A Competitive-Collaborative Approach for Introducing Software Engineering in a CS2 Class

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

Abstract

Introductory Computer Science (CS) classes are typically competitive in nature. The cutthroat nature of these classes comes from students attempting to get as high a grade as possible, which may or may not correlate with actual learning. Further, there is very little collaboration allowed in most introductory CS classes. Most assignments are completed individually since many educators feel that students learn the most, especially in introductory classes, by working alone. In addition to completing "normal" individual assignments, which have many benefits, we wanted to expose students to collaboration early (via, for example, team projects). In this paper, we describe how we leveraged competition and collaboration in a CS2 class to help students learn aspects of computer science better — in this case, good software design and software testing — and summarize student feedback.

1. Introduction

Students at top US universities grow increasingly competitive — racing against each other to achieve better grades. Within coursework, this can translate to students feeling like they must always be working to one-up the other when studying, completing assignments, and on exams. One part of the problem in such situations is that grades in a classroom is typically a zero-sum game, where it's either not possible or highly unlikely that all students get the highest grade. Many universities suggest a "curve" for the class to ensure uniformity in grading for students and to prevent grade inflation. This high-pressure situation can discourage communication between students and may result in an unpleasant learning environment. Further, this kind of competition may or may not lead to better learning of the class material. One way to address this is to use the inherent competitive aspects to help students learn certain parts of the syllabus better.

Another way to ameliorate this problem would be to introduce collaborative assignments in the curriculum, where students work together to complete assignments and learn the material. However, in introduction-level computer science courses, educators have shied away from collaborative work, citing concerns of individual student accountability [17].

In addition to lacking collaborative work, introductory CS courses also typically focus very little on teaching software testing [9, 13], even though previous work has found it highly beneficial [8, 13]. We sought to address these concerns by creating a competitive-collaborative approach to teaching software engineering in an introductory level CS course. This paper describes our approach and provides an evaluation of its usage in a CS2 class at Columbia University.

978-1-4673-5140-9/13 © 2013 IEEE CSEE&T 2013, San Francisco, CA, USA

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

2. Background

COMS 1007, Object Oriented Programming and Design with Java, is the second course in the track for CS majors and minors at Columbia University. The first author taught this course in Spring (January-May) 2012[1]. The course goals are "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" [6]. The prerequisite for the course is familiarity with programming and Java (demonstrated through a successful completion of the CS1 course at Columbia or another university, or passing marks on the AP Computer Science Exam).

In Spring 2012 the class enrollment was 129, which consisted largely of freshmen and sophomores (first and second year undergraduates). The list of topics covered were Object Oriented Design, Design Patterns, Interfaces, Graphics Programming, Inheritance and Abstract Classes, Networking, and Multithreading and Synchronization. There were five roughly biweekly assignments, which contained both theory and programming, and one midterm, and one final exam.

3. Approach and Evaluation

3.1. Competition — 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 2-player board game where each player has a 10x10 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's 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 omitted from this paper for brevity, and are shown in our accompanying technical report [16].

To reinforce the notion of "programming to an interface, not to an implementation" [10], 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 that 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),

[1]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 combination of the extra credit and the competitive aspect made almost the entire class participate in the tournament. 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.,* didn't crash) and competed in the tournament.

3.2. Competition — Gamification using HALO

Introductory CS classes typically do not focus on software testing [9, 13]. A lot of students' mental model when they start learning programming is that "if it compiles and runs without crashing, it must work fine." Despite numerous attempts to introduce testing early in CS programs and many known benefits to inculcating good testing habits early in one's programming life [8, 13], students remain averse to software testing as there is low student interest in software testing [9].

