UNIVERSITY OF CALIFORNIA,
IRVINE

# SimSE: A Software Engineering Simulation Environment for Software Process Education

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Emily Navarro

Dissertation Committee:
Professor André van der Hoek, Chair
Professor David Redmiles
Professor Debra J. Richardson

2006

This dissertation of Emily Navarro
is approved and is acceptable in quality and form for
publication on microfilm and digital formats:

_____

_____

_____

Committee Chair

University of California, Irvine
2006

**DEDICATION**

To My Family.

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ACKNOWLEDGEMENTS

I would first like to thank the person who has been the most direct help and support to me throughout the process of getting my Ph.D., my advisor, André van der Hoek. You always pushed me to do my best work and achieve my fullest potential. Even when it was hard, I always looked back on the things we did together and was thankful for the extra push—each time it turned out better because of it. You have always expressed the utmost confidence and belief in me, and I sincerely appreciate it. You know how to temper your push to succeed with the right amount of understanding, especially of the unique circumstances that come with being a woman in this community. Somehow I was able to plan a wedding, get married, have a baby, and get my Ph.D. in five years! I know this is in large part because of your understanding and patience, which always motivated me to give my research my best effort in spite of all these extenuating circumstances. I not only value our advisor-student relationship, but also our friendship. We have had a lot of fun together, and I look forward to more of that in the years to come.

I would also like to thank the other members of my committee, David Redmiles and Debra Richardson, for their wise input that has helped shape my research for the better, and for the time and effort that they have put into serving me and the research community in this invaluable way. I especially appreciate the letters of support written for me during these last five years, and the funding opportunities they have facilitated.

The ARCS foundation and the NSF have both provided financial support for my education during these last five years, and for that I am extremely grateful.

Many thanks to my research group, who spent several hours playing and giving me feedback about SimSE. Special thanks to Alex Baker for building one of the SimSE models and for helping with some of the experiments. Extra special thanks to Anita Sarma and Scott Hendrickson, who are not only my colleagues, but who have also become very dear friends of mine during these last five years.

To my dear, sweet husband, thank you for putting up with me while I got my Ph.D. As you know, I often had a level of stress that made life kind of miserable (for both of us) at times, but you constantly amazed me with the patience and love you showed me in return. I often feel I don't deserve you! If I did not have the happiness and contentment that have come from being married to you, I doubt I could have achieved this.

My precious daughter Mollie, you will not remember this past year that we have had together, finishing Mommy's research and writing her dissertation, but I will never forget it. We make a great team, you and me! Your very existence fills my heart to overflowing. Thank you for being such a good baby and a good napper so Mommy could get work done!

Thank you to my parents for the many years of love and support that made my education possible. Mommy, I cannot even begin to express how essential your friendship, support,

prayers, daily phone calls, helping out with Mollie, and just always being there for me have been to this process. My goal is to do as good a job of helping Mollie in her life's endeavors as you have done for me. You are my role model in so many ways. Daddy, I doubt I ever would have even considered getting a Ph.D. if it wasn't for you. Your unwavering belief in me gave me the confidence to make it through, and knowing that you are proud of me is one of my greatest joys.

Thank you to my dear sisters, Erica and Elizabeth, for playing school with me in my toddler and preschool years so that I was able to read, add, subtract, multiply, and divide before I started kindergarten. I credit you both with giving me a jumpstart on my education that made it possible for me to go this far in school.

To my in-laws, Margie, Natalie, Danny, Dean, Gloria, Luisita, and Bob, thank you for accepting me into your family as if I had always been part of it. The peace that comes with knowing I have a strong, loving family around me helped to make this all possible.

To my Bible study group, the Bakers, the Chous, the Henrys, the Huffmans, the Martinezes, and the Murrays, thank you for your love and support, and especially your prayers. They lifted me up during some difficult times. I am so grateful for them.

My beloved dog, Roger, you should get your own diploma for being such a faithful, loving companion all these years. From the time I started college until now, you have always been right by my side while I worked, showing your support in the only way you knew how—laying your head in my lap, keeping my feet warm, or just being there. I love you and am so grateful for all the time we have had together throughout these years.

Ultimately, I thank my God for so abundantly blessing me with all of these people who believed in me, and for His mercy and grace, which I so desperately need every day. I truly believe that every talent, gift, and ability I have comes from my Creator, so to Him I give the biggest "Thank You!" of all.

## CURRICULUM VITAE

**Emily Navarro**

## EDUCATION

| | |
|---|---|
| 2001 – 2006 | Doctor of Philosophy in Computer Science<br>University of California, Irvine<br>Research Area: Software engineering education |
| 2001 – 2003 | Master of Science<br>University of California, Irvine<br>Area: Software engineering |
| 1994 – 1998 | Bachelor of Science<br>University of California, Irvine<br>Major: Biological Sciences |

## EMPLOYMENT

| | |
|---|---|
| 2000 – 2006 | University of California, Irvine, Donald Bren School of Information and Computer Sciences, Irvine, CA<br>Position (2000 – 2006): *Graduate Research Assistant*<br>Position (2002 – 2005): *Teaching Assistant* |
| 2005 Summer | Google Inc., Santa Monica, CA<br>Position: *Software Engineering Intern* |
| 1999 – 2000 | The Jesus Film Project, San Clemente, CA<br>Position: *Statistical Research Assistant* |
| 1998 – 1999 | Arco Products Co., La Palma, CA<br>Position: *Help Desk Coordinator* |

## REFEREED JOURNAL PUBLICATIONS

J.2 E. Oh Navarro and A. van der Hoek, *Software Process Modeling for an Educational Software Engineering Simulation Game*, Software Process Improvement and Practice special issue containing expanded best papers from the Fifth International Workshop on Software Process Simulation and Modeling: 10 (3), pp. 311-325. 2004.

J.1 A. Baker, E. Oh Navarro, and A. van der Hoek, *An Experimental Card Game for Teaching Software Engineering Processes*, Journal of Systems and Software special issue containing invited and expanded best papers from the 2003 International Conference on Software Engineering & Training: 75 (1-2), pp. 3-16. 2005.

## REFEREED CONFERENCE AND WORKSHOP PUBLICATIONS

C.13 T. Birkhoelzer, E. Oh Navarro, and A. van der Hoek. *Teaching by Modeling instead of by Models*. Sixth International Workshop on Software Process Simulation and Modeling, May 2005.

C.12 E. Oh Navarro and A. van der Hoek, *Design and Evaluation of an Educational Software Process Simulation Environment and Associated Model*, Eighteenth Conference on Software Engineering Education & Training, April 2005.

C.11 E. Oh Navarro and A. van der Hoek, *Scaling up: How Thirty-two Students Collaborated and Succeeded in Developing a Prototype Software Design Environment*, Eighteenth Conference on Software Engineering Education & Training, April 2005.

C.10 E. Oh Navarro and A. van der Hoek, *SimSE: An Interactive Simulation Game For Software Engineering Education*, IASTED Conference on Computers and Advanced Technology in Education, August 2004, pages 12–17 **(nominated for best paper)**.

C.9 E. Oh Navarro and A. van der Hoek, *SimSE: An Educational Simulation Game for Teaching the Software Engineering Process*, SIGCSE Conference on Innovation and Technology in Computer Science Education, June 2004, page 233.

C.8 E. Oh Navarro and A. van der Hoek, *Software Process Modeling for an Interactive, Graphical, Educational Software Engineering Simulation Game*, Fifth International Workshop on Software Process Simulation and Modeling, May 2004, pages 171–176.

C.7 A. Baker, E. Oh Navarro, and A. van der Hoek, *Teaching Software Engineering using Simulation Games*, International Conference on Simulation in Education, January 2004, pages 9–14.

C.6 A. Baker, E. Oh Navarro, and A. van der Hoek, *Problems and Programmers: An Educational Software Engineering Card Game*, Twenty-fifth International Conference on Software Engineering, May 2003, pages 614–619.

C.5 A. Baker, E. Oh Navarro, and A. van der Hoek, *An Experimental Card Game for Teaching Software Engineering*, Sixteenth International Conference on Software Engineering Education and Training, March 2003, pages 216–223 **(selected as one of best papers, leading to J.1)**.

C.4 E. Oh Navarro and A. van der Hoek, *Towards Game-Based Simulation as a Method of Teaching Software Engineering*, Thirty-second ASEE/IEEE Frontiers in Education Conference, November 2002, page S2G-13.

C.3 E. Oh, *Teaching Software Engineering Through Simulation*, Twenty-fourth International Conference on Software Engineering Doctoral Symposium, May 2002, pages 38-40.

C.2 E. Oh and A. van der Hoek, *Adapting Game Technology to Support Individual and Organizational Learning*, 2001 International Conference on Software Engineering and Knowledge Engineering, June 2001, pages 347–354.

C.1 E. Oh and A. van der Hoek, *Challenges in Using an Economic Cost Model for Software Engineering Simulation*, Third International Workshop on Economics-Driven Software Engineering Research, May 2001, pages 45–49.

## OTHER PUBLICATIONS

O.3 E. Oh Navarro, *A Survey of Software Engineering Educational Delivery Methods and Associated Learning Theories*, UC Irvine, Institute for Software Research Technical Report, UCI-ISR-05-5, April 2005.

O.2 A. Baker, E. Oh Navarro, and A. van der Hoek, *Introducing Problems and Programmers, an Educational Software Engineering Card Game*, Software Engineering Notes, March 2003, pages 7–8.

O.1 E. Oh and A. van der Hoek, *Teaching Software Engineering through Simulation*, Online Proceedings of the Workshop on Education and Training, July 2001.

## PRESENTATIONS

P.13 August 2005, *Google Inc*, Santa Monica, CA (Intern tech talk)

P.12 May 2005, *International Workshop on Software Process Simulation and Modeling*, St. Louis, MO

P.11 April 2005, *Eighteenth International Conference on Software Engineering Education and Training*, Ottawa, Canada

P.10 November 2004, *Twelfth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Newport Beach, CA (tutorial)

P.9 August, 2004, *IASTED Conference on Computers and Advanced Technology in Education*, Kauai, HI

P.8 June 2004, *SIGCSE Conference on Innovation and Technology in Computer Science Education*, Leeds, United Kingdom

P.7 May 2004, *International Workshop on Software Process Simulation and Modeling*, Edinburgh, United Kingdom

P.6 March 2004, *Federal University of Rio de Janeiro*, Rio de Janeiro, Brazil

P.5 November 2002, *Frontiers in Education Conference*, Boston, MA

P.4 May 2002, *International Conference on Software Engineering Doctoral Symposium*, Orlando, FL

P.3 July 2001, *Workshop on Education and Training*, Santa Barbara, CA

P.2 June 2001, *International Conference on Software Engineering and Knowledge Engineering*, Buenos Aires, Argentina

P.1 May 2001, *International Workshop on Economics-Driven Software Engineering Research*, Toronto, Canada

## TEACHING

| | | |
|---|---|---|
| *Teaching Assistant* | ICS 52 | Introduction to Software Engineering |
| | ICS 125 | Project in Software System Design |
| | ICS 127 | Advanced Project in Software Design |

## UNDERGRADUATE STUDENTS ADVISED

| | |
|---|---|
| Kuan Sung Lee | B.S. 2004, Information and Computer Science, University of California Irvine |
| Kenneth Shaw | B.S. 2004, Information and Computer Science, University of California Irvine |

| | |
|---|---|
| Beverly Chan | B.S. 2005, Information and Computer Science, University of California Irvine |
| Barbara Chu | B.S. 2005, Information and Computer Science, University of California Irvine |
| Calvin Lee | B.S. 2005, Information and Computer Science, University of California Irvine |
| Terry Fog | Senior, Information and Computer Science, University of California Irvine |

## SERVICE TO THE RESEARCH COMMUNITY

### Program Committee Member

Eighteenth International Conference on Software Engineering Education and Training
Nineteenth International Conference on Software Engineering Education and Training
Twentieth International Conference on Software Engineering Education and Training

### Journal Reviews

IEEE Software (2006)
Software Process Improvement and Practice (2004)

### Conference Reviews

Thirty-second ASEE/IEEE Frontiers in Education Conference (FIE 2003)
Thirty-third ASEE/IEEE Frontiers in Education Conference (FIE 2005)

### Other

Chair of Student Participation, Nineteenth International Conference on Software Engineering Education and Training (CSEE&T 2006)

## TECHNICAL SKILLS

| | |
|---|---|
| *Languages* | Java, C++, OpenGL, HTML XML; familiar with LISP, Prolog |
| *Operating Systems* | Windows NT/XP/2000/9x, Unix (Solaris, Linux) |
| *Tools* | Eclipse, Visual Café, JPad, MS Visual C++, MS Visual J++, SPSS, XML-Spy, Dreamweaver, Subversion, CVS, MS Office |

## HONORS

2006 UC Irvine Donald Bren School of Information and Computer Sciences Dissertation Fellowship Recipient
2005 Session Chair at Eighteenth Conference on Software Engineering Education and Training
2005 Google 2005 Anita Borg Scholarship Finalist
2004 Achievement Rewards for College Scientists (ARCS) Fellowship Recipient
2004 UC Irvine Donald Bren School of Information and Computer Sciences Fellowship Recipient
2003 Achievement Rewards for College Scientists (ARCS) Fellowship Recipient
2003 UC Irvine Department of Information and Computer Science Departmental Fellowship Recipient
2002 Graduate Assistance in Areas of National Need (GAANN) Fellowship Recipient
2002 National Science Foundation Graduate Research Fellowship Honorable Mention
2001 UC Irvine Department of Information and Computer Science Departmental Fellowship Recipient
2001 Session Chair at Fourteenth Conference on Software Engineering Education and Training
2001 UC Irvine Undergraduate Research Opportunities Grant Recipient
2001 Dean's Honor List (June)
2001 Dean's Honor List (March)
2000 Dean's Honor List

**ABSTRACT OF THE DISSERTATION**

SimSE: A Software Engineering Simulation Environment

for Software Process Education

By

Emily Navarro

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2006

André van der Hoek, Chair

The typical software engineering education lacks a practical treatment of the *processes* of software engineering—students are presented with relevant process theory in lectures, but have only limited opportunity to put these concepts into practice in an associated class project. Simulation is a powerful educational tool that is commonly used to teach processes that are infeasible to practice in the real world. The work described in this dissertation is based on the hypothesis that simulation can bring to software engineering education the same kinds of benefits that it has brought to other domains. In particular, we believe that software process education can be improved by allowing students to practice, through a simulator, the activity of managing different kinds of quasi-realistic software engineering processes.

To investigate this hypothesis, we used a three-part approach: (1) design and build SimSE, a graphical, interactive, educational, customizable, game-based simulation environment for software processes, (2) develop a set of simulation models to be used in seeding the environment, (3) evaluate the usage of the environment, both in actual

software engineering courses, and in a series of formal, out-of-class experiments to gain an understanding of its various educational aspects. Some of the educational aspects explored in these experiments included how SimSE compares to traditional teaching techniques, and which learning theories are employed by students who play SimSE.

Our evaluations strongly suggest that SimSE is a useful and educationally effective approach to teaching software process concepts. Students who play SimSE tend to learn the intended concepts, and find it a relatively enjoyable experience. These statements apply to students of different genders, academic performance levels, and industrial experience backgrounds. However, in order for SimSE to be used in the most effective way possible, our experience has demonstrated that it is crucial that it be used complementary to other educational techniques and accompanied by an adequate amount of direction and guidance given to the student. Our evaluations also suggested a number of promising directions for future research that can potentially increase the effectiveness of SimSE and be applied to educational simulation environments in general.

# 1. Introduction

While the software industry has had remarkable success in developing software that is of an increasing scale and complexity, it has also experienced a steady and significant stream of failures. Most of us are familiar with public disasters such as failed Mars landings, rockets carrying satellites needing to be destroyed shortly after takeoff, or unavailable telephone networks, but many more "private" problems occur that can be equally disastrous or, at least, problematic and annoying to those involved. Examining one of the prime forums documenting these failures, the Risks Forum [4], provides an illuminating insight: a significant portion of documented failures can be attributed to software engineering *process breakdowns*. Such breakdowns range from individuals not following a prescribed process (e.g., not performing all required tests, not informing a colleague of a changed module interface), to group coordination problems (e.g., not using a configuration management system to coordinate mutual tasks, not being able to deliver a subsystem in time), to organizations making strategic mistakes (e.g., choosing to follow the waterfall process model where an incremental approach would be more appropriate, not accounting for the complexity of the software in a budget estimate). As a result, it is estimated that billions of dollars are wasted each year due to ineffective processes and subsequent faulty software being delivered [79].

We believe the root cause of this problem lies in education: current software engineering courses typically pay little to no attention to students being able to practice issues surrounding the software engineering *process*. The typical software engineering course consists of a series of lectures in which theories and concepts are communicated, and, in an attempt to put this knowledge into practice, a small software engineering

project that the students must develop. Although both of these components are necessary—lectures as a source for the basic knowledge of software engineering and projects as a way to gain hands-on experience with some of the techniques of software engineering, this approach fails to adequately teach the overall software *process*, a key part of software engineering.

The underlying issue is the constraints of the academic environment—while relevant process theory can be and typically is presented in lectures, the opportunities for students to practically and comprehensively experience the presented concepts are limited. There are simply not enough time and resources for the students to work on a project of a large enough size to exhibit many of the phenomena present in real-world software engineering processes. In addition, the brevity of the quarter, semester, or even academic year leaves little room for the student to try (and possibly fail at) different approaches in order to learn which processes work best for which situation. Most course projects simply guide students through a linear execution of the waterfall model (requirements, design, implementation, testing) in which students are left with little discretion. Students cannot decide which overall life cycle model to follow, whether or not to first build a rapid prototype, or even when to set the milestones for their deliverables—these and other decisions are usually made by the instructor. The focus strongly remains on creating project deliverables such as requirements documents, design documents, source code, and test cases, and little room is left to illustrate or experience the principles, pitfalls, and dimensions of the software process. The overall result is that students are unable to build a practical intuition and body of knowledge about the software process, and are ill-equipped for choosing particular software processes, for recognizing potentially

troublesome situations, and for identifying approaches with which to address such troublesome situations.

This lack of process education is evident in the way industry repeatedly complains that recent graduates of computer science programs are unprepared for tackling real-world software engineering projects [27, 36, 98, 130]. Academia has also recognized this deficiency and has attempted to remedy it with a wide range of innovations designed to make class projects more closely resemble those in industry. These have included such things as intentionally introducing real-world complications into a project, (e.g., causing hardware and software to crash when a deadline is looming [45]), maintaining a large-scale, ongoing project that different groups of students work on from semester to semester [97], requiring students to work on a real-world project sponsored by an industrial organization [66], incorporating multiple universities and disciplines into the project [21], and many others. However, in each of these approaches, the time and scope constraints imposed by the academic environment still remain, and prevent most of the phenomena involved in real world software engineering processes from being exhibited (although they do succeed in highlighting a few of these issues). So far, no single approach (or set of approaches) has been accepted as a sufficient solution to the problem.

Simulation is a powerful educational tool that has been widely and successfully used in a number of different domains. Before airline pilots fly an actual jet plane full of passengers, they extensively train in simulators [118]. Military personnel practice their decision-making and leadership abilities in virtual reality simulation environments [92]. Students in hardware design courses use simulators to practice designing new, state-of-the-art CPU's [33]. In all of these cases, simulation provides significant educational

benefits: valuable hands-on experience is accumulated without incurring the high cost of actual exercise and without the risk of dramatic consequences that may occur in case of failure. Moreover, unknown situations can be introduced and practiced, experiences can be repeated, alternatives can be explored, and often a general freedom of experimentation is promoted in the training exercise, allowing the student to gain deeper insights with each simulation run [90].

On top of these known benefits, educational simulations are also known to embody a number of different well-known and well-understood learning theories [5, 20, 34, 56, 110, 123], a characteristic that suggests it has a great deal of educational potential that should be explored. In spite of this, simulation has been significantly under-explored in the field of software process and software engineering in general.

The goal of this work is to understand whether simulation can bring to software engineering education the same kinds of benefits that it has brought to other domains. We hypothesize that software engineering education can be improved, specifically in the domain of software engineering *processes*, by using simulation. In particular, we believe that this improvement can be brought about by allowing students to practice, through a simulator, the activity of managing different kinds of quasi-realistic software engineering processes. While we certainly do not anticipate nor claim that this will address all of the educational deficiencies that typically lead to software process breakdowns, we have carefully chosen the focus of this hypothesis to be on what we believe is one of the root causes of these breakdowns: the lack of practice a student has in managing software processes from a project manager's perspective.

To investigate this hypothesis, our approach was threefold: (1) build a graphical, interactive, educational, customizable, game-based simulation environment for software processes, (2) develop a set of simulation models to be used in seeding the environment, (3) evaluate the usage of the environment, both in actual software engineering courses, and in a series of formal, out-of-class experiments to gain understanding of its various educational aspects.

Out of our technical development came SimSE, a computer-based environment that facilitates the creation and simulation of software engineering processes. SimSE allows students to virtually participate in realistic software engineering processes that involve real-world components not present in typical class projects, such as large teams of people, large-scale projects, critical decision-making, personnel issues, multiple stakeholders, budgets, planning, and random, unexpected events. In so doing, it aims to provide students with a platform through which they can experience many different aspects of the software process in a practical manner without the overarching emphasis on creating deliverables that is inherent in actual software development.

Along with the environment, we also developed a set of simulation models to be used in SimSE. These models cover a number of different software engineering processes, such as the waterfall model, Extreme Programming, and a code inspection process. In each of these, the player is rewarded for following that process model's "best practices" and penalized for deviating from them.

We developed these models using SimSE's model builder tool, a critical part of our environment that we created with the express intent of allowing instructors to build customized simulation models. Using this tool, instructors can encode the software

process lessons they want their students to learn, and then generate a customized simulation game based on those lessons. Because there exist a wide variety of different software process models, several different schools of thought about what are software process "best practices" [119], and numerous instructors with varied teaching objectives, one of SimSE's fundamental goals was the ability to support customization of the software processes it simulates.

Because the purpose of this work is to improve software engineering education, the third part of our approach started where learning primarily takes place: the classroom. Namely, we investigated the potential for simulation's incorporation into an actual software engineering curriculum by putting SimSE into use in introductory software engineering courses. As the students used SimSE, we tested how well they were able to learn the software process concepts it was designed to teach, and carefully observed and collected their reactions and attitudes about the experience.

We also evaluated SimSE in a series of formal, out-of-class experiments to look into educational aspects that were independent of SimSE's in-class usage. In particular, we performed three such experiments: (1) A pilot experiment to evaluate the initial educational potential of SimSE and its first simulation model by having undergraduate computer science students play the game and provide us with their feedback; (2) A comparative study between students who played SimSE, students who read from a textbook, and students who listened to lectures (noting the differences in their attitudes, observations, and gain in software process knowledge); and (3) An in-depth observational study of the learning process students go through while playing SimSE that also served to

evaluate the effectiveness of SimSE's explanatory tool in providing students with insight into their simulation runs, and hence, into the process being simulated as well.

To summarize, this work addresses the following set of incremental research questions, each of which has driven the development, usage, and evaluation of SimSE:

1. **Can a graphical, interactive, educational, customizable, game-based software engineering simulation environment be built?** We have successfully built such an environment, although, by necessity, certain tradeoffs had to be made between these qualities (graphics, interactivity, educational factors, "fun factors", customizability) to create a balance that could effectively and feasibly be developed.
2. **Can students actually learn software process concepts from using such an environment?** If given adequate background knowledge and guidance, which has proven to be crucial, students who use such an environment do seem to glean from the simulation models the concepts they are designed to teach.
3. **If students can learn software process concepts from using such an environment, how does the environment facilitate the learning of these concepts?** The most common learning theories employed by players of SimSE are Discovery Learning, Learning through Failure, and Constructivism. Learning by Doing and Situated Learning are also significant, but seen slightly less. Certain aspects of Keller's ARCS theory of motivation are employed strongly (attention and satisfaction) while others are only moderately employed (relevance and confidence).

4. **How can such an environment fit into a software engineering curriculum?** Simulation in a software engineering curriculum seems to fit best as a complementary component to lectures, projects, and readings. One option that has proven useful is to use simulation as an optional extra-credit assignment in a course that provides the background knowledge required to understand the simulation models. In our experience using SimSE in this manner, the majority of students chose to complete the assignment, enjoyed it for the most part, and seemed to learn the concepts the models are designed to teach. (Of course, this is only one option. As part of the follow-on work to this dissertation we plan to experiment with others, such as making it mandatory or optional. See Chapter 12 for further information.)

Based on the answers to these research questions that have been suggested by our experience and the data we have collected, the work described in this dissertation provides the following contributions:

1. The insight that simulation can be beneficial to software engineering process education, but with two crucial caveats: First, simulation needs to be used complementary to other educational techniques (such as lectures, projects, and readings) so that students will have adequate background knowledge to successfully use the simulation in such a way that they will learn the lessons it is designed to teach. Second, it is absolutely crucial that an adequate amount of direction and guidance be given with a simulation assignment, in order for the simulation to be used by the students correctly and effectively.

2. An implementation of a graphical, interactive, educational, customizable, game-based software engineering simulation environment, along with a set of simulation models, that has been put through both in-class use and out-of-class formal evaluations.
3. Insight into the role and potential of an explanatory tool in an educational simulation, as well as an implementation of such a tool.
4. Experience with the use of simulation in software engineering education, including the lessons learned and promising directions for future research.

The remainder of this dissertation is organized as follows: Chapter 2 frames SimSE in its research context by providing an overview of the background research areas from which it stems. Chapter 3 presents the approach we took to addressing the problem our work aims to tackle. In Chapter 4, we describe the modeling capabilities of our simulation approach. Chapter 5 presents SimSE's model builder tool. In Chapter 6, we discuss SimSE, including details of its game play, design, and implementation. Chapter 7 details the various simulation models that have been built using the model builder tool. Chapter 8 introduces SimSE's explanatory tool. Chapter 9 describes our experience with actual usage and evaluation of SimSE. In Chapter 10, we provide an overview of related work. Chapter 11 presents the conclusions we can draw from this work and in Chapter 12 we describe our plans for future work.

# 2. Background

In order to frame the context of our approach, this chapter will take a broad look at two major research areas from which this work stems: software engineering education and learning theories. First, we will present an overview of how other educators have attempted to address the problem of under-preparedness on the part of graduates starting their careers in industry. Then, we will survey the well-known learning theories that are applicable to the discipline of software engineering education. Finally, we will present a categorization of the surveyed approaches in terms of the learning theories they employ, focusing in particular on what this can teach us about the potential for educational software engineering simulation approaches.

## 2.1 Software Engineering Educational Approaches

In surveying the software engineering educational literature, it is clear that nearly every approach to teaching the subject is based on the same two components: lectures, in which software engineering theories and concepts are presented; and projects, in which students must work in groups to develop a (generally small) piece of software. However, judging from the dissatisfaction of industrial organizations that hire recent graduates (mentioned previously), it is clear that this approach is not sufficiently preparing future software engineers for jobs in the real world.

The academic community has recognized this problem and, in response to it, has created a wealth of innovations that build on the standard lectures plus project approach. These approaches fall into three major categories. The first involves attempts to make the students' project experience more closely resemble one they would encounter in the real

world ("realism"). The second category includes approaches that teach one or more specific subjects a particular instructor feels are currently missing (e.g., usability testing or formal methods), and are crucial to effectively educating the students ("missing piece"). The final category is simulation approaches, in which educators have students practice software engineering processes in a (usually) computer-based simulated environment. The remainder of this section describes these categories and their approaches in greater detail.

### 2.1.1 Adding Realism to Class Projects

It is clear from looking at the software engineering literature that the most common method of improving the educational experience involves modifying certain aspects of the class project to make it more closely resemble the experience students will face in their future software engineering careers. As the academic environment differs so greatly from the industrial, there are numerous angles from which educators have approached this issue in terms of aspects of the academic environment that they have tried to make more realistic.

Some of these involve an industrial organization as a participant in the project, either by modifying the organization's existing software [57], using one of their representatives in an advisory role [11], inviting one of their representatives to give guest lectures and/or mentor the students [125, 147], using one of their projects to be examined as a case study [60, 85], or by having an industrial participant actually function as a customer for the students' project [54, 65, 66]. Through the extra pressure of having a non-academic party involved, these "industrial partnership" approaches aim to teach students a greater

appreciation of quality, give them an opportunity to learn a real application domain, and motivate them more thoroughly to do their best work.

Another approach uses only maintenance- or evolution-based projects instead of building a system from scratch [7, 57, 97]. In some of these, the maintenance project is ongoing over a number of semesters or quarters, and each class extends and builds on the previous classes' modifications [97, 126, 128, 146]. In others, the piece of software being modified is unique to that particular class and/or semester [7, 57, 82, 107, 109]. These approaches generally argue that, since the majority of real-world projects are maintenance projects, students will be better prepared for the real world by becoming familiar with these types of projects during their university education.

Other "realism" approaches focus on the nature and composition of the student teams that work on the project, making them more closely mirror the team dynamics in real-world software engineering situations so that students will learn the skills necessary to work in teams when they enter their industrial careers. These approaches have done such things as making the same people work together for multiple projects and/or semesters [125], making the student teams very large [15], distributing the members of a team across courses [132], majors [43], universities [21], or even countries [50], or enforcing formal structure and communication protocols [140].

