An Experimental Card Game for Teaching Software Engineering

Alex Baker, Emily Oh Navarro, and André van der Hoek
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697–3425 USA
abaker@uci.edu, emilyo@ics.uci.edu, andre@ics.uci.edu

Abstract

The typical software engineering course consists of lectures in which concepts and theories are conveyed, along with a small "toy" software engineering project which attempts to give students the opportunity to put this knowledge into practice. Although both of these components are essential, neither one provides students with adequate practical knowledge regarding the process of software engineering. Namely, lectures allow only passive learning and projects are so constrained by the time and scope requirements of the academic environment that they cannot be large enough to exhibit many of the phenomena occurring in real-world software engineering processes. To address this problem, we have developed Problems and Programmers, an educational card game that simulates the software engineering process and is designed to teach those process issues that are not sufficiently highlighted by lectures and projects. We describe how the game is designed, the mechanics of its gameplay, and the results of an experiment we conducted involving students playing the game.

1. Introduction

It is now well known that software engineering professionals working in industry are generally unsatisfied with the level of real-world preparedness possessed by recent university graduates entering the workforce [2, 5, 9, 12]. Their frustration is understandable – in order for these graduates to be productive in an industrial setting, organizations that hire them must supplement their university education with extensive on-the-job training and preparation that provides them with the skills and knowledge they lack [5]. The root of the problem seems to lie in the way software engineering is typically taught: theories and concepts are presented in a series of lectures, and students are required to complete a small, toy project in an attempt to put this newfound knowledge into practice. Although both of these components are necessary and useful parts of educating future software engineers, they lack an adequate treatment of many of the critical issues involved in the overall *process* of software engineering. Specifically, the time and scope constraints inherent in an academic setting prohibit the project from being of a sufficient size to exhibit most of the phenomena present in real-world software engineering processes – those that involve large, complex systems, large teams of people, and other factors such as management, workplace issues, and corporate culture. Although the instructor can explain most of these issues in lectures, students do not have an opportunity to participate in an entire, realistic software engineering process firsthand.

To address this problem, we have developed a unique approach to teaching the software engineering process: Problems and Programmers, an educational card game that simulates the software engineering process from requirements specification to product delivery. Problems and Programmers provides students with an overall, high-level, practical experience of the software engineering process in a rapid enough manner to be feasibly used repeatedly in a

limited amount of time (i.e., a quarter or semester). Furthermore, it takes the focus off of actual deliverable artifacts and highlights the overall process by which they are developed.

It is our intention that 1 or 2 class periods in a course would be dedicated to learning and playing the game as a way to supplement the material already learned. Surely, lectures are still needed to teach the fundamental concepts and theories of software engineering, and projects still provide students with useful experience in creating deliverables, but the addition of this game could enrich the curriculum. As an initial evaluation of the game's feasibility and worth as a complementary teaching tool, we recruited a group of students who had passed an introductory software engineering course to play the game, and collected their feedback.

The remainder of this paper is organized as follows: Section 2 outlines the overall objectives of Problems and Programmers, both as a game and as an educational tool. Section 3 details the design and mechanics of the game. Section 4 briefly describes the experiment we performed to evaluate the effectiveness of the game, as well as lessons learned from it. We end in Section 5 with our conclusions and directions for future work.

2. Objectives

Problems and Programmers is a teaching tool, and as such its purpose is to educate. One possible approach to teaching any subject is to create a simulation. In the case of Problems and Programmers, the game simulates the overall software engineering process. This allows students to get a good feel for the process as a whole, but also allows for individual lessons learned about how cards work in the game to be easily translated to real-world lessons.

In general, each event in the game needs to be associated with a corresponding event in the real world. Accomplishing this goal has a twofold benefit. First of all, the connections between the game's rules and lessons learned make the rules more intuitive and easy to remember. Secondly, these associations allow the teachings of the game to be more relevant to the real world, and thus more useful.

When setting up the simulation, it is important that the game reward good software engineering practices and punish persistent deviations from them. After all, if the game were designed such that forgoing all requirements and design specification was consistently successful, the game would either be perceived as unrealistic, or worse yet teach that such a thing was a good idea! Because of this, it is necessary to ensure that the paths to victory in the game represent good software engineering practices.

