Teaching Software Engineering Using Simulation Games

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

ABSTRACT

A typical software engineering course fails to teach its students many of the skills needed in software development organizations. Because lectures and class projects alone cannot adequately teach about the software process, we have developed a pair of games in which the process is simulated, giving students an opportunity to practice it firsthand. Problems and Programmers is an educational software engineering card game and SimSE is an educational computer simulation of the software process.

Keywords: software engineering education, educational games, software engineering simulation, simulation games

INTRODUCTION

A large difference exists between the software engineering skills taught at a typical university and the skills that are desired of a software engineer by a typical software development organization [3, 5-7]. This problem seems to stem from the way software engineering is typically introduced to students: general theory is presented in lectures and put into (limited) practice in an associated class project. Although both lectures and projects are essential, they lack a practical, in-depth treatment of the overall *process* of software engineering. In particular, lectures allow only passive learning, and the size and scope of class projects are too constrained by the academic setting to exhibit many of the fundamental characteristics of real-world software engineering processes.

To address this problem, we have been in the process of researching, designing, building, and experimenting with two game-based simulation tools for teaching software engineering: Problems and Programmers, a physical card game that simulates a software engineering process; and SimSE, a computer-based environment that allows the creation and simulation of software engineering processes. Both allow students to "virtually" participate in a realistic software engineering process that involves real-world components not present in class projects, such as teams of people, large-sized projects, critical decision-making, personnel issues, multiple stakeholders, budgets, planning, and random, unexpected events. Moreover, the rapid and flexible nature of simulation allows experiences to be repeated, different situations to be introduced and practiced, and promotes a general freedom of experimentation and "play" in the training exercise. The remainder of this paper further details these two educational simulation tools.

PROBLEMS AND PROGRAMMERS

Problems and Programmers is organized as a turn-based competitive game, in which two players take on the roles of project leaders in the same company. They are both given the same project and aim to complete it as quickly as possible. The player who completes the project first is 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. There are several possible approaches to the game's challenges, and different players will apply different strategies as they see fit.

A distinct advantage of the competitive nature of Problems and Programmers is the fact that it encourages interaction. Because different players will follow different strategies, more than one approach will be demonstrated in a game. This allows players to not only evaluate their own strategy, but also to compare and discuss strategies followed by others. As a result, players learn from each other, which enhances the educational value of Problems and Programmers.

As shown in Figure 1, Problems and Programmers follows the waterfall lifecycle model. While we experimented with allowing players to choose a lifecycle model, the specific rules required were shown to add more complexity than value to the game. As it stands, the waterfall model is the one that students will be most familiar with and still demonstrates nearly all of the principles that we were striving for.

Each turn, players choose a single phase and are then able to work on their project's requirements, design, implementation or integration accordingly. While all of these options are available throughout the game, players will quickly learn that the rules of the game are set up to encourage following the waterfall lifecycle model as closely as possible. For example, choosing the requirements phase late in the game will cause a player to lose some of their design progress. This represents the fact that as requirements change, a program's design will need to be updated or reworked.

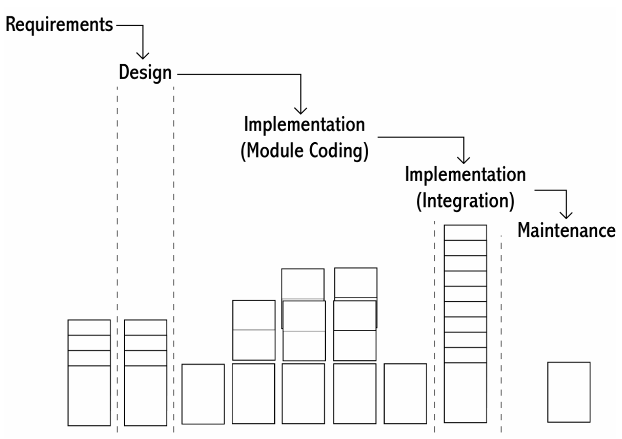


Figure 1. Different phases in Problems and Programmers.

