An Experimental Card Game for Teaching Software Engineering Processes

Alex Baker, Emily Oh Navarro*, and André van der Hoek
School of Information and Computer Science
Department of Informatics
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 game play, and the results of an experiment we conducted involving students playing the game.

Keywords: Software engineering education; Educational games; Software engineering simulation; Simulation games

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 (Callahan and Pedigo, 2002; Conn, 2002; McMillan and Rajaprabhakaran, 1999; Wohlin and Regnell, 1999). 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 (Conn, 2002). 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

* Corresponding author. Tel: 949-824-3100; Fax: 949-824-1715

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 first-hand.

In recent years, software engineering academics have put much effort into mitigating this problem by devising new ways of teaching software engineering: requiring students to work on projects sponsored by an external organization (Hayes, 2002; Mayr, 1997), intentionally applying real-world complications during the class project, such as changing requirements while the design is in progress (Dawson, 2000), incorporating multiple universities and disciplines into the project (Burnell et al., 2002), maintaining a large-scale, ongoing project that different groups of students work on from semester to semester (Sebern, 2002), and many others. All of these approaches share the same goal: to bridge the disconnect between theory and practice. While certainly an improvement over traditional class projects, the constraints imposed by the academic environment (namely, limited time and a dominating focus on deliverables rather than on the process used to create them) still apply and prevent many of the issues present in real-world software engineering processes from being experienced (although each approach does succeed in highlighting a few of these issues).

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 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.

Aside from these, Problems and Programmers has a number of other qualities that contribute to its learning effectiveness. First, it is competitive: each player takes on the role of project manager, and must complete the project before any of the opponents do. Not only does competition motivate students to play the game, but it also encourages collaborative learning, an educational technique that is know to have significant advantages (Bruffee, 1983). Second, the game is physical, meaning it is played using actual cards and with face-to-face interaction between players. This physical nature further encourages collaborative learning and also ensures that all of the underlying mechanics of the software engineering process being simulated are visible (and therefore more learnable). Lastly, Problems and Programmers has a fun and engaging nature, a quality that is known to be highly conducive to learning (Ferrari et al., 1999). Entertaining character descriptions, humorous character illustrations, and unexpected situations further add to this quality.

It is our intention that one or two 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. Section 5 provides an overview of existing research related to our work, and we end in Section 6 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 virtually any subject is to create a simulation. In the case of Problems and Programmers, the game should simulate the software engineering process, as this is the particular subject that we aim to teach.

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, the connections between the game's rules and lessons learned make the rules more intuitive and easy to remember. Second, these associations allow the teachings of the game to be more relevant to the real world, and thus more useful. For example, code cards are placed into play facedown. Players must take time to "inspect" them before they can flip them face up and reveal whether the code contains any bugs. Because players are able to associate the cards with uninspected, untested code, they are able to understand why the cards are not trivially revealed. After all, anyone who programs knows that bugs take time to find. Further, once players have spent turns to inspect code cards, they will be able to transfer this lesson back to the real world and hopefully not take for granted that code just finished is "complete".

The goal of our simulation extends to individual cards. Once the correspondence to the software process has been built up in a student's mind, individual cards can be used to teach them specific lessons about that process. For example, the Misinformed Design card states that a player with more than one unclear requirement card must lose two of their code cards and one of their design cards. This sounds very abstract. But in the framework of the game it represents that this player did not dedicate enough time to clarifying parts of their requirements document and thus created a design that did not meet the customer's needs. Because of this, part of that design is rendered useless, as is some of the coding that was done using that design as a basis. Hopefully, players having this card played on them will take away from it the importance of a solid requirements specification before proceeding to design.

It is worth noting that the causes of the game's results need to be visible. Even if the player who always follows software engineering practices always wins, players should be able to recognize this as the reason for the victory. Also, as they are playing, students should be able to recognize specific software engineering decisions made in the game as good or bad, receiving immediate feedback that is more immediate than the eventual outcome of the game. Problem cards will provide negative results, giving a reason for the loss of progress immediately, while most concept cards have positive effects on the game when they are used. This ensures that players will understand the reasons for the intermediate and end results of the game and will realize the rights and wrongs of the game and the real world.

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 total time it takes for development, 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 (Anderson et al., 1995; McKendree, 1990).

- ***The game should promote proper software engineering practices.*** It is important that the game reward good software engineering practices and punish persistent deviations from them. 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 (Chi et al., 1994).
- ***The game should be relatively easy to learn and quick to play.*** One of the game's main strengths should be 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 (Ferrari et al., 1999; Randel et al., 1992).
- ***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 (Ferrari et al., 1999). After all, it is more fun to see your opponent's programmer "get fired for slacking off" than it is to think of such a situation as "a programmer card being discarded for having a low personality score". The greater degree to which the players feel that they are leading a project, rather than playing cards, the more fun they will have.

