Secret Ninja Testing with HALO Software Engineering

Jonathan Bell, Swapneel Sheth, Gail Kaiser
Department of Computer Science, Columbia University, New York, NY 10027
{jbell, swapneel, kaiser}@cs.columbia.edu

ABSTRACT

Software testing traditionally receives little attention in early computer science courses. However, we believe that if exposed to testing early, students will develop positive habits for future work. As we have found that students typically are not keen on testing, we propose an engaging and socially-oriented approach to teaching software testing in introductory and intermediate computer science courses. Our proposal leverages the power of gaming utilizing our previously described system HALO. Unlike many previous approaches, we aim to present software testing in disguise - so that students do not recognize (at first) that they are being exposed to software testing. We describe how HALO could be integrated into course assignments as well as the benefits that HALO creates.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer Science Education*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*

General Terms

Human Factors

Keywords

Software Testing Education, Social Testing, Serious Games

1. INTRODUCTION

Introductory computer science courses traditionally focus on exposing students to basic programming and computer science theory, leaving little or no time to teach students about software testing [6, 11, 13]. However, exposure to testing even at a very basic level, can be very beneficial to the students [5, 11]. In the short term, they will do better on their assignments as testing before submission might help them discover bugs in their implementation that they hadn't realized. In the long term, they will appreciate the importance of testing as part of the software development life cycle. Testing can be tedious and boring, especially for students who just want their programs to work.

While there have been a number of approaches to bring testing to students early in the curriculum [5, 12, 13], there have been significant setbacks due to low student engagement and interest in testing [6]. Past efforts to teach students introductory testing practices have focused on formal testing practices, including approaches using test driven development [5, 6].

Kiniry and Zimmerman [14] propose a different approach to teaching another topic that students are often disinterested in - formal methods for verification. Their approach, which they call "secret ninja formal methods" aims to teach students formal methods without them realizing it (in a "sneaky" way). We combine this "secret ninja" methodology with a social environment and apply it to testing in order to expose students to testing while avoiding any negative preconceptions about it.

We propose a social approach to expose students to software testing using our game-like environment HALO (Highly Addictive, sociaLly Optimized) Software Engineering [19]. HALO uses MMORPG (Massively Multiplayer Online Role Playing Game) motifs to create an engaging and collaborative development environment. HALO can make the software development process and in particular, the testing process, more fun and social by using themes from popular computer games such as World of Warcraft [2]. By hiding testing behind a short story and a series of quests, HALO shields students from discovering that they are learning testing practices. We feel that the engaging and social nature of HALO will make it easier to expose students to software testing at an early stage. We believe that this approach can encourage a solid foundation of testing habits, leading to future willingness to test in both coursework and in industry.

2. BACKGROUND AND MOTIVATION

2.1 HALO Software Engineering

HALO, or Highly Addictive sociaLly Optimized Software Engineering, represents a new and social approach to software engineering. Using various engaging and addictive properties of collaborative computer games such as World of Warcraft [19], HALO's goal is to make all aspects of software engineering more fun, increasing developer productivity and satisfaction.

HALO represents software engineering tasks as *quests* and uses a storyline to bind multiple quests together - users must complete quests in order to advance the plot. Quests can either be individual, requiring a developer to work alone, or group, requiring a developer to form a team and work collaboratively towards their objective. This approach follows a growing trend to "gamify" everyday life (that is, bring game-like qualities to it), and has been popularized by alternate reality game proponents such as Jane McGonigal [15].

We leverage the inherently competitive-collaborative nature of software engineering in HALO by providing developers with social rewards. These social rewards harness operant conditioning - a model that rewards players for good behavior and encourages repeat behavior. Operant conditioning is a technique commonly harnessed in games to retain players [3, 20]. Multi-user games typically use peer recognition as the highest reward for successful players [20].

Simple social rewards in HALO can include titles - prefixes or suffixes for players' names and levels, both of which showcase players' successes in the game world. For instance, a developer who successfully closes over 500 bugs may receive the suffix "The Bugslayer." For completing quests, players also receive experience points that accumulate causing them to "level up" in recognition of their ongoing work. HALO is also designed to create an immersive environment that helps developers to achieve a flow state, a technique that has been found to lead to increased engagement and addiction [17]. While typically viewed as negative behavior, controlled addiction can be beneficial, when the behavior is productive, as in the case of software testing addiction. These methods try to motivate players similar to what's suggested in [9].