Clearly, a simulation can take on many forms and must make many tradeoffs between faithfulness to reality, simplicity and fun factors. If the game were to be unrealistic or overly complex, it would lose most of its effectiveness as a teaching tool. Therefore we used the following guidelines in the design of the game:

- ***The game should teach both general and specific lessons about the software engineering process***. General lessons include ideas such as the fact that multiple stakeholders will guide a project's direction, or that software engineering is a non-linear process. Specific lessons include that rushing coding often increases the time it takes or that unclear requirements documents can lead to inappropriate designs. These lessons should be taught through intermediate, as well as "end-of-the-game" feedback in the form of visible consequences [1, 8].
- ***The game should promote proper software engineering practices.*** Misusing resources, cutting corners, or otherwise straying from usual procedures should be, at best, a risky proposition. Unwise actions should be met with negative consequences, with as much visibility as possible as to why the consequences occurred, maximizing the teaching effectiveness of the game [3].

- ***The game should be relatively easy to learn and quick to play.*** One of the game's main strengths is its ability to give a high-level view of the software engineering process in a condensed timeframe. The simulation's value would be significantly reduced if learning and playing the game took too long [7, 10].
- ***The game should be fun.*** While this goal will be secondary to some of those above, it is certainly important that the players would want to play the game. The fun of the game will be a large part of what will make the lessons learned more memorable [7].

These goals can be summarized as: the game should be practical and enjoyable and teach good lessons and good practices. We believe that we have succeeded in meeting these goals, and in creating an innovative and effective teaching tool. Our specific approach to creating Problems and Programmers is detailed below.

3. Overall Design

The game is organized as a competitive game, in which students take on the roles of project leaders in the same company. They are both given the same project and are instructed to complete it as quickly as possible. The player who completes the project first will be the winner. However, players must balance several competing concerns as they work, including their budget and the client's demands regarding the reliability of the produced software.

In completing their project, players play cards based on the waterfall lifecycle model, as shown in Figure 1. While we had experimented with allowing players to choose from alternative lifecycle models, the rules required to do so violate our goal of simple gameplay and had to be forgone. As it stands, the waterfall model is the one that most students will be most familiar with and will still demonstrate nearly all of the principles that we were striving for.
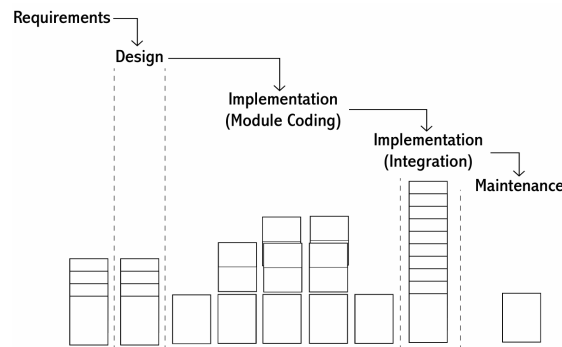

Figure 1: Phases and corresponding play areas in problems and programmers.

While most actions are available at any time, players are encouraged to create their requirements document early on, as working on it later can require reworking of other deliverables. Similarly, other deviations from the lifecycle will have adverse consequences.

As players move through the lifecycle phases, they place cards in areas from left to right, as seen in Figure 1. First, players create a column of requirements cards. Then they play design cards in a column to the right of this, and then have their programmers create code cards during implementation. Finally, all of these code cards are collected into a column of integrated code to the right. In this way, the progress through the phases is indicated in a physical and straightforward manner, and players can easily track their progress.

In the following subsections we will describe the game's play from beginning to end and briefly go over the choices and lessons presented to the players.