These goals can be summarized as: the game should be practical, enjoyable, and teach good lessons and good practices. We believe that we have succeeded in meeting these goals, in the process 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 essence, they must strive to follow proper software engineering practices in order to avoid any adverse consequences that might cause them to fall behind their opponent in the race to complete the project. What are considered proper and improper software engineering procedures is based upon a compendium of 85 "rules of software engineering" that we have collected by surveying software engineering literature (Abdel-Hamid and Madnick, 1991; Cook and Wolf, 1998; Dawson, 2000) and practitioners' experience reports (Brooks, 1995; Davis, 1995; Glass, 2003). These rules represent a mixture of both academic and industrial "best practices", and have been gathered with the intention of teaching important academic lessons while remaining faithful to reality. A full description of these rules is outside the scope of this paper, but is provided elsewhere at: http://www.ics.uci.edu/~emilyo/SimSE/se_rules.html.

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 game play and had to be forgone in favor of making an initial, baseline version of the game to be tested for its educational effectiveness. As it stands, the waterfall model is the one that students will be most familiar with and will still demonstrate nearly all of the principles that we were striving for. Furthermore, the waterfall model reflected in the game is not simply a strict linear one –

players are allowed to return to earlier development phases (with a penalty), as well as skip phases, if so desired. We are currently addressing the issue of additional lifecycles in a redesign of the game, as explained further in Section 6.

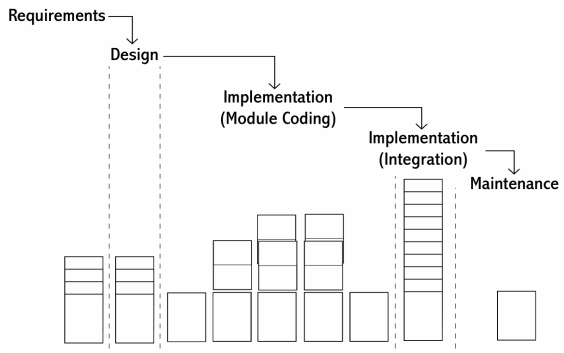

Figure 1: Phases and corresponding play areas in problems and programmers.

While most actions are available at any time, players are encouraged by the rules to follow the traditional waterfall model when they can. For example, even at the very start of the game, a player is able to obtain requirements cards, design cards or code cards. However, creating requirement cards causes any existing design document to be rendered partially obsolete, and some design cards are lost. In this way, players will learn that they should finish up at least a significant part of their requirements document before working on design, lest they be forced to rework their design.

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.

While the game is designed to encourage players to follow the traditional lifecycle model, they are still given a large amount of freedom in how to progress. Different players are allowed to take different approaches to the challenge of completing a software engineering project. Some players will carefully build up a thorough requirements document, work long and hard on design, and carefully code and inspect each module before integrating them. Others will rush their requirements, design, and coding in an effort to get the project in as quickly as possible. Both approaches have their disadvantages. The former player may find their opponent outpacing them, while the latter will potentially be thwarted by disastrous problem cards as a result of their haste. The benefit of a competitive game is that players will be able to observe the differences between their own strategies and those of their opponents. The results of these differences will be a powerful example of the consequences of actions taken in a software engineering setting.

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. This gives the attributes of the project that the players will be completing, including its length, complexity, and budget (see Figure 2 for an example). The project's complexity represents how difficult writing the project's code

is, and this determines how much skill programmers need to obtain code cards. The project's length meanwhile determines how much code is needed to complete the project, while the quality requirement determines how much of this code needs to be checked for bugs at the project's completion. Finally, the budget of the project restricts how many programmers and concepts a player can have, and will force them to make some decisions about what they really need to complete their project. Each project's set of attributes will require a slightly different approach, and will create varying scenarios over multiple game sessions.

Once a project card has been chosen and players have had a moment to think about how they will approach the project, each of them 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 shown in Figure 2. Concept cards represent decisions that a player may make regarding their approach to the project. For example, the Walkthrough concept card allows for unclear requirements cards to be reworked, while a Reusable Code concept card allows for a free code card to be added. Programmer cards are the player's workhorses and are necessary to write, inspect and fix code. They have a skill level that determines the amount of work they are able to do in a turn, as well as a personality that determines how well they follow software engineering practices, how well they work with others, and just how friendly they are. A player must weigh these factors against the salary of the programmer and the constraints of their budget when deciding which programmers to hire.