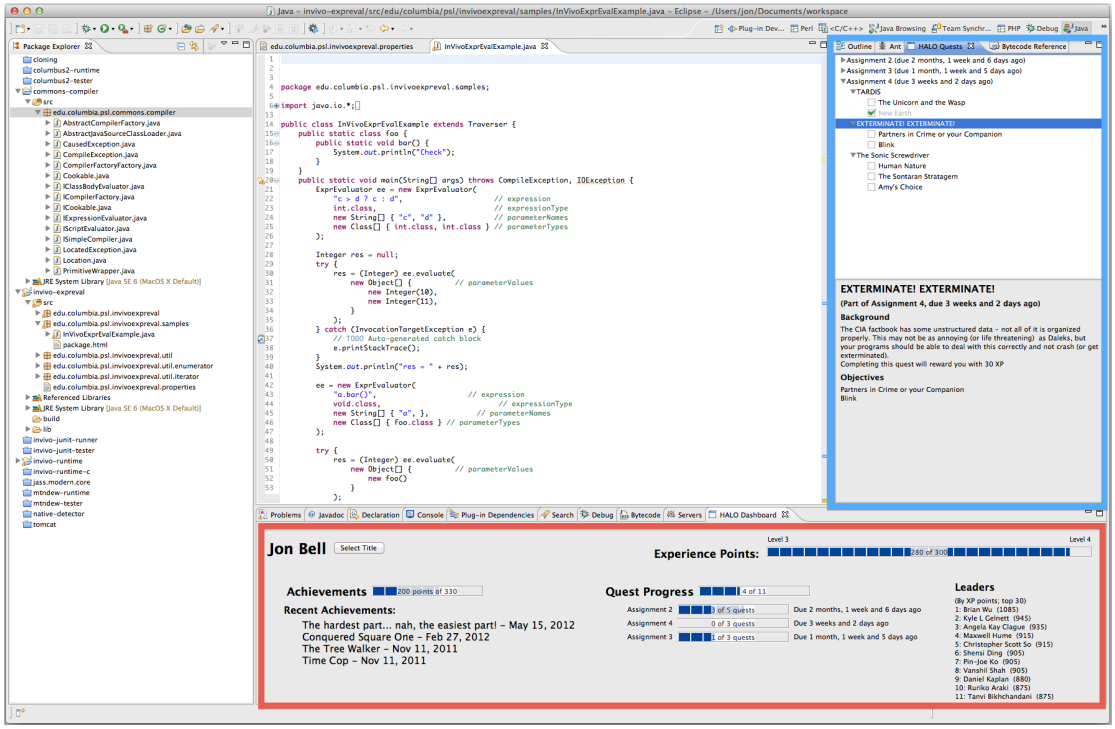


Figure 1: The HALO Eclipse plugin: The bottom part (highlighted in red) shows the dashboard, which keeps track of the achievements, experience points, leaderboards; The top right part (highlighted in blue) shows the quest list and progress

To address this problem, we used an internally developed research system called HALO — "Highly Addictive sociaLly Optimized Software Engineering" [2]. Our previous work describes early prototypes of HALO; in this paper, we describe how we used it for the CS2 class and the feedback from real users. HALO uses game-like elements and motifs from popular games like World of Warcraft [4] 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. For example, in HALO, students are given a number of "quests" that they need to complete. These quests are used to disguise standard software testing techniques like 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. While 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 plugin for Eclipse and a screenshot is shown in Figure 1.

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 (described further in Section 3.3), making it difficult for us to pre-define HALO quests, since 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.

3.2.1. 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 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: 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 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 *5* countries with the highest electricity consumption per capita. (Electricity consumption % population) 7. There are certain countries that are entirely landlocked by a single country. Find these countries. Note: For the italicized parts in the above, the code had to be able to deal with any similar input (*e.g.,* from a user). This should not be hard coded.

