CHAPTER 4

A Gameful Approach to Teaching Software Design and Software Testing*

Swapneel Sheth, Jonathan Bell, and Gail Kaiser

CONTENTS

* This chapter is based on an earlier work "Secret Ninja testing with HALO software engineering," in *Proceedings of the 4th International Workshop on Social Software Engineering*, 2011. Copyright ACM, 2011. http://doi.acm.org/10.1145/2024645.2024657; A competitive-collaborative approach for introducing software engineering in a CS2 class, in *26th IEEE Conference on Software Engineering Education and Training*, 2013. Copyright IEEE, 2013. http://dx.doi.org/10.1109/CSEET.2013.6595235. AQ 1

4.1 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 [1,2,3]. A great deal of students' mental model when they start learning programming is that "if it compiles and runs without crashing, it must work fine." Thus, exposure to testing, even at a very basic level, can be very beneficial to the students [4,5]. 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 had not realized. In the long term, they will appreciate the importance of testing as part of the software development life cycle.

However, testing can be tedious and boring, especially for students who just want their programs to work. Although there have been a number of approaches to bring testing to students early in the curriculum [3,4,5], there have been significant setbacks due to low student engagement and interest in testing [1]. Past efforts to teach students the introductory testing practices have focused on formal testing practices, including approaches using test-driven development [1,4].

Kiniry and Zimmerman [6] propose a different approach to teaching another topic that students are often uninterested in—formal methods for verification. Their approach, which they call *secret ninja formal methods*, aims to teach students formal methods without their 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 highly addictive, socially optimized (HALO) software engineering [7]. HALO uses massively multiplayer online role-playing game (MMORPG) motifs to create an engaging and collaborative development environment. It 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 [8]. 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 industry.

In addition to testing, we also want to inculcate good software design principles in early CS classes. We used a competitive tournament for this AQ 3
purpose—participation in the tournament for the students would be contingent upon their following good software design principles for their assignment. We describe our experiences on using these approaches in a CS2 class taught by the first author at Columbia University.

4.2 BACKGROUND AND MOTIVATION

4.2.1 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 during the summer of 2008. 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 did not test for it) rather than due to technical inability to implement the assignment. We reviewed the aggregate performance of the class (15 students) across four 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. Similar student testing habits have also been observed at other institutions [9].

4.2.2 HALO Software Engineering

HALO 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 [7], HALO's goal is to make all aspects of software engineering more fun, increasing developer productivity and satisfaction. It 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 be either individual, requiring a developer to work alone, or group, requiring a developer to form a team and work collaboratively toward their objective.

This approach follows a growing trend to "gamify" everyday life (i.e., bring gamelike qualities to it) and has been popularized by alternate reality game proponents such as Jane McGonigal [10]. These engaging qualities can be found in even the simplest games, from chess to Tetris, and result in deep levels of player immersion [10]. Gamification has also been studied in education, where teachers use the engaging properties of games to help students focus [11].

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 [12,13]. Multiuser games typically use peer recognition as the highest reward for successful players [13].

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 [14]. Although 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 is suggested in Reference [15].

4.2.3 Software Design

In our experience, students in early CS classes do not understand or appreciate software design. We believe that this is largely because all the early programming they have done focuses on "getting it working." Further, typical early CS assignments are a few hundred lines of Java code. Finally, most introductory CS courses at Columbia University (and other universities [16]) typically have only individual assignments and allow no collaboration on the assignments. When programs become larger and when you have to work in large teams, software design becomes a lot more critical.

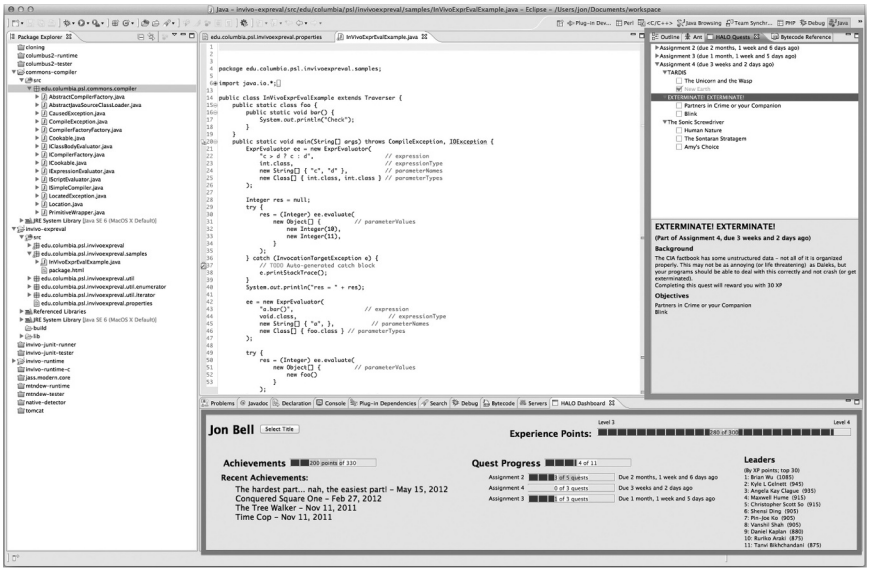
Our goal was to inculcate good software design principles via a competitive tournament where participation would be contingent based on the students' code adhering to good design principles.