2.2 Student Software Testing

We have anecdotally observed many occasions in which students do not sufficiently test their assignments prior to submission, and conducted a brief study to support our observations. We looked at a sampling of student performance in the second level computer science course at Columbia University, COMS 1007: "Object Oriented Programming and Design in Java." This course focuses on honing design and problem solving skills, building upon students' existing base of Java programming knowledge. The assignments are not typically intended to be difficult to get to "work" - the intention is to encourage students to use proper coding practices.

With its design-oriented nature, we believe that this course presents an ideal opportunity to demonstrate students' testing habits. Our assumption is that in this class, students who were missing (or had incorrect) functionality did so by accident (and didn't test for it), rather than due to technical inability to implement the assignment. We reviewed the aggregate performance of the class (15 students - a summer session) across 4 assignments to gauge the opportunities for better testing.

We found that 33% of the students had at least one "major" functionality flaw (defined as omitting a major requirement from the assignment) and over 85% of all students had multiple "minor" functionality flaws (defined as omitting individual parts of requirements from assignments) in at least one assignment. We believe that this data shows that students were not testing appropriately and suggests that student performance could increase from a greater focus on testing (note that while other classes explicitly teach testing, this one did not). Similar student testing habits have also been observed at other institutions [7].

3. HALO FOR TESTING

As we have found that students don't test as thoroughly as they ought to, HALO can make testing more engaging for students. HALO's social nature and interesting quests will make testing more attractive to students leading to fewer bugs in their code. It also aims to make testing techniques more accessible to new programmers. We now describe HALO's role using a sample assignment and show the potential benefits of using HALO in the classroom.

3.1 A sample assignment

This assignment, taken from an introductory course at Columbia University, requires students to create a command-line implementation of a "Metrocard Vending Machine", or MVM. MVMs are provided in the subways in New York City to dispense subway cards to passengers. The basic requirements for the machine are as follows: (1) The MVM accepts money (bills or coins) and credit cards (MasterCard, Visa, American Express, and Discover) and dispenses subway cards and possibly change. (2) The MVM can dispense cards with a variable declining balance ($2.50 to $80) and weekly or monthly passes. (3) The MVM can refill variable declining balance cards with more credit. (4) The MVM contains a fixed number of subway cards and coins.

Graders run a battery of tests on student code to verify input validation and general program logic. For example, for requirement 1, they verify that users must insert enough money to complete a transaction. If using cash, then the machine must have enough of each necessary coin to make change before processing the transaction (requirement 4). If using credit cards, then the machine must validate the format of the credit card number and the issuer of the card (a simplification from how a real MVM would work, of course). The tester also tries to refill cards and test all boundary conditions on input.

We have created a storyline and set of quests for helping students understand what sort of testing they should do. The quests are designed to guide students toward determining boundary conditions on their own, allowing for a "discovery" process.

3.2 Sample Quests

We now describe a few sample quests that could be created for the above assignment. These quests are designed to have three goals: (1) To help students test their implementations vis-à-vis the requirements. (2) To make them as comprehensive as possible (i.e., if the students complete all the quests and their implementation works correctly, this would imply that they have tested their code for edge cases and so on). (3) To make them engaging and fun.

The setup is as follows: The Justice League is in distress - the Hall of Justice (their usual hideout) is undergoing repairs and is currently uninhabitable. Batman pulled some of his Gotham City strings and rented some temporary space on the cheap from the city. We join our heroes as they arrive in Gotham and are trying to get to the hideout. To look like "superheroes of the people," the team decides to travel via the subway, rather than with their powers. Your quests:

Wonder Woman's quest: Wonder Woman parks her invisible jet in the park and enters the subway. She approaches the MVM and considers what sort of card to buy. She de-

cides to purchase a weekly "unlimited ride" card, realizing that she'll want to do some shopping and see the sights, and would probably use the card many times over her week in Gotham. The card costs $29 and she inserts two $20 bills. How does your machine respond?

Flash Gordon's quest: Flash wasn't expecting to have to pay for transportation and doesn't have any money with him, so he convinces Wonder Woman to spot him $4.50 - the price of two trips in the system. He selects a $4.50 declining balance card and inserts a $5 bill. How does your machine respond?

Batman's quest: Batman whips out his BatCard credit card and slides it into the vending machine. Throwing caution to the wind, Batman tries to select a $100 valued card (note that the maximum value allowed on a card is $80). Batman settles on an $80 card and attempts to charge it to his credit card (note that the machine does not accept BatCard issued credit cards). How does your machine respond?

Superman's quest: When Superman arrives, the machine indicates that there is only one card left in its stock and that it is out of change, but that's OK because Superman can use exact change. Superman selects to receive a $30 card and inserts two $10 bills, sighing as he realizes that that's all of the cash that he has. After your machine responds that he hasn't inserted enough money, he selects a $20 card, and re-inserts his bills. How does your machine respond?