Game Play Summary

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

Players start by drawing five cards from the game's 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 Reusable Code concept card allows for a free code card to be added, while a Walkthrough card allows for unclear requirements cards to be reworked.

Programmer cards, meanwhile, 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. These factors must be weighed, along with the programmer's salary and subsequent budget impact, when deciding which programmers to hire.

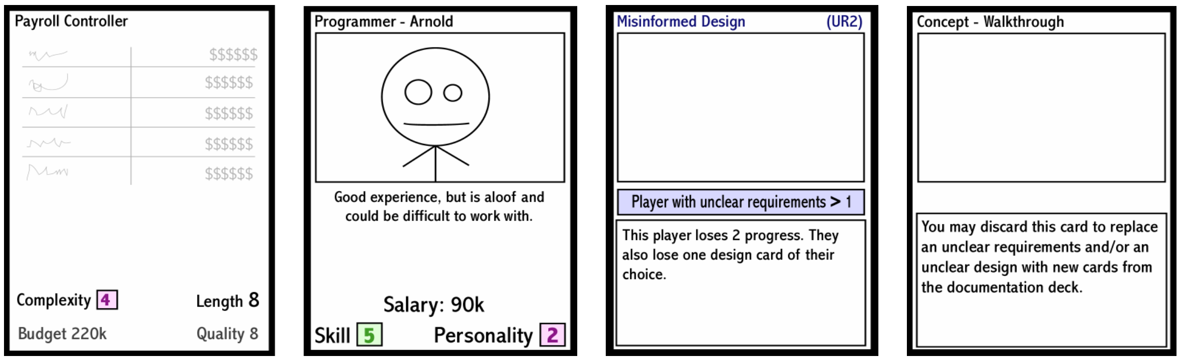
Finally are problem cards, which are at the heart of Problems and Programmers' game play. These are cards that are played by one player against the other. If the receiving player meets the condition on the card, they must suffer the consequences described. For instance, the Misinformed Design card can be played on a player with at least one unclear requirement 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. Through such situations, players are able to get specific lessons with each card and will learn not only what software engineering pitfalls to avoid, but also the specific consequences of failing to do so.

Once players have had a chance to look over their cards, a project card is chosen, which represents the project that both players will be striving to complete. A project card specifies a project in terms of its complexity, length, quality, and budget (see the Payroll Controller system project card example in Figure 2). The complexity and length are parameters defining the nature of the application to be developed. Complexity influences the amount of progress each player can make during the implementation phase (the higher the complexity, the less code each programmer can produce per turn) and length defines the size of the application to be developed as the number of code cards that must eventually be integrated to form the application.

The quality and budget represent other stakeholders in the process. Quality represents the customer's demands and defines the number of code cards that will be inspected for bugs. The player that is first in completing the integration process and succeeding in having all inspected code cards be bug free, wins the game. Finally, budget represents the development organization and constrains the project in how many programmers can be hired.

Together with the overall need to be the first to complete a project in order to win the game, the four parameters of a project capture the typical tension that exists between time (need to win the game), quality (must pass the acceptance test inspections), and money (cannot exceed the budget). The game, thus, illustrates two more important lessons regarding the software process: it always involves multiple stakeholders and must balance multiple, conflicting goals at the same time.

Once a project has been chosen, the game can begin. On



Good experience, but is aloof and could be difficult to work with.

This player loses 2 progress. They also lose one design card of their choice.

You may discard this card to replace an unclear requirements and/or an unclear design with new cards from the documentation deck.

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

each of their turns, a player goes through the following steps:

1. *Allow your opponent to play a problem card on you, if they have one that you are vulnerable to.*
2. *Draw cards.*
3. *Choose one phase and take action as appropriate.*
4. *Play any programmer and concept cards.*
5. *Discard any unneeded cards.*

This turn structure keeps cards moving from the deck, to players' hands and into play and the discard pile as the players choose. 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 the real-world situation in that it takes time for programmers to get used to the environment and the program at hand.