4.3 GAMEFUL TESTING USING HALO

As we have found that students do not test as thoroughly as they ought to, we use HALO to make software testing more enjoyable and fun. For example, students are given a number of "quests" that they need to complete. These quests are used to disguise standard software testing techniques such as white and black box testing, unit testing, and boundary value analysis. Upon completing these quests, the students get social rewards in the form of achievements, titles, and experience points. They can see how they are doing compared to other students in the class. Although the students think that they are competing just for points and achievements, the primary benefit of such a system is that the students' code gets tested a lot better than it normally would have. Our current prototype implementation of HALO is a plug-in for Eclipse and a screenshot is shown in Figure 4.1.

4.3.1 HALO Plug-in for Eclipse

We used HALO in a class taught at Columbia University: COMS 1007 Object-Oriented Programming and Design with Java. The rest of this section describes some background about the class, how HALO was used, and the results of our case study.



AQ 4 FIGURE 4.1 The HALO eclipse plug-in: The bottom part shows the dashboard, which keeps track of the achievements, experience points, and leaderboards; The top right part shows the quest list and progress.

4.3.2 COMS 1007—Object-Oriented Programming and Design with Java

COMS 1007—Object-Oriented Programming and Design with Java was the second course in the track for CS majors and minors at Columbia University. The class was also required for majors in several other engineering disciplines, including electrical engineering and industrial engineering, and was used by other students to satisfy their general science or computer science requirement. The first author taught this course in spring (January–May) 2012.* The course goals were as follows: A rigorous treatment of object-oriented concepts using Java as an example language and Development of sound programming and design skills, problem solving and modeling of real world problems from science, engineering, and economics using the object-oriented paradigm [17]. The prerequisite for the course was familiarity with programming and Java (demonstrated through a successful completion of

* The introductory sequence of courses has undergone a change and COMS 1007 has become an honors version of the CS1 course since fall 2012.

the CS1 course at Columbia or another university, or passing marks on the AP Computer Science Exam).

In spring 2012, the class enrolment was 129, which consisted largely of freshmen and sophomores (first- and second-year undergraduates, respectively). The list of topics covered was object-oriented design, design patterns, interfaces, graphics programming, inheritance and abstract classes, networking, and multithreading and synchronization. There were roughly five biweekly assignments, which contained both theory and programming, one midterm exam, and one final exam.

As explained above, HALO uses gamelike elements and motifs from popular games such as World of Warcraft [8] to make the whole software engineering process and, in particular, the software testing process more engaging and social. HALO is not a game; it leverages game mechanics and applies them to the software development process. We now describe how we used HALO in our class.

4.3.3 A Case Study with HALO

In this class, we used HALO for three assignments. In the first two cases, HALO was not a required part of the assignment; students could optionally use it if they wanted to. For the last case, students could earn extra credit (10 points for the assignment, accounting for 0.8% of the overall course grade) by completing the HALO quests.

The final course assignment allowed students to design their own projects, making it difficult for us to predefine HALO quests, because each project was different. Instead, students were offered extra credit in exchange for creating HALO quests for their projects, thus emphasizing the *learning by example* pedagogy. Out of the 124 students who submitted Assignment 5, 77 students (62.1%) attempted the extra credit, and 71 out of these 77 students (92.21%) got a perfect score for the HALO quests that they had created.

4.3.3.1 An Assignment on Java Networking: Getting and Analyzing Data from the Internet—The CIA World Factbook

We now describe an assignment that was given to the class and the HALO quests that were created for it. The rest of the assignments and the quests are described in our technical report [18].