Some other "realism" approaches focus on non-technical skills such as communication, group process, interpersonal competencies, project management, and problem solving [62], rather than traditionally taught skills like design and coding. These approaches identify such "soft" skills as what are most lacking in university graduates, and hence argue that this kind of emphasis is crucial for their education.

Others have tried to mimic common less-structured real-world software engineering situations by making the project purposely open-ended and/or vague. This is done in two main ways: either by allowing the students to define their own requirements (giving students the pseudo-experience of new product development based on market research) [100], or by allowing them to define their own process (giving students experience in not only following a process, but in designing the process that they follow) [63].

A somewhat radical approach that has been used is a "practice-driven" approach in which the curriculum is largely lab- and project-based [67, 104, 140]. In these approaches, lectures are used only as supporting activities. These approaches argue that theory is something that cannot be taught in a lecture, but instead must be built in each individual through experience and making mistakes.

Another approach that encourages learning through mistakes, although more explicitly, is deliberate sabotage. In this approach, the instructor purposely sets the students up for failure by introducing common real-world complications into projects, the rationale being that students will then be prepared when these situations occur in their future careers. Some of these sabotage tactics have included providing inadequate specifications, instructing the customer to be purposely uncertain when describing their needs, or purposely crashing the hardware just before a deadline [45].

Finally, in a somewhat different sort of sabotage, some have assigned projects that had been known to fail in the past due to software process problems [12]. In all cases, the students also failed, providing a perfect opportunity for the instructor to explain the

rationale behind the best practices of software processes, as well as a way for the students to learn the consequences of not following these practices firsthand.

### 2.1.2 Adding the "Missing Piece"

While the "realism" approaches all mainly focus on changing the *manner* in which software engineering concepts are taught, there is another large school of thought that concentrates instead on changing the *content* of what is taught—in particular, these approaches believe that software engineering education is lacking in effectiveness due to the omission of one (or a few) important subject(s). What this "missing piece" is varies from approach to approach, but all generally believe that the addition of this subject to the curriculum (either to an existing course or as an entirely new and separate course) will make the students' education much more complete, better preparing them for the real world.

Some of these approaches believe that formality is underemphasized, and propose to teach more formal methods [3] or to make traditional engineering education a larger part of software engineering curriculum [40]. Others believe that students should be taught a specific software process (such as the Personal Software Process (PSP) [69, 70], the Team Software Process (TSP) [116, 138], the Rational Unified Process (RUP) [54, 64], or Extreme Programming (XP) [67, 131]) and be required to follow that process in their academic software engineering projects. Still others propose that students should not only be required to follow a specific software process, but to also practice process engineering and project management techniques to create their own software processes and use process improvement techniques to improve upon them [24, 63, 77].

Rather than focus on process as a whole, other "missing piece" approaches focus on specific parts of the process (e.g., requirements analysis, testing), and specific techniques for performing that part (e.g., scenario-based requirements engineering [39], usability testing [143]). Other approaches, rather than teach software engineering in general, focus on a specific type of software engineering, such as maintenance-based software engineering [7, 133], component-based software engineering [53, 105], or software engineering for real-time applications [84, 86]. Still others believe it is certain non-technical aspects of software engineering that should be added to the software engineering curriculum, such as communication [60, 62], interacting with stakeholders [108], Human-Computer Interaction [68, 142], or the business aspects of software engineering (e.g., intellectual property, product marketing, and financial models) [126].

### 2.1.3 Simulation

While the majority of the software engineering educational approaches focus on adding realism to class projects or critical topics to the curriculum, a number of others argue that the only feasible way to provide students with the experience of realistic software engineering processes within the academic environment is through simulation, as used in conjunction with lectures and projects. While these approaches vary in terms of the processes they simulate and their specific purposes, they are all designed to allow students to practice and participate in software engineering processes on a larger scale and in a more rapid manner than can be feasibly done through an actual project. Within the realm of software engineering simulation, there are three varieties: industrial

simulation brought to the classroom, group process simulations, and game-based simulations.

In industrial simulation brought to the classroom, a simulator that is used in industry to predict the effects of process planning decisions is brought into the classroom for the students to practice on [35, 106]. The models run in these simulators are generally based strictly on empirical data. They also typically have a non-graphical interface (meaning they display a set of gauges, graphs, and meters rather than characters and realistic surroundings) and a relatively low level of interactivity, taking a set of inputs such as person power, project size, and/or process plan, and outputting a set of results, such as budget, time, and defect rate. Use of these highly-realistic simulations in the classroom is designed to illustrate to students, using real-world data, the overall life cycle and project planning phenomena of software engineering.

Group process simulations portray structured group discussion and interaction processes that are typically present in real-world software engineering situations [103, 136], such as code inspections and requirements analysis meetings. In these cases, the student engages in a discussion in which some or all of the other participants are simulated. Such simulations are designed to give students experience in these kinds of discussions, which, these approaches argue, is one area in which new graduates are typically unprepared.

The final category is game-based simulation, in which software engineering processes are practiced by "playing" them in a game-based environment [9, 44, 47, 78, 101, 129]. In these software engineering simulation games, the player is generally presented with a task to complete in the simulated world (normally to complete a software engineering

project within certain constraints), and must interact with the game to drive the simulation in order to complete the task. Most of these simulation games have graphical user interfaces in which the simulated physical surroundings are displayed, creating a fun, game-like atmosphere. The general hope in the game-based approach is that the additional enjoyment provided by the game features and dynamics will make learning about the particular software engineering process being modeled more memorable, and hence, more effective.

## 2.2 Learning Theories

When discussing and evaluating educational approaches, it is only appropriate that the discussion is tied back to the roots of educational theory: learning theories. Learning theories are theories that describe how people learn. One of the main purposes of learning theories is their use as a guide in evaluating and modifying existing educational approaches, as well as in creating new ones. In this section, we will present and describe some of the most widely accepted and well-known learning theories that are relevant to the domain of software engineering education. We chose the following set of learning theories because of three criteria: wide acceptance across fields beyond software engineering, orthogonality among the factors defining the theory, and relevancy to software engineering. That is, we wanted theories that apply beyond software engineering but still bore hands-on applicability in structuring our methods of teaching (thereby ignoring general theories such as cognitive dissonance, which focus on conflict resolution in the mind [52]), and we wanted to avoid listing numerous theories that vary ever so slightly (those that are similar we implicitly grouped under a single “learning theory”).

One of the most well-known learning theories is Learning by Doing, a theory based upon the premise that people learn a task best not only by hearing about it, but also by actually *doing* it [8, 110, 117, 123, 124]. The implication of this theory for educational approaches is the following: the learner should be provided with ample opportunity to actually *do* what they are learning about, not simply absorb the knowledge through a lecture, book, or some other medium. Furthermore, they should be encouraged to reflect upon their actions through analysis, synthesis, and evaluation activities.

Situated Learning [5, 23, 56, 115, 135, 137] is an educational theory that builds upon the Learning by Doing approach. However, while Learning by Doing focuses on the specific learning activities that the student performs, the Situated Learning theory is concerned with the environment in which the learning by doing takes place. In particular, Situated Learning is based on the belief that knowledge is situated, being in large part a product of the activity, context, and culture in which it is developed and used. Therefore, the environment in which the student practices their newly learned knowledge should resemble, as closely as possible, the environment in which the knowledge will be used in real life.

Like Situated Learning, Keller's ARCS Motivation Theory [81] also focuses on motivating students to learn. However, rather than focusing on the physical environment in which they learn, Keller's ARCS Motivation Theory concerns itself with promoting certain feelings in the learner that motivate them to learn. In particular, these feelings are attention, relevance, confidence, and satisfaction.

- **Attention:** The attention and interest of the learner must be engaged. Proposed methods for doing so are: introducing unique and unexpected events; varying

aspects of instruction; and arousing information-seeking behavior by having the learner solve or generate questions or problems.

- **Relevance:** Learners must feel that the knowledge is relevant to their lives. The theory suggests that knowledge be presented and practiced using examples and concepts that are relevant to learners' past, present, and future experiences.
- **Confidence:** Learners need to feel personal confidence in the learning material. This should be done by presenting a non-trivial challenge and enabling them to succeed at it, communicating positive expectations, and providing constructive feedback.
- **Satisfaction:** A feeling of satisfaction must be promoted in the learning experience. This can be done by providing students with opportunities to practice their newly learned knowledge or skills in a real or simulated setting, and providing positive reinforcements for success.

Anchored Instruction [20] is another theory that deals with teaching techniques. In particular, Anchored Instruction says educators should center all learning activities around an "anchor"—a realistic situation, case study, or problem. Presentation of general concepts and theories should be kept to a minimum. Instead, Anchored Instruction believes that knowledge is best learned by exploration of these realistic case studies or problems.

The Discovery Learning theory [5, 110] takes a similar approach to Anchored Instruction in that it believes that an exploratory type of learning is best. Discovery Learning is based on the idea that an individual learns a piece of knowledge most effectively if they discover it on their own, rather than having it explicitly told to them.

This theory encourages educational approaches that are rich in exploring, experimenting, doing research, asking questions, and seeking answers.

Along the same lines as the Discovery Learning theory is the Learning Through Failure theory [123]. This theory is based on the assumption that the most memorable lessons are those that are learned as a result of failure. Learning through failure also provides more motivation for students to learn, so as to avoid the adverse consequences that they experience firsthand when they do not perform as taught. Failure can also engage students, as they are motivated to try again in order to succeed. Proponents of the theory argue that students should be allowed to (and even set up to) fail to encourage maximal learning.

While most of the learning theories discussed so far focus mainly on the learner as an independent being who is responsible for fostering their knowledge on their own (using the proper learning materials/activities), the Learning through Dialogue theory [38] gives the teacher a much more active and pivotal role in the learner's education. Learning through Dialogue suggests that dialogue between student and teacher is necessary for effective learning and retention. According to the theory, this dialogue should consist of the teacher encouraging reflection, assessing the student's aptitudes and learning style, and tailoring their teaching strategy accordingly.

Like Learning through Dialogue, the Aptitude-Treatment Interaction [41] theory also recommends that the instructor take an active role in assessing the characteristics of the learner and modify their teaching style accordingly. Aptitude-Treatment Interaction focuses primarily on the aptitude of the learner, and states that the learning environment should be tailored to this particular characteristic. Specifically, low-ability learners need

highly-structured learning environments that incorporate a high level of control by the instructor, concrete and well-defined assignments, and specific sequences to follow for completing them. High-ability learners, on the other hand, tend to be more independent, which implies that a less structured approach is more effective for this type of student.

Like the Aptitude-Treatment Interaction theory, the theory of Multiple Intelligences [55] also deals with the diverse learning needs and styles of individuals. However, rather than focusing on the *aptitude* of the learner, the Multiple Intelligences theory is instead focused on the particular learning modalities that are unique to each individual. In particular, the theory identifies seven different learning modalities: linguistic, musical, logical-mathematical, spatial, body-kinesthetic, intrapersonal (metacognition and insight), and interpersonal (social skills). Whenever possible, instruction should be individually tailored to each student to target the particular learning modalities that are most effective for them.

The theory of Learning through Reflection is primarily based on Donald Schön's work suggesting the importance of reflection activities in the learning process [127]. In particular, Learning through Reflection emphasizes the need for students to reflect on their learning experience in order to make the learning material more explicit, concrete, and memorable. Some common reflection activities include discussions, journaling, or dialogue with an instructor [83].

While Learning Through Reflection is primarily concerned with what individuals do with knowledge once they have received it, the theory of Elaboration [113] is focused on how that information is presented to the learner in the first place. In particular, it states that, for optimal learning, instruction should be organized in order of complexity, from

least complex to most complex. Simplest versions of tasks should be taught first, followed by more complicated versions.

The Lateral Thinking theory [46] is concerned with how students are encouraged to think about the information presented. Specifically, Lateral Thinking states that knowledge is best learned when students are presented with problems that require them to take on different perspectives than they are used to and practice "out of the box" thinking. The theory suggests that students be challenged to search for new and unique ways of looking at things, and in particular, these views should involve low-probability ideas that are unlikely to occur in the normal course of events. It is only through this type of relaxed, exploratory thinking that one can obtain a firm grasp on a problem or piece of knowledge.

## 2.3 Software Engineering Educational Approaches and Learning Theories

Table 1 presents the frequency of each software engineering educational approach discussed here, including a breakdown of each approach's subcategories. Looking at the number of approaches that fall into the "Projects Plus Realism" category (53 out of 109 total) and the "Missing Piece" category (48 out of 109), it is obvious that these are the two most popular approaches to addressing the problem of adequately preparing students for their future careers in software engineering. Simulation is by far the category of approach that is least often used.

If we then compare these teaching strategies with the set of learning theories discussed previously, the results are shown in Table 2. An 'X' in the table indicates that there have been approaches within that category that have embodied that theory (either

**Table 1: Frequency and Breakdown of Each Software Engineering Educational Approach.**

| **Realism** | **53** | **Simulation** | **8** | **Missing Piece** | **48** |
|---|---|---|---|---|---|
| **Industrial Partnerships** | **16** | **Industrial** | **2** | **Formality** | **3** |
| - Modify real software | 1 | **Game-Based** | **4** | - Formal methods | 2 |
| - Industrial advisor | 1 | **Group Process** | **2** | - Engineering | 1 |
| - Industrial mentor/lecturer | 2 | | | **Process (Specific)** | **21** |
| - Case study | 5 | | | - Personal Software Process | 14 |
| - Real project / customer | 7 | | | - Team Software Process | 2 |
| **Maintenance/Evolution** | **9** | | | - Rational Unified Process | 3 |
| - Multi-semester | 4 | | | - Extreme Programming | 2 |
| - Single-semester | 5 | | | **Process (General)** | **6** |
| **Team Composition** | **13** | | | - Process engineering | 3 |
| - Long-term teams | 1 | | | - Project management | 3 |
| - Large teams | 3 | | | **Parts of Process** | **3** |
| - Different C.S. classes | 1 | | | - Scenario-based requirements | 1 |
| - Different majors | 2 | | | - Code reviews | 1 |
| - Different universities | 2 | | | - Usability testing | 1 |
| - Different countries | 1 | | | **Types of Software Eng.** | **8** |
| - Team structure | 3 | | | - Maintenance/Evolution | 3 |
| **Non-Technical Skills** | **2** | | | - Component-based SE | 2 |
| **Open-Endedness** | **7** | | | - Real-time SE | 3 |
| - Requirements | 2 | | | **Non-Technical Skills** | **7** |
| - Process | 5 | | | - Social/logistical skills | 3 |
| **Practice-Driven** | **3** | | | - Interact w/ stakeholders | 1 |
| **Sabotage** | **2** | | | - Human-Computer Interaction | 2 |
| **Project Failures** | **1** | | | - Business aspects | 1 |

accidentally or deliberately), and a ‘P’ represents that there is an obvious potential for that particular type of approach to employ that learning theory (in and of itself, not combined with any other approach), but there have been no known cases of it. The presence of both an ‘X’ and a ‘P’ indicates that perhaps one or two approaches in the category have taken advantage of the theory, but most have not, so there is significant potential for further exploitation. (See [99] for a more thorough explanation of this categorization).

The first eight rows of results illustrate the correlation between learning theories and advances in the eight subcategories of the “realism” category. It should be clear that, although all learning theories are covered, each approach only covers a subset of the

**Table 2. Learning Theories and Different Software Engineering Educational Approaches.**

| | *Learning by Doing (and similar) [117]* | *Situated Learning (and similar) [23]* | *Keller's ARCS [81]* | *Anchored Instruction [20]* | *Discovery Learning [5]* | *Learning Through Failure [123]* | *Learning Through Dialogue [38]* | *Aptitude-Treatment Interaction [41]* | *Learning Through Reflection [127]* | *Elaboration [113]* | *Lateral Thinking [46]* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Industrial Partnerships** | X | X | X | | | | X/P | | X/P | P | |
| **Maintenance / Evolution** | X | X | | | | | P | | P | P | |
| **Team Composition** | X | X | | | | | P | | P | X/P | P |
| **Open-Endedness** | X | X | X | | X | X | P | | P | | |
| **Non-Technical Skills** | X | X | | | | | P | | P | P | |
| **Practice-Driven** | X | | | X | X | X | X/P | P | X/P | P | |
| **Sabotage** | X | X | | | | X | P | | P | P | |
| **Project Failures** | X | X | | | | X | P | | P | P | |
| **Missing Piece** | X | | | | | | | | | | |
| **Simulation** | X | X | X | P | X | X | X/P | P | X/P | X/P | X/P |

surveyed learning theories. Approaches of the "missing piece" variety are worse off (and therefore grouped together). Because these approaches tend to focus on exposing students to a particular technology or topic, little time is spent in framing such exposures in learning theories. Exposure itself is typically considered a sufficient advance in and of itself.

What is interesting to this dissertation, however, is the relationship between simulation and learning theories: all of the theories considered apply in some way or another. While it certainly is not the case that any teaching method that addresses more learning theories than another is better than that other method (consider a haphazard combination of strategies put together in some teaching method versus one well-thought-out and tightly-focused method cleverly leveraging one very good strategy), an approach that naturally addresses factors and considerations of multiple learning theories is one that is most definitely worth exploring. Simulation is such an approach, but one that, as

we have seen, has been significantly under-explored in the field of software process and software engineering in general—something that our approach aims to correct.

# 3. Approach

This dissertation is based on the hypothesis that simulation can bring to software engineering education many of the same benefits it has brought to other educational domains. Specifically, we believe that software engineering *process* education can be improved by using simulation to allow students to practice managing different kinds of "realistic" software engineering processes. As discussed in Chapter 1, software process is a key part of software engineering that is not adequately addressed in typical software engineering educational approaches. The constraints of the academic environment prevent students from having the opportunity to practice many issues surrounding the software engineering process. Accordingly, our approach focuses on providing this opportunity through the use of a new educational software engineering simulation environment, SimSE.

As simulation environments have become widely recognized as educationally beneficial and thus, have become a standard part of many curricula, there is a significant body of experience that can be drawn from in developing a new educational simulation approach. Rather than focusing on individual projects, we discuss collective lessons learned from these projects—lessons that identify some of the critical success factors for educational simulations, and thus, have driven the development SimSE:

- *Simulation must be used complementary to existing teaching methods.* It is important to introduce topics in class lectures first in order to create a basic set of knowledge and skills that students use during simulations. Similarly, it is important to carry out class projects for the sake of Learning by Doing [117], since having hands-on confirmation of at least a few of the lessons learned

during simulation make these lessons that much more powerful and believable [51].

- *Simulation must provide students with a clear goal.* Precisely defined objectives not only guide students through a simulation, but also pose a challenge that many students find hard to resist. Achieving the goal becomes a priority and Discovery Learning [110] is employed as creative thinking is sparked in coming to an approach that eventually achieves that goal [93, 112].
- *Simulation must start with simple tasks and gradually move towards more difficult ones.* In line with the Elaboration learning theory [113], in order for simulation to be effective over multiple sessions, students first must become familiar with a simulation environment and achieve some early and successful results. Otherwise, they quickly become disenchanted and are not likely to complete any kind of larger simulation task [51].
- *Simulation must be engaging.* In order to retain the attention of students, a simulation should provide them with interesting situations to be addressed that are adequately challenging (making it likely that they learn through failure at times) but not impossible, promoting eventual success that leads to confidence in the learning material and satisfaction in the experience. Moreover, it should sometimes provide surprising twists and turns, and have a visually interesting user interface that grabs the student's attention [51]. As stated in the Keller's ARCS learning theory [81], combining all of these qualities results in a learning experience that is highly motivating for the student.

- *Simulation must provide useful feedback on a regular basis.* One of the common mistakes in using simulation for educational purposes is to not provide feedback until the end of a simulation. Research has demonstrated that intermediate feedback is at least as important in contributing to an effective learning experience [6, 51, 96].
- *Simulation must be accompanied by explanatory tools.* Simulation relies heavily on independent learning: students draw their own conclusions regarding the relationship between their inputs and the resulting outputs. To aid in this process, explanatory tools must help illustrate and elucidate the cause and effect relationships triggered by student input [32].

Adherence to these six guidelines establishes simulation environments and broader educational approaches that promote effective learning, enhance a student's knowledge and skills in a fun way [112], and are known to increase the interest, education, and retention rate of students [29, 76].

## 3.1 Research Questions

It was these lessons for successful educational simulations that drove and helped shape our approach to using simulation in the domain of software engineering education. In particular, we applied these lessons to our particular domain (software engineering education) to formulate the following research questions, which have guided the development of our approach:

1. **Can an educational software engineering simulation environment that is rooted in principles for effective educational simulations be built?** In other words, can we successfully apply these principles to the domain of software

engineering education to create a simulation environment that follows these principles? Is it possible to create a maximal combination of all of the desired qualities, or are there tradeoffs that must be made between them?

2. **Can students actually learn software process concepts from using such an environment?** As the ultimate goal of such a simulation environment is for students to learn certain lessons from it, it is crucial to determine whether this goal is achieved.
3. **If students can learn software process concepts from using such an environment, how does the environment facilitate the learning of these concepts?** Answering this question can provide insights into the learning process students undergo when using such an environment, which can inform future work in educational simulation in software engineering, as well as in educational simulation environments in general. Moreover, it can validate whether or not the learning theories that simulation environments are thought to embody are actually employed by students who use them.
4. **How can such an environment fit into a software engineering curriculum?** Does it work well as a voluntary, extra-credit, or mandatory exercise? How much guidance is needed, both by the game itself and by the instructor, and how much should the students be required to figure out by themselves through independent learning?

## 3.2 Key Decisions

To answer these research questions, we studied the domain of software engineering education to discover what its unique needs are, and combined these with the principles

for successful educational simulations. Through this combination we designed a new educational simulation approach that relies on the following key decisions, which characterize it and set it apart from existing approaches:

1. **Construct our simulation approach as a game.** We could have chosen to base our simulation approach on the industrial simulation or group process simulation paradigms described in Section 2.1.3, but instead we chose the game paradigm. As one of the successful educational simulation principles states, there is a clear link between the level of engagement of an educational exercise and its effectiveness [51]. We deliberately chose to capitalize on this and the interest in computer games that is typical of college-age students by giving our simulation approach a distinct game-like feel. In designing our simulation environment and its simulation models, we made liberal use of graphics, interactivity, interesting, life-like challenges, and other game-like elements such as humorous employee descriptions and dialogues, and surprising random events. Moreover, the game paradigm allows us to naturally follow the principle of providing students with a clear goal: a game is, in essence, a set of precisely defined objectives that a player is asked to achieve in a game world.
2. **Create our simulation approach with a fully graphical user interface.** To further adhere to the principle that educational simulations must be engaging, we chose to design a fully graphical, rather than textual interface. The focal point of this interface is a typical office layout in which the simulated process is "taking place", including cubicles, desks, chairs, computers, and employees who "talk" to the player through pop-up speech bubbles over their heads (see

Figure 1). In addition, graphical representations of all artifacts, tools, customers, and projects along with the status of each of these objects are visible. Besides holding the attention of the learner, being able to see simulated software engineering situations portrayed graphically also leverages the theory of Situated Learning—the learner is provided with a visual context that corresponds to the real world situations in which the learned knowledge would typically be used [23].

3. **Make our simulation approach highly interactive.** Keeping the interest of the learner engaged is not only done by making a user interface visually appealing, but also by involving the learner continually throughout the simulation. Thus, rather than designing our simulation approach as a continuous simulation that simply takes an initial set of inputs and produces some predictive results, we have designed it in such a way that the player must make decisions and steer the simulation accordingly throughout the entire simulated process. Our simulation approach operates on a step-by-step, clock tick basis, and every clock tick the player has the opportunity to perform actions that affect the simulation. Not only does this continuous interaction with the simulation keep the player engaged, but it allows us to follow another educational simulation principle: provide useful feedback on a regular basis. Every clock tick, the player has the opportunity to receive feedback about their performance through specialized feedback mechanisms we have built into our simulation environment.
4. **Create a simulation approach with customizable simulation models.** This feature was primarily necessitated by the unique needs of the domain of

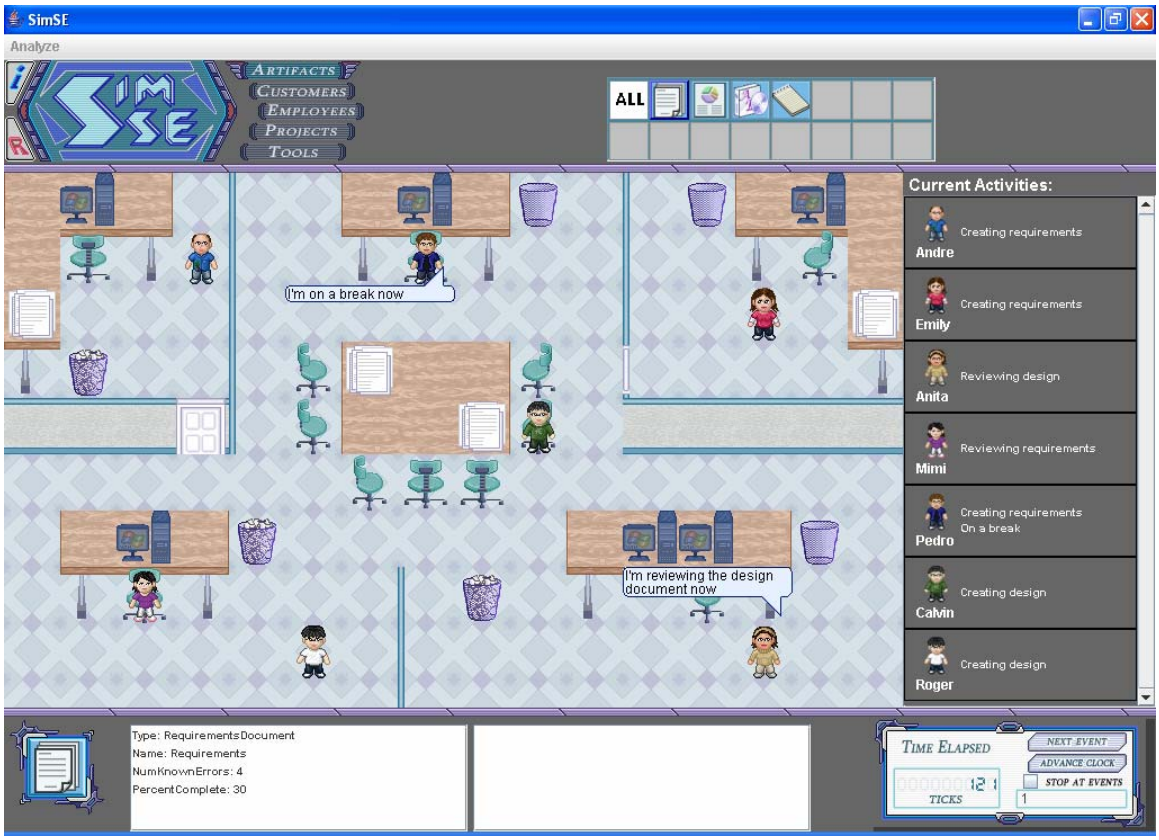


**Figure 1: Graphical User Interface of SimSE.**

software engineering education. Multiple software process models exist and are regularly taught in software engineering courses. Thus, one of our chief goals in the design of our simulation approach was to facilitate the modeling and simulation of different software process models. Having a customization feature also allows for models of different complexities to be built so that the principle of starting with simple simulation tasks and gradually moving towards more difficult ones can be followed. This customization was accomplished through the inclusion of a model builder tool and associated modeling approach that allow an instructor to build simulation models and generate customized games based on these models.

5. **Create a new modeling approach for creating graphical, interactive, game-based software process models.** Developing a game-based simulation approach that was also customizable required us to create a new modeling approach that was specifically tailored to the needs of our environment. In particular, our approach had to support the modeling of game-based, graphical, interactive models that are both predictive (i.e., can predict and execute the effects of player actions) and prescriptive (i.e., can specify a set of allowable next steps that the player can take), a combination that has not been achieved by other software process modeling approaches to date.
6. **Design a modeling approach that is deliberately more specific than other general-purpose software process modeling approaches for the sake of a simpler and more straightforward model building process.** A careful balance between flexibility and specificity was orchestrated to create a modeling approach that adequately meets the needs of our particular domain and targeted audience—software engineering instructors.
7. **Root our simulation models in results from the research literature.** We collected the rules and lessons we have encoded into our simulation models by scouring the research literature to discover what is commonly believed and taught about software engineering processes. Although most of this does not include hard numbers that are able to be directly encoded into a simulation (e.g., "integration is 65% faster when there is a design document" versus simply, "integration is faster when there is a design document"), we were able to incorporate rules such as these into our models by experimenting with different

values to come up with ones that are effective enough at conveying each particular lesson in the simulation (see Section 4.2).