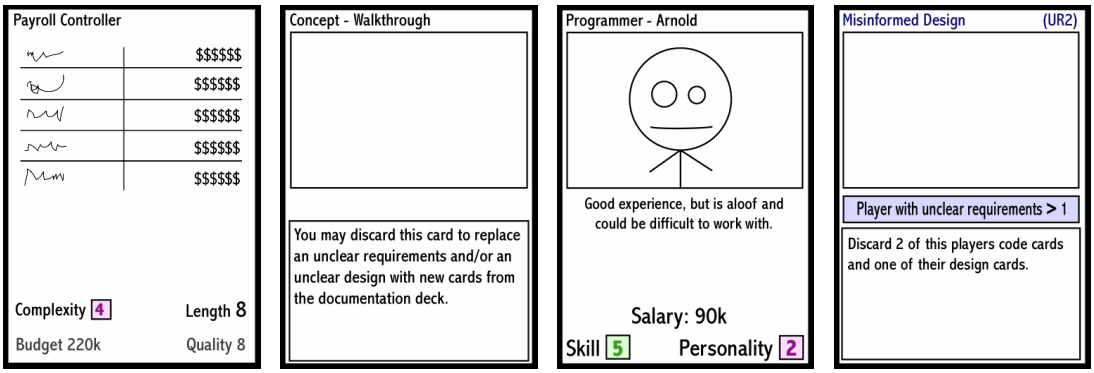

Figure 2: Examples of a project card, concept card, programmer card, and problem card.

Finally are problem cards, which are at the heart of Problems and Programmers' game play. 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 of the card. For example, the Misinformed Design card, mentioned in Section 2, can be played on a player with unclear requirements and causes them to lose both a design card and two code cards. Many other problem cards exist that highlight all kinds of problems such as not dedicating enough time to requirements specification or hiring irresponsible programmers.

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 from 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 the 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" step. The exact sequence of events in this step will depend on the lifecycle phase that the player is in. It is in this step that work actually gets done, and where the flow of the game is shaped into the lifecycle model. The following is a detailed description of each of the actions that a player can take in this step, in the order they are normally taken in the waterfall lifecycle model.

3.3 Requirements

Players are encouraged to stay in the requirements phase early on, and to spend this time to play requirements cards. They are not required to spend even a single turn working on their requirements document, but new players will soon realize that spending time on such a document is necessary in the game and in reality alike. During this phase players may draw two cards from the documentation deck per turn. 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 two 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 one 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.

Players must also avoid spending too long on requirements, as they have time constraints to consider. As players pass about two turns-worth of requirements, they begin to see diminishing returns on the number of problem cards they are protecting themselves from (per turn). Eventually, the difference between having five and six clear requirements is nearly insignificant. So players need to balance their needs for completeness and safety against their need for speed.

3.4 Design

The design phase is handled in a similar manner as the requirements phase, 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 require-

ments, 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. These problems have different names and effects than requirements problems and instead teach lessons about improper design and its pitfalls.

3.5 Implementation

Once players have decided that they have done enough design, they may move onto their implementation phase. In this phase, 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 or not this code has bugs 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).

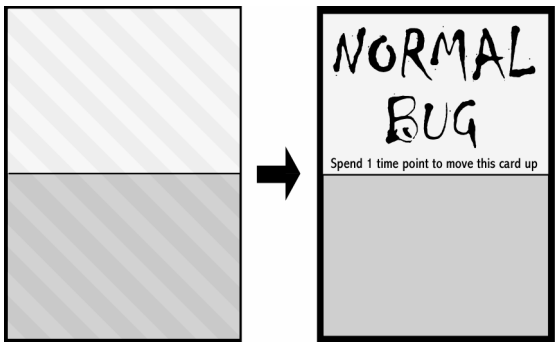

Figure 3: A piece of rush code that is inspected and revealed to have a bug in it.

While some cards have bugs on both ends of the card, and some are bug-free on both sides, many will only have bugs on their rush code side. Thus a programmer who creates rush code is much more likely to have bugs in their software.

In addition, rush code tends to have more severe bugs. There are three types of bugs in the game: *Simple bugs* can be replaced with new code cards for one point. *Normal bugs* must be "moved up": Each skill point spent on a normal bug swaps it with the code card above it, and once the bug is at the top of the code column, it can be replaced with new code. Thus, finding and resolving normal bugs early is emphasized, and therefore so are frequent code inspections in general. Likewise enforcing this idea are *nasty bugs*. When they are revealed they immediately cause the code card above them to be discarded. This enforces the idea that code does

not work in isolation, and that if code is written badly enough, code that relies on it may need to be redone.