The Central Intelligence Agency (CIA) has an excellent collection of detailed information about each country in the world, called the CIA *World Factbook*. For this assignment, students had to write a program in Java to analyze data from the CIA *World Factbook* website, interacting

directly with the website. The student programs had to interactively answer questions such as the following:

1. List countries in *South America* that are prone to *earthquakes*.
2. Find the country with the lowest elevation point in *Europe*.
3. List all countries in the *southeastern* hemisphere.
4. List all countries in *Asia* with more than *10* political parties.
5. Find all countries that have the color *blue* in their flag.
6. Find the top *five* countries with the highest electricity consumption per capita (electricity consumption as a percentage of population).
7. Find countries that are entirely landlocked by a single country.

   For the italicized parts in the above list, the code had to be able to deal with any similar input (e.g., from a user). This should not be hard coded.

4.3.3.2 HALO Quests

We now describe the HALO quests that we used for the above assignment.

1. *TARDIS*—To interact with the CIA *World Factbook*, it would be nice to have a TARDIS. No, not like in the show, but a Java program that can transfer and read data from Internet sites. Completing this quest will reward you with 30 XP. This quest has two tasks:

   AQ 5

   a. *New Earth*—This will probably be your first program that talks to the Internet. Although this is not as complex as creating a new Earth, you should test out the basic functionality to make sure it works. Can your program read one page correctly? Can it read multiple pages? Can it read all of them?

   b. *The Unicorn and the Wasp*—Just like Agatha Christie, you should be able to sift through all the information and find the important things. Are you able to filter information from the web page to get only the relevant data?

2. *EXTERMINATE! EXTERMINATE!*—The CIA fact book has some unstructured data—not all of it is organized properly. This may not be as annoying (or life threatening) as Daleks, but your programs should

be able to deal with this correctly and not crash (or get exterminated). Completing this quest will reward you with 30 XP and unlock Achievement: Torchwood. This quest has two tasks:

a. *Partners in Crime or Your Companion*—You can get help for parsing through the HTML stuff—you could do it yourself, you could you [sic] regular expressions, or you could use an external HTML parsing library. Regardless of who your partner in crime is, are you sure that it is working as expected and not accidentally removing or keeping information that you would or would not need, respectively? AQ 6

b. *Blink*—Your program does not need to be afraid of the Angels and can blink, that is, take longer than a few seconds to run and get all the information. However, this should not be too long, say 1 hour. Does your program run in a reasonable amount of time?

3. *The Sonic Screwdriver*—This is a useful tool used by the Doctor to make life a little bit easier. Does your code make it easy for you to answer the required questions? Completing this quest will reward you with 40 XP. This quest has three tasks:

   a. *Human Nature*—It might be human nature to hard code certain pieces of information in your code. However, your code needs to be generic enough to substitute the italicized parts of the questions. Is this possible?

   b. *The Sontaran Stratagem*—For some of the questions, you do not need a clever strategy (or algorithm). However, for some of the latter questions, you do. Do you have a good code strategy to deal with these?

   c. *Amy's Choice*—You have a choice of two wild card questions. Did you come up with an interesting question and answer it?

4.3.3.3 Student-Created HALO Quests

We now describe one of the HALO quests that some students created for their own project. This highlights that students understood the basics of software testing, which was the goal with HALO. We include a short description of the project (quoted from student assignment submissions) along with the quests, because students could define their own project.

4.3.3.3.1 Drawsome Golf Drawsome Golf is a multi-player miniature golf simulator where users draw their own holes. After the hole is drawn, users take turns putting the ball toward the hole, avoiding the obstacles in their path. The person who can get into the hole in the lowest amount of strokes is the winner. There are four tasks to complete for the quest for Drawsome Golf:

1. Perfectly Framed (Task): Is the panel for the hole situated on the frame? Is there any discrepancy between where you click and what shows up on the screen? Is the information bar causing problems?
2. Win, Lose, or Draw (Task): Are you able to draw lines and water? Are you able to place the hole and the tee box? Can you add multiple lines and multiple ponds? Could you add a new type of line?
3. Like a Rolling Stone (Task): Does the ball move where it is supposed to? Do you have a good formula for realistic motion of the ball?
4. When We Collide (Task): Does the ball handle collisions correctly? Is the behavior correct when the ball hits a line, a wall, the hole, or a water hazard?

4.3.3.3.2 Matrix Code Encoder/Decoder The user will select a text file that he or she would like to encode or decode and will select the alphabet and numerical key for use. Encoded messages can be sent to a designated user using the networking principles we have learned in class.

1. I'll Handel It: Are your classes passing each other the correct information? Make sure there is no overlap between the calculations performed by one class and those of another. Are variables updated correctly to reflect user input?
2. Liszt Iterators: During the matrix multiplication process, it is necessary to keep track of several iterators simultaneously. Is each of these iterators incrementing and/or resetting at appropriate moments? Does each one accomplish a specific task?
3. What are you Haydn? Encapsulation is key! Encapsulation makes it much easier to understand code and to make changes later on. Have you broken tasks into subtasks, each united by a mini-goal? How can you break up the encoding and decoding methods? Can you break the GUI into bite-sized pieces?

