice a week for 110 minutes over the span of the 10-week quarter. Most class sessions have three components: a BotD discussion, a formal presentation related to testing, and a time slot devoted to a lab activity.

III. BUG OF THE DAY

A significant component of being a software engineer is how to create software that is high quality. Understanding the proper testing procedures and technologies is paramount for producing high quality software. Reviewing and understanding past instances of where defects occurred in software is beneficial for students for several reasons. Comprehending these past events and the reasoning behind them will allow students to gain experiences and knowledge which they may use when they encounter similar situations in their future careers. Additionally, these prior events will provide a real world context for the students and help to both foster their interest in software testing and reaffirm the course topics being discussed. For this reason, we created the BotD activity.

A. BUGS TO A SOFTWARE ENGINEER

System faults cause significant aggravation to end users, may lead to overall system failures and have led to billions of dollars in lost revenue [9]. Many of these defects may have been eliminated through the use of software testing [10]. Software bugs occur for a variety of reasons. Some of which include simple developer coding errors, incorrect specification understanding or incorrect requirements [11]. One of the primary goals of a software engineer is to create applications that are as defect free as possible [12, 13].

The BotD activity was a way for us to demonstrate the importance of software testing to future software engineers. Additional goals were to expose students to various testing and defect prevention techniques, to get them thinking about ways to resolve issues should they encounter a similar situation in industry and to foster student interest in the topic.

Each BotD activity was comprised of several aspects. These included a brief description of the bug, its effects and resolution taken.

B. GUIDING PRINCIPLES

The BotD examples were initially selected before the Fall term. Based upon student feedback and requests, the selected cases have been modified in the two course offerings that utilized this activity. When selecting a BotD example, several guiding principles were used. All cases should be relevant, simple and have good discussion points.

Relevant: A BotD example should represent a situation which the students could possibly face in industry. This serves two primary functions. The first is that the discussed scenario will help to educate students to instances which they are likely to encounter. Additionally, students will likely recognize these as relevant cases and thus be more inclined to both pay more attention to the bug and contribute more to the subsequent class discussion in a meaningful manner.

The instructor may strive to maintain relevance for the examples by ensuring that the discussed bugs are reasonably recent and if code is discussed make sure that it is in a language that a significant portion of the students understand. Student feedback indicated that they enjoyed discussing bugs which were fairly recent and had a technical context which they reasonably understood.

Simple: The discussed BotD should be reasonably understandable in only a brief period of time. While a student is not expected to comprehend every minute detail of the bug, they should be able to firmly understand the major points in only a five minute reading. We have found that the best examples are approximately 500 to 1000 words in length. However, this number should be shortened if the case encompases technical details.

Every student taking the course will not be at the same technical level. Some students will be far more experienced in various technologies and programming languages than others. We kept this in mind when selecting examples. The goal of the BotD activity was not to acclimate students with new technologies. The purpose is ensure that students actively think and discuss the reasons for the bug and how it could have been avoided. An overly technical or complicated example would detract from this goal.

Discussion Points: When selecting BotD examples, we strove to select cases which would have interesting and clear talking points. Some of the primary discussion points were the root cause of the error, how it could have been avoided, how its root cause could have been more quickly discovered and what would the student's recommendations be to the software group in order to avoid such instances in the future.

Instances which had reasonably clear solutions were often avoided. Examples with several possible remedies were frequently chosen since they would typically lead to a classwide, student led, informal debate over what the proper solution to the situation would be. This type of active thinking was a primary objective of the BotD activity. In the course feedback, several students indicated that while they typically do not speak up in most classes, they found themselves regularly contributing to these BotD discussions.

C. EXAMPLE #1: MARS CLIMATE ORBITER

The Mars Climate Orbiter was launched in December 1998 and was intended to collect data for approximately two earth years and serve as a relay station for five years. The cost for this NASA project was approximately $327 million. In the Fall of 1999, the orbiter reached Mars and needed to decelerate. Precise speeds and events that needed to occur were calculated in advance. After the first burn to slow the orbiter down, contact could not be reestablished and the orbiter was presumed lost [14].

A NASA analysis of the failure determined the root cause to be due to a confusion in units. A development group assumed the units of measurement to be in pounds, while the other in Newtons. One Newton is worth approximately .2248 pounds force. This miscalculation caused the orbiter to take a different trajectory than planned, one which was 170 km closer to Mars than intended. This caused the orbiter to likely burn in the upper martian atmosphere [15].