3.6 Integration

After a player has worked to create code cards they can begin to integrate their code. Each turn of integration allows for all of *one* programmer's code to be integrated and set aside. Only when the necessary code has been both completed and integrated can the project be considered finished. This means that projects that have had numerous programmers work on them throughout their lifecycle will take longer to integrate. This emphasizes the difficulties in integrating code from a large number of programmers and demonstrates the benefits of using a small, skillful team of employees when possible.

3.7 Product delivery

The final phase of a player's turn is product delivery. It is important to note that, while implementing, the code created need not have its bugs fixed nor even be inspected if the player does not desire it. But if bugs are discovered when the product is delivered, the player may feel their customer's wrath. In this phase, the player shuffles all of his or her code cards and reveals a number of them equal to the project's quality requirement. If any are found to have bugs, these bugs must be fixed, and if they are severe enough the game can be lost altogether.

If only normal and/or simple bugs are found, the code must be reworked and reintegrated, which will cost the player precious time. If a nasty bug is found however, this player has lost the game. Nasty bugs represent bugs that create problems so severe that no product that is released with them could be acceptable. Thus players will learn that some risks are acceptable, but that software engineering is serious business, and throwing code blindly at a client will often end in disaster.

But, if all of the revealed code cards are-bug free, the customer is satisfied and the game is won! In summary, the final stage of the game does have a luck factor involved, and on projects that have a more lenient quality requirement it may be possible to get away with a bug or two. However, the odds of this are not good even on the most lax projects and the fact that a single nasty bug spells disaster makes this risk an unattractive option. In the end, it is almost always going to be the appropriately cautious player who succeeds, and the most efficient one who wins. Players may take several games to come to the ideal combination of caution and speed, but during this time they will learn a great deal about software engineering.

3.8 Game Play Example

To illustrate what playing Problems and Programmers is actually like, we will show a brief example scenario, demonstrating several of the causes and effects that one can expect to encounter.

In the example shown in Figure 4, the player has already worked to create two requirements cards, as demonstrated by the two cards in the column on the left. To the right of that, the player has created five design cards, representing a large amount of work on the design document. Notice that none of this documentation is marked as "unclear", so the player has ensured the quality of the work they have done. The player has also hired Maria, a programmer with a good personality and a small amount of programming knowledge. They have also hired Carl, who is a strong programmer but with a very bad attitude, as demonstrated by his personality score of "1". These programmers have completed some code, though Carl completed some rush code that, upon inspection, ended up having a bug.

At the beginning of the turn, this player's opponent has an opportunity to play a problem card on them. In this case, their opponent has chosen to play the Better Job card on this player, specifically on Carl (see Figure 5). This card only works on programmers with a personality of three or less, and because Carl has a low personality score, he is vulnerable to this problem. As a result, the "Carl" card is discarded. His code, however, remains. Anyone who wishes to try to fix the bug he has left behind will have to pay a "help penalty" (skill points of the "helper" are reduced by two for that turn) to work on his code. In addition, his code still represents another block of code that will have to be integrated into the final project.

Now that the problem phase has been weathered, this player is able to take normal actions. In this case, they have chosen to implement code. This means that Maria has two time points to spend (based on the skill level of 2 listed on her card), and the player decides to use them to have her create two rush code cards (see Figure 6). These cards are placed at the top of her column of code, representing that they are her most recently completed pieces of code. They are placed face down, indicating that they have not been inspected. A programmer's time points will need to be used to inspect them and find out if they have any bugs. (However, that will have to wait until next turn, since for now Maria is the only programmer this player has, and she has used all of her time points.)

After they have taken action (implemented code in this case), the player is able to play any programmer or concept cards. In this case, they have chosen to play Rick, a more reliable programmer that will hopefully help them to complete code more quickly (he is also seen in the right side of Figure 6). The player can then discard cards, ending their turn and marking the beginning of their opponent's turn. However, these steps are ignored here, for the sake of space, and we advance to the beginning of the same player's next turn.

At the start of this player's second turn, their opponent plays another problem card on them, as shown in Figure 7. In this case, the card is an Incorrect Assumptions card. The metaphor for this card is that the targeted player did not make a thorough enough requirements document, and as a result they made some incorrect assumptions about what needed to be done when coding. Thus, some of their work will have to be redone. In game terms, this problem card states that any player with less than three requirements cards is vulnerable to this problem, and loses two code cards as a result. In this case, this player's opponent chooses those two pieces of inspected, good code that Maria had completed, and discards them. Obviously this is quite a setback, and just one of many potential problems that could be encountered as a result of putting too little work into one's requirements document.