8. **Construct simulation models that teach by rewarding good software engineering practices and penalizing bad ones.** Because our simulation approach is a game, the goal of the player is to "win" the game by attaining a good score. Although it depends to a large degree on the simulation model being used, our environment is designed with the intent that players receive a good score when they follow proper software engineering practices and a bad score when they deviate from them. In this way the player can discover the lessons being taught by associating their (high or low) score with the actions they took and infer which ones are good practices and which ones are not. In addition to the score received at the end of the game, players are also rewarded or penalized throughout the game through various forms of intermediate feedback. For example, a player who skips requirements and goes straight to design will immediately see that design is slow and the design document is full of errors, hinting that skipping requirements was not the proper thing to do.
9. **Include an explanatory tool as part of the simulation environment.** We have directly implemented the principle for successful educational simulations which states that simulation must be accompanied by explanatory tools. An integral part of SimSE is its novel explanatory tool that provides players with a visual representation of how the simulated process progressed over time and explanations of the rules underlying the game.

10. **Use and evaluate our simulation approach in a classroom setting.** Because one of the educational simulation principles states that simulation should be used complementary to existing teaching methods, a fundamental part of our approach is to use our simulation environment in conjunction with actual courses, so that it can be evaluated in the context of its ideal and intended usage.

## 3.3 Detailed Approach

These key decisions translate into the following three-part approach: (1) building a graphical, interactive, educational, customizable, game-based simulation environment for software processes (SimSE), (2) developing a set of simulation models to be used in seeding the environment, (3) evaluating the usage of the environment, both in actual software engineering courses, and in formal out-of-class experiments to gain understanding of its various educational aspects.

The first part of our approach is SimSE, an educational software engineering simulation environment. SimSE is a single-player game in which the player takes on the role of project manager of a team of developers. The player is given a software engineering task to complete, which is generally a particular (aspect of a) software engineering project. In order to complete this task, they must perform various management activities such as hiring and firing, assigning tasks, monitoring progress, purchasing tools, and responding to (sometimes random) events, all through a graphical user interface that visually portrays all of the employees and the office in which they work (see Figure 1). In general, following good software engineering practices will lead to positive results while ignoring these practices will lead to failure in completing the project.

As stated in Section 3.2, one of the fundamental goals of SimSE is to allow customization of the software processes it simulates. Thus, its architecture was designed to support this customization, as can be seen in Figure 2. An instructor uses the model builder tool to create a simulation model that embodies the process and lessons they wish to teach their students. The generator component interprets this model and automatically generates Java code for a state management component, a rule execution component, a simulation engine, an explanatory tool, and the graphical user interface, which comprise the simulation environment. A student uses this custom-generated environment to practice the situations captured by the model.

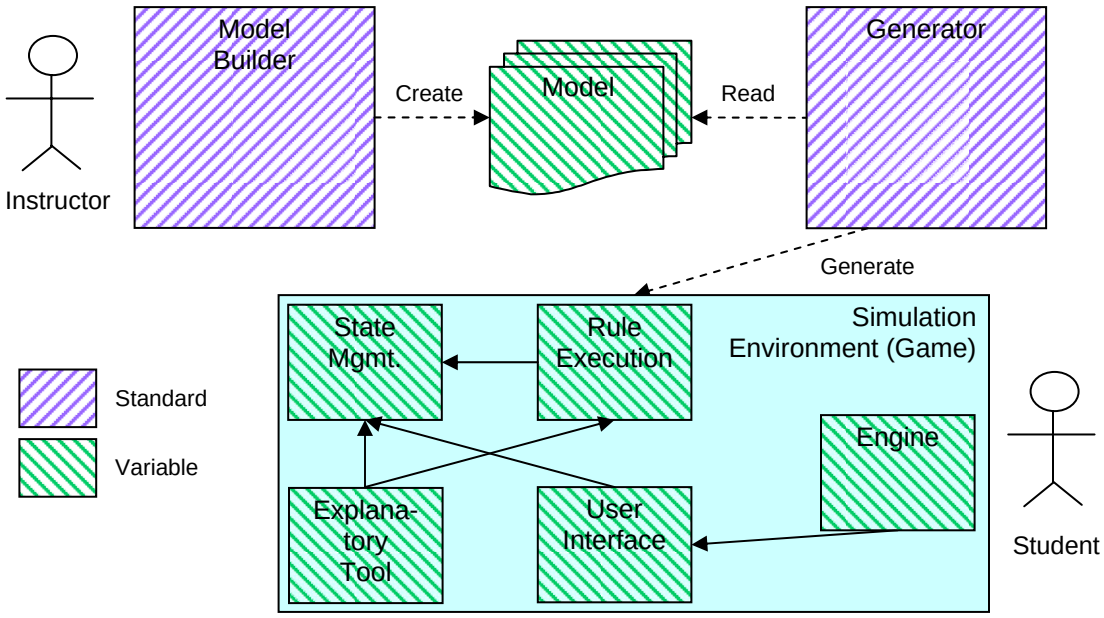


**Figure 2: SimSE Architecture.**

In order to aid students in understanding the process being simulated and how their actions during the game affect it, a critical part of this simulation environment is the

explanatory tool. This is a tool that the student can run at the end of a game to view a trace of events, rules, and attribute values that were recorded during the game.

To provide a set of models to be used in the simulation environment, as well as to test and refine the environment's model-building capacities, the second part of our approach is a set of six simulation models that each portray a different software engineering process (or sub-process). Together, these models represent a wide spectrum of different software processes that vary in size, scope, and purpose. They comprise a library of models that can be either used as-is, or modified to meet the needs of a particular situation and/or instructor.

The third and final part of our approach is a set of evaluations designed to determine the educational effectiveness of SimSE from various angles. These include both usage of SimSE in conjunction with a software engineering course, and a series of formal experiments done in controlled settings. Each evaluation was designed to assess a different aspect of our approach so that collectively, the results could be used to make conclusions about the overall educational effectiveness of SimSE.

# 4. Modeling/Simulation Capabilities

Motivated by our key decision to make SimSE's simulation models customizable, the first step in designing SimSE was determining exactly what kinds of things it should be able to model. Therefore, before going into detail on the game play aspects and inner workings of SimSE in later chapters, we will first present the modeling and simulation capabilities of SimSE.

As a first step in determining what an educational software engineering simulation environment would have to model and simulate, we performed a survey of existing software engineering literature, talked to software engineering professionals, perused the lecture notes and textbooks for the introductory software engineering classes at UC Irvine, and looked at other software engineering simulations to see what kinds of phenomena they model. The result of these activities is a compendium of 86 "fundamental rules of software engineering" (see Appendix A) that have driven the design of SimSE's modeling and simulation capabilities. The following is a representative sample of the breadth of lessons that comprise these rules.

1. *In a waterfall model of software development, do requirements, followed by design, followed by implementation, followed by integration, followed by testing* [134].
2. *At the end of each phase in the waterfall model, perform quality assurance activities (e.g., reviews, inspections), followed by correction of any discovered errors. Otherwise, errors from one artifact will be carried over into subsequently developed artifacts* [134].
3. *If you do not create a high quality design, integration will be problematic* [134].

4. *Developers' productivity varies greatly depending on their individual skills, and matching the tasks to the skills and motivation of the people available increases productivity* [18, 26, 121].
5. *The greater the number of developers working on a task simultaneously, the faster that task is finished, but more overall effort is required due to the growing need for communication among developers* [22].
6. *Software inspections find a high percentage of errors early in the development life cycle* [141].
7. *The better a test is prepared, the higher the amount of detected errors* [134].
8. *The use of software engineering tools leads to increased productivity* [134].
9. *The average assimilation delay, the period of time it takes for a new employee to become fully productive, is 80 days* [2].
10. *In the absence of schedule pressure, a full-time employee allocates, on average, 60% of his working hours to the project (the rest is slack time: reading mail, personal activities, non-project related company business, etc.)* [2].

The compendium as a whole covers a broad variety of rules—rules that agree with each other, rules that conflict with each other, rules that are precise, rules that are imprecise, rules that cover issues specific to software engineering, and rules that apply to a wide range of business processes. While this is certainly not a comprehensive set of *all* existing software engineering rules and processes, together they form a representative set that can be selected from as necessary to form different software process simulation models.

Our next step in designing a modeling approach was choosing several of these rules to incorporate into a preliminary prototype version of SimSE. These rules were selected based on a desire to cover several of the different dimensions present in the compendium, as well as the need to form a cohesive model of a software engineering process. The resulting version of SimSE was highly simplified compared to the current version in two major ways: First, it was non-graphical, using only tables, text boxes, and drop-down lists to portray the process to the player (see Figure 3). Second, it was non-customizable. The set of rules that we incorporated into this version were hard-coded and could not be modified except through changing the source code of the simulation. Moreover, the number of rules included in this version was smaller than many of our current models—enough to demonstrate the feasibility of the approach but not so many as to require a large amount of unnecessary effort in programming this preliminary prototype. Basically, this model was a simplified version of our current waterfall model (see Section 7.1), including only its core set of rules and simplified versions of its objects and actions.

After completing development of this non-graphical prototype, we then informally tested it out by observing a group of graduate students playing it. From this we gathered useful feedback that gave us good ideas to incorporate into the current version of SimSE (and also gave us confidence that this prototype was playable).

After determining that building an educational software engineering simulation based on the types of rules we collected was feasible, we proceeded to abstract away from the hard-coded model the generic constructs that would be needed to model this and other software processes—constructs that would allow a user to choose and build different sets of rules into different models. These constructs are described in detail in the following.

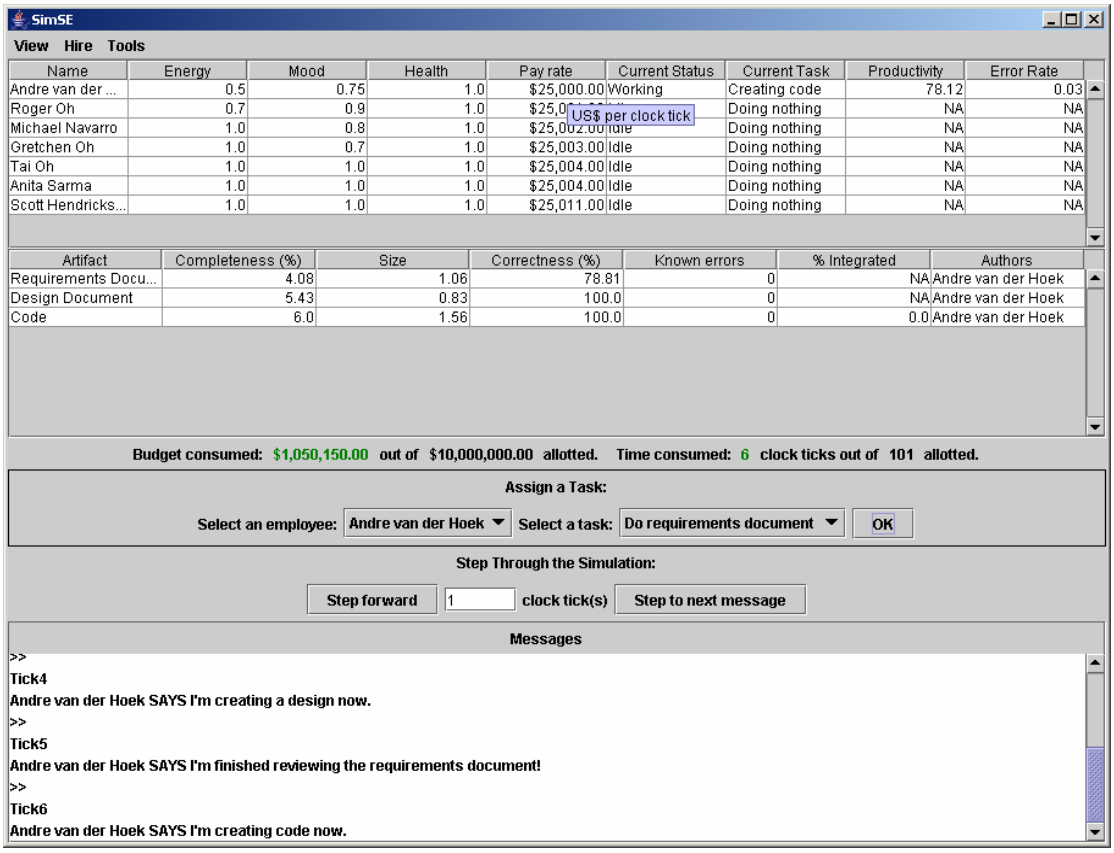


**Figure 3: SimSE Non-graphical Preliminary Prototype User Interface.**

## 4.1 Modeling Constructs

A SimSE model consists of five parts. *Object types* define templates for all objects that participate in the simulation. The *start state* of a model is the collection of objects present at the beginning of the simulation. Each object in the start state instantiates an object type. Start state objects participate in a*ctions*, which are the activities represented in the simulated process. Each action has one or more r*ules* that define the effects that action has on the rest of the simulation. Each object in the simulation is represented by g*raphics*, which also provide visualizations of the relevant actions occurring in the simulation. Figure 4 illustrates the relationships between the different parts of a model. The following subsections discuss each of these parts of the overall modeling approach in further detail.

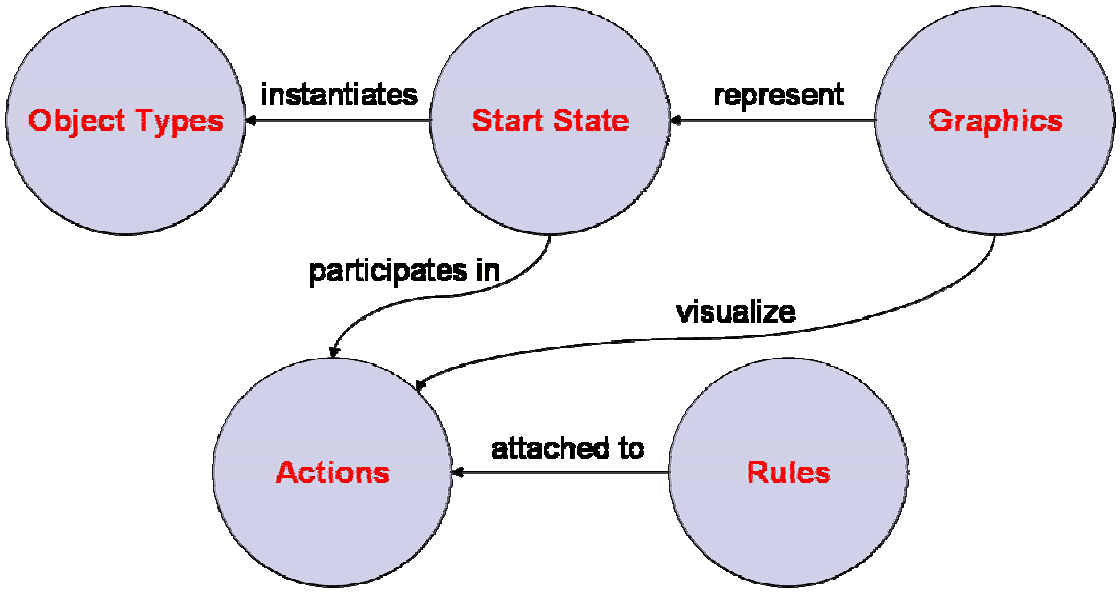


**Figure 4: Relationships Between Modeling Constructs.**

### 4.1.1 Object Types

The core of a SimSE model is the set of object types to be used in the model. Each major entity participating in the simulation is an instantiation of an object type. Every object type defined must descend from one of five *meta-types*: Employee, Artifact, Tool, Project, or Customer. Each of these meta-types has very limited semantics in and of itself, except for where objects of each type are displayed in the GUI of the simulation, and how the player can interact with each type of object. Specifically, only objects descending from Employee will display overhead pop-up messages during the game and have right-click menus associated with them so the player can command their activities.

An object type consists of a name and a set of typed attributes. For each attribute, in addition to the name and type (String, Double, Integer, or Boolean), the following metadata must be specified:

- *Meta-type:* whether this object type is an Employee, Artifact, Tool, Project, or Customer.
- *Key*: a Boolean value indicating whether or not this attribute is the key attribute for the object type.
- *Visible*: a Boolean value denoting whether this attribute should be visible to the player throughout the game.
- *VisibleAtEnd*: a Boolean value indicating whether or not this attribute should be visible at the end of the game. An attribute that was hidden throughout the game but revealed at the end can give further insight to the player about why they received their particular score.
- *MinVal*: the minimum value for this attribute (for Double and Integer attributes only).
- *MaxVal*: the maximum value for this attribute (also for Double and Integer attributes only).
- *MinDigits*: the minimum number of digits after the decimal point to display for this attribute's value (for Double attributes only).
- *MaxDigits*: the maximum number of digits to display (also for Double attributes only).

Three sample object types, a "Programmer" of type Employee, a "Code" of type Artifact, and an "SEProject" of type Project are shown in Figure 5. If we take a closer look at one of these, the Code object type, we can see how this metadata is used in practice. A Code artifact has a name, which is its key value, to distinguish it from other Code objects. It also has two types of error attributes: unknown errors

```
Programmer Employee
{
  name:
    type: String
    key: true
    visible: true
    visibleAtEnd: true
  energy:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: 1.0
    minDigits: 1
    maxDigits: 2
  productivity:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: 1.0
    minDigits: 1
    maxDigits: 2
  errorRate:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: 1.0
    minDigits: 1
    maxDigits: 2
  hired:
    type: Boolean
    key: false
    visible: true
    visibleAtEnd: true
  payRate:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 2
    maxDigits: 2
}
```

```
Code Artifact
{
  name:
    type: String
    key: true
    visible: true
    visibleAtEnd: true
  numUnknownErrors:
    type: Double
    key: false
    visible: false
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 0
    maxDigits: 0
  numKnownErrors:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 0
    maxDigits: 0
  size:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 1
    maxDigits: 1
  percentComplete:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: 100.0
    minDigits: 1
    maxDigits: 1
  }
```

```
SEProject Project
{
  description:
    type: String
    key: true
    visible: true
    visibleAtEnd: true
  requiredSizeOfCode:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 0
    maxDigits: 0
  budget:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 0
    maxDigits: 2
  allottedTime:
    type: Integer
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
  score:
    type: Integer
    key: false
    visible: false
    visibleAtEnd: true
    minVal: 0
    maxVal: 100
  }
```

**Figure 5: Programmer, Code, and Project Object Types.**

("numUnknownErrors"), which are those that the developers have not discovered, and known errors ("numKnownErrors"), which are those that the developers have discovered.

The known errors are visible during the game, while the unknown errors are hidden during the game, since this is a value that would not be known in a real life software engineering situation. However, to give the player some insight into where they might have gone wrong in the process, the unknown errors are revealed at the end of the game (*visibleAtEnd* is equal to true). Both the unknown errors and the known errors are stored as Doubles but displayed to the player as Integers, as both have a *maxDigits* value of 0 (meaning no digits after the decimal place are shown). This is done so that errors can be added, "discovered", and removed fractionally behind the scenes, but appear to the player as if these values are changing as they would in a real-life situation, by whole numbers. For example, if developers are inspecting a Code artifact, their productivities might dictate that they only discover ¼ of an error per clock tick. Thus, one whole error would be discovered after four clock ticks, at which point the player would see the number of known errors increase by one (hidden digits are truncated and resulting digits are not rounded off).

In addition to these attributes, the Code artifact in Figure 5 also has a size attribute and a percent complete attribute, which are both visible to the player throughout the simulation. The percent complete attribute has a minimum value of 0 and a maximum value of 100 to enforce the standard percentage values of 0 to 100.

It should be noted that the format of this example and the examples throughout this chapter are shown in a "shorthand" version of the actual SimSE modeling language format, which is XML-like and difficult to read. However, since this language is completely hidden from the user by our model building tool, we have accordingly

omitted it from this dissertation. (See Chapter 5 for a presentation of the model builder tool.)

### 4.1.2 Start State

The start state refers to the set of objects that are present when the simulation begins. Each one of these objects must be an instantiation of one of the object types defined for the model, and starting values for all attributes must be assigned—no default values are automatically given. Figure 6 shows sample instantiated objects for the "Programmer", "Code", and "SEProject" object types from Figure 5. The first object is a high-energy (0.9 out of 1.0) Programmer Employee named Roger, with moderate productivity (0.6 out of 1.0) and a relatively low error rate (0.3 out of 1.0), who makes $100 per clock tick and is currently hired. The second object is a Code Artifact named "My Code" that seems to have already been developed some. It has 18 unknown errors, 7 known errors, a size of 25,600, and a completeness level of 10%. The final object is an "SEProject" Project described as "Rocket Launcher Software", with a required code size of 256,000 (hence the 10% completeness of the code with a size of 25,600), a budget of $2,500,000, no money spent, an allotted time of 692, no time used, and score (which represents the current score for the player of the game) of 0.

### 4.1.3 Actions

The actions in a SimSE model represent the set of activities in which the objects in the simulation can participate. For example, to model a situation in which programmers are building a piece of code using an integrated development environment (IDE), one would create a "Coding" action, in which the participants include a "Code" Artifact, one or

```
Object Programmer
Employee
{
  name = “Roger”
  energy = 0.9
  productivity =
    0.6
  error rate = 0.3
  hired = true
  payRate = 100.00
}
```

```
Object Code Artifact
{
  name = “My Code”
  numUnknownErrors =
    18
  numKnownErrors = 7
  size = 25600.0
  percentComplete =
    10.0
}
```

```
Object Project
SEProject
{
  description =
    “Rocket Launcher
    Software”
  requiredSizeOfCode =
    256000
  budget = 2500000.00
  moneySpent = 0.00
  allottedTime = 692
  timeUsed = 0
  score = 0
}
```

**Figure 6: Instantiated Programmer, Code, and SEProject Objects.**

more “Programmer” Employees and one or more “IDE” Tools. As another example, an Employee of any type could participate in a “Break” action, referring to the activity of taking a break, during which he or she rests and does not work. These two examples are shown in Figures 7 and 8, respectively, and will be referred to throughout the remainder of this subsection.

For each action, the following information is specified:

- *Name*: name of the action (e.g., “Coding” or “Break”).
- *VisibleInSimulation*: whether or not the player should be able to see that the action is occurring during the simulation (in the “Current Activities” pane on the right-hand side of the user interface), and, if true, a short textual description of that action to display in the game’s user interface. This value is true for both the “Coding” and “Break” actions, as these are actions that, in a real-life situation, situation, would be visible to a project manager. An example of an action that would typically not be visible would be an “UpdateProjectAttributes” action that occurs every clock tick and simply updates the time used and money spent. Obviously seeing that this sort of action

```
Action Coding
{
  VisibileInSimulation: true
  SimulationDescription: “Creating code”
  VisibleInExplanatoryTool: true
  ExplanatoryDescription: “Software
    engineers create a piece of code.”

  Participant Coder
  {
    quantity: at least 1
    allowableTypes: Programmer, Tester
  }

  Participant CodeDoc
  {
    quantity: exactly 1
    allowableTypes: Code
  }

  Participant IDE
  {
    quantity: at most 1
    allowableTypes: Eclipse, JPad
  }

  Trigger userTrigger
  {
    type: User-initiated
    menuText: “Start coding”
    overheadText: “I’m coding now!”
    game-ending: false
    priority: 8
    conditions
    {
      Coder:
        Programmer:
          hired == true
        Tester:
          hired == true
          health >= 0.7

      IDE:
        Eclipse:
          purchased == true
        JPad:
          purchased == true
    }
  }
```

```
Destroyer autoDestroyer
{
  type: Autonomous
  overheadText: “I’m finished
    coding!”
   game-ending: false
   priority: 10
   conditions
   {
     percentComplete == 100
   }
}

Destroyer userDestroyer
{
   type: User-initiated
   menuText: “Stop coding”
   overheadText: “I’ve
   stopped coding.”
   game-ending: false
   priority: 11
   conditions {}
 }
}
```

**Figure 7: Sample “Coding” Action with Associated Triggers and Destroyers.**

is taking place would take away from the realism of the environment and would not be of any use to the player so it would be best to keep it invisible.

```
Action Break
{
  VisibileInSimulation: true
  SimulationDescription: “On a Break”
  VisibleInExplanatoryTool: true
  ExplanatoryDescription: “The employee
    rests and does no work in order to
    regain his/her energy.”

  Participant Breaker
  {
    quantity: exactly 1
    allowableTypes: Programmer, Tester
  }

  Trigger autoTrigger
  {
    type: Autonomous
    overheadText: “I’m taking a break now!”
    game-ending: false
    priority: 2
    conditions
    {
      Coder:
        Programmer:
          hired == true
        energy <= 0.2
        Tester:
          hired == true
        energy <= 0.2
     }
  }
```

```
Destroyer autoDestroyer
{
  type: Autonomous
  overheadText: “I’m going
    back to work now!”
  game-ending: false
  priority: 1
  conditions
  {
    Coder:
      Programmer:
        energy == 1.0
      Tester:
        energy == 1.0
  }
 }
}
```

**Figure 8: Sample “Break” Action with Associated Trigger and Destroyer.**

- *VisiblelnExplanatoryTool*: whether or not the player should be able to see occurrences of the action when running the explanatory tool, and, if true, a more detailed description of that action to display in the explanatory tool user interface. Both the “Coding” and “Break” actions are denoted as visible in the explanatory tool since it would be useful for the player to view these actions in the context of the explanatory tool. At first glance it may seem that any action that is visible in the simulation should be visible in the explanatory tool and vice-versa. However, there are some cases where it is useful to make an action invisible in the simulation and visible in the explanatory tool, typically when it

is an action that would not necessarily be visible to a project manager in real-life, but is appropriate for the player to see for educational reasons. An example of this is an action named "DoubleProductivity" in our code inspection model (see Section 7.6). This action is triggered autonomously whenever the ideal number of people (four) are participating in a code inspection [145], and has the effect of doubling the productivity of the inspection, causing bugs to be found twice as fast. So as not to give too much away during game play and maintain the realism of the simulation, this action is hidden during the simulation but revealed in the explanatory tool.

- *Participant(s)*: roles in the action that can be filled by one or more objects of one or more possible object types. In the "Coding" action there are three participants: (1) "Coder" (the person(s) working on the code), which can be filled by one or more Programmer and/or Tester Employees; (2) "CodeDoc" (the code artifact being worked on), which must be filled by exactly one Code Artifact; and (3) "IDE" (the integrated development environment being used for coding), which can be filled by at most one Eclipse or JPad tool. The "Break" action consists of only one participant: the "Breaker", exactly one employee of type Programmer or Tester that is taking the break.
- *Trigger(s)*: what causes the action to begin to occur in the simulation. Three distinct classes of triggers exist: *autonomous*, *user-initiated*, and *random*. Autonomous triggers specify a set of conditions (based on the attributes of the participants in the action) that cause the action to automatically begin, with no user intervention. For instance, in the "Break" action, the employee

automatically takes a break when his or her energy level drops to 0.2 or below. User-initiated triggers also specify a set of conditions, but include a menu item text string, which will appear on the right-click menu for an Employee when these conditions are met. This menu item corresponds to this action, and when the menu item is selected, the action begins. For example, in the "Coding" action, a menu item with the text "Start coding" will appear on the menus of all "Programmer" and "Tester" Employees who meet the specified conditions (hired and, for testers, health level greater than or equal to 0.7). When this menu item is selected by the player, the action will begin. Random triggers provide the opportunity to introduce some chance into the model, specifying both a set of conditions and a frequency that indicates the likelihood of the action occurring whenever the specified conditions are met. For instance, a "Quit" action might have a 75% chance of occurring every clock tick that an Employee's energy level is below 0.1, meaning that employees are likely to quit when they have been worked too hard, but may not always do so. As another example, a random trigger with a very small frequency (e.g., 0.5%) might be attached to an action that causes a rare disastrous event to occur, such as a catastrophic system failure that results in a significant portion of the project being lost. Finally, for every trigger that has one or more Employee participants, the modeler can specify overhead text that will appear to come from the employees participating in the action when the trigger executes. For the "Coding" action this text is "I'm coding now!" and for the "Break" action the employee will announce, "I'm taking a break now!"

- *Destroyer(s)*: An action destroyer works in a manner similar to an action trigger, but has the opposite effect: whereas a trigger *starts* an action, a destroyer *stops* an action. Destroyers can be of the same types as triggers (autonomous, random, or user-initiated), but have one additional type: *timed*. A timed destroyer specifies a "time to live" value for an action—once an action starts, it exists for a number of simulation clock ticks equal to this value, and is then automatically destroyed. The "Coding" action has associated with it two destroyers: an autonomous one that will cause the action to stop when the code is 100% complete, and a user-initiated one that allows the player to make the action cease at any time they wish, by choosing the "Stop coding" menu option. These destroyers have different overhead text associated with them to distinguish the different scenarios—"I'm finished coding!" indicates that the code is complete and "I've stopped coding" indicates that they have simply stopped the activity but have not necessarily completed the task. The "Break" action has only one destroyer—an autonomous one that causes the break to end when the employee's energy level is back up to its maximum value (1.0), at which point the employee will announce, "I'm going back to work now!"

Triggers and destroyers have two additional pieces of information associated with them: *priority* and *game-ending*. The priorities of triggers and destroyers determine the order in which each trigger/destroyer will be checked, and, if all conditions are met, executed. All triggers in a model are prioritized in relation to all other triggers, and are checked in ascending order of priorities, e.g., one is the highest priority. Analogously, all destroyers are prioritized in relation to all other destroyers, and are also checked in

ascending order. So that the order of execution is always deterministic, no two triggers or destroyers can have the same priority. It is not required that triggers and destroyers be prioritized—non-prioritized triggers/destroyers will execute in an undetermined order, after all of the prioritized triggers/destroyers have executed in their specified ordering.