The notion that such an error could occur is often surprising to students. How could such an expensive NASA project fall prey to such a simple failure? This serves several important lessons to students. First of all, no software cases are too simple to test. Even the most basic situations are capable of leading to massive and catastrophic system failures. Additionally, if such a simple error can thwart such an expensive NASA project, then it can happen to any undertaking. Finally, in the realm of general software engineering, this case serves to solidify the importance of team communication.

Other than basic team communication, at least three other tests could have helped to avoid this bug. The first is a code review by a somewhat science-knowledgeable individual. Someone reasonably familiar with the difference between pounds and Newtons would have likely discovered this issue reasonably easily. A second possible test would be if the developed code was checked against an expected model of how the code should function or even a mock object. Finally, proper unit testing would have likely found this error at the code level.

D. EXAMPLE #2: GOOGLE MALWARE SNAFU

In early 2009, a human error at Google caused all websites to be briefly identified as possibly malicious. For a short period of time, all searches resulted in the warning “This site may be harmful to your computer”. Google soon stated that the issue was caused by “human error” when a developer designated “/” as a site that was attempting to install malware on a user’s computer. Google’s malware detection system understood this to mean that all sites on the internet were possibly malicious. Fortunately for Google, they were able to quickly roll back the update so the duration of the problem was only about 40 minutes [16].

There are several discussion points from this issue. The first is that even though this was a very minor syntax glitch, it had wide ranging ramifications and affected a significant number of internet users. This helps drive home the notion that nothing is too small to test for. If something can go wrong, it should be accounted for. Another topic of conversation is how this issue could have been avoided. While a simple response could be for the developers to ensure that they don’t make such a mistake, this is not reasonable to expect in all situations. The matter of putting a system in place to reasonably avoid an instance like this in the future is not a simple solution with a straightforward answer. Situations like this that make students think outside the box not only keep them interested in the course, but help them to come up with a wide range of possible solutions for a particular problem.

IV. STUDENT FEEDBACK

The students made their opinions clear that they both enjoyed the activity and extensively learned from it as well. At the conclusion of the course offered in the winter of 2012-2013, students were asked to anonymously fill out a survey regarding the BotD activity, the results of which are shown in Table 1. Out of 32 students who responded, all of them stated that they enjoyed the activity. 84% of the students stated that they learned a significant amount from the activity while half stated that it was their favorite part of class. Students indicated that they enjoyed the real world nature of the issues being discussed along with the impacts that software bugs may have on applications. Students also stated their enjoyment of the lighthearted nature of the activity and how the discussion was done in an informal, non-graded fashion. This made the students more at ease during this classwide exchange. Students also stated they enjoyed more recent bugs as opposed to software issues that had occurred a relatively long time ago. One interesting portion of received feedback is that a significant number of students inversely stated that either too little or too much time was devoted to the activity.

TABLE I. STUDENT COURSE EVALUATION DATA

Did you enjoy it?	Did you learn from it?	Favorite Part of Class?
32/32 (100%)	27/32 (84%)	16/32 (50%)

Following are representative samples of written feedback we have received:

The activity connected software testing concepts to reality. Discussions are fun. Became aware of some high-risk problems like handling time.

I enjoyed reading about different bugs in the real world and how how companies responded to them.

It provided us with concrete examples of how what we were studying applied in the real world, helped to demonstrate the importance of thoroughly and effectively testing applications at all levels and all stages of development, and

also showed how often proper testing is not fully done in the real world.

A primary goal for the activity was to put student's at ease, encourage them to ask questions, make mistakes and come up with alternative solutions. A desire was to have the students want to learn, not force them to learn. This was accomplished by allowing the students to propose their own bug examples and have the students provide input on the type of issues they would like to analyze. The instructor also acted as a moderator for the discussion and created a conducive environment for a student led discussion. Students have provided substantial feedback that they enjoyed the fact that this activity was not graded, thus allowing for a more easy going, friendly and non-judgmental environment. This notion is well exemplified by the following student feedback:

The fact that there was no quiz on it and that we were able to take a break from lecture. Its nice change of pace from all the rest of the classes where you can just read something over for the first time, discuss it, learn from it, and not have to worry about being graded.

I liked being able to actively get involved in discussion and exploring different reasoning and solutions.

V. RELATED WORK

We are not aware of any existing courses which use an activity related to our "Bug of the Day" exercise. A similar activity is conducted in an Engineering Secure Software class at RIT with positive results [5]. In the context of software testing, there are numerous institutions which offer courses pertaining to software testing. While many of them have activities that discuss current software techniques and technologies, none of them are known to actively discuss real world cases in such a manner on a routine basis [6, 7, 17].