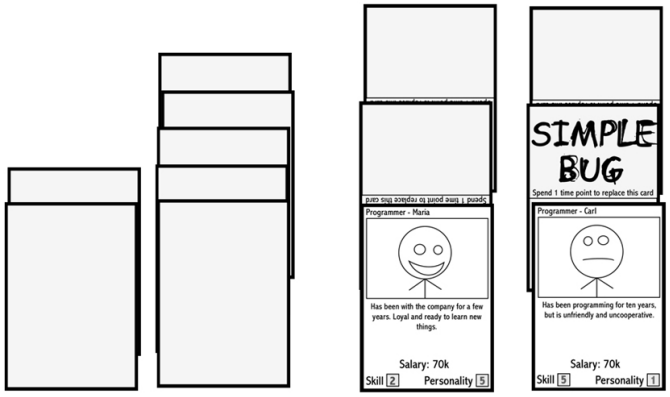

Figure 4: Our game play example's starting position.

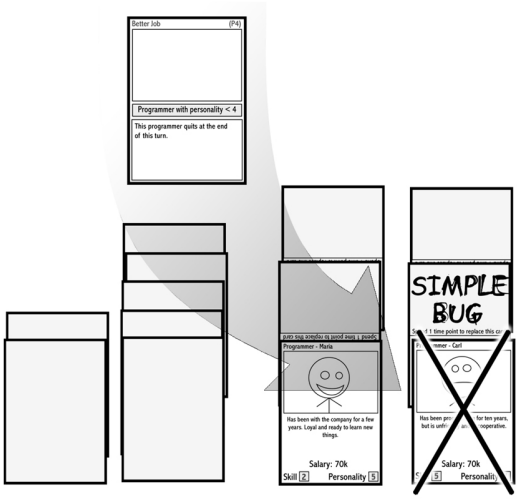

Figure 5: Carl must quit.

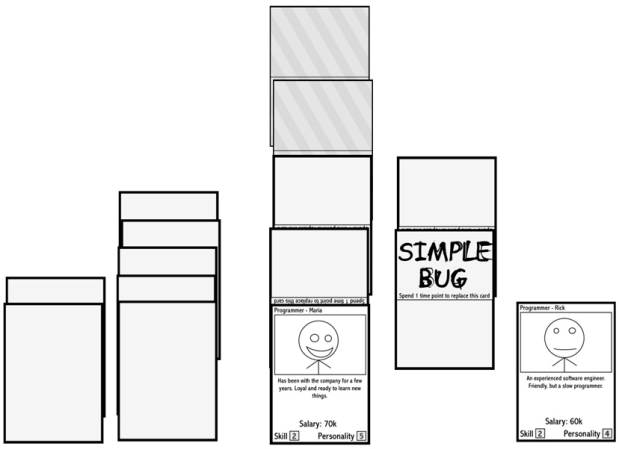

Figure 6: Coding and hiring.

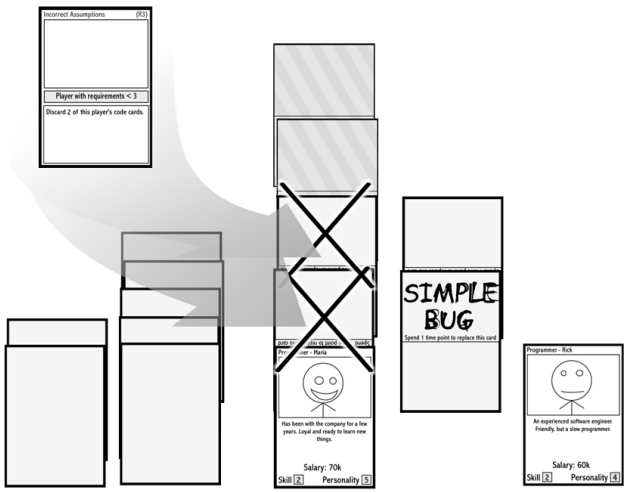

Figure 7: Incorrect Assumptions lead to the loss of code cards.

3.9 Discussion of Game Play Example

In this brief example, we have demonstrated some of the consequences a player may face for their actions, as well as hinted at some of the ways that the software process is simulated. If we evaluate this example in terms of our previously stated objectives for the game, it is apparent that a proper balance between all of them is achieved: Both general lessons (the common flow of the waterfall model; the notion that later development artifacts depend on the quality of previous ones; the importance of hiring quality programmers) and specific lessons (not doing enough requirements may lead to incorrect assumptions in the code, which must then be redone; rushing the coding process generally results in more buggy code) about the software engineering process are taught. Proper software engineering practices are promoted, both through rewarding such practices (when our sample player spent extra time to complete high quality code, this code was revealed to be bug-free upon inspection, unlike some of the rush code that was created) and punishing deviations from proper practices (not spending enough time on requirements later caused the player to lose some code). The game follows a well-defined turn structure throughout game play, making it easy to learn, and can simulate an entire software engineering process in a relatively short period of time. Lastly, the strong correspondence between all of the events in the game and real-world software engineering situations, the unique characters, and unexpected situations makes the player feel like they are leading a project, rather than playing cards, creating a fun and engaging experience. It should be noted that in this example we have only shown a few turns – in an actual game, the player encounters many more situations, and also sees the actions their opponent takes, in the process learning numerous lessons about software engineering.