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

i) **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: 1) *New Earth* — This will probably be your first program that talks to the Internet. While this isn't as complex as creating a new Earth, you should test out the basic functionality to make sure it works. Can you program read one page correctly? Can it read multiple pages? Can it read all of them? 2) *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 webpage to get only the relevant data?

ii) **EXTERMINATE! EXTERMINATE!** — The CIA factbook 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: 1) *Partners in Crime or your Companion* — You can get help for parsing through the HTML stuff - you could do it yourself, you could use regular expressions, you could use an external HTML parsing library. Regardless of who your partner in crime is, are you sure that it's working as expected and not accidentally removing or keeping information that you would or wouldn't need, respectively? 2) *Blink* — Your program doesn't need to be afraid of the Angels and can blink, i.e., take longer than a few seconds to run and get all the

information. However, this shouldn't be too long, say 1 hour. Does your program run in a reasonable amount of time?

iii) **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:
1) *Human Nature* — It might be human nature to hard code certain pieces of information in your code. But your code needs to be generic enough to substitute the italicized parts of the questions? Is this possible? 2) *The Sontaran Stratagem* — For some of the questions, you don't need a clever strategy (or algorithm). But for some of the latter questions, you do. Do you have a good code strategy to deal with these? 3) *Amy's Choice* — You have a choice of 2 wild card questions. Did you come up with an interesting question and answer it?

3.2.3. 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, since students could define their own project.

"**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 towards 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 for when the ball hits a line, a wall, the hole, or a water hazard?"

3.3. Collaboration — Team Projects

Most introductory CS courses at Columbia University (and other universities [17]) typically have only individual assignments and allow no collaboration on the assignments. On the other hand, most graduate classes and real world projects typically are done in (large) teams. We felt that it would be beneficial for students to work in teams as early as possible. Thus, for the last assignment of the class, the students could optionally work in a team of up to three people. If they chose this option, they could define any project that they liked subject to a few constraints described below. The alternative would be to do an individual "default" project that was defined by the teaching staff.

In terms of constraints for the custom projects, we provided them with two categories, which corresponded to the material they had learnt in the course of the semester. The teams needed to choose at least two topics from each category and use them in their project in a meaningful way. Category 1 was roughly "design" and Category 2 was roughly "functionality," and their specific requirements are described in Table 1.

One further pedagogical incentive for custom projects is to allow students the freedom to choose to do something that they really like and importantly, deal with the ambiguity that results out of it. Typically, most assignments in universities are very well defined. There is a specific and well-outlined set of requirements that students need to accomplish to do well on the assignments.

Table 1: Categories for project constraints

Category 1	Category 2
Formal Design (CRC, Class diagrams, Sequence/State Diagrams)	Java Graphics
Interfaces (Define and use)	Networking
Inheritance	Multithreading
Design Patterns	Advanced Java (Data Structures, Reflection, External Libraries, etc.)

However, this does not mimic most real-world projects where requirements are often ambiguous or uncertain, may change over time, and so on.

Thus, through this assignment (and a similar flavor to some of the earlier assignments), students would learn how to deal with ill-defined programming assignments. We saw this with almost all the teams — many had ideas that changed and evolved over time; some had to modify the scope because they didn't have enough time to do what they wanted; some had to change part of the project as it was not technically feasible or would require a time-intensive manual effort; a few abandoned the project idea and decided to do the default option as it was "easier."

41 of the 129 (31.78%) students ended up doing the default individual assignment. Out of the rest, 11 did the custom project, but with a team size of one. Thus, 69 students (53.49%) did the custom project in teams of two or more.

There were a lot of interesting projects that resulted out of this. A nice side-effect of letting students choose their own projects was that the projects ended up being something that the students really cared about — resulting in many projects that were socially relevant and/or arty. The students wanted to build something that other people would use or something that would help them do a certain task better. Many of the projects had relevance to something outside of this class. Some of the most creative projects are described below (quoted from the project summary submitted by the teams):

"**Meal Planner**: A meal planner that uses the USDA nutrient database to pull nutrient profiles from food. It allows a user to create recipes, share them, and learn about the % values of the foods. It's intended to simplify nutrition."