In the "Coding" action, the autonomous destroyer ("autoDestroyer") has priority 10, while the user-initiated destroyer ("userDestroyer") has priority 11, indicating that when a "Coding" action is occurring, the conditions for the autonomous destroyer will be checked first. This sequence is specified so that if the code is 100% complete, the action will cease (as a result of the autonomous destroyer) before the user-initiated destroyer is checked and the "Stop coding" choice is put on an Employee's menu. The "Break" trigger has priority 1 so that if an employee is tired, they will go on a break before they can get involved in any other task (by being triggered into another action).

Any trigger or destroyer can also be designated as *game-ending*, meaning that when that trigger or destroyer occurs, the game will be over. A game-ending trigger or destroyer must have exactly one of its participant's attributes specified as the *score* attribute, indicating that the value of that attribute at the time that trigger or destroyer is executed will be given as the player's score. A typical game-ending trigger might be attached to a user-initiated "DeliverProductToCustomer" action in which the score is designated as the "score" attribute of an "SEProject" participant.

### 4.1.4 Rules

Each action can have attached to it one or more rules that define the effects of that action—how the simulation is affected when the action is active. Three example rules

attached to the "Coding" action are shown in Figure 9 and will be referred to in the remainder of this subsection.

We distinguish three types of rules in a SimSE model: *create objects rules*, *destroy objects rules*, and *effect rules*. As its name indicates, a create objects rule causes new objects to be created in the game. For example, the "Coding" action has associated with it a create objects rule that creates a new "Code" Artifact object with its size and number of errors equal to zero. This indicates that a new piece of code comes into existence as a result of programmers participating in a "Coding" action.

In contrast to a create objects rule, the firing of a destroy objects rule results in the destruction of existing objects. For instance, a "Fire" action might have associated with it a destroy objects rule that removes an Employee from the game, indicating that they have been fired.

An effect rule is the most powerful and expressive type of rule in SimSE. Rules of this type specify the complex effects of an action on its participants' states, including the values of their attributes and their participation in other actions. For instance, the first effect rule attached to the "Coding" action decreases the energy and productivity levels of the coders as they work, and adjusts their error rates based on their current energy levels. The second effect rule in this action: (a) causes the size of the code to increase by the additive productivity levels of all of the programmers currently working on it; (b) causes the number of unknown errors in the code to increase based on the error rates of the currently active coders; and (c) updates the completeness level of the code. As another example, shown in Figure 10, a "Break" action has one effect rule attached to it that deactivates the employee from all other actions in which he or she is currently

```
Rules
{
  Action: Coding // action that these rules are attached to
  CreateObjectsRule
  {
    timing: trigger
    visibleInExplanatoryTool: true
    explanatoryToolDescription: “A new piece of code is created.”
    priority: 1
    createdObjects
    {
      Object Code Artifact
      {
        name =“My Code”
        numUnknownErrors = 0
        numKnownErrors = 0
        size = 0.0
        percentComplete = 0.0
      }
    }
  }

  EffectRule
  {
    timing: continuous
    visibleInExplanatoryTool: true
    explanatoryToolDescription: “Each employee’s energy is decreased as
      they expend energy working. As a result, their productivity
      accordingly decreases and their error rate increases.”
    priority: 13
    Coder:
      Programmer/Tester:
        energy = this.energy – 0.05
        productivity = this.productivity – 0.0375
        errorRate = (1 - this.energy) * 0.4
  }

  EffectRule
  {
    timing: continuous
    visibleInExplanatoryTool: true
    explanatoryToolDescription: “The size of the code is incremented by
      the employees’ productivities in coding, and the number of
      unknown errors is incremented by their error rates in coding.”
    priority: 14
    CodeDoc:
      Code:
        size = this.size + allActiveProgrammerCoders.productivity
        numUnknownErrors = this.numUnknownErrors +
          allActiveProgrammerCoders.errorRate
        percentComplete = (this.size /
          allSEProjectProjects.targetCodeSize) * 100
  }
}
```

**Figure 9: Example Create Objects Rule and Example Effect Rules for the “Coding” Action.**

```
Rules
{
  Action: Break // action that these rules are attached to

  EffectRule
  {
    timing: trigger
    visibleInExplanatoryTool: true
    explanatoryToolDescription: “As the employee goes on a break,
      they are deactivated from all of their other actions.”
    priority: 1
    Breaker:
      Programmer/Tester:
        effectOnOtherActions: deactivate All
  }

  EffectRule
  {
    timing: continuous
    visibleInExplanatoryTool: true
    explanatoryToolDescription: “The energy of the employee is
      increased as they enjoy their break.”
    priority: 3
    Breaker:
      Programmer/Tester:
        energy = this.energy + 0.1
  }

  EffectRule
  {
    timing: destroyer
    visibleInExplanatoryTool: true
    explanatoryToolDescription: “As the employee returns to work from
      their break, they are reactivated into all of their previous
      actions.”
    priority: 1
    Breaker:
      Programmer/Tester:
        effectOnOtherActions: activate All
  }
}
```

**Figure 10: Example Effect Rules for the “Break” Action.**

participating for the duration of the “Break” action, one that increases the energy of an employee while they are on a break, and one that reactivates them into all of their other actions when the break is over.

In specifying an effect rule, the modeler can use a number of different constructs as parameters in an effect’s expression. These include participant attribute values, the

number of participants in an action, the number of other actions in which a participant is involved, the time elapsed in the simulation, random values, numbers, user inputs, and mathematical operators.

In addition to a rule's general type (*create objects*, *destroy objects*, or *effect)*, each rule is also assigned a *timing* type, indicating when and how often that rule will be executed. There are three possible timing types: *trigger, destroyer*, or *continuous*. A trigger rule will execute only once, at the time the action is triggered, while a destroyer rule will also execute only once, but at the time the action is destroyed. A continuous rule, on the other hand, will fire every clock tick that the action is active. Only *effect rules* can be continuous, since there is no need to create the same object multiple times (using a *create objects* rule), or destroy the same object multiple times (using a *destroy objects* rule). Table 3 summarizes these various combinations. In the "Coding" rules shown in Figure 9, the new Code Artifact is created once, at the time the action is triggered, since the create objects rule is assigned a trigger timing. Because the effect rules in this action are assigned continuous timings, however, their expressions are evaluated every clock tick that the action is active, and the "Coder" and "CodeDoc" attributes are updated accordingly. In the "Break" action's rules shown in Figure 10, each of the different rule timings is represented: a trigger rule deactivates the employee from all of their other actions when their break starts, a continuous rule increases their energy level each clock tick during the break, and a destroyer rule reactivates them into all of their previous actions when the break ends.

Like action triggers and destroyers, each rule may also be assigned a priority in order to specify the order in which it should be executed in relation to other rules. The

**Table 3: Timing of Execution of Each Different Type of Rule.**

| | | Rule Type | | |
|---|---|---|---|---|
| | | **Create Objects** | **Destroy Objects** | **Effect** |
| **Rule Timing Type** | **Trigger** | Once, at trigger time | Once, at trigger time | Once, at trigger time |
| | **Destroyer** | Once, at destroyer time | Once, at destroyer time | Once, at destroyer time |
| | **Continuous** | N/A | N/A | Once every clock tick that the action is active |

mechanism of prioritization varies depending on the timing of the rule. A trigger rule is prioritized in relation to the other trigger rules attached to the same action. A destroyer rule is prioritized in relation to the other destroyer rules attached to the same action. A continuous rule is prioritized in relation to all other continuous rules in the simulation. Like triggers and destroyers, all rules in a prioritization must have unique priorities to ensure a predictable ordering. Also like triggers and destroyers, the prioritization of a rule is optional. All prioritized rules will execute first (in their specified ordering), after which the non-prioritized rules will execute in an undetermined order.

For example, the first continuous effect rule attached to the "Coding" action (the one that decreases employee energy and productivity) has a priority of 13 while the second one (the one that updates the progress on the code based on the employees' productivity) has a priority of 14. This means that the employees' productivities will be correctly updated *before* these productivity values are used to calculate the current progress on the code.

Finally, for each rule it must also be specified whether or not the rule should be visible in the explanatory tool—whether the user should be able to see that this rule was executed during the game. If this value is true, a textual description of the rule must also be given, to be displayed in the user interface of the explanatory tool.

### 4.1.5 Graphics

Because the user interface of SimSE is fully graphical, graphics are an integral part of our modeling approach, and are woven throughout the different parts of a model. For instance, as mentioned previously, each action trigger and destroyer can have associated with it a string of text to appear in pop-up bubbles over the heads of that action's employee participants when the action either begins (trigger) or ends (destroyer). Additionally, effect rules can have specified with them *rule inputs* that cause a dialog to appear during the simulation, prompting the user for input. Figure 11 shows an effect rule for a "GiveBonus" action that takes a rule input. As can be seen from the figure, each rule input has associated with it the following metadata:

- *Name:* A name for the input ("BonusAmount").
- *Type:* Whether the input is a String, a Boolean, an Integer, or a Double. The "BonusAmount" input for the "GiveBonus" action is a Double, since it represents a monetary quantity.
- *Condition:* If the type is either Integer or Double, this field can specify a condition on the input. For the "BonusAmount" input, the condition is that it must be greater than 0.0, since logically, an amount of money cannot be 0 or negative.
- *Prompt:* The text that will appear when the player is prompted to enter the input. For instance, the player who is giving the bonus to their employee will be prompted with the text, "Please enter bonus amount".

A rule input can be used as a parameter in any of that effect rule's expressions. In the "GiveBonus" action, the "BonusAmount" input is used to recalculate the employee's

```
Rules
{
  Action: GiveBonus // action that this rule is attached to

  EffectRule
  {
    timing: trigger
    visibleInExplanatoryTool: true
    explanatoryToolDescription: “The employee's energy is increased
      by an amount that is proportional to the amount of the bonus
      compared to the employee's pay rate (larger bonus -> larger
      energy increase).
    Recipient:
      Programmer/Tester:
        energy = this.energy + (input-BonusAmount / this.payRate)

    ProjectWithBudget:
      SEProject:
        moneySpent = this.moneySpent + input-BonusAmount

    Rule Input:
      Name: “BonusAmount”
      Type: Double
      Condition: > 0.0
      Prompt: “Please enter bonus amount”
  }
}
```

**Figure 11: Example Effect Rule for the “GiveBonus” Action.**

energy, increasing it by an amount dependent on the magnitude of the bonus in relation to the employee’s pay rate. For example, a bonus amount that is 10% of the employee’s pay rate will increase the employee’s energy by 10% of their maximum energy (0.1). In this same rule, the “BonusAmount” input is also used to add the amount of the bonus to the project’s “moneySpent” value.

In addition to these graphical aspects woven throughout the model, there are two distinct parts of a model that are purely graphical. The first part is the assignment of specific images to each object in the start state, shown in Figure 12. Each model must have associated with it one directory that contains all of the icons that are to be used for representing objects in the simulation (denoted by the “iconDirectoryPath” field in Figure 12). Each object in the start state, as well as each object created by a *create objects rule*,

must be assigned an image file contained in this directory. In addition, each Employee object must also be assigned an x, y location in the map of the simulated office in which the process takes place. For example, in Figure 12, the Programmer Employee named Roger will be represented by the image contained in the file "roger.gif" and will appear in tile 0, 5 in the office; the Code Artifact named "My Code" will be represented by the image named "code.gif"; and the SEProject Project with the description "Rocket Launcher Software" will be represented by the image "project.gif."

The second distinctly graphical part of a SimSE model is the map, which defines the layout of the simulated office that makes up the main portion of the user interface. In particular, the map specifies locations for all of the surroundings of the employees such as desks, walls, computers, and chairs. The images for all of these surroundings are predefined by SimSE, while, as already mentioned, the images for simulation objects are defined by the modeler (although a download of SimSE includes a set of icons that can be used for this purpose).

The map is a 16 x 10 grid of tiles, a size that was chosen based on its fit into the rest of the SimSE graphical user interface. We considered making the map larger, or else customizable per model, but in the models we have built thus far, a larger map has not been needed. Moreover, making the map larger than 16 x 10 would require that the map be either scrollable (which might make user interaction more awkward) or the tiles be made smaller (which might make the images harder to see). Still, in future work we plan to experiment with making the map size customizable per model to see if this adds any benefit to SimSE (see Chapter 12).

```
Images
{
  iconDirectoryPath = “C:\SimSE\Models\SampleModel\Icons”

  Object Programmer Employee
  {
    keyAttributeValue: “Roger”
    imageFilename: “roger.gif”
    x-Position: 0
    y-Position: 5
  }

  Object Code Artifact
  {
    keyAttributeValue: “My Code”
    imageFilename: “code.gif”
  }

  Object SEProject Project
  {
    keyAttributeValue: “Rocket Launcher Software”
    imageFilename: “project.gif”
  }
}
```

**Figure 12: Sample Image Assignments to Objects in SimSE.**

For each tile in the map, two pieces of data may be specified: a base image and a fringe image. The base image is what appears as the bottom-most image in the tile and the fringe is what appears directly above the base image. (If a tile is designated as the location for an employee, that employee’s image will appear above the fringe, in the object layer.) SimSE designates some of its predefined office surrounding images as base images (walls, doors, tables, and floor tiles) and some as fringe images (computers, chairs, trash cans, and papers).

Figure 13 shows a portion of a sample map definition, for tiles 0, 0 through 0, 5. In this example, the first tile will display a floor tile, the second tile will display the left portion of a table with a computer sitting on top of it, the third tile will display the right portion of a table with papers on top of it, the fourth tile will display an empty trash can on the floor, the fifth tile will display only a floor tile, and the sixth tile will display a

```
Map
{
  0,0:
    base: FLOOR
    fringe: // none
  0, 1:
    base: TABLE_TOP_LEFT
    fringe: COMPUTER
  0, 2:
    base: TABLE_TOP_RIGHT
    fringe: PAPERS
  0, 3:
    base: FLOOR
    fringe: TRASH_CAN_EMPTY
  0, 4:
    base: FLOOR
    fringe: // none
  0, 5:
    base: FLOOR
    fringe: CHAIR

  // etc…
}
```

**Figure 13: Sample Map Definition in SimSE.**

chair on the floor. Per the example shown in Figure 12, the Programmer Employee "Roger" will also appear in this tile, on top of the chair.

### 4.1.6 Modeling Sequence

The order in which the constructs of a model must be defined is partially variable and partially constrained. Object types are the core of the model and therefore must be defined first, before any other construct can be created (except the map). This is not to say that *all* object types must be defined before moving on to define any other constructs—it is fully expected that models are developed iteratively. A few object types are generally created first, followed by the start state objects, actions, rules, and graphics that involve those object types. This sequence is then repeated as model development progresses.

Similarly, because rules are attached to actions, an action must be defined before the rule(s) attached to that action are defined. A start state object must, of course, also be created before the graphical counterpart of the object (the image representing the object and, if it is an employee, its location in the map) is assigned. Figure 14 summarizes the dependencies between modeling constructs in terms of order of development.

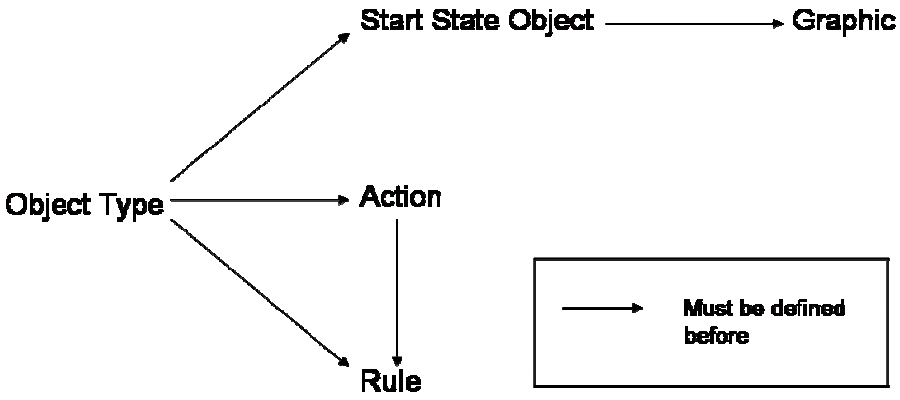


**Figure 14: Dependencies of Modeling Construct Development.**

### 4.1.7 Summary of Modeling Constructs

To summarize our presentation of SimSE's modeling language constructs, a UML-like diagram representing the language is shown in Figure 15. Modeling constructs are denoted as rectangles, with the name of the construct in bold in the top part of the rectangle and its attributes listed below. Attributes that are either optional or only present for certain types of that particular construct (e.g., *minVal* is only present for Integer or Double object types) are shown in parentheses. Relationships between two constructs are indicated by an arrow drawn between them, and each arrow is labeled with the type of relationship it represents. The cardinality of a relationship is specified at each end of the corresponding arrow (with the exception of the "type of" relationship, which has no

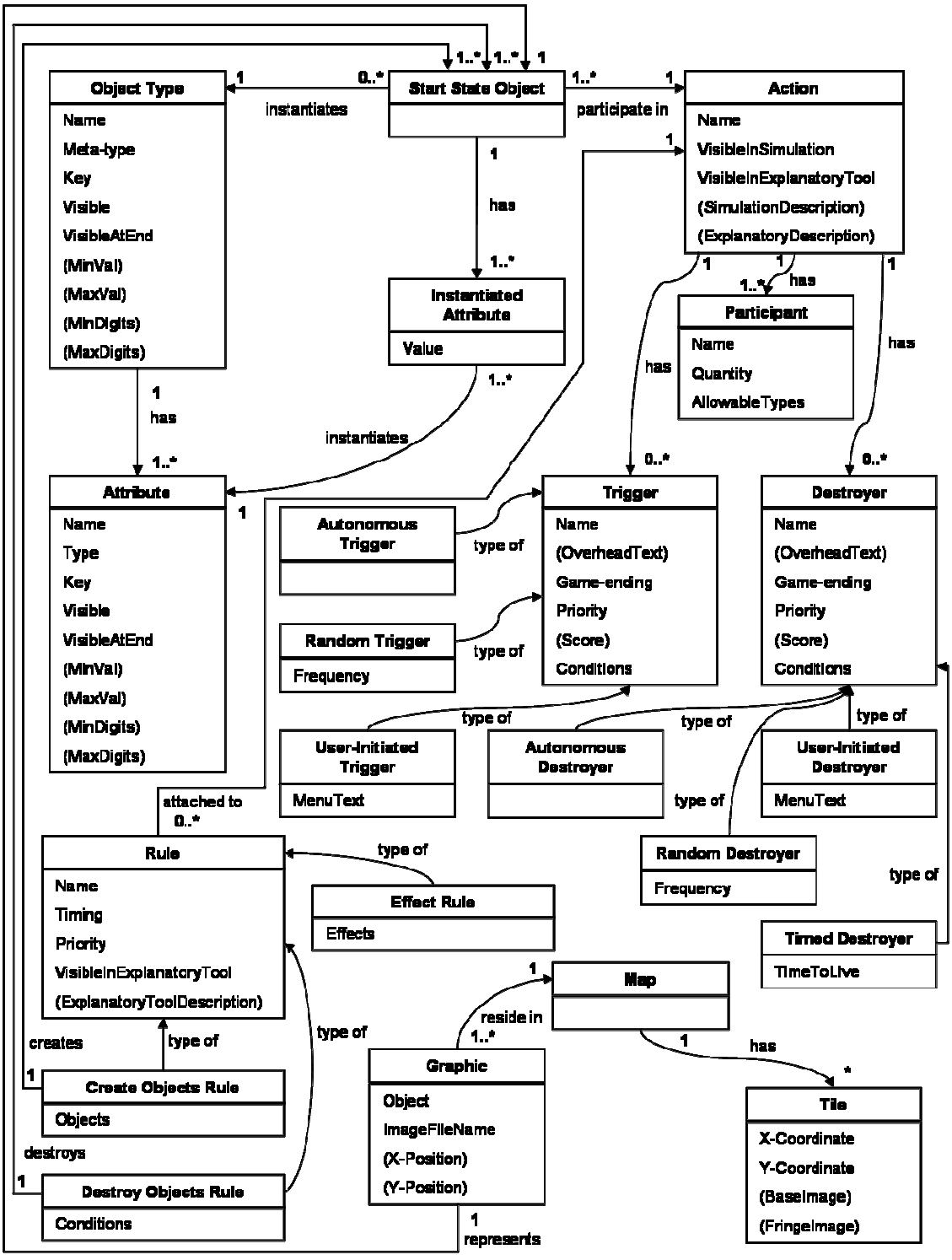


**Figure 15: A UML-like Representation of SimSE's Modeling Language.**

cardinality). As an example, let us consider the object type and start state object constructs and their relationships, shown in the top part of the diagram. An object type

has the mandatory attributes *Name*, *Meta-type*, *Key*, *Visible*, and *VisibleAtEnd*, and the optional attributes *MinVal*, *MaxVal*, *MinDigits*, and *MaxDigits*. Each object type also has one or more attributes. Zero or more start state objects can instantiate each object type. Each start state object has one or more instantiated attributes, each of which have one attribute, *Value*. One or more instantiated attributes can instantiate one attribute. One or more start state objects can participate in an action, etc. etc.

## 4.2 Sample Implementation

In order to more thoroughly demonstrate the modeling constructs of SimSE, we now present a few implemented SimSE rules from some of our completed simulation models. These rules represent some of the "fundamental rules of software engineering" discussed at the beginning of this chapter and listed in Appendix A. The first rule is a continuous effect rule attached to the "CreateDesign" action that modifies the "size" attribute of the design document artifact being created:

```
1  DesignDoc:
2   Design:
3    size = this.size +
4     (allActiveSoftwareEngineerDesigners.productivityInDesign
5      * (1 – (.01 * (numDesigners * (numDesigners – 1) / 2)))
6       * (1 + this.completenessDiffRequirementsDoc)
7        * (1 +
8         allActiveDesignEnvironmentTools.productivityIncreaseFactor))
```

In short, this rule says that as a design is being created, the size will increase by an amount dependent on the additive productivity of the designers (line 4), the communication overhead of the number of designers working on it (line 5), the difference in completeness between the requirements document and the design document (line 6), and the productivity increase factor of any design environment tool used (lines 7 and 8). The amount of increase is primarily based on the additive productivity of the designers,

and each of the other factors serve as multipliers to either raise or lower this amount. We can see in this rule the implementation of a number of the software engineering rules presented in the beginning of this chapter. In the first multiplier, listed on line 4, we can see the implementation of rule 5 (the greater the number of developers working on a task simultaneously, the faster that task is finished, but more overall effort is required due to the growing need for communication among developers). The amount of increase is reduced by 1% for each communication link between two people who are working on the design. (Note that because there exists no empirical data for this value, we assigned it to 1% after trying several different values and playing the game repeatedly in order to determine which value produced the most educationally effective result. As mentioned in Section 3.2, this same process was used to formulate many of the rules in our models for which there exists no empirical data.) In the second multiplier (line 6) we can see the implementation of rule 1, which enforces the sequential nature of the waterfall model. The design document's "completenessDiffRequirementsDoc" attribute is an integer attribute with minimum value 0 and maximum value 1 (hence, it must be either 0 or 1). This value is set in another effect rule that is executed before the one shown here, which sets it to 0 if the requirements document is less complete than the design document, or 1 otherwise. Hence, the amount of increase in the size of the design document is doubled if the features the developers are designing have been specified first. Otherwise, there is no effect. Again, no empirical data was available regarding the exact magnitude of this effect (how much faster design should be if requirements are done first) and so the multiplier was set at this particular value (100% speed increase) through experimentation and play-testing.

In the third multiplier (line 8), we can see the implementation of rule 8, which states that tools increase productivity. The amount of increase in the size of the design document is increased according to the productivity increase factor of the design environment tool (which, in this particular model, was set to 0.5).

The next rule is also a continuous effect rule attached to the "CreateDesign" action, but this one modifies the design document's "numUnknownErrors" attribute:

```
1  Design Doc:
2   Design:
3    numUnknownErrors = this.numUnknownErrors +
4     (allActiveSoftwareEngineerDesigners.errorRateInDesign
5       * (1 – (.01 * (numDesigners * (numDesigners – 1) / 2)))
6        * (1 + (allActiveRequirementsDocuments.PercentErroneous / 100
7         * 10))
8          * (1 + (1 – this.completenessDiffRequirementsDoc))
9           * (1 –
10            allActiveDesignEnvironmentTools.errorRateDecreaseFactor))
```

This rule represents the effect that, as the design is being created, a number of unknown errors are being introduced into the design document. This number is primarily based on the designers' additive error rate in design (line 4), and is affected by the communication overhead between the designers (line 5), the number of errors in the requirements document (lines 6 and 7), the completeness of the requirements document (line 8), and the error rate decrease factor of any design environment tool used (lines 9 and 10). In this rule, we can again see rule 5 implemented (the cost of communication overhead) in line 5—the amount of errors the designers can introduce is tempered by the communication overhead (as the rate at which the designers work slows down, the rate at which they can introduce errors slows down as well). The next multiplier (lines 6 and 7) illustrates rule 2, which states that any errors that are not corrected in one artifact will be carried over into the next artifact. In this expression, the amount by which the design document's unknown

errors will increase will be (x * 10)% higher, where x is the percentage of the requirements document that is erroneous. (This multiplication by ten is obviously an exaggeration—See Section 7.7 for a discussion on the tradeoff between accuracy and educational effectiveness.)

The next multiplier (line 8) again illustrates the sequential nature of the waterfall model stated in rule 1. It represents that the number of unknown errors introduced into the design document will be doubled if the requirements document is less complete than the design document (completenessDiffRequirementsDoc = 0), but will otherwise have no effect (completenessDiffRequirementsDoc = 1).

Finally, the last multiplier (lines 9 and 10) again implements rule 8 (tools increase productivity), but affects the artifact's errors rather than the artifact's size, as in the previous rule. This expression represents that the number of unknown errors introduced into the design document will be decreased according to the error rate decrease factor of the design environment tool.

## 4.3 Discussion

The unique educational, interactive, game-based, and graphical nature of SimSE required that we design a new process modeling language that fit our particular needs, rather than adopt an existing one. Specifically, these four goals and characteristics of SimSE (educational, interactive, game-based, and graphical) impose three unique requirements upon its process modeling language.

First, it must be simultaneously predictive—allow the modeler (instructor) to specify causal effects that the player's actions will have on the simulation, and prescriptive—support the specification of the allowable next steps the player can take at any given time.

Being both predictive and prescriptive serves three purposes in terms of the objectives of SimSE:

1. Interactivity is maximized in that the player is able to both affect how the process plays out and be guided in their enacting of the process, rather than simply one or the other.
2. A game-like feel is promoted, as computer games are typically both predictive and prescriptive.
3. Educational effectiveness is maximized by providing two different avenues through which the simulation can teach the player: the player learns how their actions affect the process as they see them played out in the simulation, and the player learns the flow of the process from the actions that are allowed at each point in the simulation.

The second unique requirement imposed upon SimSE's process modeling language is that it must be interactive, meaning that it should operate on a step-by-step basis, accepting user input and providing feedback constantly throughout the simulation. The player should feel that they are an active and constantly involved participant in the simulated process, rather than simply an observer of the simulation. Such a quality is known to strongly engage the player and hence, increase educational effectiveness [51].

Finally, because SimSE is a fully graphical simulation, the third requirement for its underlying modeling language is that it must allow the modeler to specify the graphical representations of the elements in the model. Our survey of existing process modeling approaches revealed that most are either predictive [2, 17, 88] or prescriptive [30, 102], but not both; few are interactive [30, 102]; few support graphics [71, 94]; and none fulfill

all of these requirements. Therefore, since no existing process modeling language fit our unique needs, we needed to develop our own approach to incorporate predictive, prescriptive, interactive, and graphical facilities into one language.

In designing SimSE's software process modeling approach however, it became apparent that some tradeoffs would have to be made. First and foremost, we acknowledge that it is not as generic or flexible as some general purpose modeling and simulation approaches [14, 71], or even domain-specific languages designed specifically for modeling software processes [30, 48, 80]. For instance, our approach requires that every object being modeled be an employee, artifact, project, customer, or tool. However, aside from the fact that none of these existing approaches met the unique needs of our educational game domain, we felt that such a level of generality and flexibility was unnecessary for our purposes. The process by which we designed our modeling approach underscores this: As mentioned previously, we surveyed the software engineering literature and extracted the widely accepted process lessons and rules that would conceivably go into a SimSE model, and then designed the modeling approach with these rules in mind. Although they include a wide range of different types of phenomena, from management issues, to organizational behavior theories, to corporate culture, to the traditional software engineering theories, all of the rules that we have collected thus far can be modeled and simulated using SimSE's modeling approach.

The chief limitations of our modeling approach lie mainly in the fact that it lacks many common programming language constructs, such as if-else statements, explicit data structures, loops, and predicates. Because the intended user of SimSE's model building facilities is the instructor, we have chosen to focus on the simplicity and rapidity of the

model building process over the flexibility and expressivity of the approach. Namely, rather than a textual process modeling language, we have chosen to provide a model builder tool that abstracts away the textual representation of a modeling language (see Chapter 5).