4.4 BETTER SOFTWARE DESIGN VIA A BATTLESHIP TOURNAMENT

The second assignment for the class focused on design principles and, in particular, using interfaces in Java. For the assignment, which constituted 8% of the overall course grade, the students had to implement a battleship game. Battleship is a two-player board game where each player has a 10×10 grid to place five ships of different lengths at the start of the game. Each player's grid is not visible to the other player, and the player needs to guess the location of the other player's ships. Thus, by alternating turns, each player calls out "shots," which are grid locations for the other player. If a ship is present at that location, the player says "hit"; else it is a "miss." The game ends when one of the players has hit all the parts of all the opponent's ships.

The students needed to implement this game in Java with an emphasis on good design and none on the graphical aspects; the students could create any sort of user interface they wanted—a simple command line–based user interface would suffice as far as the assignment was concerned. To emphasize good design, we provided the students with three interfaces as a starting-off point for the assignment. The three interfaces, `Game`, `Location`, and `Player`, are shown in Listings 1.1, 1.2, and 1.3.

To reinforce the notion of "programming to an interface, not to an implementation" [19], there was a tournament after the assignment submission deadline. For the tournament, the teaching staff would provide implementations of the `Game` and `Location` interfaces and use each student's `Player` implementation. (In particular, the students were told to provide two implementations of the `Player`—a human player who is interactive and can ask the user for input and a computer player that can play automatically; this latter player would be used for the tournament.) As long as the students' code respected the interfaces, they would be able to take part in the tournament.

The tournament logistics were as follows. First, all student players played 1000 games against a simple AI written by the teaching staff. From these results, we seeded a single-elimination bracket for the student players to compete directly. Thus, players with good strategies would progress through the rounds and defeat players with weaker strategies. As an added extra incentive, there were extra credit points awarded to students based on how well they performed in the tournament.

Even though the extra credit was not a lot (accounting for only 0.8% of the total course grade), the combination of the extra credit and the competitive aspect made almost the entire class participate in the tournament.

Only 116 out of 129 students (89.92%) of the class elected to take part in the tournament, and of those that wanted to be in the tournament, 107 (92.24%) had implementations that functioned well enough (e.g., did not crash) and competed in the tournament.

Listing 1.1: The Game Interface

```
1 /**
2  * The game interface - this will control the
   Battleship game.
3  * It will keep track of 2 versions of the "board"
   - one for each player.
4  * It will let players take turns.
5  * It will announce hits, misses, and ships sunk
   (by calling the appropriate methods in the Player
   interface/class).
6  * @author swapneel
7  *
8  */
9 public interface Game {
10
11       int SIZE = 10;
12
13       int CARRIER = 5;
14       int BATTLESHIP = 4;
15       int SUBMARINE = 3;
16       int CRUISER = 3;
17       int DESTROYER = 2;
18
19       /**
20       * This method will initialize the game.
21       * At the end of this method, the board has
         been set up and the game can be started
22       * @param p1 Player 1
23       * @param p2 Player 2
24       */
25       void initialize(Player p1, Player p2);
26
27       /**
28       * This is the start point of playing the game.
29       * The game will alternate between the players
         letting them take shots at the other team.
```

```
30        * @return Player who won
31        */
32        Player playGame();
34}
```

AQ 7

Listing 1.2: The Location Interface

```
1 /**
2  * The Location interface to specify how x and y
   coordinates are represented.
3  * This can be used to represent the location of a
   ship or a shot.
4  * If the location is a shot, the isShipHorizontal()
   method can return an arbitrary value.
5  * @author swapneel
6  *
7  */
8 public interface Location {
9
10        /**
11        * Gets the x coordinate
12        * @return the x coordinate
13        */
14        int getX ();
15
16        /**
17        * Gets the y coordinate
18        * @return the y coordinate
19        */
20        int getY ();
21
22        /**
23        * This method will indicate whether the ship
          is horizontal or vertical.
24        * Can return an arbitrary value if the location
          is used to indicate a shot (and not a ship)
25        * @return true if ship is horizontal, false
          otherwise
26        */
27        boolean isShipHorizontal ();
28
29}
```