The most important step of each turn is the third "take actions" step. The exact sequence of events in this phase will depend on the lifecycle phase that the player chooses. It is in this phase that work actually gets done, and where students will make decisions about how to approach their software engineering project. We will now discuss the different actions that may be taken by a player when they choose each of the possible software lifecycle phases.

In completing their project, players play cards based on the waterfall lifecycle model, playing cards in areas from left to right as they move through the lifecycle phases. Most players will start by choosing the requirements phase, allowing them to work on their project's requirements. This means that they are able to create a column of requirements cards, each card representing a unit of work they have spent making their requirements document thorough and complete. When working on design, players place cards in a column to the right of this, in a similar manner to the requirements cards.

Having more cards in these columns represents more thorough documentation, but players must also be concerned about the clarity of their work. Both requirements cards and design cards are drawn at random from a "documentation" deck. Some documentation is marked "unclear", in which case the player must spend extra time to replace these cards or risk being vulnerable to certain problem cards. A variety of problem cards exist that only affect players who have less than a certain number of requirements or design cards, or only affect players who have too many unclear requirements or design cards. Thus, players will learn that spending more time getting their documentation to be thorough and clear will save them from having problems later on in the project.

Once they feel that their documentation is complete, players will want to work on implementing their project. To the right of the design cards, programmer cards are laid out in a row. Each turn that a player chooses to work on implementation allows them to use their programmers to meticulously create good code, more quickly create poor code, inspect code or fix bugs. A programmer's coding progress is represented by code cards, which are placed face-down above the programmer used to create them. In order to reveal these code cards, programmers must inspect their code. Doing so allows the code cards to be flipped over, revealing whether or not the code contains any bugs. Bugs can then be fixed by the player's programmers.

By using these actions in different combinations, players are able to code in a variety of coding styles. A programmer can be made to slowly produce good code and inspect it, fixing bugs as they are found. On the other hand, a programmer can create large sections of rush code and then inspect them all at the end. However, the rules are set up to encourage strategies with more real-world validity.

Once all of the code needed for a project is finished, players may choose the integration phase to consolidate one programmer's code per turn. This means that the greater the number of total programmers who worked on a project, the longer it will take for the program to be tied together. Finally, a player may submit their project to the customer. As long as the code they have created does not contain too many bugs, this player is the winner! However, if the customer is dissatisfied, the player may yet still have work ahead of them, a setback that could cost them the game. Thus, players will be forced to reconcile their need to complete their project quickly with the need to complete high-quality documentation and code.

Evaluation

We conducted an experiment in which 28 undergraduate software engineering students were recruited to play Problems and Programmers once or twice and asked to fill out a questionnaire stating their impressions of the game. Most of these questions asked for a numerical answer on a 1 to 5 scale. In general, students' feelings about the game were favorable, as shown in Table 1.

Question	***1***	***2***	***3***	***4***	***5***
How enjoyable is it to play?			6	13	9
How difficult/easy is it to play?		3	10	12	3
How well does it reinforce knowledge?		6	9	6	6
How well does it teach new knowledge?	7	8	6	3	4
How well does it teach the software process?	1	4	8	12	3
Incorporate it as standard part of SE course?	1	6	3	12	6
As an optional part?	1	5	4	10	8
As a mandatory part?	1	6	8	11	2

Table 1. Questionnaire results.

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). Perhaps most indicative is the 3.7 average score for the question as to whether the game should be incorporated as a standard part of a software engineering course, clearly a vote of confidence by the students who participated in the experiment.

Students were also asked to answer some open-ended questions about the game. Their responses to these 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."

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

Concerning how well the game teaches software engineering process issues, students commented:

"Consequences are more drastic than mentioned in class. We could clearly see this in the game."

"You need to put the time into earlier phases (design) or else it will come back to get you."

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 lecture 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.