The lack of language features makes it necessary at times to use some non-intuitive, roundabout techniques to achieve the desired effect. One example of this is in the existence of the "completenessDiffRequirementsDoc" attribute attached to a design document object, discussed in the sample implementation presented in Section 4.2. In any programming language, such an attribute would be unnecessary—an if-else statement with a predicate could simply be used to check whether the completeness of the requirements document was greater than or equal to the completeness of the design document, and, if so, adjust the multiplier in question accordingly. Instead, in our approach, we have to first create this hidden attribute, specify that it can only be equal to either 0 or 1 by making it an integer with minimum value 0 and maximum value 1, and then create a rule that sets it to the correct value using additional mathematical manipulations.

Another instance of this sort of limitation was revealed when we attempted to model the following software engineering rule: Error correction is done most efficiently by the document's author [47]. In a full-fledged modeling language this might be modeled by keeping an array of employee names or IDs with the document/artifact object, indicating that those people had been authors of that document. When an employee would then go to correct that document, this array would simply be searched for that employee's name/ID, and, if found, correction would speed up accordingly. In our approach,

however, there is not a simple way to perform such a task, due to the absence of arrays, loops, and if-else statements. Another roundabout workaround could accomplish the same effect, however: Each employee could have an integer attribute called, "authorOfDocument" (where "Document" is replaced by the name of the document in question) that must be either 0 or 1. Each employee's "authorOfDocument" attribute is set to 0 to begin with, but when an employee authors the document, this attribute is set to 1. Then, when that document is being corrected, the progress in correction made each clock tick can be multiplied by the sum of all the correctors' "authorOfDocument" values plus 1. In this way, if none of the correctors were authors there would be no effect (progress would simply be multiplied by 1), but each corrector that was also an author would cause progress to increase by 100% (or some other value as desired). For example, if there are three employees correcting the document, and two of them were also authors of that document (meaning their "authorOfDocument" value is 1), the progress in correction made each tick would be multiplied by 3, increasing progress by 200%.

Another example of an effect that is difficult to model in SimSE is the influence of the work environment on productivity. For instance, [141] states that improving the work environment by doing such things as giving employees enclosed offices and providing common areas where employees can participate in "water cooler" conversations increases productivity. Because our modeling approach currently uses graphics mainly for decorative purposes, it does not directly support this kind of phenomenon. However, a "quick and dirty" workaround could be that the modeler assigns each employee start state object their productivity value based on their location in the office. For instance, the modeler could assign a high productivity to someone who has a large, comfortable,

enclosed office adjacent to a water cooler, and a low productivity to someone whose office is simply a desk and a chair in the middle of a hallway, far away from any water cooler. Although this is not an ideal approach in which the simulation itself could calculate productivity modifiers based on employee surroundings, this workaround would probably still communicate the effect somewhat.

So, while we are aware that the specificity of our modeling approach results in certain effects being difficult to model in SimSE, we also consider this an acceptable tradeoff, as most of the rules we collected, and especially those that we consider the most fundamental principles of software engineering, can be modeled in SimSE. Many of the effects that seemed to be infeasible to model in SimSE could be modeled, but required somewhat of a different mode of thinking—in terms of the SimSE modeling constructs provided, rather than the programming language constructs to which most people are used. In order to assist with these difficulties, we have provided a "tips and tricks" guide (see Appendix B) along with the model builder's documentation. This document, compiled from lessons we have learned in building our models, provides guidelines for how common effects can be modeled that might not be intuitive at first, as well as generally helpful hints on the model-building process. Based on the number of simulation models that have been successfully developed and used, we believe that the added simplicity of the model builder tool, along with its documentation, offsets most of the drawbacks of the absence of programming language constructs.

Another fundamental tradeoff we have made in designing our modeling approach is one between graphics and customizability. Namely, we have chosen to forego much of the sophistication typical of commercial computer game graphics for the sake of having

easily customizable models. Currently, the graphical extent of our approach is a simple icon attached to each object, a two-dimensional grid-based map, and textual pop-up messages. Using a more complex graphical model that includes such things as three-dimensional graphics, animation, and sound would undoubtedly make SimSE more appealing to students, but would also make it significantly more difficult to build a model, as all of these graphical components would need to be customized as well. Because the purpose of SimSE is education, and the intended users of the modeling approach are software engineering instructors, we concluded that this was an acceptable tradeoff to make. However, in our usage of SimSE with students, we found that many of them did express their desire for more sophisticated graphics to make the game more interesting. Therefore, we plan to investigate possible ways of adding some simple animation capabilities without too much added complexity in the modeling approach (see Chapter 12).

# 5. Model Builder

Motivated by our key decision to make SimSE's simulation models customizable, we have developed a model builder tool to facilitate the model building process. The model builder completely hides the underlying textual representation of the modeling language from the modeler, and provides a graphical user interface for specifying the object types, start state, actions, rules, and graphics for a model. Figure 16 shows the user interface for the model builder, with the tab for defining object types in focus. The tabs for the other parts of the model builder are not shown in this figure, but they are similar in appearance to the object builder in that they all facilitate building a model using buttons, drop-down lists, menus, and dialog boxes—no programming is required. Once a model is specified, the model builder then generates Java code for a complete, customized simulation game based on the given model. The following sections detail each part of the model builder, as corresponding to the five parts of a SimSE model.

## 5.1 Object Types Tab

The object types tab, shown in Figure 16, allows the user to create new object types and edit existing ones. A new object type can be defined by first choosing a meta-type (employee, artifact, tool, project, or customer) for the object using the drop-down list in the upper portion of the user interface, and then clicking on "OK". The user will then be prompted to enter a unique name for the object type (e.g., "Code"). Attributes can then be added to the object type by clicking the "Add New Attribute" button. For each new attribute created, the user will be prompted to enter the attribute information detailed in Section 4.1.1, using the interface shown in Figure 17. After each new attribute is added, it

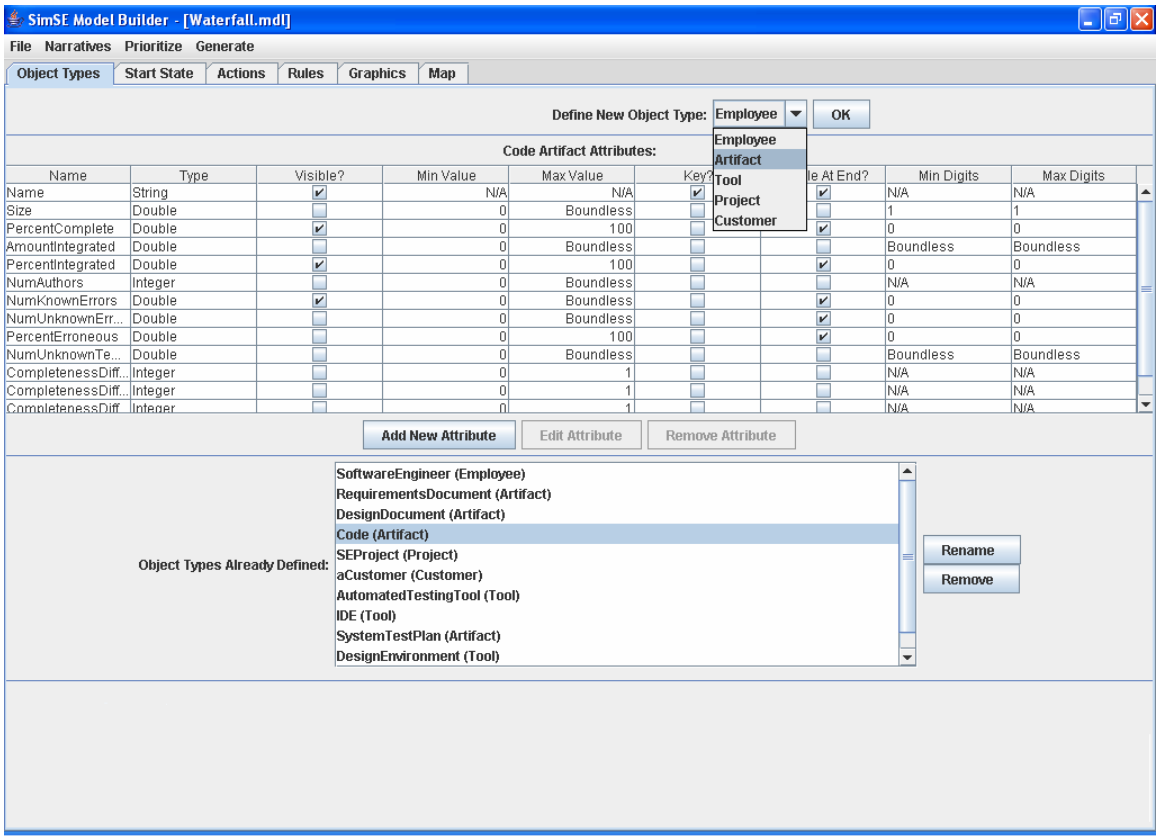


**Figure 16: Model Builder User Interface.**

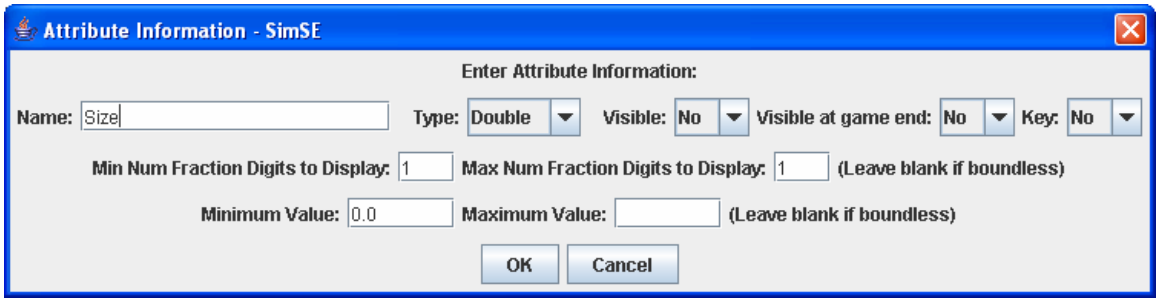


**Figure 17: User Interface for Entering Attribute Information.**

will appear in the table of attributes in the middle of the object tab user interface, with its detailed information shown in the columns of the table. Attributes can be edited by double-clicking on them or by clicking the “Edit Attribute” button, and attributes can be deleted by clicking the “Remove Attribute” button.

All of the object types that have been defined in the model appear in the list at the bottom of the object tab. Any of these object types can be brought into focus by clicking on them, and an object type can be renamed or removed using the “Rename” and “Remove” buttons.

## 5.2 Start State Tab

The start state tab, shown in Figure 18, is the portion of the model builder that facilitates the creation of start state objects—those objects with which the simulation begins. A new start state object can be created by first choosing an object type for the object from the drop-down list at the top of the interface. Each item in this list refers to an object type that was created in the object types tab for this model. Once an object type is chosen, the user will be prompted to enter a value for that object’s key attribute (e.g., a name for a software engineer employee). The new object is then created and its attributes appear in the table in the upper half of the start state tab. The last column in this table, titled “Value”, lists the starting value for each attribute. A value for each attribute can be entered by either double-clicking on the row (attribute) or by using the “Edit Starting Value” button.

All of the objects created in the start state for the model are displayed in a list in the middle part of the interface. An object from this list can be brought into focus by clicking on it, and an object can be removed by using the “Remove” button.

Finally, the bottom portion of the start state tab will list any warnings about inconsistencies in the model that involve the start state objects. In particular, these warnings notify the user if any of the object types on which the start state objects are based have changed, causing some part of the start state objects to be invalidated (e.g., a

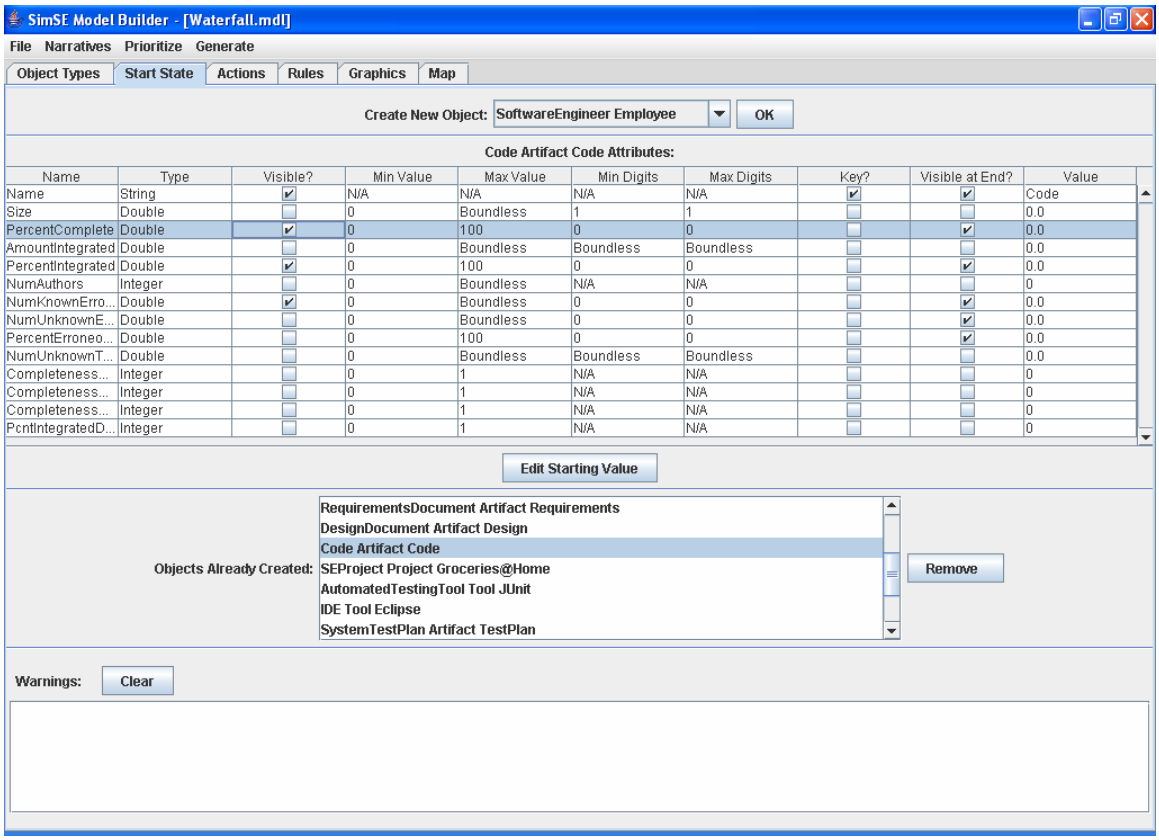


**Figure 18: Start State Tab of the Model Builder.**

string attribute changed to an integer attribute after a string starting value has already been assigned to that attribute; an object type being removed after start state objects that are based on that object type have already been created). Such inconsistencies are checked every time the focus is switched to a different tab, or when an attempt is made to save the model (a model with inconsistencies can be saved, but the warnings will re-appear to the user the next time they open the model).

## 5.3 Actions Tab

The actions tab allows the user to define the actions in a model, and can be seen in Figure 19. A new action can be created using the “Create New Action Type” button, at which point the user will be asked to give the action a unique name (e.g., “CreateCode”).

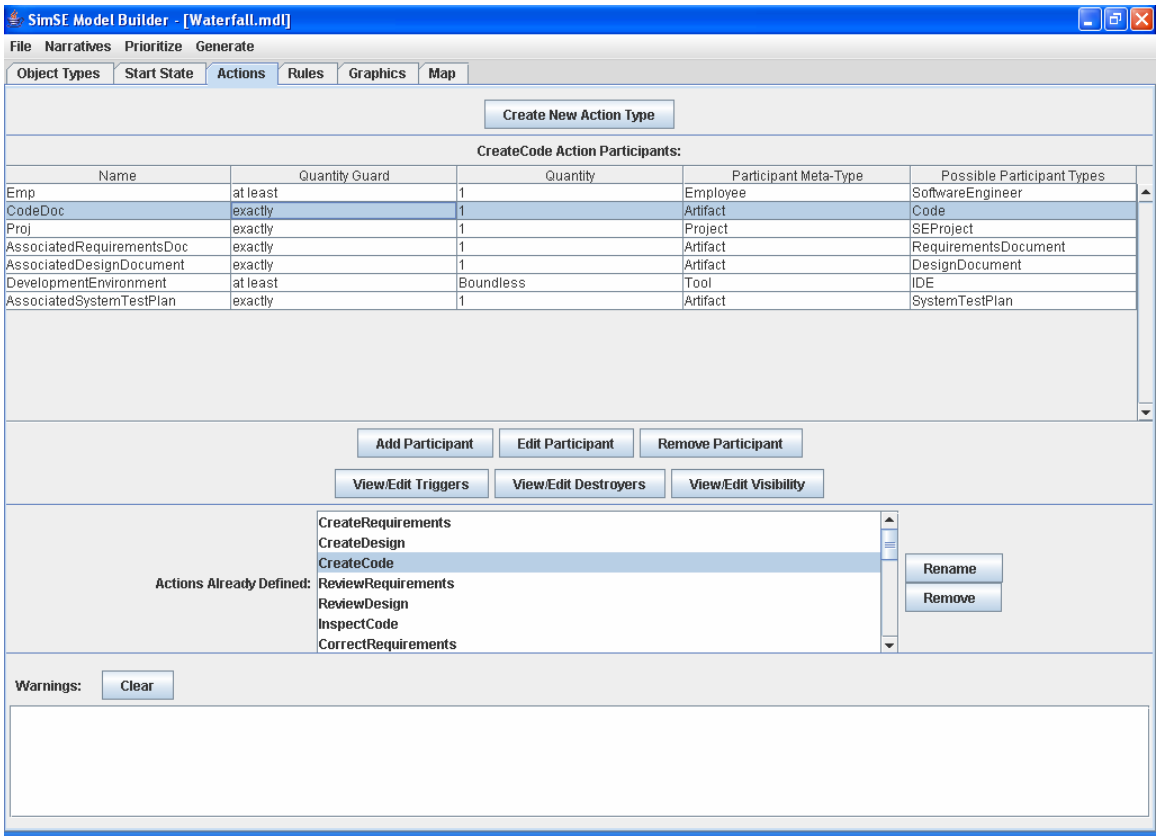


**Figure 19: Actions Tab of the Model Builder.**

Participants can then be added to the action using the "Add Participant" button, which will result in prompting the user to provide the necessary information about the participant, including its name, quantity restrictions, and allowable types through the form shown in Figure 20. Once a participant is added, it will appear in the participant table in the upper half of the interface. A participant can be edited or removed using the "Edit Participant" and "Remove Participant" buttons, respectively.

Once all of the participants have been added to an action type, the next step is to define one or more triggers for the action. A trigger can be defined by clicking the "View/Edit Triggers" button, which will cause a trigger management window to appear (see Figure 21a). All of the triggers that are attached to the action will appear here, and

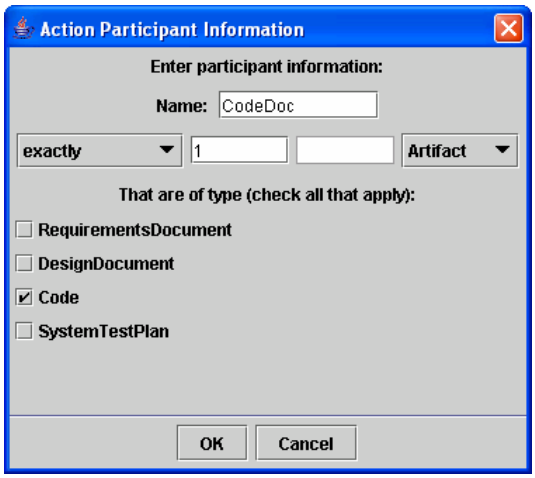


**Figure 20: Action Participant Information Form.**

any one of them can be viewed and/or edited using the "View/Edit" button, or removed using the "Remove" button. A new trigger can be added using the "Add New Trigger" button, after which a "Trigger Information" window will appear, as shown in Figure 21b. First, the trigger type can be chosen from the "Choose the trigger type" drop-down list. If the random trigger type is chosen, it will prompt for a frequency to be entered, which must be a number between 0 and 100, denoting the percent chance this action has of occurring when all of the specified conditions are met. If the user-initiated trigger type is chosen, as in Figure 21b, it will prompt for menu text to be entered. The text that is entered here will be displayed on the menus of all possible participants in this action that are of meta-type Employee (when the trigger conditions are met). If at least one of the participants in the action is of meta-type Employee, there will be a prompt for entering "overhead text", referring to the text that will be displayed in a pop-up bubble over the head of all Employee participants when this action begins in the simulation. Checking the "Game-ending trigger" box indicates that, when this trigger occurs in the game, the game will end and a score will be given to the player.

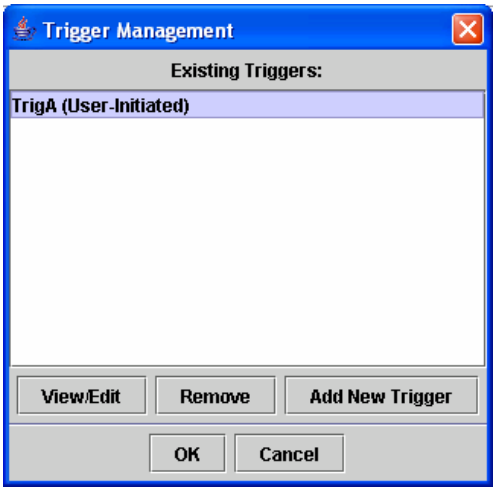


**Figure 21a:** Trigger management window.

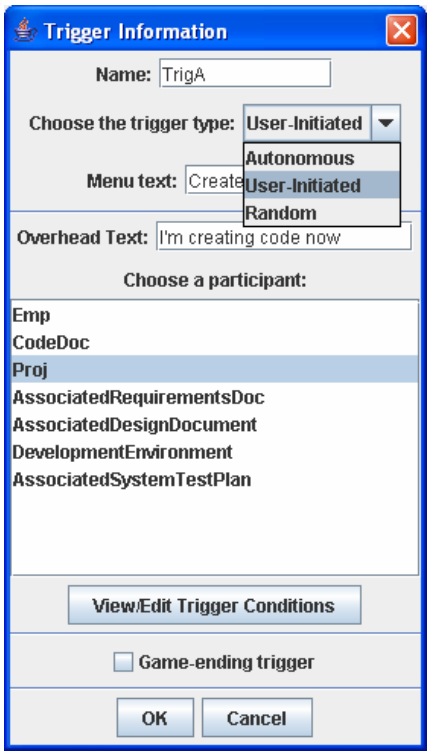


**Figure 21b:** Trigger information window.

The trigger conditions can be specified using the "View/Edit Trigger Conditions" button. This will bring up a new window in which these conditions or constraints can be entered, as shown in Figure 22. All of the allowable object types that this participant can be are listed in the "Allowable Types" list in the upper part of the window. In the example shown in Figure 22, this particular participant can only be of type "SoftwareEngineer", so only that type is shown in the list. If other types, such as "Manager" and "Tester" had been chosen when defining this participant, they would also appear on the list. The bottom half of the window allows entry of the constraints for each attribute of the object type currently selected in the "Allowable Types" list. The comparison operator (>, <, >=, <=, =) can be selected, and a value entered for each

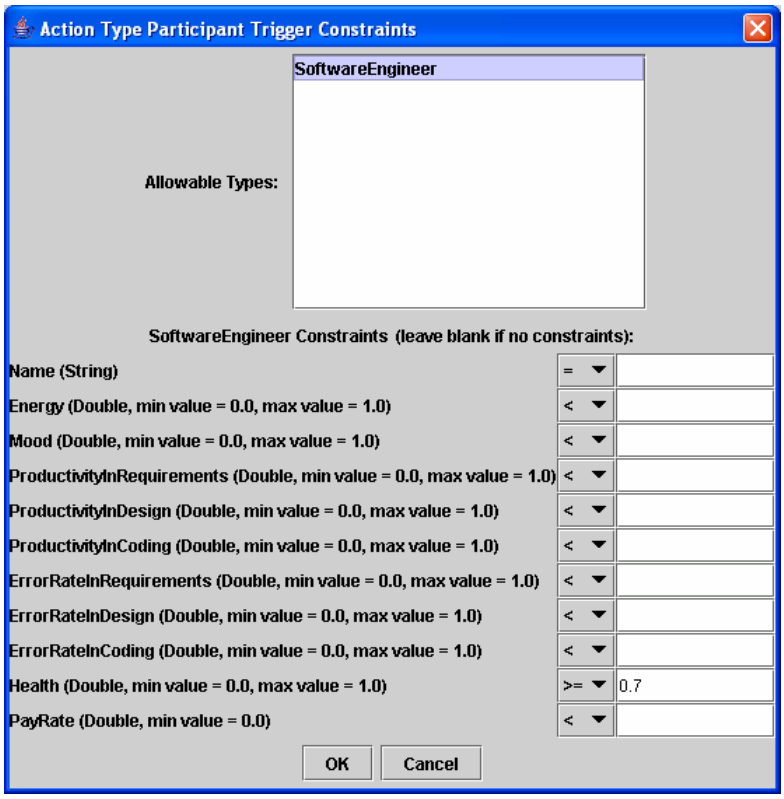


**Figure 22: Window for Entering Participant Trigger Conditions.**

attribute. In the example shown in Figure 22, the only condition for this participant is that the employee's health must be greater than or equal to 0.7.

If the trigger is a game-ending trigger, an additional column of radio buttons, marked "Score?" will appear to the left of each attribute, as shown in Figure 23. Choosing one of these indicates that the corresponding attribute's value will be given as the score to the player when the game ends.

The interfaces for defining an action destroyer are exactly identical to defining an action trigger, aside from the additional capability to define a "timed" destroyer by specifying a time to live value.

The visibility for an action can be specified using the "View/Edit Visibility" button in the center portion of the actions tab (see Figure 19). This will bring up a form for entering

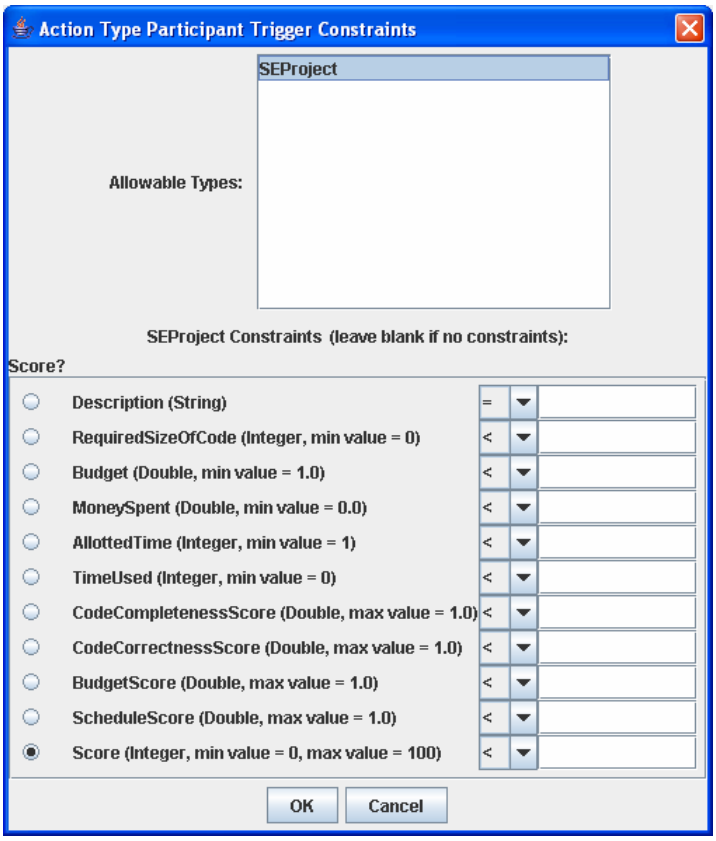


**Figure 23: Participant Trigger Conditions Window for a Game-Ending Trigger.**

both the simulation visibility and the explanatory tool visibility of the action, including optional descriptions for each (see Figure 24).

All actions that have already been defined for the model are displayed in the list in the middle portion of the actions tab. Actions can be brought into focus by clicking on them, and can be renamed or removed using the “Rename” and “Remove” buttons.

Like the start state tab, the actions tab also has an area to display warnings. Warnings will be shown here if a change was made to an object type that invalidated some part of a defined action. For instance, if a trigger condition of “size=85.5” was specified, and then the size attribute was changed to an integer, a warning would appear notifying the user that this trigger condition is no longer valid, since the value of 85.5 is not an integer.