Listing 1.3: The Player Interface

```
1 /**
2  * The Player interface
3  * Each player will get to choose where to place the 5
   ships and how to take turns shooting at enemy ships
4  * @author swapneel
5  *
6  */
7 public interface Player {
8
9        /**
10       * This method will place a ship on the grid.
11       * This method should guarantee correctness of
         location (no overlaps, no ships over the edge
         of the board, etc.)
12       * @param size the size of the ship to place
13       * @param retry if an earlier call to this method
         returned an invalid position, this method will
         be called again with retry set to true.
14       * @return The Location of the ship
15       */
16       Location placeShip (int size, boolean retry);
17
18       /**
19       * This method will get the new target to aim
for
20       * @return The Location of the target
21       */
22       Location getTarget ();
23
24       /**
25       * This method will notify the Player of the
         result of the previous shot
26       * @param hit true, if it was a hit; false
         otherwise
27       * @param sunk true, if a ship is sunk; false
         otherwise
28       */
29       void setResult (boolean hit, boolean sunk);
30
31}
```

4.5 FEEDBACK AND RETROSPECTIVES

In this section, we describe the feedback about the course structure given by the students and our thoughts and retrospectives on using gameful approaches for CS education.

4.5.1 Student Feedback

The student feedback comes from various sources such as midterm and end of semester surveys, public reviews of the class, and e-mail sent to the first author.

HALO received mixed reviews—many students found that it was very useful; other students found that it was not beneficial. Figure 4.2 shows the students' reasons on why HALO was beneficial. Figure 4.3 shows why students thought that it was not beneficial. The main takeaway for us with HALO was the following: because it was either completely optional or only for extra credit, typically only students who are doing really well in the class will use it. Students who are having a hard time in the class will not want to do something that is optional. In an analogous manner, students will only do the extra credit if they have managed to complete the assignment early enough and sufficiently well. Thus, HALO quests needed to be more oriented toward the students

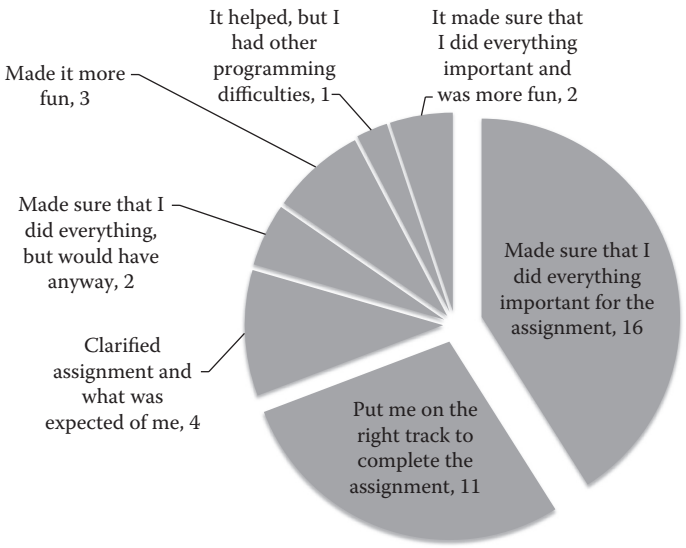


FIGURE 4.2 Reasons why HALO helped students (n = 39).

AQ 8

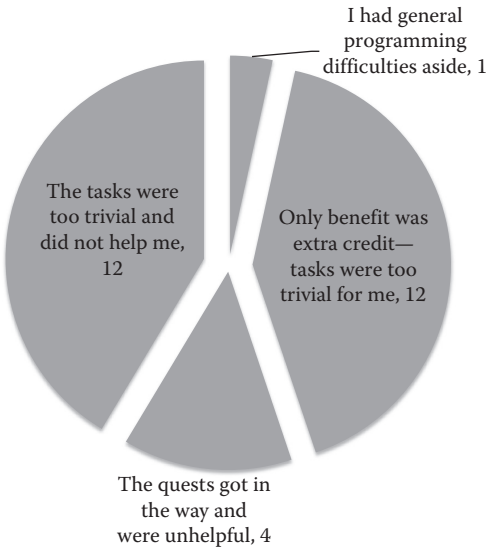


FIGURE 4.3 Reasons why HALO was not beneficial to students (n = 29).

doing well in the assignment. However, HALO quests need to entice the struggling students as they might benefit the most by being able to complete the basic tasks of the assignment. Ideally, we would like to have some adaptability or dynamic nature of the quests where the difficulty of the quests will self-adjust based on what the students would find it most useful for. For example, students who are struggling with the assignment might want quests for very basic things, whereas students who are doing well might want quests for the more challenging aspects of the assignment.