3.1 Setup

At the start of the game, a project card is selected (see Figure 2). This gives the attributes of the project that the players will be completing, including its length, complexity, and budget. Then, each player draws five cards from the main deck. Here they will find three types of cards: concepts, programmers and problems. Examples of each of these are also shown in Figure 2. Concept cards represent decisions that a player may make regarding their approach to the project. For example, the Reusable Code concept card allows for a free code card to be added, while a Walkthrough card allows for unclear requirements cards to be re-worked. Programmer cards are the player's workhorses and are necessary to write, inspect and fix code. Finally are problem cards, which are at the heart of Problems and Programmers' gameplay. These are cards that are played by one player against the other. If the receiving player meets the condition on the card, they must suffer the consequences described.
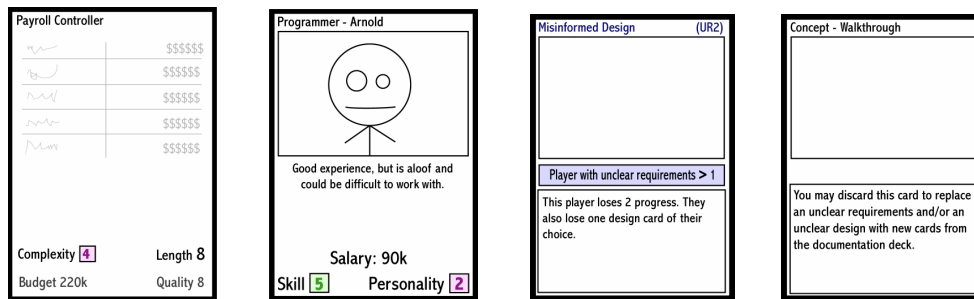

Figure 2: Examples of a project card, programmer card, problem card, and concept card.

3.2 Turn structure

Once each player has their five cards, they begin the game. Each turn players go through the following steps:

1. *Decide whether or not to move to the next phase of the life cycle.*
2. *Draw cards.*
3. *Take actions, as allowed in the respective phase.*
4. *Play any programmer and concept cards.*
5. *Discard any unneeded cards.*

This turn structure keeps cards moving between the decks, into the players' hands and into the play areas. It is also arranged specifically to make the turnover of concepts and programmers difficult. If players are using up their entire budget, for example, they cannot fire programmers to free up money until the end of their turn. At this point they have missed their chance to hire any new programmers until next turn, and those programmers will not be able to act until the turn following that. This represents that in the real world it takes time for programmers to get used to the environment and the project at hand.

The most important step of each turn is the "take actions" phase. The exact sequence of events in this step will depend on the lifecycle phase that the player is in.

3.3 Requirements

Players are encouraged to stay in the requirements phase early on, and to spend this time to play requirements cards. These are placed in front of the player and used to represent work they have spent making their requirements document thorough and complete. In game terms,

the more of these cards the player acquires, the less problem cards they will be vulnerable to. For example, by working on requirements for one turn at the start of the game and acquiring two requirements cards, a player makes themselves immune to any problem cards with the conditions that a player have "less than 2 requirements cards".

While most documentation cards are blank, sometimes players will reveal one that is marked "unclear". Some problem cards will cause problems for players with 1 or more "unclear" requirements cards, their total number of requirements notwithstanding. Players are able to replace these cards, but doing so counts towards their two-card-per-turn limit. This represents to players that there are multiple desirable qualities for the requirements document, but also brings a bit of tactics to the game. Software engineers will sometimes need to spend more time on their requirements than they had planned if things are not going smoothly.

3.4 Design

The design phase is handled in a similar manner, but players instead produce cards that represent the thoroughness of a design document. The procedure of the game is the same as in the requirements to promote learnability. In addition, the same documentation deck is used to keep the play area as uncluttered as possible. As with requirements, players are not forced to spend any time in their design phase. Again however, there are numerous cards that will allow the opponents of reckless players to thwart them.

3.5 Implementation

Once players have decided that they have done enough design, they may move onto their implementation phase. Once they have done so, their programmers are able to take action based on the number of skill points they have. Their options include:

- *Produce Good Code:* Which takes time based on the project's complexity.
- *Produce Rush Code:* Taking half the time of good code.
- *Inspect Code:* For one point, a piece of code can be inspected, and is flipped face-up.
- *Fix Bugs:* For one point a programmer can also work towards fixing one of their bugs. The exact action will depend on the type of bug, which will be discussed below.

By using these actions in different combinations, players are able to use a variety of coding styles. A programmer can methodically produce good code and inspect it, fixing bugs as they are found. Or, a programmer can create a mass of rush code and then inspect it all at the end. However, the rules are set up to encourage strategies with more real-world validity.