There are numerous other works which purport the need for a further emphasis on software testing and quality in the student curricula [3, 4, 18, 19]. In order to foster student interest in software testing, Preston examines the use of real world projects in information technology education. Using real world projects help in motivating students in addition to providing a more real world experience for the students [20]. Harrison [21] discusses teaching software testing from two perspectives, one being the developer and the other as the software tester. The main benefit of this is approach is that students witness the importance of testing from both viewpoints and they learn to apply testing techniques from both perspectives. Goldwater [22] discusses several fun ways to incorporate software testing into a curriculum. In his example, student projects are evaluated against a common test set in a competitive fashion.

VI. SUMMARY

Future software engineers need to understand the importance of testing in developing high-quality software, as well as the principles and practices that are the foundation of a successful testing regime. In addition to standard material on testing techniques, our course incorporates a Bug of the Day activity to reinforce the cost of letting defects seep into production software. The time devoted to in-class discussion serves to sharpen student appreciation of the need for testing, and to give them the opportunity to see how appropriate testing can be critical to system success. Students generally appreciate the inclusion of these discussions. Indeed, based on preliminary discussions of this technique, at least one colleague at another university plans to incorporate BotD in their testing course. We encourage others to consider this approach in the courses they teach.

REFERENCES

[1] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *Computer,* vol. 26, pp. 18-41, 1993.

[2] E. L. Jones, "Software testing in the computer science curriculum -- a holistic approach," presented at the Proceedings of the Australasian conference on Computing education, Melbourne, Australia, 2000.

[3] C. Kaner and S. Padmanabhan, "Practice and Transfer of Learning in the Teaching of Software Testing," presented at the Proceedings of the 20th Conference on Software Engineering Education \& Training, 2007.

[4] N. Clark, "Peer testing in Software Engineering Projects," presented at the Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30, Dunedin, New Zealand, 2004.

[5] A. Meneely and S. Lucidi, "Vulnerability of the Day: Concrete Demonstrations for Software Engineering Undergraduates," *Int' Conference on Software Engineering, Education Track,* 2013.

[6] L. Pollock, "CISC 879 Software Testing and Maintenance," 2004.

[7] D. Marino. (2013, 3/25/2013). *CS498DM: Software Testing*. Available: https://wiki.engr.illinois.edu/display/cs498dmsp12/Syllabus

[8] S. Thebault, " SYLLABUS: CEN 4072/6070 SOFTWARE TESTING AND VERIFICATION," 2013.

[9] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," presented at the Proceedings of the 1st workshop on Architectural and system support for improving software dependability, San Jose, California, 2006.

[10] W. E. Wong, A. Bertolino, V. Debroy, A. Mathur, J. Offutt, and M. Vouk, "Teaching software testing: Experiences, lessons learned and the path forward," presented at the Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training, 2011.

[11] R. Patton, *Software Testing*, 2nd ed. Indianapolis, IN: SAMS, 2006.

[12] F. Kazemian and T. Howles, "A software testing course for computer science majors," *SIGCSE Bull.,* vol. 37, pp. 50-53, 2005.

[13] R. Pressman, *Introduction to Software Engineering*, 7 ed.: McGraw-Hill Science, 2009.

[14] K. Sawyer. (1999, 2/30/2013). *Mystery of Orbiter Crash Solved* [10/1/1999]. Available: http://www.washingtonpost.com/wp-srv/national/longterm/space/stories/orbiter100199.htm

[15] S. Marshal, " Software Engineering: Mars Climate Orbiter."

[16] C. Metz. (2009, 3/28/2013). *Google mistakes entire web for malware*. Available: http://www.theregister.co.uk/2009/01/31/google_malware_snafu/

[17] S. Roach, " CS5387:Software Verification and Validation," 2012.

[18] S. H. Edwards, "Teaching software testing: automatic grading meets test-first coding," presented at the Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA, USA, 2003.

[19] T. Ostrand and E. Weyuker, "Software testing research and software engineering education," presented at the Proceedings of the FSE/SDP workshop on Future of software engineering research, Santa Fe, New Mexico, USA, 2010.

[20] J. A. Preston, "Utilizing authentic, real-world projects in information technology education," *SIGITE Newsl.,* vol. 2, pp. 1-10, 2005.

[21] N. B. Harrison, "Teaching software testing from two viewpoints," *J. Comput. Small Coll.,* vol. 26, pp. 55-62, 2010.

[22] M. H. Goldwasser, "A gimmick to integrate software testing throughout the curriculum," presented at the Proceedings of the 33rd SIGCSE technical symposium on Computer science education, Cincinnati, Kentucky, 2002.