Some of the student comments are shown below:

- I really liked the class tournaments. If only there was a way to make them like mandatory.
- The assignments are completely doable, and he helps us with them by giving us Halo quests which provide a checklist of things one should be doing (they're themed, so the last one was Doctor Who themed!).
- I think it's awesome that you're sneaking your taste in music into the HALO quests. The Coldplay references are hilarious. PLEASE make every HALO quest music-themed. It keeps me awake and happy as I do my homework.

4.5.2 Thoughts on CS Education a Year Later

Our experience with using gameful elements in a classroom has been largely positive. The first author taught COMS 1007 again, which is now an honors CS1 class, in spring 2013. For that semester, we made a few changes based on our experiences from the previous year. We now describe our decisions and rationale behind these changes.

4.5.2.1 Reflections on HALO

First, we decided not to use HALO in the class. Our decision was largely based on two aspects: resource constraints and research challenges. AQ 9

4.5.2.1.1 Constraints The second author developed the HALO eclipse plug-in and helped students set it up during the first iteration of COMS 1007. For use in future semesters, the plug-in would need to be updated and maintained. Creating student accounts and having them install and use the plug-in is straightforward albeit very time consuming, especially for a class with over 100 students. Note that the students in our classes are typically freshmen and sophomores with very little experience with Java and Eclipse. One option would be to get an extra TA, if possible, for future courses if we want to use HALO.

4.5.2.1.2 Assignment and Quest Design A big advantage with HALO is that there are no constraints with assignment creation and design. However, there are a few implications as far as quest design is concerned. First, creating good and fun quests takes a significant amount of time. In our experience, quest creation took about 30% time that it takes to create an assignment. In other words, creating quests for three to four assignments is about the same time as creating an entire new assignment. This needs to be factored in with the other time constraints that an instructor might have. Second, quest creation is much easier and faster if the same person creates both the assignment and the quests for it. In our case, there were a few assignments that were designed or conceived by TAs, and unfortunately, they could not create the quests themselves as they had not taken a software engineering or software testing class yet. This meant that quest creation had to be done by the first author, and hence needed much more time.

4.5.2.1.3 Adaptive Quests As the student feedback shows, we need HALO quests that can target, both, struggling students and students who are doing well. One option would be to create specific sets of quests for the different

target demographics. This is not an ideal option however; we would need to spend much time on quest creation. Further, unless this particular assignment has been used in previous classes, it might be hard to know *a priori* what parts of it will be easy and what parts will be difficult, thereby making it challenging to design appropriate quests. In our experience, instructor and student opinions on the ease or difficulty of assignments do not always converge. The better option would be to have some way of automatically "scaling" the difficulty of quests, but we are not completely sure of what this entails and much more research is required on this topic.

4.5.2.2 Reflections on Tournaments

Second, we continued using the tournament structure for encouraging students to use good design. The tournament required significant resources as well. Because we had over 100 students who would participate in the tournament, the second author wrote a generic framework that would take all the student code and create and run the tournament as described above. Automating the entire process certainly was essential as it helped save a lot of time and the code could be reused for future tournaments. In spite of being able to reuse this for the spring 2013 semester, a significant amount of additional time and effort was still needed. The first reason was to create the game framework and tournament AI for the students to compete against. We could not reuse this as we changed the assignment to use Othello instead of Battleship. The second reason was actually running the tournament—we ran 1000 games for each student in the first round. The official timeout policy was 1 minute per game, that is, if the game took longer than 1 minute to complete, the player would be disqualified. Using a very conservative estimate that each game takes 1 second to run, we would still need roughly 28 CPU hours to run just the first round for 100 students. Typically, we try to release homework grades 1 week after homework is due. If a tournament is to be run, these numbers need to factor into time and resource allocation constraints for the class, instructor, and TAs.

4.6 RELATED WORK

There has been ongoing work in studying how best to teach students testing. Jones [3,5] proposed integrating software testing across all computer science courses 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 [4] 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. 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 in each other's code [20]. Elbaum et al. [1] presented Bug Hunt—a tool to teach software testing in a self-paced, web-driven environment. 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 [6] proposed teaching formal verification through *secret ninja formal methods*—an approach that avoids students' apprehension to use complex mathematics by hiding it from them. The secret ninja approach differs from those mentioned earlier 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 adapted their *secret ninja* method for HALO.

Much work has also been done to create games to teach software engineering concepts. Horning and Wortman [21] created Software Hut, turning the course project itself into a game, played out by all of the students together. SimSE and card game were games created to teach students software engineering through a game environment [22,23]. Eagle and Barnes [24] introduced a game to help students learn basic programming techniques: basic loops, arrays, and nested for loops. However, none of these games focused specifically on testing practices. There has been research into teaching aspects such as global software development [25] in a classroom, but these do not focus on software testing.