Future Work

Currently in development are an online, computer-based version of Problems and Programmers that will allow students to play against each other over the Internet, as well as a re-designed version of the physical card game. This new version will represent the software process in a very different way, but will be easier to learn and still teach a wide variety of lessons. Once this version of the game is completed, we plan to perform similar experiments to the one we conducted for the initial version of the game, in order to evaluate the benefits and drawbacks of each version.

SIMSE

SimSE is a computer-based simulation environment for teaching the software engineering process, and is a single-player game in which the player takes on the role of project manager of a team of developers. The player must manage these employees to complete a particular (aspect of a) software engineering project. Management activities include, among others, hiring and firing, assigning tasks, monitoring progress, and purchasing tools. As in Problems and Programmers, following good software engineering practices will generally lead to positive results while blithely ignoring these practices will lead to miserable failure in completing the project.

The user interface of SimSE is fully graphical, displaying a "virtual" office in which the software engineering process is taking place, including typical office surroundings (e.g., desks, chairs, computers, meeting rooms), employees, customers, and project information (e.g., budget and time), as well as representations of software engineering artifacts (e.g., requirements documents, design documents, and source code) that include such information as that artifact's completeness, correctness, and other similar qualities. Information about the status of individuals is provided through automatic pop-up "bubbles" over the heads of individuals (for example, to express surprise in response to a player's action) and through explicitly querying an individual (for example, to ask how busy they are or how happy they are with their salary). Players use information gleaned from these sources to make decisions and take actions, driving the software engineering process to complete a project within budget, schedule, and at or above the customer's desired quality requirement.

Because many different schools of thought exist about software engineering, and the educational needs and objectives of different instructors vary, the models of the software process that execute within SimSE must be customizable. Therefore, an integral part of SimSE is a model builder that an instructor can use to specify the particular software engineering process that he or she wishes to teach, including the graphical representations to be used in the simulation. A customized simulation that the students can play is then generated.

Architecture

Figure 3 illustrates the architecture of SimSE. Models are created using a model builder that allows the specification of: 1) major entities in the simulation, i.e., employees, artifacts, projects, tools, and customers; 2) the actions, or activities that these entities can participate in (e.g., codes, integrates, reviews the requirements document, takes a break); 3) the rules that specify the effects of these actions on the rest of the simulation (e.g., for every clock tick that occurs during coding, the size of that piece of code increases by the additive productivity of all the employees working on it; the energy level of all of these employees also decreases by 5%); 4) the graphical representation of each of the major entities in the simulation; and 5) the entities that the game is to start with, or the start state. Rules specify both predictive [1, 2, 9] aspects (as magnitudes of causal effects) and prescriptive [4, 8] aspects (as allowable next steps). Based upon a particular choice of model, a generator interprets the model and automatically generates code for a state management and a rule execution component that are inserted into the generic simulation environment, such that the student can practice the situations captured by the chosen model.