One of the primary ways that these types of strategies are encouraged is through the bug system. Each code card that is completed is placed into play above the programmer that created it, with the red "rush" code or the blue "good" code side up as appropriate. Whether this code has bugs or not is hidden on the other side of the card until that code is inspected. When code is inspected, it is flipped over, with its orientation maintained (see Figure 3).
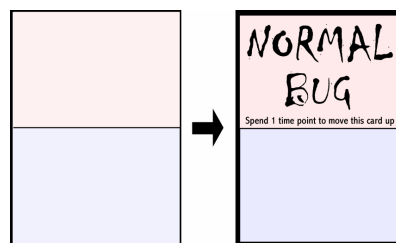

Figure 3: A piece of rush code that is inspected and revealed to have a bug in it.

The "rush" side of the cards is more likely to contain bugs, and these bugs tend to be more severe. There are 3 types of bugs in the game: simple bugs can be replaced with new code fairly quickly. Normal bugs take time to fix, based on how early in the project they are discovered, while nasty bugs can render more recent code obsolete and require it to be redone.

3.6 Implementation (Integration)

Once a player has completed the required number of code cards they can begin to integrate, spending one turn per programmer they had working on the project. Only when the necessary code has been both completed and integrated can the project be considered finished.

3.7 Product delivery

The final phase of a player's turn is product delivery. In this phase, the player shuffles all of his or her code cards and reveals some of them at random. If any are found to have bugs, these bugs must be fixed, and if they are severe enough the game can be lost altogether. Overall, players may be able to get away with submitting slightly faulty code, but usually they will get caught and pay the price.

But, if all of the revealed code cards are-bug free, the customer is satisfied and the game is won! There is some luck at the end, but this not so unrealistic. It is still almost always the more thorough player that wins, and the concepts of the game are certainly still reinforced.

4. Evaluation

4.1 Experiment design

We recruited 28 undergraduate students who had passed the introductory software engineering course at U.C. Irvine. They were matched randomly into groups of 2, received instruction on how to play the game, and then played against each other for approximately 1½ hours, completing 1 to 2 games. Following this, they completed a questionnaire stating their thoughts and feelings about the game in general, their opinions about the pedagogical effectiveness of the game in teaching software engineering process issues, and their educational and professional background in software engineering.

4.2 Experiment results

In general, students' feelings about the game were favorable, as summarized in Table 1. On average, students found the game quite enjoyable to play (4.1 rating out of 5) and relatively easy to play (3.5). They also felt that it was moderately successful in reinforcing software engineering process issues taught in the introductory software engineering course they had taken (3.5) and equally successful in teaching software engineering process issues in general (3.4). For the most part, they agreed that Problems and Programmers would be helpful to teaching software engineering concepts if incorporated into the introductory software engineering course (3.6).

Students' answers to the open-ended questions also reflected their positive feelings about Problems and Programmers. Regarding the enjoyability of the game, some students remarked:
- *"Because this game is fun, I think students will tend to learn more. It's interesting how such a card game can teach one about software engineering concepts."*
- *"[It] makes me think there is hope to make learning fun one day."*

- *"[I like] the various strategies you can employ. I guess this speaks to the depth of the game."*

Regarding how well the game teaches software engineering process issues, students wrote:
- *"Consequences are more drastic than mentioned in class. We could clearly see this in the game."*
- *"It was easy to understand the process because it was a game."*
- *"You need to put the time into earlier phases (design) or else it will come back to get you."*

Table 1: Questionnaire results.

Question	1	2	3	4	5	Avg
How enjoyable is it to play? (1=least enjoyable, 5=most enjoyable)	0	0	6	13	9	**4.1**
How difficult/easy is it to play? (1=most difficult, 5=easiest)	0	3	10	12	3	**3.5**
How well does it reinforce knowledge of SE process taught in class? (1=not at all, 5=definitely)	0	6	9	7	6	**3.5**
How well does it teach new SE process knowledge? (1=not at all, 5=definitely)	7	8	6	3	4	**2.6**
How well does it teach the SE process? (1=not at all, 5=very much so)	1	4	8	12	3	**3.4**
Incorporate it as standard part of SE course? (1=not at all, 5=very much so)	1	6	3	12	6	**3.6**
As an optional part? (1=not at all, 5=very much so)	1	5	4	10	8	**3.7**
As a mandatory part? (1=not at all, 5=very much so)	1	6	8	11	2	**3.3**