TankBrains [26] is a collaborative and competitive game used in a CS course where students competed to develop better AIs. Bug Wars [27] is a classroom exercise where students seed bugs in code, swap examples, and compete to find the most bugs (in each other's code). Although TankBrains and Bug Wars are specific programming activities, we present a general approach to teaching introductory computer science that is both cooperative and competitive.

There have been several approaches toward integrating games into CS curricula. One of the earliest such attempts was Software Hut, where the authors formulated their project-based software engineering course as a game [21]. Groups competed to be the most "profitable"—where performance

was tracked by "program engineering dollars" (a fictional currency). This technique is similar to ours in that we both tracked student performance with points and added in other game concepts, such as quests and achievements. KommGame is an interface that encapsulates many collaborative software development activities such as creating documentation or reporting and resolving bugs and tracks each student with karma points [28]. This social and collaborative environment represented real-world open-source development environments.

SimSE [23] and Problems and Programmers [22] are two simulation-oriented games that give students a "real-world" software engineering experience. Somewhat similar, Wu's Castle [24] is a game to teach students basic programming constructs such as loops. These three projects are games, whereas we have built a game layer on top of the regular course environment.

4.7 CONCLUSION

In this chapter, we described how we incorporated gameful elements for teaching software testing and software design in a CS2 class. Students learnt software testing using a social learning environment. We described our HALO prototype, an assignment, and the accompanying quests for HALO to enhance teaching of software testing in a CS2 class. Students learnt software design via a competitive tournament, and we described details of our assignment and on how the tournament was run. The feedback from the students for both these aspects was largely positive.

We believe that our approach will make testing and design more engaging and fun for students, leading to better systems. We also feel that this will inculcate good software engineering habits at an early stage.

ACKNOWLEDGMENTS

We thank Joey J. Lee of Teachers College's Games Research Lab and Jessica Hammer of Carnegie Mellon University's Human-Computer Interaction Institute, for their assistance with the pedagogical aspects of this work. We thank all the students who participated in the COMS 1007 class. We thank the teaching assistants, Lakshya Bhagat, Amrita Mazumdar, Paul Palen, Laura Willson, and Don Yu, for helping with the class.

The authors are members of the Programming Systems Laboratory (PSL) at Columbia University. PSL is funded in part by NSF CCF-1302269, NSF CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

REFERENCES

AQ 10
AQ 11

1. Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. Bug hunt: Making early software testing lessons engaging and affordable. *Proceedings of the 29th International Conference on Software Engineering*, pp. 688–697, IEEE Computer Society, Washington, DC, 2007.
2. Ursula Jackson, Bill Z. Manaris, and Renée A. McCauley. Strategies for effective integration of software engineering concepts and techniques into the undergraduate computer science curriculum. *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, pp. 360–364, ACM, New York, 1997.
3. Edward L. Jones. Integrating testing into the curriculum arsenic in small doses. *SIGCSE Bulletin*, 33:337–341, 2001.
4. Stephen H. Edwards. Rethinking computer science education from a test-first perspective. *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 148–155, ACM, New York, 2003.
5. Edward L. Jones. An experiential approach to incorporating software testing into the computer science curriculum. In *Proceedings of the 31st Annual Frontiers in Education Conference*, vol. 2, pp. F3D–7–F3D–11, IEEE Computer Society, Washington, DC, 2001.
6. Joseph R. Kiniry and Daniel M. Zimmerman. Secret ninja formal methods. *Proceedings of the 15th International Symposium on Formal Methods*, pp. 214–228, Springer-Verlag, Berlin, Germany, 2008.
7. Swapneel Sheth, Jonathan Bell, and Gail Kaiser. HALO (Highly Addictive, sociaLly Optimized) software engineering. *Proceedings of the 1st International Workshop on Games and Software Engineering*, pp. 29–32, ACM, New York, 2011.
8. Blizzard Entertainment. World of Warcraft. http://us.battle.net/wow/en.
9. David Ginat, Owen Astrachan, Daniel D. Garcia, and Mark Guzdial. "But it looks right!": The bugs students don't see. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pp. 284–285, ACM, New York, 2004.
10. Jane McGonigal. *Reality Is Broken: Why Games Make Us Better and How They Can Change the World*. Penguin Press, New York, 2011.
11. Joey J. Lee and Jessica Hammer. Gamification in education: What, how, why bother? *Academic Exchange Quarterly*, 15(2):2, 2011.
12. John P. Charlton and Ian D.W. Danforth. Distinguishing addiction and high engagement in the context of online game playing. *Computers in Human Behavior*, 23(3):1531–1548, 2007.
13. Patricia Wallace. *The Psychology of the Internet*. Cambridge University Press, New York, 2001.
14. SungBok Park and Ha Hwang. Understanding online game addiction: Connection between presence and flow. In *Human-Computer Interaction. Interacting in Various Application Domains*, *Lecture Notes in Computer Science*, vol. 5613, pp. 378–386, Springer, Berlin, 2009.
15. Tracy Hall, Helen Sharp, Sarah Beecham, Nathan Baddoo, and Hugh Robinson. What do we know about developer motivation? *IEEE Software*, 25(4):92–94, 2008.