4. Evaluation

4.1 Experiment design

In order to perform an initial evaluation of the game, we designed a simple experiment in which students were taught to play the game and then asked to submit written feedback in the form of answers to structured questions. While more subjective than some other evaluation methods, we feel that this was well suited to an initial evaluation of the concept, and allowed us flexibility in the information that we gathered. In the future, we plan to utilize more formal approaches – namely, performing comparative studies in actual software engineering courses between the aptitudes of students who played the game and those who did not.

For this experiment, we recruited 28 undergraduate students who had passed the introductory software engineering course at U.C. Irvine. They were matched randomly into groups of two, received instruction on how to play the game, and then played against each other for approximately one-and-a-half hours, completing one to two 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. Some of these questions asked for a numerical answer on a one to five scale, while others allowed students to write out their responses in free form.

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 soft-

ware 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).

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**

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."*

- *"I think I learned that having many programmers = more time in integrations."*

- *"Tells me why it is important to create <u>quality</u> code."*

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. (The redesign of the game that we are currently working on (see Section 6) directly

addresses this issue.) 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. Finally, students generally felt that the game was not as successful in teaching *new* software engineering process knowledge that was not introduced in class (2.6). However, in the free-form question section of the questionnaire, many of these students gave answers that suggested they did learn at least *some* new concepts. While reinforcing concepts taught in lectures is a useful benefit in and of itself, the game 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.

Further analysis of the data revealed some interesting correlations between the different backgrounds of the students and their opinions of the game. The first significant difference was between those students who had *industrial* software engineering experience and those who did not. In particular, those students who had at least one year of software engineering experience in an industrial setting seemed to generally have a more favorable opinion of the game than those with no outside experience. In particular, they found the game more enjoyable, easier to learn, more strongly believed that it both reinforced and taught new process knowledge, and were more in favor of incorporating it as a part of software engineering courses. This suggests that the game is teaching a valid body of knowledge that closely reflects phenomena that occur in real-world software processes (such as those that the experienced students have participated in). However, the fact that those without industrial experience seem to have less of an appreciation of the game is disappointing, since these students have an even greater need to learn what the game teaches. We plan to investigate this further, to try to determine how to make the game equally, if not more effective for students without industrial experience.

The second significant difference was between students who had *educational* software engineering experience beyond the introductory course and those who did not. Particularly, the students who had only taken one software engineering course (the introductory one) appeared to have learned more *new* software process knowledge from the game than those who had taken two or more software engineering courses. What this suggests is that Problems and Programmers should be introduced early in a software engineering curriculum, most likely into an introductory software engineering course, and that doing so will help to teach concepts that normally would not be learned until later in the student's education.

5. Related Work

This research draws from three main areas of related work: simulation in education, software engineering education, and software process simulation. The following subsections describe the relationships between Problems and Programmers and these existing bodies of work.

5.1 Simulation in education

Simulation is a powerful educational tool that is widely used in a number of different domains such as flight simulation (Rolfe, 1988), military training (Lindheim and Swartout, 2001), and hardware design (Skrien, 2001). The success of simulation in education can be attributed to its unique qualities that set it apart from other pedagogical approaches: First, simulation allows students to gain valuable hands-on experience of the process being simulated without any of the potential monetary costs or harmful effects that can result from actual

real-world experience. As a result, students are free to repeat experiences and experiment with different approaches, their only concern being the *simulated* consequences (rather than real life ones) in case of failure. Second, the relative ease with which simulations are configurable allows the educator to introduce a wide variety of unknown situations for the student to experience. Finally, because simulations operate at a faster pace than real life, students can practice the process many more times than would be feasible in the real world.