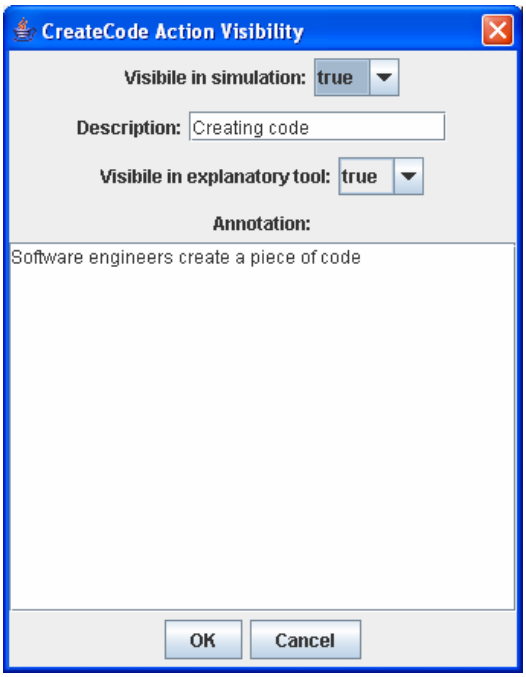


**Figure 24: Interface for Specifying an Action's Visibility.**

## 5.4 Rules Tab

The rules tab of the model builder is the tool that allows the creation of SimSE rules, and is shown in Figure 25. All of the actions for the model are listed in the "Actions" list in the middle portion of the user interface. When one of these actions is selected, the rules attached to that action (if any) appear in the table in the upper portion of the interface. These rules can be viewed, edited, renamed, or removed using the buttons to the right of the table. Like the start state and actions tab, the rules tab also has an area at the bottom for warnings that will appear if a change was made to another part of the model that creates an inconsistency with a defined rule.

The rules tab allows the creation of the three types of rules in SimSE: *create objects rules*, *destroy objects rules*, and *effect rules* (see Chapter 4.1.4). A *create objects rule* can be created by first choosing an action to which the rule will be attached, and then using

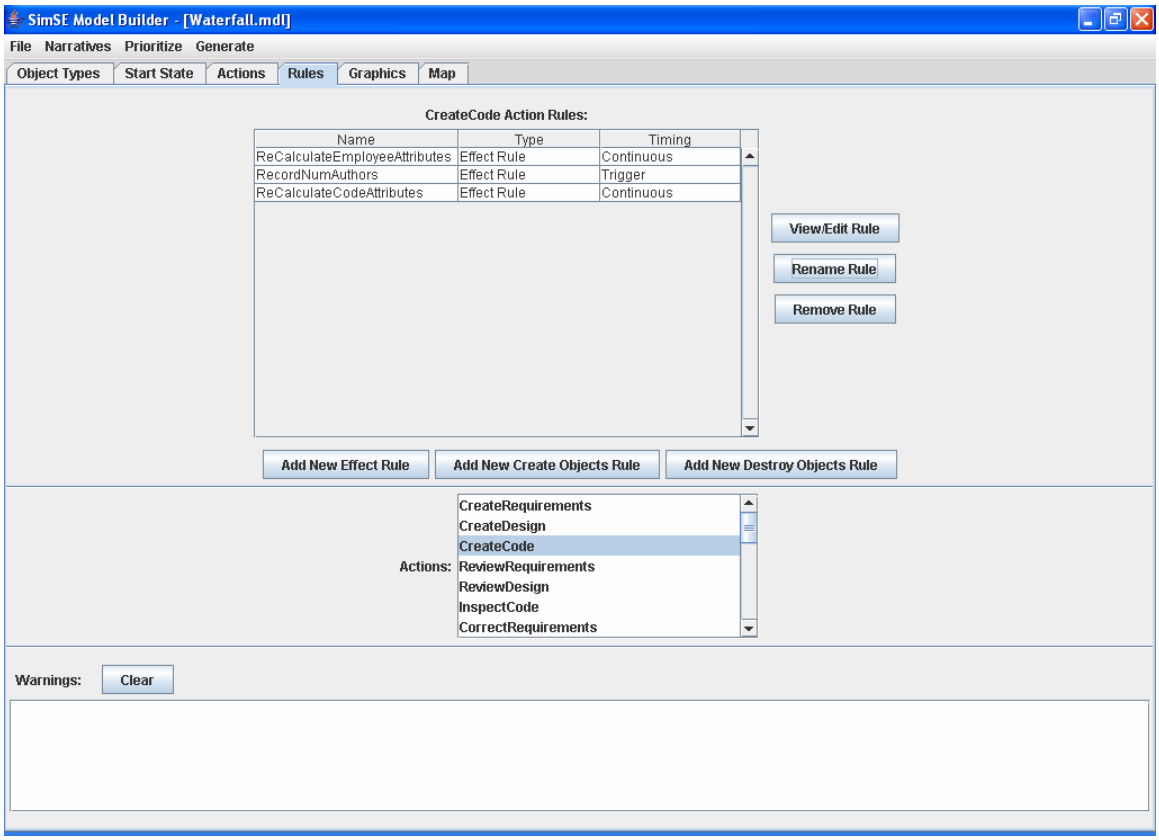


**Figure 25: Rules Tab of the Model Builder.**

the "Add New Create Objects Rule" button. The user will first be prompted to specify a name for the rule, and then a form will appear through which the specifics of this *create objects rule* can be defined (see Figure 26).

The timing for the rule (continuous, trigger, or destroyer, discussed in Chapter 4.1.4) can be chosen through the "Timing of Rule:" radio buttons. The object type for an object to be created must first be chosen from the drop-down list at the top of the window, and the "OK" button must be clicked. A form will then appear in which valid starting values for each of this attribute's values must be entered (attribute types and min/max values are enforced), and when the "OK" button is clicked, the object is added to this rule. Once an object is added to the rule, it will appear in the "Created Objects" list (see Figure 26).

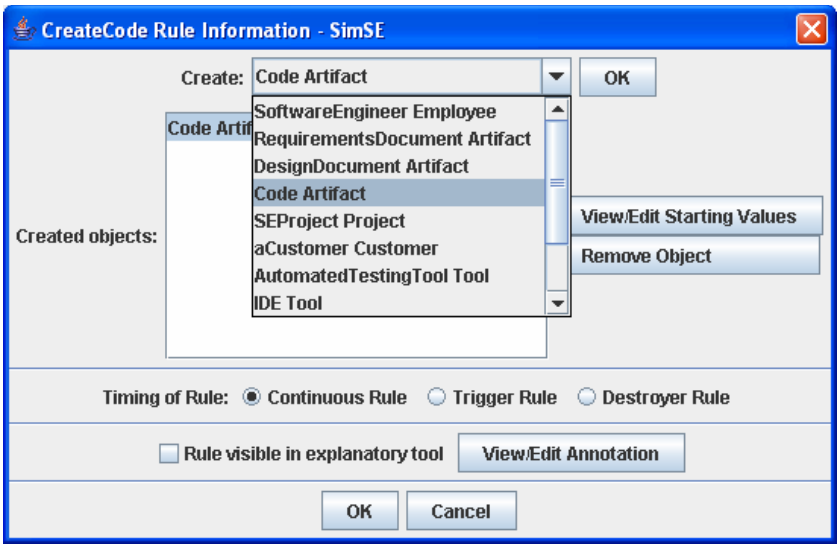


**Figure 26: *Create Objects Rule* Information Window.**

Any of these objects can then be clicked on and then viewed and edited using the "View/Edit Starting Values" button or removed using the "Remove Object" button.

A *destroy objects rule* can be created using the "Add New Destroy Objects Rule" button, which will cause the user to be prompted to enter a name for the rule, followed by the appearance of the form shown in Figure 27, which displays all of the participants in the rule's associated action. A set of conditions that must be met by an object's attributes in order for that object to be destroyed by the *destroy objects rule* can be specified using the "View/Edit Participant Conditions" button. This will bring up a new window in which these conditions can be entered, as shown in Figure 28. All of the allowable object types for this participant are listed in the "Allowable Types" list in the upper part of the window. The bottom half of the window allows entry of the conditions/constraints for each attribute of the object type currently selected in the "Allowable Types" list. The comparison operator (>, <, >=, <=, =) can be selected, and a value entered for each

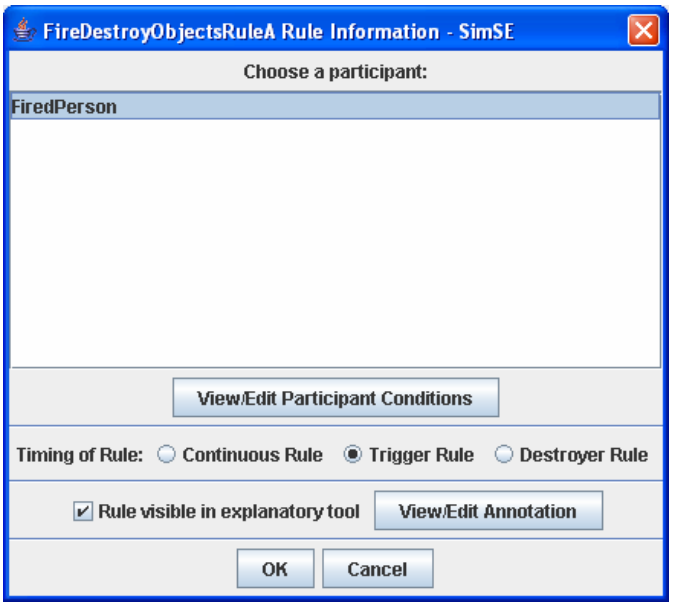


**Figure 27: *Destroy Objects Rule* Information Window.**

attribute. In the example shown in Figure 28, the "SoftwareEngineer" object will be destroyed if its energy is less than or equal to 0.5.

Effect rules can be created and added to an action using the "Add New Effect Rule" button, which prompts the user to enter a name for the rule, followed by the appearance of the window shown in Figure 29. The middle part of this window contains a list of all of the participants in the action for which this rule is being created, along with all of the possible object types for each. An effect for one of these participants can be specified by bringing into focus one of the object types for the participant, and then using the "View/Edit Effects" button. The effects for each attribute of that object type (if any) will then appear in the top part of the window, as is the case in Figure 29 for the "Code" participant.

Once a participant object type (e.g., "CodeDoc" Code Artifact) is brought into focus, two types of effects for this participant object type can be defined: (1) the effect on the participant's other actions; and (2) the effects on each of this object type's attributes. The

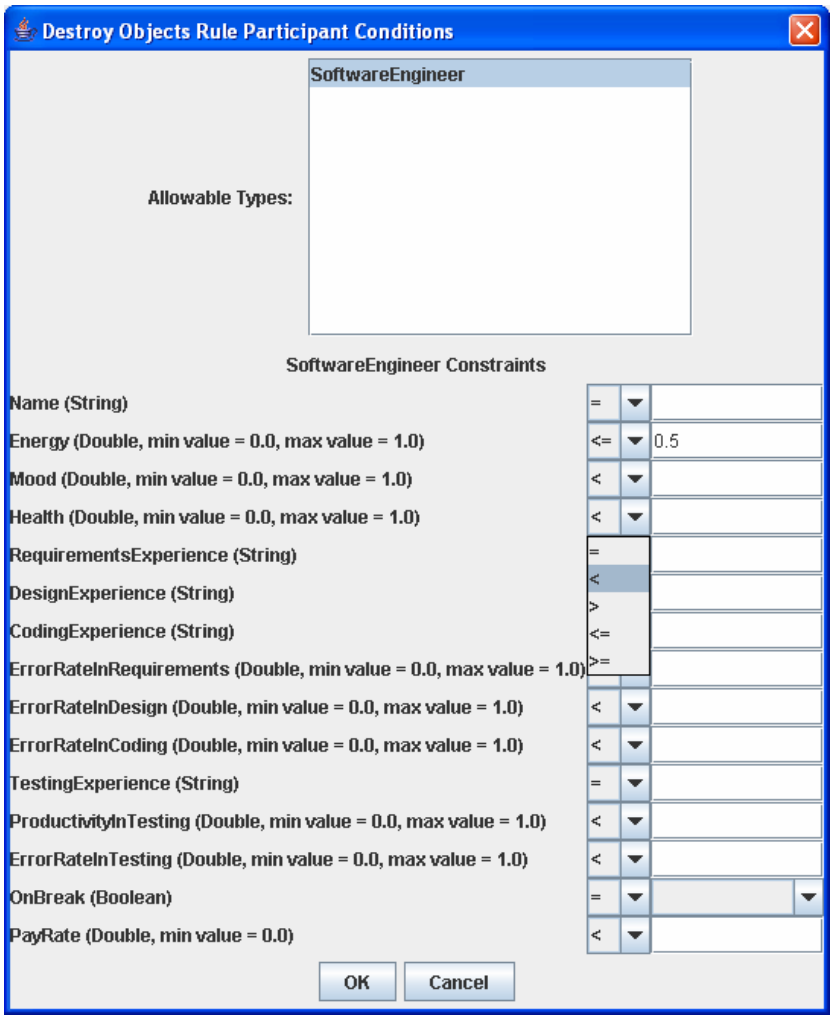


**Figure 28: Window for Entering Participant Conditions for a *Destroy Objects Rule.***

effect on the participant's other actions can be specified by selecting one of the choices next to the text: "Effect on Participant's Other Actions". This refers to what will happen every time this effect rule is fired. Every participant in an action at run-time is either active or inactive. The first choice, "Activate all other actions," causes this participant to become active in all of their actions (besides the one that this rule is attached to) in which they were previously inactive. The second choice, "Deactivate all other actions," has the opposite effect. The "none" indicates that there is to be no effect on the participant's other actions. The modeler is currently limited to these two choices (activate/deactivate

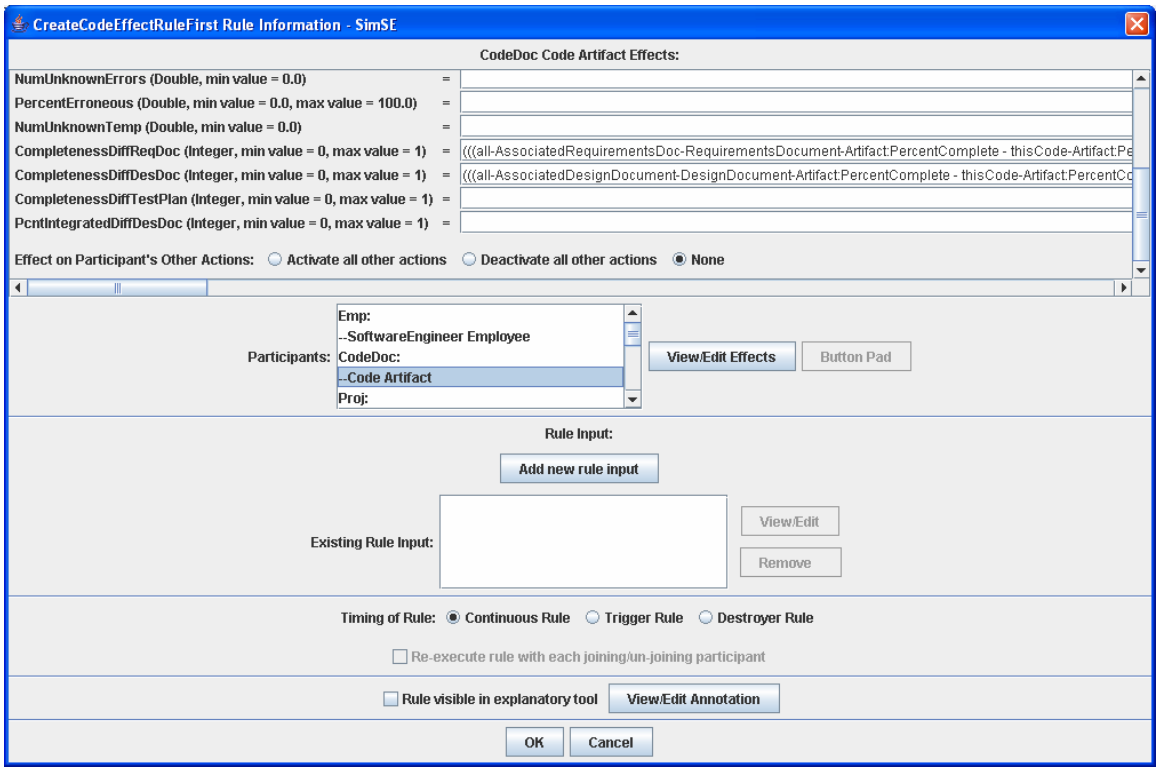


**Figure 29: *Effect Rule* Information Window.**

all or none), but our future plans include adding the ability to choose specific actions to activate or deactivate (see Chapter 12).

The effects on each of the participant object type's attributes can be specified as follows: In the top part of the *effect rule* information window, next to each of the attributes, there is a text box in which expressions can be entered that specify what the values of each of these attributes will be set to each clock tick that the action is active. As shown in Figure 29, both "CompletenessDiffReqDoc" and "CompletenessDiffDesDoc" attributes for the "CodeDoc" Code Artifact will be set to the evaluated value of the expression in the text box next to it.

The button pad, shown in Figure 30, is the interface through which the user can specify an *effect rule* expression. In order to specify an expression, the text box can be


**Figure 30: Button Pad for Entering *Effect Rule* Expressions.**

double-clicked or the "Button Pad" button can be used, at which point the button pad will appear. Each click of the button pad will result in text being inserted into the *effect rule* expression. Aside from the digit (0 through 9) buttons, the operator (+, -, *, /) buttons, and the backspace button, which are self-explanatory, the meaning and use of the other buttons are as follows, from left to right, top to bottom on the button pad:

- **Rule Input:** One of the rule inputs (see Chapter 4.1.5) that have been defined in this *effect rule* can be chosen to be inserted, and when this rule fires, the current value of the rule input will be evaluated in this expression.
- **Attributes – this participant:** One of the attributes in this participant object type can be chosen to be inserted (including the attribute for which the effect is being edited), and when this rule fires, the current value of that attribute will be evaluated in this expression.
- **Attributes – other participants:** This refers to the attributes of the other participants in this action. Upon clicking this button, it will prompt for three

selections: *status*, *participant*, and *attribute*. *Status* refers to whether active, inactive, or all (both active and inactive) participants' attribute values should be included. *Participant* refers to the name of the participant whose attribute is being chosen. *Attribute* simply refers to the attribute being chosen.

- **Num participants in action:** This button will insert a value that refers to how many participants are in this action. Once again, an all/active/inactive status can be chosen, as well as a participant.
- **Num actions – this participant:** This button corresponds to the number of actions that this participant is in. An all/active/inactive status must be chosen, along with either an action type or the "*" choice, the latter indicating that all actions should be included regardless of type. For example, if "All Inactive" and "Designing" are chosen, and the participant is currently inactive in two "Designing" actions, the number 2 would be inserted at run-time.
- **Num actions – other participants:** This button is similar to the "Num actions – this participant" button, but instead of the number of actions that this participant is in, it corresponds to the number of actions that another participant in this action is in. The all/active/inactive status must be selected, along with the participant and the action (or "*").
- **totalTimeElapsed:** This is equal to the total number of clock ticks that have executed in the simulation.
- **actionTimeElapsed:** This is equal to the total number of clock ticks that have executed since this action began.

- **Random(min, max):** When a min and a max are entered, this will generate a random number at runtime that is between min (inclusive) and max (exclusive).
- **String:** A literal string value can be entered.
- **Boolean:** A Boolean value can be entered.

We aimed to augment the simplicity of the button pad and provide some guidance to the user by designing it so that buttons are enabled or disabled depending on what input is allowed at any given time. For example, once one mathematical operator is inserted, the buttons for the other mathematical operators become disabled (except for the '-' button, which can be used as a negative sign as well). As another example, if the attribute for which an expression is being created is a non-numerical attribute, the buttons that result in a number being inserted into the expression are disabled. Once an expression is entered, the user will also receive a warning if that expression is not valid, such as in the case of a missing closing parenthesis or an expression ending with an operator. For the user who wishes to learn the syntax or wants to quickly insert a simple expression parameter, the text fields can also be directly edited.

The area below the participant list in the *effect rule* information window (see Figure 29) deals with rule inputs. A new rule input can be defined using the "Add new rule input", after which it will prompt for a name for the rule input. Following this, a rule input information form will appear, as shown in Figure 31. The type (String, Boolean, Double, or Integer) of the input must be chosen, a condition can be specified on the input if it is of type Double or Integer, and a prompt must be entered. The prompt will be the text the user of the simulation will see when they are asked to enter the input (e.g., the amount of a bonus an employee is to be given). Whether or not the rule input can be

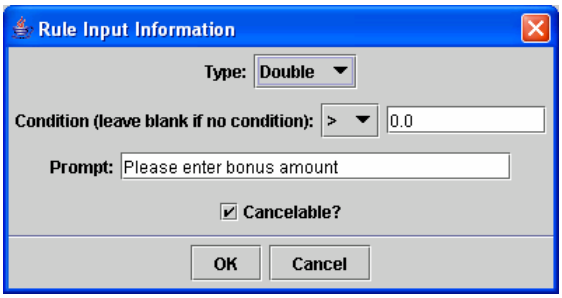


**Figure 31: Rule Input Information Form.**

cancelled by the player must also be specified. After clicking "OK", the rule input will be added to the rule, and it will then be accessible through the button pad for use in the *effect rule* expressions.

Below the rule input area in the *effect rule* information window (see Figure 29) is the area that allows the selection of the rule timing (either continuous, trigger, or destroyer) through the "Timing of Rule" radio buttons. Finally, the visibility of the rule, along with a textual description of the rule can be entered using the checkbox and the button below the rule input area.

## 5.5 Graphics Tab

The graphics tab, shown in Figure 32, is used to assign an image to each object in the start state, as well as each object that is created by a c*reate objects rule*. The image assigned will be used to represent the object in the graphical user interface of the simulation.

Operation of the graphics tab is straightforward. The first step to assigning graphics is to specify the directory that contains the icons to be used, using the "Icon Directory" button in the upper portion of the graphics tab. This directory must contain the images (50 x 50 pixels or smaller) to be matched to objects. Once the directory is chosen, the

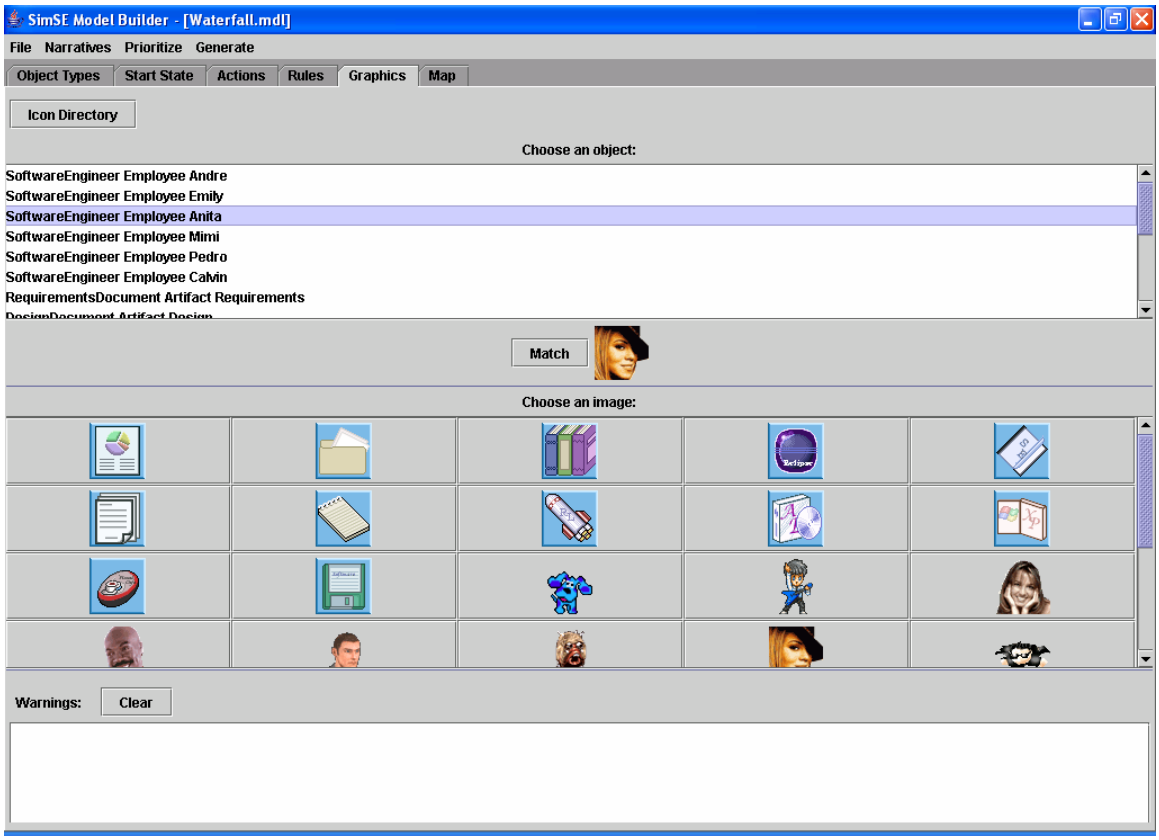


**Figure 32: Graphics Tab of the Model Builder.**

images appear in the grid in the middle part of the interface. Images can then be matched to objects by simply choosing an object from the list and choosing an image, and then clicking the "Match" button.

Again, like the other tabs, the graphics tab also contains a warning area that will display warnings about any inconsistencies between the objects assigned to images and the rest of the simulation (e.g., if the object type for an object that was assigned to an image is deleted).

## 5.6 Map Tab

The map tab of the model builder is used to specify the layout of the office in the generated game, and is also quite straightforward in operation. The map is represented as

a 16 x 10 grid, and each square in the grid may be assigned an image using a right-click menu (see Figure 33). Using this menu, the user can place office furniture, doors, walls, floor tiles, and employees (both those in the start state and those created by *create objects rules*) in the office. Employees created by *create objects rules* will appear inside a pink box with a blue hue around them, to differentiate them from employees in the start state. For instance, the employee in the lower left-hand corner of the map in Figure 33 is an employee created by a *create objects rule*. Of course, both types of employees will look the same at run-time, in the game's user interface.

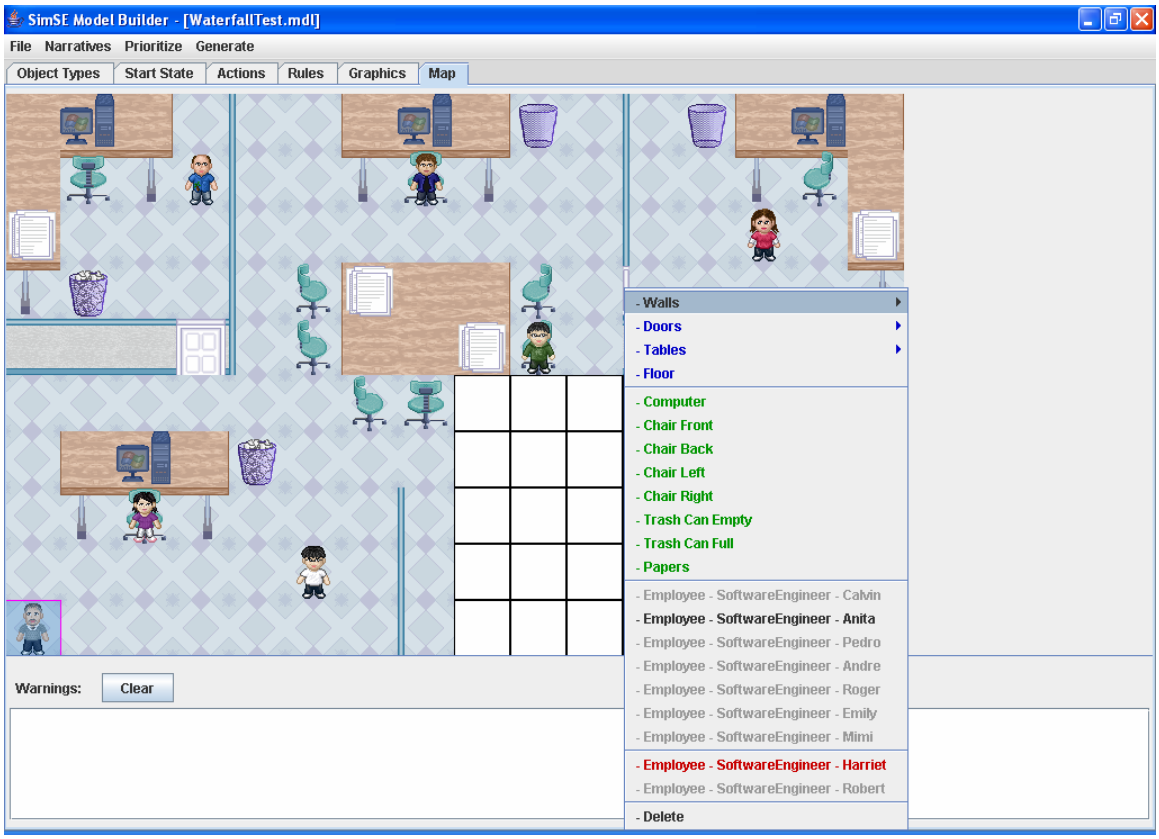


**Figure 33: Map Tab of the Model Builder.**

As discussed in Section 4.1.5, there are three allowable image layers per tile: base (bottom), fringe (middle), and object (top). The right-click menu is accordingly organized

in terms of these layers: The topmost section, listed in blue, contains all of the base images. The section below this, listed in green, contains all of the fringe images. Below this are the employees, which are placed in the object layer. Start state employees are listed first, followed by employees created by *create objects rules*, which are listed in red in their own section of the menu. Finally, the bottommost option in the menu is "Delete", which clears all images from a tile.