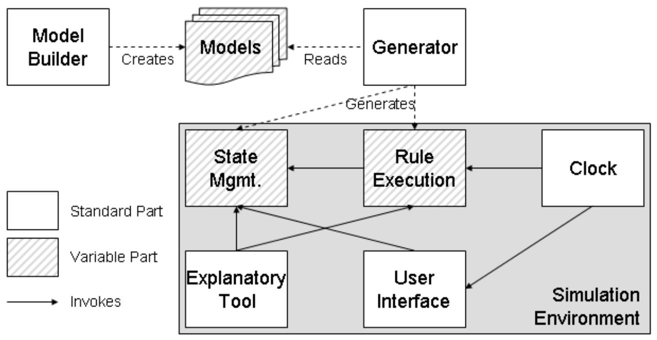


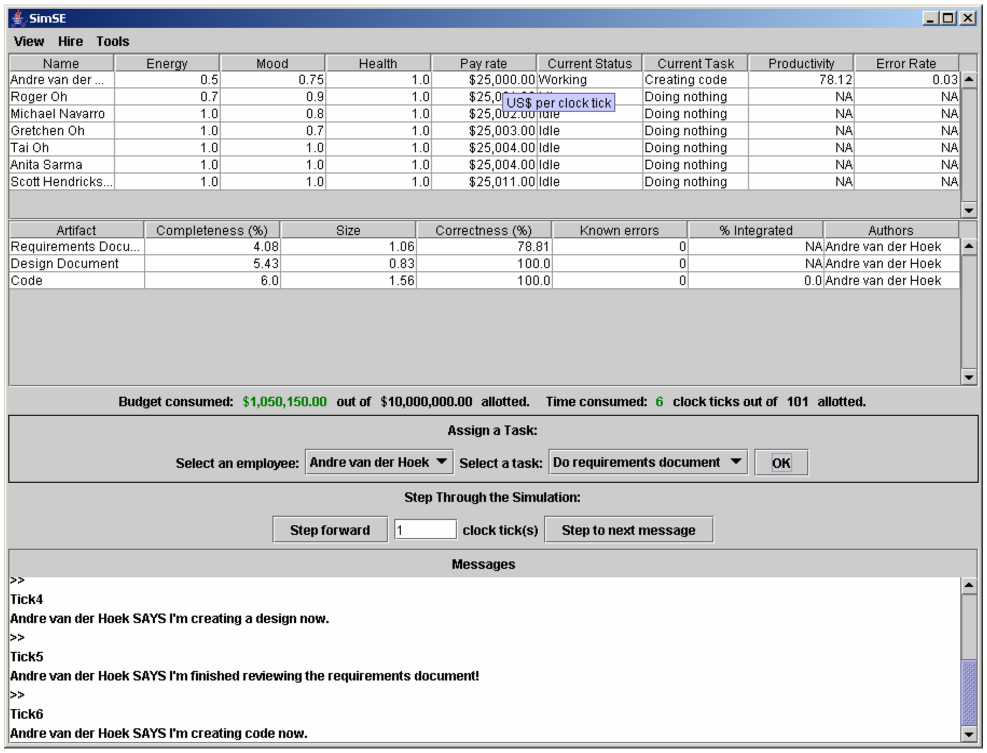
Figure 3. SimSE Architecture.

The simulation loop itself operates in the following manner: The clock drives the simulation by emitting ticks at equal time intervals. At every clock tick, the rule execution component checks which actions are currently executing by querying the state management component. It then executes the rules associated with these actions, and in turn propagates the effects of these rules on the entities and actions in the state management component. After this update is completed, the clock then signals the user interface to update the display. The user interface then queries the state management component and updates itself to reflect the current state.

In addition to these standard simulation components, the educational nature of SimSE also requires the addition of a unique component: SimSE's explanatory tool. This feature will provide a student with the ability to, at any time, request a visual trace of events. This trace will contain a time-ordered log of all inputs provided, the levels of the various meters as they progressed over time, and indicators as to which rules were triggered at which point in time. Additionally, the trace will indicate the length of time and impact for each rule triggered. In making the rules and cause-and-effect relationships clearly visible, students can compare different traces to evaluate their performance and determine which approaches lead to success and failure for different models.

Current Status

SimSE is a work in progress. To date, a proof-of-concept, non-graphical prototype version of SimSE has been built that displays information about employees, artifacts, projects, tools, and customers in tables and textual messages, and is built on a specific model of the software engineering process. The user interface for this prototype is shown in Figure 4. The topmost table contains information about the employees, including their name, energy level, pay rate, current activity, and productivity. Below that is the artifact status table, which displays each artifact's completeness, size, correctness, number of known errors, integration status, and authors. Underneath this is a bar that shows the project



Name	Energy	Mood	Health	Pay rate	Current Status	Current Task	Productivity	Error Rate
Andre van der ...	0.5	0.75	1.0	$25,000.00	Working	Creating code	78.12	0.03
Roger Oh	0.7	0.9	1.0	$25,0[illegible]	[illegible]	Doing nothing	NA	NA
Michael Navarro	1.0	0.8	1.0	$25,002.00	Idle	Doing nothing	NA	NA
Gretchen Oh	1.0	0.7	1.0	$25,003.00	Idle	Doing nothing	NA	NA
Tai Oh	1.0	1.0	1.0	$25,004.00	Idle	Doing nothing	NA	NA
Anita Sarma	1.0	1.0	1.0	$25,004.00	Idle	Doing nothing	NA	NA
Scott Hendricks...	1.0	1.0	1.0	$25,011.00	Idle	Doing nothing	NA	NA