Green Lantern's quest: After a wave of superheroes all using the same metrocard vending machine, Green Lantern arrives to see that your machine has neither change nor cards. Luckily for him, he has an old card from the last time that he was in Gotham City and selects to add $20 to it with his MasterCard. How does your machine respond?

Aquaman's quest: Aquaman finally gets to the subway station and realizes that he completely missed out - your vending machine is out of cards and out of change! Aquaman wants to buy a monthly unlimited card ($104) with 6 $20 bills. Help Aquaman get a subway card by using one of the other vending machines in the station (that is, one created by a classmate). Note that while he is trying to use the machine, Aquaman is not used to vending machines and sometimes pushes extra buttons randomly. How does your classmate's machine respond?

3.3 Social Testing

We envision that students will use Eclipse as the Integrated Development Environment (IDE) and HALO will be an Eclipse Plugin. This is not required however as HALO, in theory, could be implemented for any IDE that supports plugins. Students will see a list of quests (similar to the ones mentioned above) that are pertinent to the current assignment. HALO will keep track of their progress through the various quests. Completing each quest will give the students "experience points," which could be redeemed for some extra credit at the discretion of the instructor (or required for completion of the assignment). In addition to experience points, there can also be "achievements" such as being the first student to complete a quest. There will also be a global leader board that will let the students track their own and other students' achievements and experience points. This social aspect will help the quests be engaging and fun.

Further, our system will also support the notion of group quests. Typically, in lower-level undergraduate classes at Columbia University, most assignments are meant to be done individually. Students can still work together to test their independent implementations. Consider two students Alice and Bob who have implemented the system as required individually. In a group quest, they could pick their favorite characters - Wonder Woman and Batman - and work their way together through the quests mentioned above. They would still be testing their own implementations but working with a friend or classmate is usually more fun than working alone. This might lead to students doing more quests and as a result, discovering more bugs in their code. For assignments where collaboration with teammates is allowed (usually in more advanced classes), team groups can work through the quest lines together and there can exist notions of team achievements and team experience points, rather than individual ones.

We feel that such a social emphasis on testing will make it fun. From a pedagogical point of view, it will also result in students learning about and doing software testing more often from the start.

4. RELATED WORK

There has been ongoing work in studying how best to teach students testing. Jones proposed integrating software testing across all computer science courses [12, 13] and suggested splitting different components of testing across different courses, teaching aspects incrementally so as not to overwhelm students all at once with testing. Edwards proposed a "test-first" software engineering curriculum, applying test-driven development to all programming assignments, requiring students to submit complete test cases with each of their assignments [5]. Our approach is similar to these in that we also propose early and broad exposure to testing.

Goldwasser proposed a highly social, competitive means to integrate software testing where students provided test cases that were used on each others code [8]. Elbaum et al. presented "Bug Hunt" - a tool to teach software testing in a self paced, web-driven environment [6]. With Bug Hunt, students progress through a set of predefined lessons, creating test cases for sample code. Both of these approaches introduce testing directly into the curriculum; with HALO, we aim to introduce testing surreptitiously.

Kiniry and Zimmerman proposed teaching formal verification through "Secret Ninja Formal Methods" - an approach that avoids students' apprehension to use complex mathematics by hiding it from them [14]. The Secret Ninja approach differs from those mentioned above in that it exposes students to new areas without them realizing it. They implemented this technique at multiple institutions, receiving positive student responses (based on qualitative evaluations). We adapt their "secret ninja" method for our approach.

Much work has also been done to create games to teach software engineering concepts. Horning and Wortman created Software Hut, turning the course project itself into a game, played out by all of the students together [10] . SimSE and Card Game were games created to teach students software engineering, through a game environment [1, 16]. Eagle and Barnes introduced a game to help students learn basic programming techniques: basic loops, arrays, and nested for loops [4]. However, none of these games focused specifically on testing practices. There has been research into teaching aspects such as Global Software Development [18] in a classroom, but these don't focus on software testing.

5. RESEARCH AGENDA

We foresee many potential challenges and research areas stemming from this proposal. Broadly, we believe that further study is needed in the design space of educational games. How do we appeal specifically to students and improve their approach to the software development life cycle? What requirements must we make sure we fulfill to appeal to instructors? Can we design quest templates that would work for many different classes? How do we evaluate such social software for teaching software engineering?