5.2 Software engineering education

As mentioned previously, software engineering educators have invented several new and innovative ways of teaching the subject in recent years. Some of these approaches have included software process simulation designed specifically for education (Collofello, 2000; Drappa and Ludewig, 2000; Sharp and Hall, 2000). To date, the most advanced of these is SESAM, a software engineering simulation environment in which students manage a team of virtual employees to complete a virtual project on schedule, within budget, and at or above the required level of quality. The student drives the simulation by typing in textual commands which can consist of hiring and firing employees, assigning them tasks, and asking them about their progress and the state of the project (Drappa and Ludewig, 2000). Although Problems and Programmers and SESAM both share the purpose of teaching students the software engineering process in a game-based setting, SESAM, unlike Problems and Programmers, lacks a visually interesting graphical user interface, which is considered essential to any successful educational simulation (Ferrari et al., 1999). Moreover, the software engineering process models developed for SESAM so far are based on a limited number of rules of interaction, and omit many of the more non-technical, workplace-related phenomena, such as personnel problems and issues involving management. As mentioned previously, the physical nature of Problems and Programmers makes the mechanics of the process being taught visible, and therefore more learnable to the players than in a computer simulation such as SESAM. Finally, the fact that Problems and Programmers is competitive and played in the physical presence of an opponent fosters collaborative learning as well as independent learning. Despite these drawbacks, SESAM's models do provide a source of some well-documented software engineering rules of behavior that we have incorporated into Problems and Programmers, and we believe that classroom use of both Problems and Programmers and a computer simulation like SESAM together would be an especially useful approach.

5.3 Software process simulation

The area of software process simulation deals with creating models of software processes in order to analyze and predict certain properties and behaviors of these processes (Abdel-Hamid and Madnick, 1991; Kusumoto et al., 1997; Raffo et al., 1999a; Raffo et al., 1999b). For example, process simulations have been created to predict the effects of different managerial decisions on the resulting project attributes, such as cycle time, cost, and quality (Rus et al., 1999). In another case, process simulations have been used to calculate how many inspections typically need to be done to keep the defect rate under a certain percentage (Madachy, 1996). Because Problems and Programmers essentially simulates a software engineering process, the work done in this area provides useful process models upon which we have based the game and can base future versions of the game. However, since these models were created for the purpose of analysis and discovery rather than education, certain aspects and portions of them are not relevant to the pedagogical goals of the game.

6. 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. We believe that when used in conjunction with lectures and projects, Problems and Programmers will allow 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 (Collofello, 2000; Drappa and Ludewig, 2000; Sharp and Hall, 2000). 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 of the game 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 three other institutions in this matter, giving us a robust test of the game's applicability in educational settings.

Based on early feedback and lessons learned from testing the game, a new version is under development which will focus on customers' requirements for a project. While playing this version, players will need to use various game mechanics to find all of the requirements cards in a "client deck", trying to ensure that no requirements are missed. These requirements cards are then moved into a requirements document stack, organized into modules, integrated and tested. In this way, rather than focusing on creating cards representing work done, a player moves cards representing the customer's requirements from phase to phase. In the end, players will create a stack of these cards that represent "the things the project does", and will aim to ensure that all of a customer's requirements cards were included. This approach will require less card types, will be less difficult to learn, and will hopefully teach some valuable complementary lessons to the current version.

Finally, we are currently developing an automated version of the game. While this approach will cost us of some of the mentioned advantages of a physical representation, it will also provide some advantages of its own. A computerized version will allow us to expand on the lessons in this initial version and simulate different process models, will allow for easier distribution of the game, and will be useful as an introduction to the game to be used in conjunction with the physical version. Finally, this automated version may allow us to gather statistics about student's choices during game play and the results of these choices. This information could then be used to give us insight into the accuracy of the simulation and how well the game teaches its lessons.

7. Availability

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

8. Acknowledgements

We thank the other members of our research group for their invaluable suggestions regarding the design and implementation of Problems and Programmers.

Effort funded by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation.

9. References

Abdel-Hamid, T. and S.E. Madnick, 1991. Software Project Dynamics: an Integrated Approach, Prentice-Hall, Inc., Upper Saddle River, NJ.

Anderson, J.R., et al., 1995. Cognitive Tutors: Lessons Learned, The Journal of the Learning Sciences. 4 (2), 167-207.

Brooks, F.P., 1995. The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley,

Bruffee, K.A., 1983. Collaborative Learning: Higher Education, Interdependence, and the Authority of Knowledge, John Hopkins University Press,

Burnell, L.J., J.W. Priest, and J.R. Durrett, 2002. Teaching Distributed Multidisciplinary Software Development, IEEE Software. 19 (5), 86-93.

Callahan, D. and B. Pedigo, 2002. Educating Experienced IT Professionals by Addressing Industry's Needs, IEEE Software. 19 (5), 57-62.

Chi, M.T.H., et al., 1994. Eliciting Self-Explanations Improves Understanding, Cognitive Science. 18, 439-477.

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.

Conn, R., 2002. Developing Software Engineers at the C-130J Software Factory, IEEE Software. 19 (5), 25-29.