"**What Should I Wear** is a program that advices the user of what clothing/items to wear/bring depending on various weather conditions. It uses weather.com as a database, so it covers any zip code found in that website. It has functionality to work for up to three days ahead (any more and the data is just too unreliable)."

"**Corn Maze Race!** Users compete one-on-one to see who can complete a series of mazes in the fewest number of moves. The mazes are randomly generated and presented to the user in a "corn maze" style, i.e. the user can only see the area around him or her, not the whole maze. When the user finds and lands on the star, the maze is complete. The server then displays the information and determines the winner."

"The **Swapagtochi** game is inspired by Tamagotchi. Like a Tamagotchi, a Swapagotchi ages as time passes and needs to be taken care of. The game does not tell the user what the Swapagotchi wants, so the use [sic] must determine items to give the Swapagotchi based on the weather icon and how much time has passed. The goal of the game is to keep the Swapagotchi happy, which can be attained by meeting his requests (for food, when hungry) and for weather protection (sunglasses when hot and sunny, umbrella when rainy). Swapagotchi is special because weather is determined by the weather of the user's zipcode (or any zipcode they decide to answer) from the weather channels website"

"**Database Searcher**: We have created a program that can sort a database of contacts. It contains name, major, email, region, school, year graduating and comments section. This directory is searchable and users can create customized searches/subsists from the main list (ex: create a list of all the CS majors who graduated in 2008 from Columbia College and live in Arizona). Information is inputted into the system through the user loading a csv file or html file or manually through the command line. We were inspired to do this by our club (the Society of Women Engineers) and plan on using it for our own club database to keep in contact with SWE alumni."

"**Synthesizer** is an application that creates song composed using Java. The compose method creates and adds Phrases and Parts to the Score. I used external music libraries for Java the main one being JMusic. I am a music and computer science double major, and I was looking for a way to realize the music I write using my computer."

Finally, many students also decided to implement games such as or similar to Sudoku, Multiplayer Minesweeper, Guess Who?, Hangman, Checkers, Online Poker, Bejeweled, Scrabble, Mafia, Monopoly, Yahtzee, Tetris, and Dance Dance Revolution.

3.4. Collaboration — Lectures in Class

The meeting schedule for the class was two 75 minute lectures per week. The goals with the class lectures were the following: first, as the students are new to CS, we wanted to ensure that stereotypes about CS being "geeky" and "boring" are not reinforced; second, very often in CS classes, students get used to the notion of very precise and well-defined assignments and when they go work for industry or pursue a Ph.D., they realize that this is typically not the case and making this transition may not be easy.

For the first goal, a very informal and collaborative classroom environment was created. From the first day of class, students were encouraged to participate, interrupt, and disagree with the material being talked about. This resulted in a lot of discussion and sharing of thoughts and ideas among the students. This is much harder to do with large class sizes, but even with 129 students in the class, the classroom environment was interactive and fun. We deliberately did not limit ourselves a lecture/presentation style of teaching, but instead leveraged aspects of active learning such as class discussions and group activities. Another strategy, which worked out well, was to get the students involved in deciding some of the material that would be taught. Since the course syllabus offers a bit of flexibility, students appreciated that they had a voice in what was being taught (which is typically not the case in most courses).

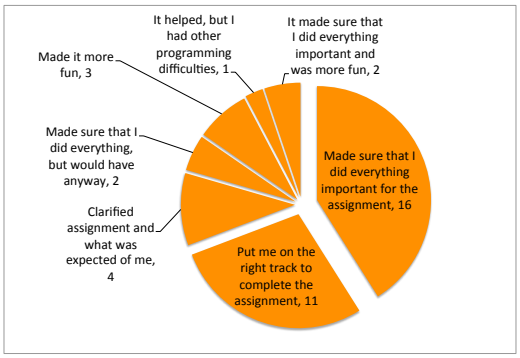
The second goal was addressed in three ways. First, all assignments had an element of uncertainty and vagueness about it. Initially, this lead to some anxiety and nervousness in the students, but towards the end of the semester, they were much more relaxed and comfortable dealing with ambiguity. Second, the custom projects that the students could do (as described in Section 3.3) further strengthened this goal. Finally, the classroom lectures reinforced all of these aspects along with the importance of being flexible and accepting that uncertainty is the nature of the beast when it comes to CS.