Artifact	Completeness (%)	Size	Correctness (%)	Known errors	% Integrated	Authors
Requirements Docu...	4.08	1.06	78.81	0	NA	Andre van der Hoek
Design Document	5.43	0.83	100.0	0	NA	Andre van der Hoek
Code	6.0	1.56	100.0	0	0.0	Andre van der Hoek

Budget consumed: $1,050,150.00 out of $10,000,000.00 allotted. Time consumed: 6 clock ticks out of 101 allotted.

```
>>
Tick4
Andre van der Hoek SAYS I'm creating a design now.
>>
Tick5
Andre van der Hoek SAYS I'm finished reviewing the requirements document!
>>
Tick6
Andre van der Hoek SAYS I'm creating code now.
```

Figure 4. SimSE Non-graphical Prototype User Interface.

status, namely, time and budget. Players use the area beneath this to assign tasks to employees by choosing the employee and the task from drop-down menus. The player can also step through the simulation by either specifying a number of clock ticks to step through, or by choosing to step through the simulation until the next message appears. The bottom section of the interface is the message window, in which employees “say” what they are doing and “answer” the player’s inquiries about their activities and the artifacts they are working on. Players are also notified of various events through this window, such as “virtual” network failure or introduction of added requirements by the customer. Finally, players can also use menu bar options to view employee qualifications, hire and fire employees, and manage tools.

The customizable, fully graphical version of SimSE is currently in development. The model builder is nearing completion and, in parallel, development of the code generator has begun. Once completed, we will be evaluating SimSE by conducting experiments similar to the one for Problems and Programmers.

CONCLUSIONS

Both Problems and Programmers and SimSE attempt to address the lack of software engineering process education present in the typical software engineering course by providing students with a practical, high-level experience of a realistic software engineering process in an engaging manner. We plan to continue work on more sophisticated versions of each game, incorporate both games into actual software engineering courses in the coming year, and further evaluate their effectiveness, refining them accordingly.

REFERENCES

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

[2] Boehm, B., Abts, C., Brown, W., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, D., and Steece, B, *Software Cost Estimation with COCOMO II*. 2000, New Jersey: Prentice Hall.

[3] Callahan, D. and B. Pedigo, "Educating Experienced IT Professionals by Addressing Industry's Needs." IEEE Software, 2002. **19**(5): p. 57-62.

[4] Cass, A.G., et al., "Little-JIL/Juliette: A Process Definition Language and Interpreter." In Proceedings of the 22nd International Conference on Software Engineering. 2000: Limerick, Ireland. p. 754-757.

[5] Conn, R., "Developing Software Engineers at the C-130J Software Factory." IEEE Software, 2002. **19**(5): p. 25-29.

[6] Ludi, S. and J.S. Collofello, "An Analysis of the Gap Between the Knowledge and Skills Learned in Academic Software Engineering Course Projects and Those Required in Real Projects." In Proceedings of the 2001 Frontiers in Education Conference. 2001.

[7] 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.

[8] Noll, J. and W. Scacchi, "Specifying Process-Oriented Hypertext for Organizational Computing." Journal of Network and Computer Applications, 2001. **24**(1): p. 39-61.

[9] Raffo, D., *Modeling Software Processes Quantitatively and Assessing the Impact of Potential Process Changes on Process Performance*, in *Graduate School of Industrial Administration*. 1996, Carnegie Mellon University: Pittsburgh, PA.

AUTHOR BIOGRAPHIES

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 UC 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.

Alex Baker is currently working towards a PhD in Information and Computer Science at the University of California, Irvine. He received his B.S. in Information and Computer Science from UC 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.

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 UC 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 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.