Cook, J.E. and A.L. Wolf, 1998. Discovering Process Models of Software Processes from Event-Based Data, ACM Transactions on Software Engineering and Methodology. 7 (3), 215-249.

Davis, A.M., 1995. 201 Principles of Software Development, McGraw-Hill,

Dawson, R., *Twenty Dirty Tricks to Train Software Engineers*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 209-218.

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.

Ferrari, M., R. Taylor, and K. VanLehn, 1999. Adapting Work Simulations for Schools, The Journal of Educational Computing Research. 21 (1), 25-53.

Glass, R.L., 2003. Facts and Fallacies of Software Engineering, Addison-Wesley,

Hayes, J.H., *Energizing Software Engineering Education through Real-World Projects as Experimental Studies*, in *Proceedings of the 15th Conference on Software Engineering Education and Training*. 2002, IEEE. p. 192-206.

Kusumoto, S., et al., *A New Software Project Simulator Based on Generalized Stochastic Petri-net*, in *Proceedings of the 1997 International Conference on Software Engineering*. 1997, ACM. p. 293-302.

Lindheim, R. and W. Swartout, 2001. Forging a New Simulation Technology at the ICT, IEEE Computer. 34 (1), 72-79.

Madachy, R., *System Dynamics Modeling of an Inspection-Based Process*, in *Proceedings of the Eighteenth International Conference on Software Engineering*. 1996, IEEE Computer Society.

Mayr, H., *Teaching Software Engineering by Means of a "Virtual Enterprise"*, in *Proceedings of the 10th Conference on Software Engineering*. 1997, IEEE Computer Society.

McKendree, J., 1990. Effective Feedback Content for Tutoring Complex Skills, Human-Computer Interaction. 5, 381-413.

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.

Raffo, D., J. Settle, and W. Harrison, *Estimating the Financial Benefit and Risk Associated with Process Changes*, in *Proceedings of the First Workshop on Economics-Driven Software Engineering Research*. 1999a.

Raffo, D., J.V. Vandeville, and R.H. Martin, 1999b. Software Process Simulation to Achieve Higher CMM Levels, The Journal of Systems and Software. 46 (2-3), 163-172.

Randel, J.M., et al., 1992. The Effectiveness of Games for Educational Purposes: A Review of Recent Research, Simulation and Gaming. 23 (3), 261-276.

Rolfe, J.M., 1988. Flight Simulation (Cambridge Aerospace Series), Cambridge University Press,

Rus, I., J.S. Collofello, and P. Lakey, 1999. Software Process Simulation for Reliability Management, The Journal of Systems and Software. 46 (3), 178-182.

Sebern, M.J., *The Software Development Laboratory: Incorporating Industrial Practice in an Academic Environment*, in *Proceedings of the 15th Conference on Software Engineering and Training*. 2002, IEEE. p. 118-127.

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.

Skrien, D., 2001. CPU Sim 3.1: A Tool for Simulating Computer Architectures for Computer Organization Classes, ACM Journal of Educational Resources in Computing. 1 (4), 46-59.

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.

Alex Baker is currently working towards a Ph.D. in Information and Computer Science at the University of California, Irvine. He received his B.S. in Information and Computer Science from the University of California, Irvine in 2002. He is the primary designer of Problems and Programmers, and continues to develop card and board games for teaching software engineering. His research interests include software design techniques, software engineering education and application of games to software engineering.

Emily Oh Navarro received her B.S. in Biological Sciences and her M.S. in Information and Computer Science from the University of California, Irvine in 1998 and 2003, respectively. She is currently pursuing a Ph.D. in computer science at the University of California, Irvine, focusing on developing game-based simulation tools for software engineering education. She is the lead developer on the SimSE project and has also contributed to the design and evaluation of Problems and Programmers. She has been a GAANN fellowship recipient for the past 2 years and is also an ARCS scholar.

André van der Hoek André van der Hoek is an assistant professor in the Informatics Department of the School of Information and Computer Science, and a faculty member of the Institute for Software Research, both at the University of California, Irvine. He holds a joint B.S. and M.S. degree in Business-Oriented Computer Science from the Erasmus University Rotterdam, the Netherlands and a Ph.D. degree in Computer Science from the University of Colorado at Boulder. His research interests include configuration management, software architecture, product line architectures, configurable distributed systems, and software engineering education. He has developed several CM systems (including NUCM, SRM, Palantír, TWICS, and MCCM), was chair of SCM-10, and is co-author of the Configuration Management Impact report. He is on the program committee for ICSE 2004, ICSE 2005, and FSE-12, and has contributed to the development of Problems and Programmers.