4. Feedback

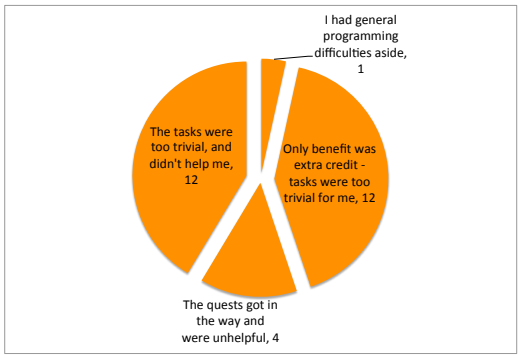
In this section, we now describe the qualitative feedback about the course structure given by the students. This feedback comes from various sources such as midterm and end of semester surveys, public reviews of the class, and email sent to the first author.

4.1. Feedback on the Competitive Aspects

HALO received mixed reviews — many students found it was very useful; other students found that it was not beneficial. Figure 2a shows the students' reasons on why HALO was beneficial.



(a) Reasons HALO helped students (n=39)



(b) Reasons HALO was not beneficial to students (n=29)

Figure 2: HALO Feedback

Figure 2b shows why students thought it was not beneficial. The main takeaway for us with HALO was the following: Since 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's optional. In an analogous manner, students will only do the extra credit if they've managed to complete the assignment early enough and sufficiently well. Thus, HALO quests needed to be more oriented towards the students doing well in the assignment. On the other hand, HALO quests need to entice the struggling students as they might benefit the most by being able 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 (both positive and negative) on the competitive aspects of the class 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."
- "He wasted my time in various ways such as having a battleship tournament in class instead of actually teaching. Instead of fully utilizing the fact that there was a computer and projector in the room, he would write some code on the board, which could take a little while. This class seriously felt like "here's a homework assignment, now go do it and become a better programmer." "

4.2. Feedback on the Collaborative Aspects

Some of the student comments (both positive and negative) on the collaborative aspects of the class are shown below:

- "I've been working at a startup for a few months now and I've come to realize how virtually everything that's of any significance at all gets done in a combination of working as a group

and as individuals; what you accomplish alone is so much smaller in scale than what you can achieve in a group. In addition, I've found that there aren't as many incredibly-well-defined assignments when you're creating a product - almost everything is a tradeoff in priorities, and it's incredibly important not to get panicky when you feel like you don't have a clear definition so you can work out (with your team) how to develop a plan going forward. You're the first teacher I've had to recognize that, to share those insights with the class, and to actually apply that kind of "well, duh" real life expertise to the way you teach class. "

- "I just wanted to say thank you for making class so engaging. It's really rare to feel like learning is a collaborative experience between the professor and the students, and even rarer to feel like it's a collaborative experience amongst the students themselves, and you've been awesome at fostering that kind of environment."
- "I don't think it would be difficult to give assignments that were somewhat more focused (and perhaps shorter but more frequent)."

Finally, although anecdotal in nature, many students said that they were now considering or decided to do a major in CS or a double-major in CS after taking this class.

5. Related Work

Collaboration has proven to be an important aspect in CS courses. TankBrains [3] is a collaborative and competitive game used in a CS course where students competed to develop better AIs. Bug Wars [5] is a classroom exercise where students seed bugs in code, swap examples, and compete to find the most bugs (in each other's code). While 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 towards 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 [12]. 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 track student performance with points, but we also 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 [14]. This social and collaborative environment represented real world open source development environments.