## 5.7 Menu Items

There are four menu items in the menu bar of the model builder: "File", "Narratives", "Prioritize", and "Generate". The "File" menu allows the user to perform the standard file management functions of opening, closing, saving, and creating new models. The "Narratives" menu allows the user to enter the starting narrative for a model (the text that will appear to the player of the simulation at the start of the game). The "Prioritize" menu, shown in Figure 34, lets the user specify the order in which they want their model's trigger, destroyers, and rules to be executed. This menu is organized into sub-menus according to the following rules: triggers are prioritized in relation to other triggers, destroyers are prioritized in relation to other destroyers, continuous rules are prioritized in relation to other continuous rules, trigger rules are prioritized in relation to other trigger rules attached to the same trigger, and destroyer rules are prioritized in relation to other destroyer rules attached to the same destroyer. When any of these menu items are selected, a window similar to the one shown in Figure 35 (the one for continuous rules) appears. The user can move a rule/trigger/destroyer from the non-prioritized list to the prioritized list (or vice-versa) using the arrow buttons in the middle.

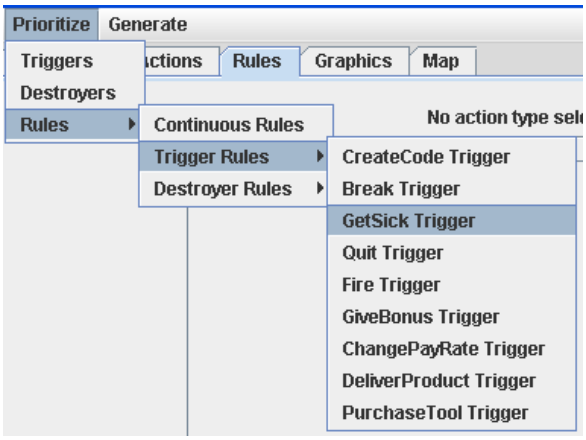


**Figure 34: The "Prioritize" Menu.**

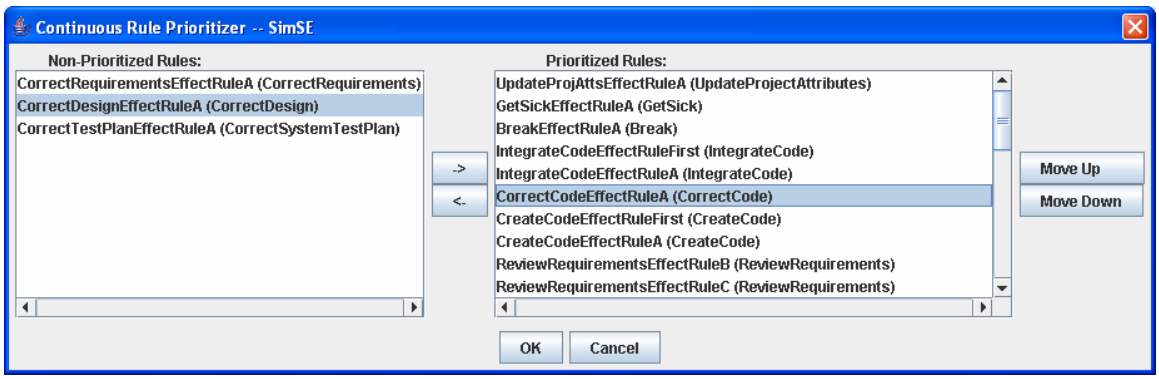


**Figure 35: The Continuous Rule Prioritizer.**

The order of the prioritized list can also be changed by selecting a list item and using the "Move Up" and "Move Down" buttons.

Finally, the "Generate" menu allows the user to generate a simulation game from a model. When this menu item is selected, the user will be prompted to specify a destination directory for the generated code. They will then be either notified that the simulation was successfully generated, or else shown an error message explaining any problems that might have occurred during code generation.

## 5.8 Design and Implementation

The model builder is comprised of approximately 50,000 lines of Java code. Its design is shown in Figure 36, and consists of two main components: the builder and the code generator. The builder is the component that facilitates the creation of a model. Within the builder, there are eight sub-components. The GUI houses all of the components of the user interface. The six sub-components in the middle row of the builder each correspond to a part of a SimSE model (object types, start state, actions, rules, graphics, map) and a tab in the model builder user interface. Each one of these sub-components is responsible for handling the creation of its corresponding model part, using the model/file manipulation component, which creates and manages the actual model in memory and on the file system.

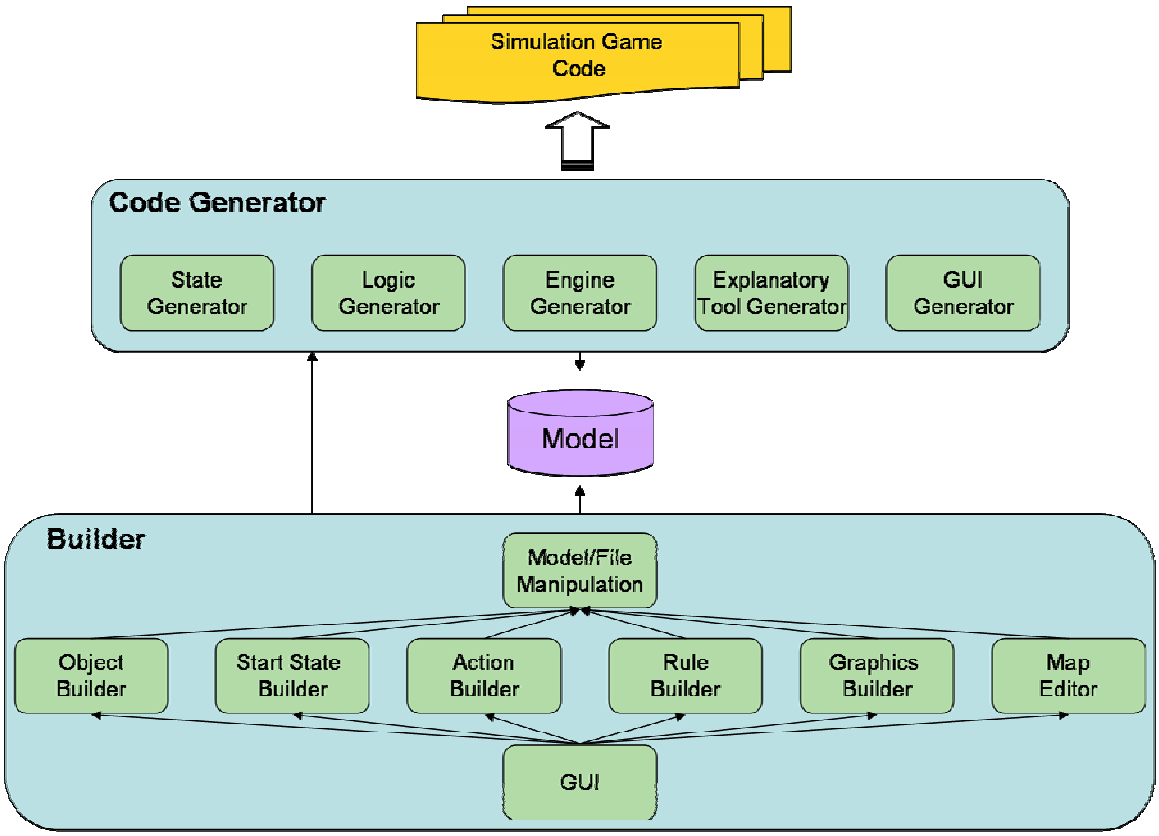


**Figure 36: Model Builder Design.**

When the user commands the model builder to generate a simulation game from a model, the builder passes this command to the code generator component. When the code generator receives this command, it reads the model and uses its sub-components to generate code for a simulation game based on this model. The code generator contains five sub-components, each corresponding to a component in the generated simulation environment (see Chapter 6.2). Each one of these sub-components is responsible for generating the code for its corresponding part of the simulation environment (e.g., the state generator generates the state component of the simulation environment, the logic generator generates the logic component, etc.)

## 5.9 Discussion

Although the model builder removes many of the inherent difficulties of a programming language (e.g., syntax and text manipulation), we recognize that building a model in SimSE is still not a trivial matter. Most notably, the difficulty of collecting software engineering phenomena and rules and translating these into SimSE actions and rules still remains. This was one of the chief purposes of building our collection of six different simulation models and making them freely available with SimSE (see Chapter 7). These models embody some of the most commonly taught software processes, and can readily be used by instructors who wish to either adapt them for their own purposes or use them directly. Our hope is that any instructor who wishes to create a new SimSE model will be able to, at minimum, either use our models as examples to follow or else take bits and pieces from them and reuse them. Although doing so would not necessarily make the process “quick and easy”, it would at least give the instructor something to work with, rather than require that they build a model from scratch.

As another resource for the model-building instructor, we have also written a 50-page user guide to accompany the model builder. This guide explains, in detail, how to use the tool, and also includes the "tips and tricks" guide presented in Appendix B. As mentioned in Section 4.3, this document includes (among other topics) suggested steps for starting a model, finishing a model, and working around the lack of common programming language constructs (e.g., if-else statements).

It is important to note that use of the model builder also does not guarantee the model is a "good" model. Rather, a strongly iterative development cycle is required. In our experience, building a model involves a significant amount of time aside from the initial construction of the model in which the model is repeatedly played and refined in order to ensure that the desired lessons and effects are illustrated, as well as to achieve the proper balance between educational effectiveness and realism (see Section 7.7 for further discussion on this issue).

# 6. SimSE

Now that we have established what a SimSE can model and simulate and how a SimSE model can be built, in this chapter we focus on how a SimSE model actually works. Specifically, we detail what the game play of a model is like and how the simulation environment is designed and implemented.

## 6.1 Game Play

SimSE is a single-player game in which the player takes on the role of project manager and must manage a team of developers in order to successfully complete an assigned software engineering task. This task may comprise an entire life cycle of a software product from inception to delivery, a small, specific activity within a software process (such as a code review), or some other aspect of a software engineering process.

At the beginning of the game, the player is presented with a description of the software engineering task they are expected to perform. This description usually includes what the goal of the game is, how much time and/or money the player is allowed, how the final score will be calculated, and perhaps some helpful hints to guide the player along the way (see Figure 37 for an example). The player then drives the process by, among other things, hiring and firing employees, assigning tasks, monitoring progress, and purchasing tools. At the end of the game the player receives a score indicating how well they performed, and additional information that was hidden throughout the game is revealed to give the player some insight into why they were given their particular score. The player can also run the explanatory tool at the conclusion of a game to gain further insight about their simulation run.

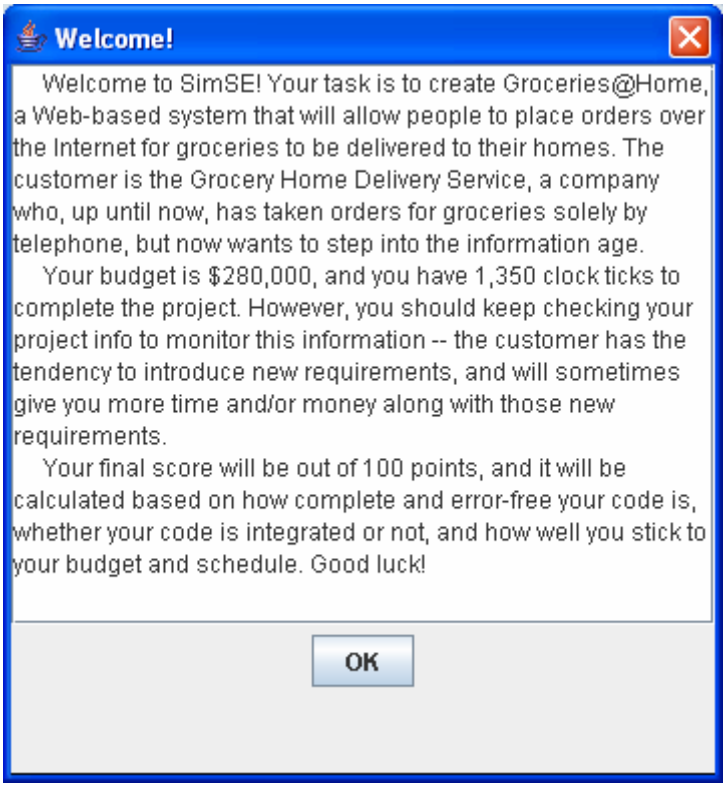


**Figure 37: SimSE Introductory Information Screen.**

Motivated by our key decision to make extensive use of graphics in our approach, the user interface of SimSE is fully graphical, as shown in Figure 38. The center part of the interface displays a virtual office in which the software engineering process is taking place, including typical office surroundings and employees. Employees "communicate" with the manager (player) through speech bubbles over their heads. Through these, they inform the player of important information, such as when they have started or completed a task, when a random event has occurred, or to express a response to one of the player's actions. In addition, depending on the particular simulation model being used, the text in these speech bubbles can also serve as a mechanism for providing the player with subtle guidance and feedback as they play the game. For instance, an employee could make a recommendation about what the next action should be after they inform the player that

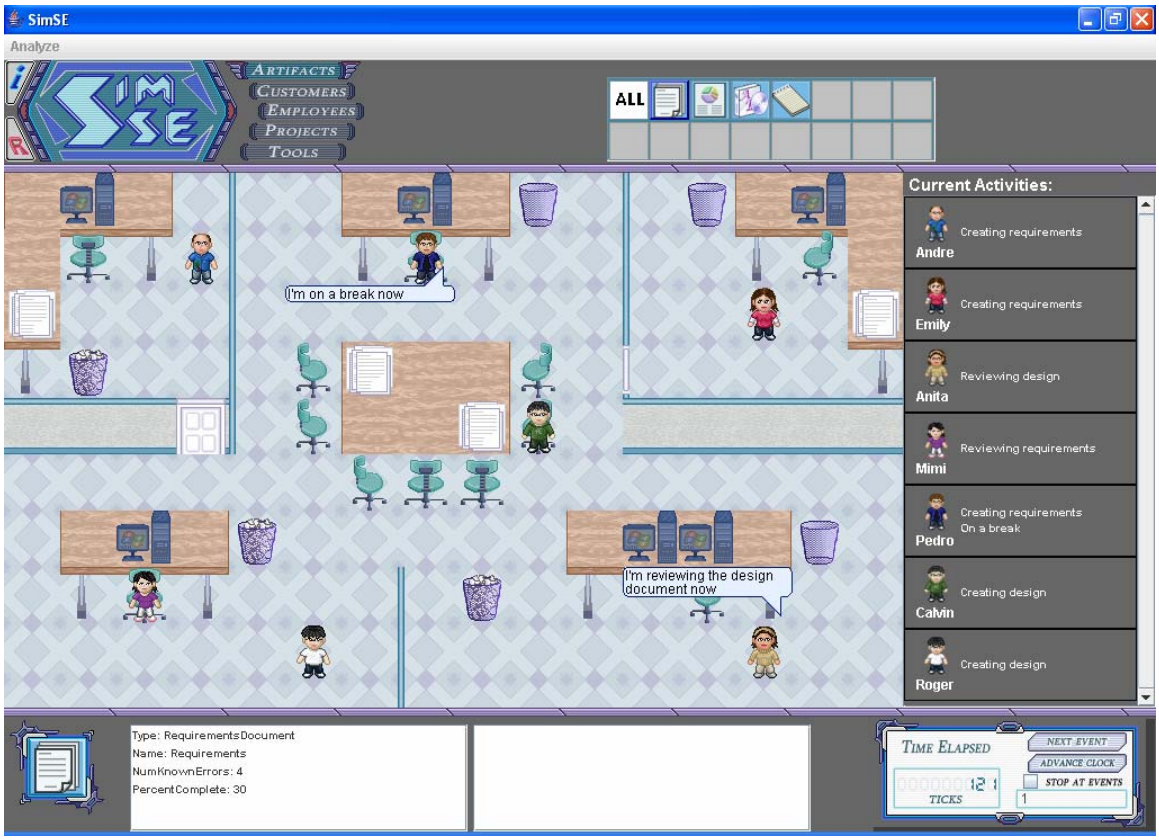


**Figure 38: SimSE Graphical User Interface (Duplicate of Figure 1).**

they have completed a task (e.g., "We just finished the requirements document. We should move onto design now."). As another example, an employee could notify the player of a mistake they made and suggest a remedy (e.g., "We ended up with a number of errors in the design because the people you assigned to create it did not have design expertise. You should now assign experienced designers to review and correct the design so we can get it into better shape."). In all cases, these "comments" by the employees provide valuable information that the player can use to make decisions and take action, steering the simulation accordingly.

SimSE has a variety of control mechanisms for playing the game. One of these is the simulation clock, the controls of which are located in the lower right corner of the user

interface, and through which the user drives the simulation. Because one of our key decisions was to make SimSE as interactive as possible, we designed SimSE to operate on a clock-tick basis, rather than as a continuous simulation in which the user provides a set of inputs, commands the simulation to run, and obtains a set of outputs such as cost and schedule. The SimSE player has two choices about how to advance through time: They can either choose to advance the clock a particular number of ticks, or advance until the next time an event occurs (when one of the employees has something to say).

The player can interact with the employees through right-click menus on each employee (see Figure 39). Using these menus, they can assign software engineering tasks (e.g., write code, review the design document), or perform other managerial activities in relation to an employee, such as firing, giving bonuses, or changing an employee's pay rate. They can also perform "global" managerial actions such as purchasing software engineering tools for the whole company, or delivering the final product to the customer.

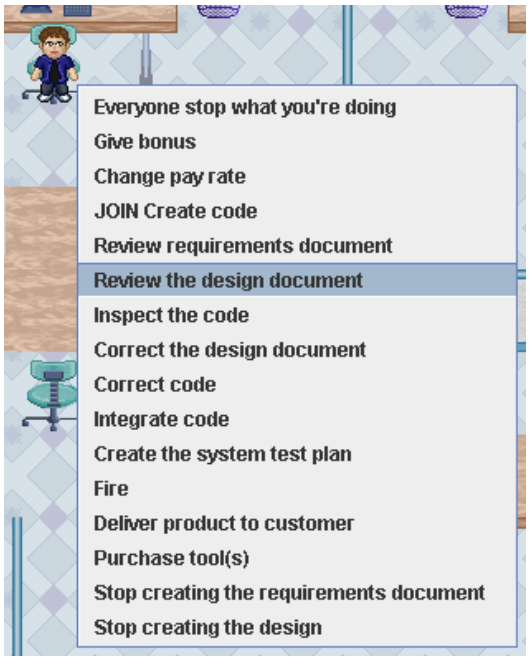


**Figure 39: Right-click Menus on Employees.**

Detailed information about each object (artifacts, customers, employees, projects, and tools) can be obtained by clicking on the corresponding tab for the object's type in the upper left hand corner of the interface, and then choosing the image representing the desired object. (In addition, to promote consistency and intuitiveness, any employee icon appearing in the office or in the panel on the right-hand side of the interface can also be clicked on to bring up their information.) Doing so brings the object's icon and all of its attributes into view in the bottom of the user interface. In Figure 38, the requirements document object is in focus with its attributes of name, number of known errors, and percent complete all shown.

As can be seen in Figure 38, for each group of objects appearing in the upper right hand side of the user interface, there is also a button labeled "ALL". This button brings up an at-a-glance, tabular view of all objects in that group. For instance, when the employees are in focus, the "ALL" button brings up a view of all of the employees and their attributes, as shown in Figure 40. By right-clicking on a column, the table can be customized by hiding and un-hiding columns to focus only on the attributes that are relevant to the current task. The at-a-glance view was designed particularly for situations in which the objects need to be compared quickly, such as the activity of allocating employees to tasks. A player can use this view to gain a rapid overview of which employees possess which strengths, and assign them to tasks accordingly.

We also included a panel on the right side of the interface that lists all of the activities in which each employee is currently participating. This is provided so that the player can be continuously aware of what everyone is doing and have this important information at hand to assist them in making decisions about their next steps.

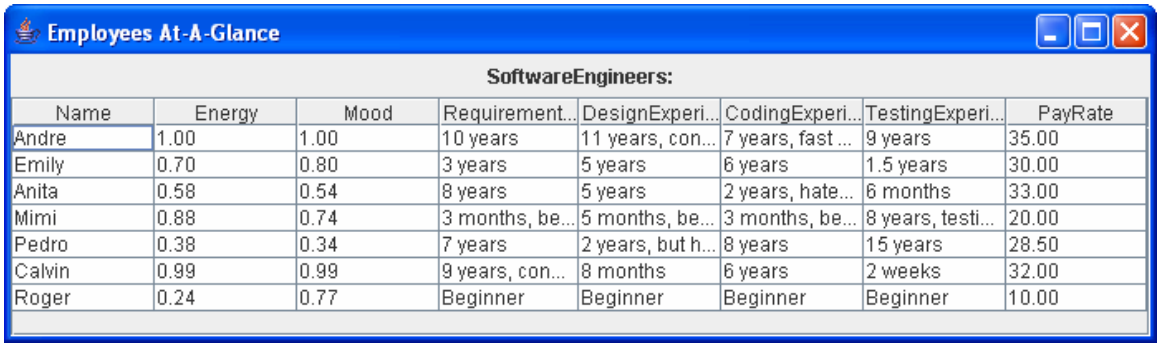


**Figure 40: At-a-glance View of Employees.**

During game play, the player also has access to the "information" and "restart" buttons in the upper left hand corner (labeled with an "i" and an "r" respectively). The "information" button brings up the starting narrative again, so that the player is always able to review the simulation's goals and success criteria, as well as any guidance that is given in the narrative to help the player along the way. The "restart" button, as its name indicates, restarts the game.

The "Analyze" menu in the upper left-hand corner launches the explanatory tool (see Chapter 8). (Currently, this menu is only enabled once a game has been finished and the score has been revealed, as the current version of the explanatory tool is designed as an end-of-game tool. However, in the future we plan to make it accessible at any point during the game so that the player can use the explanatory tool to see intermediate traces of their game in progress, as will be discussed in Chapter 12.) While the explanatory tool is running, the player can also navigate around the game to view any information needed (although the clock and the employee right-click menus are disabled at this point).

### 6.1.1 Game Play Example

To provide an example of what a SimSE simulation game is like, we will use a brief scenario of how a student may use SimSE in completing the task of developing a

software product from requirements specification to product delivery. In addition, this example will illustrate how some software engineering lessons are exhibited during game play and demonstrate our key decision to make SimSE teach by rewarding good software engineering practices and penalizing deviant ones. This particular scenario is based on SimSE's waterfall model, presented in Section 7.1.

When the game begins, the player sees a starting narrative that describes to them the goals of the simulation. In this example, the starting narrative is the following (taken from Figure 37):

> *"Welcome to SimSE! Your task is to create Groceries@Home, a Web-based system that will allow people to place orders over the Internet for groceries to be delivered to their homes. The customer is the Grocery Home Delivery Service, a company who, up until now, has taken orders for groceries solely by telephone, but now wants to step into the information age. Your budget is $280,000, and you have 1,350 clock ticks to complete the project. However, you should keep checking your project info to monitor this information – the customer has the tendency to introduce new requirements, and will sometimes give you more time and/or money along with those new requirements. Your final score will be out of 100 points, and it will be calculated based on how complete and error-free your code is, whether your code is integrated or not, and how well you stick to your budget and schedule. Good luck!"*

The first step this player takes is to go through some of their resources and assess what they have to work with. The player brings into focus the at-a-glance view of all

employees and views their skill levels in each area, noting who is good and bad at the different tasks. They then look at each tool and note its cost. The player sees that two of the tools, the JUnit automated testing tool and the Eclipse IDE, have a cost of zero, so they immediately "purchase" those two.

Since this is the waterfall model, the player decides to start out having their employees specify the requirements for the product. Because the employees Anita, Calvin, Pedro, and Andre (see Figure 40) have the most experience in requirements, these are the ones the player assigns to start creating the requirements. The player then steps the simulation forward 20 clock ticks, after which they see that the requirements document is 7% complete. Now that some requirements have been specified, some of the other employees can start reviewing them. The player assigns the rest of the employees, Mimi, Roger, and Emily, to review the requirements document. The player steps forward 20 more clock ticks, and sees that the requirements document is now 14% complete, and the reviewers have discovered three errors (see Figure 41). At this point the player is thinking that requirements specification is going a bit slower than they would like, so they decide that it might be worth the $10,000 to purchase the requirements capture tool, which they do. They then step forward another 20 clock ticks, and see that the requirements document is now 25% complete, and are pleased that their purchase seems to have sped things up by a factor of about 1.5.

We now fast forward a bit, and assume that the employees finished the requirements document, reviewed it, and the player has their requirements experts, Anita, Calvin, Pedro, and Andre correcting it. The player now decides to move on to the design phase. Since the requirements tool seemed to be so helpful, the player also purchases a design

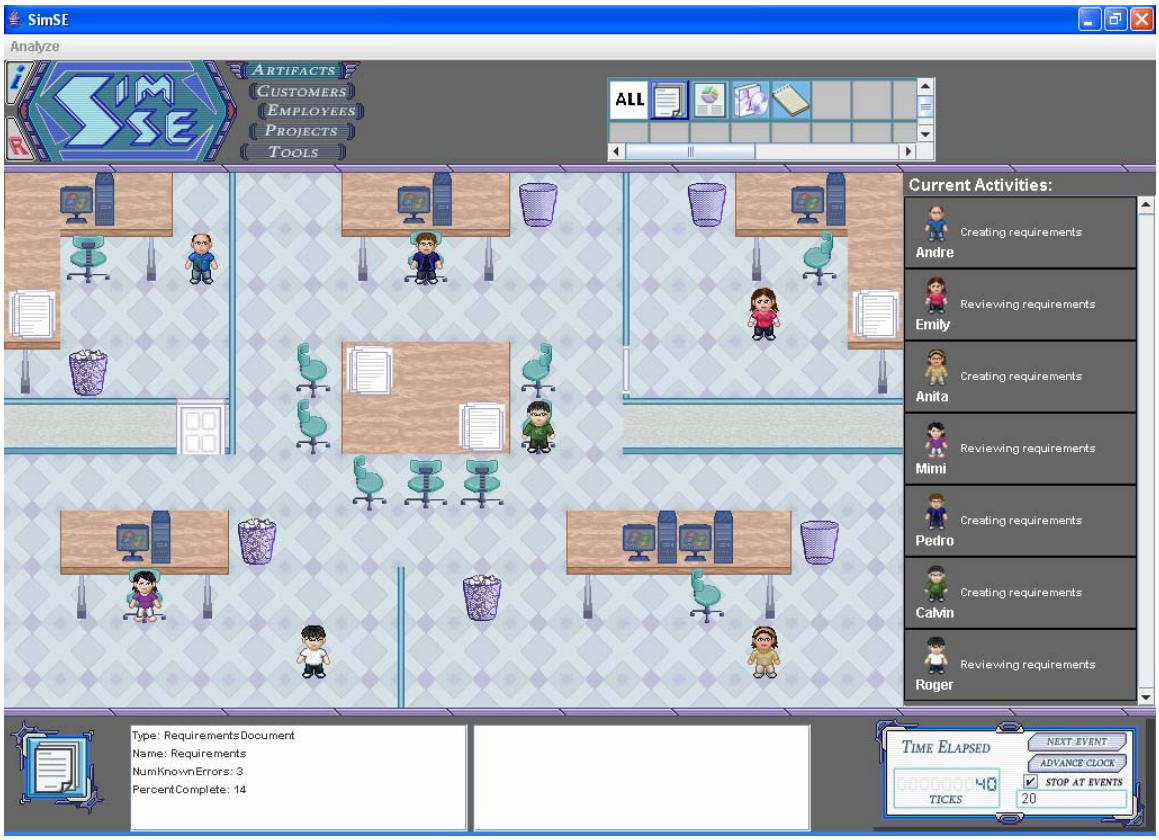


**Figure 41: Requirements Creation and Review.**

environment tool for $5,000. Unfortunately, two of the three experts in design (Andre and Anita) are also requirements experts, so they are already engaged in correcting the requirements document. As a result, the player assigns only one employee who is experienced in design (Emily) along with two less-experienced designers, Roger and Mimi, to start creating the design document. They continue this until the other employees are finished correcting the requirements document. At this point, the designers have been designing for 33 clock ticks, and they are only 4% finished. Now that the other two expert designers, Andre and Anita, are freed up, the player adds them to the designing task, and has all of the other employees start reviewing the design. The player then steps

forward 20 clock ticks and is pleased to see that this reallocation of tasks has sped up design tremendously—the design document is now 10% complete.

The player continues like this until the design document is 100% complete. They then (unwisely) figure that since they had such qualified people working on the design, and they would like to try to finish the project as quickly as possible, they stop the design review process and have all of the employees correct the design errors that have already been found. They move on to the coding phase, assigning all of the coding experts to coding, and they complete the code. Once the player begins inspection, however, they realize that they have made a bad decision somewhere, because inspection seems to be endless, taking 230 clock ticks and finding 194 errors (see Figure 42). (Many of these are errors that were carried over from the design document, which the player will find out later.)

The player has their employees correct all of these errors, and then integrate the code (which also seems to be awfully slow). They then prepare the system test plan, test the system, which reveals 108 errors, and correct these errors. Due to all of the errors in the code, which required extra time spent on inspection, testing and correction, the project is now slightly late (145 more clock ticks than allotted) and $20,580 over budget. As can be seen in Figure 43, the player delivers the product to the customer, and receives a score of 81 out of 100—not a bad score, but it could have been better.