Although responses were positive for the most part, it is clear that some aspects of the game need to be improved. For instance, several students felt that the requirements and design phases of the game were boring. Clearly, more breadth needs to be added to this part of the game play, possibly in the form of new types of problems that can be played during these phases. Moreover, many believed that the learning curve for the game was too steep. Perhaps the instruction process can be streamlined or the game made simpler to alleviate this problem. Most importantly, students generally felt that the game was not very successful in teaching new software engineering process knowledge that was not introduced in class. While reinforcing concepts taught in lectures is a useful benefit in and of itself, the tool would be even more valuable if it could also introduce new knowledge. An investigation will be required to determine whether this can be done by incorporating more software engineering process issues into the game (running the risk of adding further difficulty to learning the game), making the existing ones more obvious, or a combination of the two.

5. Conclusions and Future Work

Problems and Programmers represents a first attempt at using a physical card game to teach students about the software engineering process. It addresses many of the weaknesses of more traditional techniques and brings additional benefits in the form of face-to-face learning and enjoyable play. When used in conjunction with lectures and projects, Problems and Programmers allows students to gain a thorough understanding of real-world lessons that might otherwise have been poorly understood or overlooked altogether.

Our card-based approach holds several advantages over existing automated simulations [4, 6, 11]. In comparison, it has a very visual nature, is simple and fun to play, allows for collaborative learning and provides almost immediate feedback to players about the lessons to be

learned. The physical nature was difficult at times and occasionally restricted the lessons we could teach, but we feel that the game represents a good balance between our stated objectives.

The results of our experiments show that students will embrace the use of the game, as most of our test subjects felt that playing the game was both a useful lesson and an enjoyable experience. Additionally, most students felt that it would be a valuable addition to a software engineering course's curriculum. We plan to examine this further by introducing the use of the game into several classroom settings. We will be collaborating with 3 other institutions in this matter, giving us a robust test of the game's applicability in educational settings.

More information about Problems and Programmers, as well as a freely downloadable version of the cards, is available at: http://www.problemsandprogrammers.com.

6. Acknowledgements

8. References

1. Anderson, J.R., et al., *Cognitive Tutors: Lessons Learned.* The Journal of the Learning Sciences, 1995. **4**(2): p. 167-207.
2. Callahan, D. and B. Pedigo, *Educating Experienced IT Professionals by Addressing Industry's Needs.* IEEE Software, 2002. **19**(5): p. 57-62.
3. Chi, M.T.H., et al., *Eliciting Self-Explanations Improves Understanding.* Cognitive Science, 1994. **18**: p. 439-477.
4. Collofello, J.S., *University/Industry Collaboration in Developing a Simulation Based Software Project Management Training Course*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*, S. Mengel and P.J. Knoke, Editors. 2000, IEEE Computer Society. p. 161-168.
5. Conn, R., *Developing Software Engineers at the C-130J Software Factory.* IEEE Software, 2002. **19**(5): p. 25-29.
6. Drappa, A. and J. Ludewig, *Simulation in Software Engineering Training*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 199-208.
7. Ferrari, M., R. Taylor, and K. VanLehn, *Adapting Work Simulations for Schools.* The Journal of Educational Computing Research, 1999. **21**(1): p. 25-53.
8. McKendree, J., *Effective Feedback Content for Tutoring Complex Skills.* Human-Computer Interaction, 1990. **5**: p. 381-413.
9. McMillan, W.W. and S. Rajaprabhakaran, *What Leading Practitioners Say Should Be Emphasized in Students' Software Engineering Projects*, in *Proceedings of the Twelfth Conference on Software Engineering Education and Training*, H. Saiedian, Editor. 1999, IEEE Computer Society. p. 177-185.
10. Randel, J.M., et al., *The Effectiveness of Games for Educational Purposes: A Review of Recent Research.* Simulation and Gaming, 1992. **23**(3): p. 261-276.
11. Sharp, H. and P. Hall, *An Interactive Multimedia Software House Simulation for Postgraduate Software Engineers*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 688-691.
12. Wohlin, C. and B. Regnell, *Achieving Industrial Relevance in Software Engineering Education*, in *Proceedings of the Twelfth Conference on Software Engineering Education and Training*, H. Saiedian, Editor. 1999, IEEE Computer Society. p. 16-25.