SimSE [15] and Problems and Programmers [1] are two simulation-oriented games that give students a "real world" software engineering experience. Somewhat similar, Wu's Castle [7] 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.

There are other approaches aimed specifically at integrating testing into the early CS curriculum. Bug Hunt [9] is a web-based tutorial application that teaches students about software testing. However, Bug Hunt did not use any game elements, unlike our approach. Goldwasser proposed a "gimmick" for teaching software testing [11], where students competed to achieve better test suites. This is similar to our work in that it involves a competitive aspect.

6. Conclusion

In this paper, we described a competitive-collaborative approach for teaching a CS2 class. The competitive aspects that we used in the class leveraged a code tournament and HALO to teach students good software design and the basics of software testing. The collaborative aspects allowed students to work in teams to create a project of their choice. This resulted in many interesting

projects, including some socially relevant and arty projects. Finally, a fun, informal, and collaborative classroom was used to prevent creating stereotypes that CS is boring. The overall student feedback for the class was very positive. Most of the students enjoyed the competitions, the HALO quests, and the team projects and many students decided to major in CS after the class.

Acknowledgment

We would like to thank all the students who participated in the COMS 1007 class. We would also like to thank the Teaching Assistants (Lakshya Bhagat, Amrita Mazumdar, Paul Palen, Laura Willson, Don Yu) for helping with the class. The authors are members of the Programming Systems Laboratory funded in part by NSF CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

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.

[2] J. Bell, S. Sheth, and G. Kaiser. Secret Ninja Testing with HALO Software Engineering. In *Proc. of the 4th Intl. workshop on Social software engineering*, SSE '11, pages 43–47, 2011.

[3] K. Bierre, P. Ventura, A. Phelps, and C. Egert. Motivating OOP by blowing things up: An Exercise in Cooperation and Competition in an Introductory Java Programming Course. In *Proc. of the 37th SIGCSE tech. symp. on Computer science education*, SIGCSE '06, pages 354–358, 2006.

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

[5] R. Bryce. Bug Wars: A Competitive Exercise to find Bugs in Code. *J. Comput. Sci. Coll.*, 27(2):43–50, Dec. 2011.

[6] Columbia Engineering — The Fu Foundation School of Engineering and Applied Science. Bulletin 2011-2012. `http://bulletin.engineering.columbia.edu/files/seasbulletin/2011Bulletin.pdf`, 2011.

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

[8] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *18th annual ACM SIGPLAN conf. on OO programming, systems, languages, and applications*, OOPSLA '03, pages 148–155, 2003.

[9] S. Elbaum, S. Person, J. Dokulil, and M. Jorde. Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable. In *Proc. of the 29th Intl. Conf. on Software Engineering*, ICSE '07, pages 688–697, 2007.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object-oriented design, 1995.

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

[12] 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.

[13] 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 *Proc. of the 28th SIGCSE tech. symp. on Computer science education*, SIGCSE '97, pages 360–364, 1997.

[14] T. Kilamo, I. Hammouda, and M. A. Chatti. Teaching collaborative software development: a case study. In *Proc. of the 2012 Intl. Conf. on Software Engineering*, ICSE 2012, pages 1165–1174, 2012.

[15] 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 conf. on Innovation and technology in CS education*, ITiCSE '04, pages 233–233, 2004.

[16] S. Sheth, J. Bell, and G. Kaiser. A Competitive-Collaborative Approach for Introducing Software Engineering in a CS2 Class. Technical Report cucs-018-12, Dept. of Computer Science, Columbia University, 2012. `http://mice.cs.columbia.edu/getTechreport.php?techreportID=1517&format=pdf&`.

[17] L. Williams and L. Layman. Lab partners: If they're good enough for the natural sciences, why aren't they good enough for us? In *Proc. of the 20th Conf. on Software Engineering Education & Training*, CSEET '07, pages 72–82, 2007.