16. Laurie Williams and Lucas Layman. Lab partners: If they're good enough for the natural sciences, why aren't they good enough for us? *Proceedings of the 20th Conference on Software Engineering Education & Training*, pp. 72–82, 2007.
17. Columbia Engineering—The Fu Foundation School of Engineering and Applied Science. Bulletin 2011–2012. http://bulletin.engineering.columbia.edu/files/seasbulletin/2011Bulletin.pdf, 2011.
18. Swapneel Sheth, Jonathan Bell, and Gail Kaiser. A gameful approach to teaching software design and software testing—Assignments and quests. Technical Report cucs-030-13, Department of Computer Science, Columbia University, New York, 2013. http://mice.cs.columbia.edu/getTechreport.php?techreportID=1557.
AQ 13 19. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*, 1995.
20. Michael H. Goldwasser. A gimmick to integrate software testing throughout the curriculum. *SIGCSE Bulletin*, 34:271–275, 2002.
21. James (Jim) Horning and David B. Wortman. Software hut: A computer program engineering project in the form of a game. *IEEE Transactions on Software Engineering*, 3(4):325–330, 1977.
22. Alex Baker, Emily Oh Navarro, and André van der Hoek. An experimental card game for teaching software engineering processes. *Journal of Systems and Software*, 75(1/2):3–16, 2005.
AQ 14 23. Emily Oh Navarro and André van der Hoek. SimSE: An educational simulation game for teaching the software engineering process. *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in CS Education*, pp. 233–233, 2004.
24. Michael Eagle and Tiffany Barnes. Experimental evaluation of an educational game for improved learning in introductory computing. *SIGCSE Bulletin*, 41:321–325, 2009.
25. Ita Richardson, Sarah Moore, Daniel Paulish, Valentine Casey, and Dolores Zage. Globalizing software development in the local classroom. *Proceedings of the 20th Conference on Software Engineering Education Training*, pp. 64–71, July 2007.
26. Kevin Bierre, Phil Ventura, Andrew Phelps, and Christopher Egert. Motivating OOP by blowing things up: An exercise in cooperation and competition in an introductory Java programming course. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pp. 354–358, 2006.
27. Renee Bryce. Bug Wars: A competitive exercise to find bugs in code. *Journal of Computing Sciences in Colleges*, 27(2):43–50, 2011.
28. Terhi Kilamo, Imed Hammouda, and Mohamed Amine Chatti. Teaching collaborative software development: A case study. *Proceedings of the 2012 International Conference on Software Engineering*, pp. 1165–1174, 2012.

Author Query Sheet

Chapter No: 4

Query No.	Queries	Response
AQ 1	We have set the **Note** provided in the chapter opener page of this chapter as note. Please confirm if this is okay.	
AQ 2	We have shortened the Running Head, please confirm.	
AQ 3	Should "CS" be set as "computer science" throughout.	
AQ 4	Please confirm the deletion of color indications in Figure 4.1 as it is gray-colored figure.	
AQ 5	Please spell out or clarify XP.	
AQ 6	"you could you [sic] regular expressions" not clear.	
AQ 7	Please confirm if line number "34" can be changed to "33".	
AQ 8	Please provide the significance of numbers used in Figures 4.2 and 4.3.	
AQ 9	Two aspects "resource constraints and research challenges" have been listed, but three aspects have been discussed, excluding research challenges. Please check.	
AQ 10	References are renumbered sequentially based on order of occurrence in citations. Please confirm.	
AQ 11	The section head **Bibliography** is changed to **References**, since all references are cited in text. Please confirm.	
AQ 12	Please provide date and location of proceedings in references 1, 2, 4, 5, 6, 7, 9, 16, 26, 28.	
AQ 13	Please provide publisher name and location for the book in reference 19.	
AQ 14	Please provide date and location of proceedings in reference 23 and also check the page range (pp. 233–233).	