While we believe that we certainly have the capabilities from a software engineering perspective to implement HALO for a course, we acknowledge that none of the authors are story designers, but we are interested in studying game design further. Specifically, we are planning to begin to implement HALO for a lower-level undergraduate course and to evaluate HALO's impact and benefit to students.

6. CONCLUSION

We have described a new approach for teaching students software testing using a social learning environment. We have outlined a sample assignment and accompanying quests for HALO to enhance teaching of software testing in a classroom environment. We believe that our Secret Ninja Software Testing approach will make testing more engaging and fun for students leading to better testing and fewer bugs. We also feel that this will inculcate good software testing habits at an early stage.

7. ACKNOWLEDGEMENTS

The authors are members of the Programming Systems Laboratory, funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 2 U54 CA121852-06.

8. REFERENCES

[1] A. Baker, E. O. Navarro, and A. van der Hoek. An experimental card game for teaching software engineering processes. *Journal of Systems and Software*, 75(1-2):3 – 16, 2005. Software Engineering Education and Training.

[2] Blizzard Entertainment. World of Warcraft. `http://us.battle.net/wow/en`.

[3] J. P. Charlton and I. D. Danforth. Distinguishing addiction and high engagement in the context of online game playing. *Computers in Human Behavior*, 23(3):1531 – 1548, 2007.

[4] M. Eagle and T. Barnes. Experimental evaluation of an educational game for improved learning in introductory computing. *SIGCSE Bull.*, 41:321–325, March 2009.

[5] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 148–155, New York, NY, USA, 2003. ACM.

[6] S. Elbaum, S. Person, J. Dokulil, and M. Jorde. Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 688–697, Washington, DC, USA, 2007. IEEE Computer Society.

[7] D. Ginat, O. Astrachan, D. D. Garcia, and M. Guzdial. "But it looks right!": the bugs students don't see. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 284–285, New York, NY, USA, 2004. ACM.

[8] M. H. Goldwasser. A gimmick to integrate software testing throughout the curriculum. *SIGCSE Bull.*, 34:271–275, February 2002.

[9] T. Hall, H. Sharp, S. Beecham, N. Baddoo, and H. Robinson. What do we know about developer motivation? *Software, IEEE*, 25(4):92 –94, July-Aug. 2008.

[10] J. Horning and D. Wortman. Software Hut: A Computer Program Engineering Project in the Form of a Game. *Software Engineering, IEEE Transactions on*, SE-3(4):325 – 330, July 1977.

[11] U. Jackson, B. Z. Manaris, and R. A. McCauley. Strategies for effective integration of software engineering concepts and techniques into the undergraduate computer science curriculum. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, SIGCSE '97, pages 360–364, New York, NY, USA, 1997. ACM.

[12] E. L. Jones. An experiential approach to incorporating software testing into the computer science curriculum. In *Proceedings of the Frontiers in Education Conference, 2001. 31st Annual - Volume 02*, pages F3D–7–F3D–11 vol.2. IEEE Computer Society, 2001.

[13] E. L. Jones. Integrating testing into the curriculum arsenic in small doses. *SIGCSE Bull.*, 33:337–341, February 2001.

[14] J. R. Kiniry and D. M. Zimmerman. Secret Ninja Formal Methods. In *Proceedings of the 15th international symposium on Formal Methods*, FM '08, pages 214–228, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] J. McGonigal. *Reality Is Broken: Why Games Make Us Better and How They Can Change the World*. The Penguin Press HC, 2011.

[16] E. O. Navarro and A. van der Hoek. SimSE: an educational simulation game for teaching the software engineering process. In *Proc. of the 9th annual SIGCSE conference on Innovation and technology in CS education*, ITiCSE '04, pages 233–233, 2004.

[17] S. Park and H. Hwang. Understanding online game addiction: Connection between presence and flow. In *Human-Computer Interaction. Interacting in Various Application Domains*, volume 5613 of *Lecture Notes in Computer Science*, pages 378–386. Springer Berlin / Heidelberg, 2009.

[18] I. Richardson, S. Moore, D. Paulish, V. Casey, and D. Zage. Globalizing software development in the local classroom. In *Software Engineering Education Training, 2007. CSEET '07. 20th Conference on*, pages 64 –71, july 2007.

[19] S. Sheth, J. Bell, and G. Kaiser. HALO (Highly Addictive, sociaLly Optimized) Software Engineering. In *Proceeding of the 1st international workshop on Games and software engineering*, GAS '11, pages 29–32, New York, NY, USA, 2011. ACM.

[20] P. Wallace. *The Psychology of the Internet*. Cambridge University Press, March 2001.