When the hidden attributes are revealed, the player discovers that there were 149 unknown errors in the design document (see Figure 43), indicating that they probably should have had the employees review and correct the design before moving on to coding. (Use of the explanatory tool would underscore this lesson, as well as provide

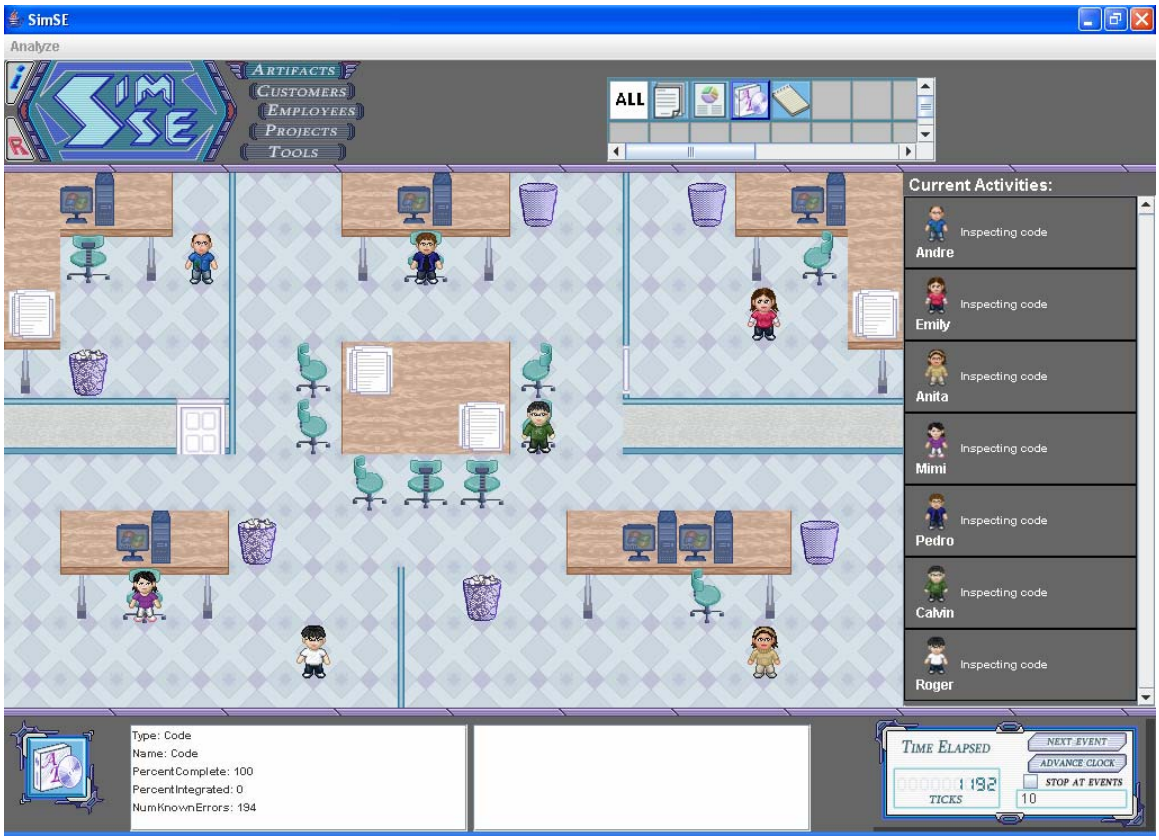


**Figure 42: 194 Errors are Found When the Code is Inspected.**

further insight—See Chapter 8.) Most likely, this player would engage in multiple simulation runs of this same model (along with use of the explanatory tool) in order to try to correct their mistakes, explore different approaches, and gain a thorough understanding of the lessons and the process being taught.

## 6.2 Design and Implementation

The overall architecture of SimSE was shown in Figure 2. In this section, we will focus on the part of the architecture that corresponds to SimSE's game play, namely the simulation environment that embodies a custom game generated from a simulation model. The internal design of the simulation environment is shown in Figure 44, and

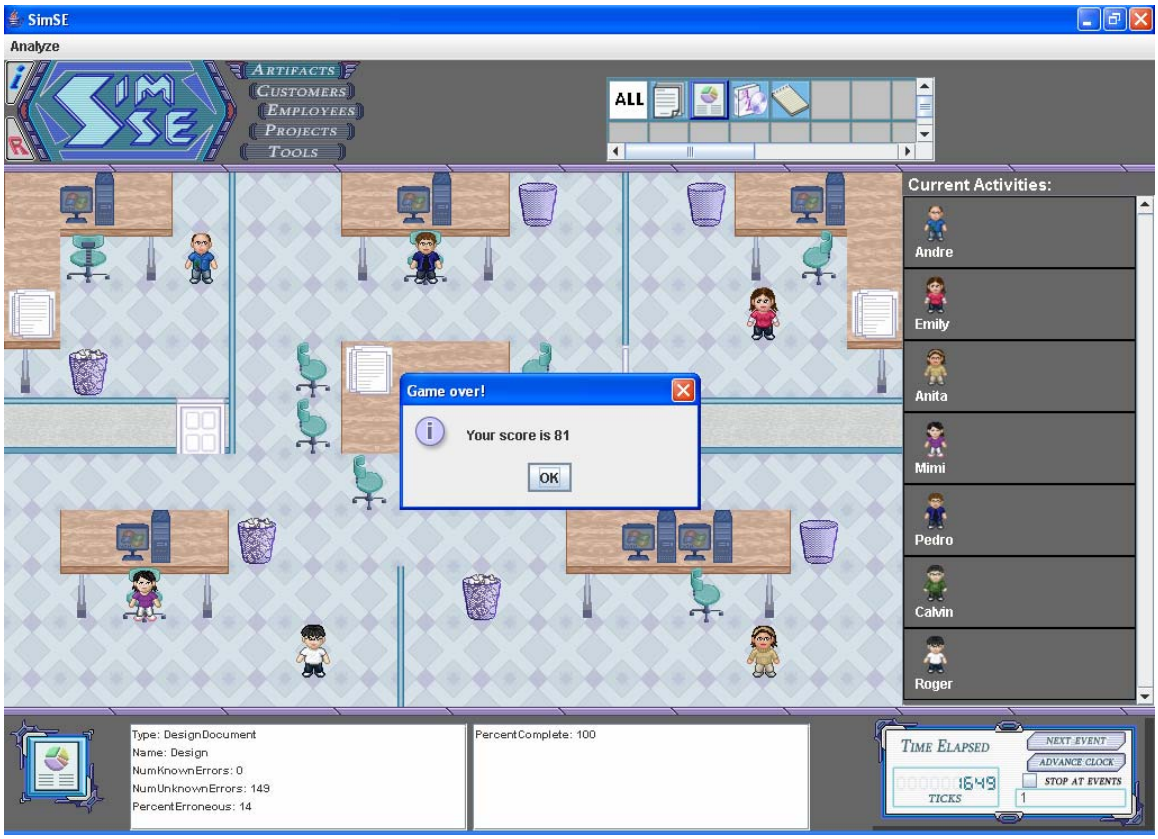


**Figure 43: A Score is Given and Hidden Attributes are Revealed.**

contains five major components: the GUI, the engine, the logic component, the state, and the explanatory tool.

The GUI is a simple component that contains all of the game's user interface components and handles user actions. The engine component has two main functions. First, when the game begins, it runs a startup script that creates the start state objects and adds them to the proper repositories in the state component. Second, during the simulation, the engine drives the simulation by responding to clock events sent by the GUI and notifying the rest of the simulation to update itself accordingly.

The logic component is the heart of the simulation, and contains five major components: the trigger checker, the destroyer checker, the rule executor, the menu input

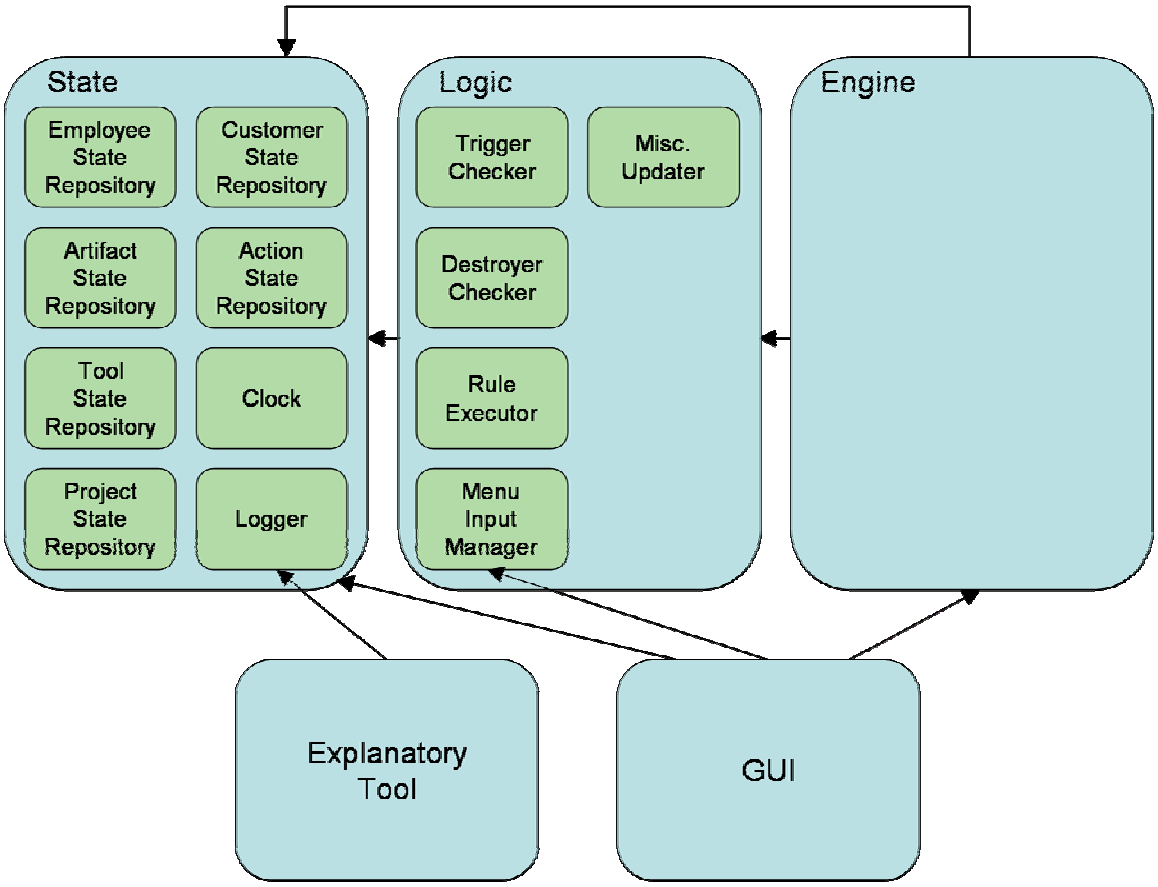


**Figure 44: Simulation Environment Design.**

manager, and a miscellaneous updater. The trigger checker knows the conditions that correspond to each trigger in the simulation, and is responsible for checking if these conditions are met by querying the state component. The trigger conditions are checked at every clock tick, as well as when an action is started, stopped, or when a rule is fired. If a trigger condition is met, that trigger is fired. In a similar manner, the destroyer checker checks which destroyers should be fired.

The rule executor contains all of the rules embodied in the simulation model. Every clock tick, it queries the state to check which actions are currently occurring. For each action that is occurring, it executes the appropriate rule(s). For instance, it would be responsible for causing the size of a code artifact to increase (by the additive productivity

levels of all the employees working on it) every clock tick that a “create code” action is occurring.

The menu input manager receives from the GUI all of the user actions performed through employee right-click menus. This component knows which menu commands correspond to which action triggers and destroyers, and causes the corresponding trigger or destroyer to fire (sometimes after asking the user for more information, such as which other participants they would like to participate in an action). Finally, the miscellaneous updater performs various maintenance and cleanup activities every clock tick, such as clearing employee overhead text and menus and telling the action state repository to update the elapsed time of each action.

The state component, as its name indicates, holds the current state of the simulation at any given time by keeping track of the state of all objects and actions. It does this by storing these objects and actions in “repositories”. Each object meta-type (employee, artifact, tool, project, and customer) has a corresponding repository (employee state repository, artifact state repository, tool state repository, project state repository, and customer state repository, respectively) where all objects of that meta-type are stored, and which can be queried at any time to find out the current attribute values of a particular object. There also exists an action state repository, which contains all of the actions that are occurring at a given time. This repository can be queried to obtain information about an action, such as who its participants are, how long the action has been occurring, and which participants are active or inactive.

The state also contains a clock, which simply keeps track of the time—the current clock tick of the simulation. The logger component (inside the state) and the explanatory

tool component both function for the express purpose of the explanatory tool, so they will be explained in Chapter 8 (the chapter that presents the explanatory tool).

The overall loop the environment performs during game play is as follows: The user drives the simulation through the GUI by commanding the clock to step forward in time. This command is passed to the engine, which responds to clock events by informing the state and logic components that time is stepping forward and that they should update themselves accordingly. When the state receives this notification to update, it tells the clock to update itself (at which point it increments the time) and the logger to update itself (see Chapter 8). When the logic receives the command to update, it tells the trigger checker, destroyer checker, rule executor, and miscellaneous updater components to perform their various updating functions—the trigger checker to check triggers, the destroyer checker to check destroyers, the rule executor to execute rules, and the miscellaneous updater to perform its updates.

The generated simulation environment code is in Java, and for the models built to date, the size of each generated package ranges from about 12,000 (inspection model) to about 47,000 (Rational Unified Process model) lines of code (not including the generated explanatory tool code).

# 7. Models

A SimSE model embodies the lessons a model developer (generally a software engineering instructor) wishes to teach to the players of the resulting generated game (generally software engineering students). We have developed a base set of models and made them available with SimSE. Doing so serves three main purposes for the research: First, these models are necessary in order for educators to be able to use SimSE with students. Second, having a set of models available to instructors who wish to use SimSE makes it easier for them to use it: Pre-existing models will provide examples that modelers can either look at to help them in building their own models, or extend and/or modify for their own purposes. Additionally, instructors who do not wish to build their own models will have a variety of ready-made ones from which to choose. Third, building a variety of different models has demonstrated the capabilities of the modeling approach and provided us with a significant amount of data about the strengths and limitations of the model builder and modeling approach, as well as possible improvements that could be made to each (see Sections 4.3 and 9.5).

To maximize SimSE's applicability and evaluate its modeling approach, it is important to not only build a significant number of models, but to also ensure that these models cover a wide variety of different processes. Hence, we have built models that fall into three distinct categories:

- **Classic approaches:** This category consists of those process models that are well-known and embody an entire software lifecycle. These include a waterfall model, an incremental model, and a rapid prototyping model.

- **Modern approaches:** These are full life cycle models that have been developed in recent years and are less traditional. Modern approaches that we have modeled are Extreme Programming and the Rational Unified Process.
- **Specific models:** In contrast to the models that embody an entire software life cycle, specific models are on a smaller scale and portray only a specific sub-process within the life cycle. We have developed one model within this category—one of a code inspection process.

The remainder of this chapter describes each of the models we have developed in detail. Because of our key decision to base SimSE's models on research literature, a model is essentially a series of lessons taken from this literature and encoded in a game. Therefore, for the most part, we will largely describe each model in terms of the lessons it teaches and how those lessons are expressed during game play.

## 7.1 Waterfall Model

The waterfall model was our initial attempt at building a SimSE model. Although the waterfall is not the most interesting or challenging life cycle model that exists, it is probably the most well-known process, and its simplicity allowed us to evaluate and demonstrate the principles of the environment. Moreover, because the waterfall model is a relatively simple and straightforward process, we were able to also include in this model several general, non-waterfall-specific software engineering lessons.

Since this particular model was purposed to emulate a waterfall process, we developed the model to reward the player for following the proper steps and practices of the waterfall model and penalize them for doing otherwise. In parallel, we aimed to teach a number of overall lessons about the software engineering process in general. These

lessons were taken from our compendium of 86 “fundamental rules of software engineering”, mentioned in Chapter 4 (and listed in Appendix A). The following two waterfall-specific lessons were implemented in this model:

- *Do requirements, followed by design, followed by implementation, followed by integration, followed by testing.* The player must adhere to this sequence, although they can do some activities in parallel, as long as they are not performing a later development activity for a requirement that has not been worked on in an earlier phase. For instance, a player may have their employees work on coding at the same time they are working on design, as long as the code is not more complete than the design document, in which case the player would incur a penalty—namely, development of the later artifact becomes slower than usual, and more errors are introduced. This enforces that while each complete phase does not have to be entirely finished by the time the next phase begins, each feature must be specified before it is designed, designed before it is coded, coded before it is integrated, and integrated before it is tested. Furthermore, if the player goes back to a previous phase and works on, say, the design document after they have already worked on the code to a completeness level greater than that of the design document (e.g., the code is 90% complete whereas the design document is only 60% complete), some new errors will appear in the code. This represents to the player that they implemented some features for which there was no design, and now that they have gone back and properly designed those features, they have found that much of this un-designed code was erroneous.

- *At the end of each phase, perform quality assurance activities (e.g., reviews, inspections), followed by correction of any discovered errors.* Although working on two phases in parallel is acceptable in certain situations (as mentioned previously), any uncorrected errors in an artifact (e.g., requirements document) will be carried over into the next phase's artifact (e.g., design document) if they are not discovered and corrected before work on the next phase begins.

In addition to these waterfall-specific lessons, the SimSE waterfall model also aims to teach the following lessons that are general to most software engineering processes:

- *If you do not create a high quality design, integration will be slower and many more integration errors will be introduced.* The speed of integration and the number of errors that are introduced into the code during integration are directly dependent on the completeness and correctness of the design. If the player spends an adequate amount of effort and resources on the design phase, they will be rewarded with a faster integration and a more correct system.
- *Developers' productivity varies greatly depending on their individual skills, and matching the tasks to the skills and motivation of the people available increases productivity* [18, 26, 121]. The employees that the player is given to manage each have different skill levels in requirements, design, coding, and testing. In any given development activity, this skill level is the greatest influencing factor on their productivity, as well as on the rate at which they introduce errors into the artifact on which they are working. Hence, when a player assigns tasks only

to employees that are skilled at those tasks, the project will be finished faster, and fewer errors will be introduced.

- *The greater the number of developers working on a task simultaneously, the faster that task is finished, but more overall effort is required due to the growing need for communication among developers (Brooks' Law)* [22]. For each development activity, the productivity of each developer is decreased by a small factor for each additional employee working on that same activity.
- *Software inspections are more effective the earlier they are performed* [141]. To demonstrate the fact that software inspections are more effective the earlier they are done during development, the more integrated the code is, the less effective an inspection will be at finding errors in the code.
- *The better a test is prepared for, the higher the amount of detected errors.* Before doing system testing, the player should ensure that their employees have developed a system test plan. The more complete and correct that test plan is, the more efficient testing will be.
- *Monetary incentives increase motivation, which leads to increased productivity (but faster expenditures)* [141]. The player can give their employees pay raises and bonuses, which will increase their mood by an amount proportional to the amount of the raise or bonus. As a result, the employee's productivity will also increase. Bonuses have a short-term effect on productivity, while pay raises have a longer one. Both pay raises and bonuses are taken out of the project budget, so players must use caution in how they dole out such incentives.

- *The use of software engineering tools leads to increased productivity* [141]. The waterfall model allows the player to obtain up to four different tools: a requirements capture tool, a design environment, an integrated development environment (IDE), and an automated testing tool. Some of these tools have a cost associated with them, while others are free (in accordance with the common practice of downloading free software engineering tools off of the Internet), so the player must also balance the potential benefit of the tool with the monetary cost. Each tool has a productivity increase factor and an error rate decrease factor, both of which are hidden from the player (until the end of the game). When a tool is used in a development task, the productivity of the developers involved in that task is increased accordingly, and at the same time the error rate is decreased.
- *New requirements frequently emerge during development since they could not be identified until portions of the system had been designed or implemented* [42]. During game play, the customer introduces new requirements at random. In some cases, they will also give the player more time and/or money to finish the project. Introducing new requirements increases the required size of the artifacts, so if the player has time it is in their best interest to go back and work these new requirements into each artifact.

In addition to these, there are a number of other general workplace issues not specific to software engineering that are included in the model to add realism and make things more interesting. For instance, employees sometimes get sick, take breaks when they are tired,

become less productive when they are tired, and quit when they are upset about something significant (e.g., a pay cut).

## 7.2 Inspection Model

After building the large waterfall model that depicted an entire software life cycle from requirements analysis to product delivery and contained numerous parallel and interacting effects, we decided to take the opposite approach with our next model. The inspection model was our initial attempt at building a model of a small, specific process within the software life cycle that teaches only a few, very focused lessons. We chose code inspection as the subject of this model because there exists real-world data in the literature regarding the best practices of this process. Because of this data, and because the process simulated in this model was so small compared to the other models, we were able to teach a small set of very well-defined lessons related to code inspections. To keep the model small and specific, we chose to concentrate only on these lessons and ignore other details of the inspection process, such as the different roles and the detailed steps involved in the process. Thus, a player of this model has three main concerns: choosing the right number of people with the right qualifications, choosing the right size of code to inspect, and choosing the right size of inspection checklist to use. These decisions are based on the following lessons, which are collectively taken from [59], [73], and [145]:

- *A code inspection takes place in a group setting.* While the office layout of the other models all portray employees working alone (for the most part) in their cubicles, this model's layout shows a large conference room with several people sitting around a table (see Figure 45).

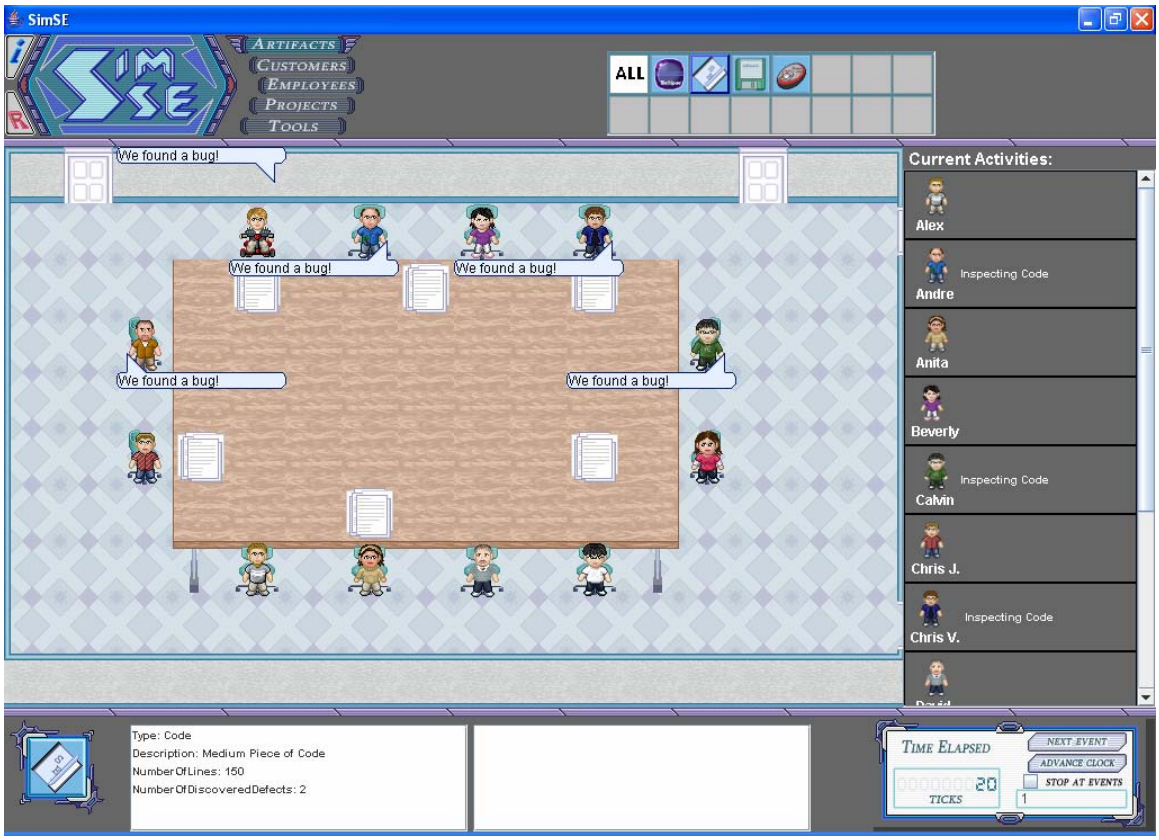


**Figure 45: Screenshot of the Conference Room Layout of the Inspection Game.**

- *The productivity of an inspector depends equally on their familiarity with the product and their inspection experience.* Each employee in this model has a project experience attribute and an inspection experience attribute. Both of these values contribute equally to the employee's effectiveness in finding bugs.
- *A four-person inspection team is ideal, and is twice as effective as a three-person team.* The player will find that the most bugs are discovered in the shortest period of time if they use four people, and only half as many are found in the same period of time if they use three people.
- *A larger inspection team does not necessarily equal a more productive inspection team.* After each bug is found, the employees discuss it. As the

number of people in an inspection team increases, it takes the group a shorter time to find each bug, but an exponentially longer time to discuss each one, as the number of communication links (and opinions) also increases.

- *A code inspection checklist should be no larger than one page.* When the player starts the inspection meeting, they must choose one of three different sized checklists to use: a quarter-page list, a one-page list, or a five-page list. Using the one-page checklist causes the inspectors to find bugs the fastest, while both the quarter-page checklist and the five-page checklist have a speed-up effect that is one-half as fast as the one-page list.
- *The piece of code being inspected should be less than or equal to 300 lines, but less than or equal to 200 lines is ideal.* At the start of the inspection meeting, the player also must choose from three different pieces of code to inspect, each with a different size (20, 150, and 1500 lines of code, respectively). Inspecting the 150-line piece of code will yield the most productive inspection meeting in terms of the speed with which bugs are found, while the productivity of inspecting the 20- and 1500-line pieces of code is only half as much.
- *An inspection meeting should last no more than 2 hours.* In the starting narrative, the player is notified that in this model, one clock tick is equal to one real-world minute. Thus, after 120 clock ticks the developers declare, “I’m so tired…” and their productivity wanes significantly.

## 7.3 Incremental Model

With our incremental SimSE model we returned to portraying an entire life cycle, but aimed to simulate a different approach, although one that is still well-known and

considered “classic.” This model was designed to embody an iterative development process that values regularly providing the customer with incremental versions of the software throughout the life cycle. We accomplished this using a module-based approach that accommodated the partial submissions of a project throughout the process. In this model, a module represents an anonymous part of the project that can be worked on and developed independently of the other parts. Development actions, such as risk analysis, requirements analysis, design, and implementation are performed on each module separately. We specifically designed the attributes of a module to facilitate teaching about incremental software process approaches. In particular, a module in this model has the following attributes (none of which are visible to the player at the beginning of the game, but instead are only revealed through various analysis activities performed on each module):

- **Value:** This represents the priority of this module to the customer, which in turn controls how much the completion of that module will help the player’s final score. Discovering this value through risk analysis can help the player to prioritize the completion of each module if time constraints prevent a complete submission.
- **Inflexibility:** This value signifies the degree to which the customer will be unwilling to accept deviations from their ideal concept of the module. The higher the inflexibility of a module, the more the player’s score will be hurt if that module is implemented incorrectly. Once again, using risk analysis to discover this value will help to guide the player in prioritizing modules.

- **Changeability:** This value corresponds to how often the module is likely to be changed by the customer and, like value and inflexibility, is also discovered using risk analysis. This value is very important to determining how the player should proceed. Overcoming frequent customer changes is one of the primary challenges of this model, so knowing which modules are most likely to change is one of the most important pieces of information that should be used when devising a strategy for playing the game.
- **Accuracy:** This represents how well the developed module corresponds to the customer's expectations for that module. Accuracy is improved by working on a module's requirements, and it is eroded whenever the customer makes changes to the module. Even if a module is complete, if its accuracy is too low, it might represent no value at all to the customer in terms of the final project. This is especially true in modules with high inflexibility values.
- **Phase difficulties:** Each module contains a difficulty value for each development action that can be performed on it (requirements, design, and implementation). These values can help a player to determine which modules might make good candidates for the basis of early, rapid prototypes. Performing difficulty analysis on a module reveals that module's difficulty values.

The central challenge in this model lies in the player dedicating time to performing various analysis activities in order to reveal these attribute values and gather the information necessary to guide their decisions in completing the process. Careful consideration of each module's attribute values to determine which tasks to perform on which modules in which order should be a player's main concern.

While there are a number of specific software process models that can be classified as incremental processes (Extreme Programming, rapid prototyping, Rational Unified Process), rather than focusing on one of these, in this model we instead endeavored to teach a number of lessons about incremental software processes in general. Surveying several publications about incremental processes [10, 19, 58, 89, 120, 122] revealed the following overarching principles which are inherent to nearly any incremental software process:

- *Software versions should be created early and often.* The player will have a much higher chance of achieving a good score each time they submit a partial build of their project to the customer during development. When a module is submitted to the user, the player gains many benefits. First, many of its hidden attributes are either revealed or clarified, as customer feedback provides insight into their valuation of, and inflexibility about, the module in question. Second, the difficulty of all other actions on a module is reduced when that module is submitted. Requirements is foremost among these, as insight from the customer feedback guides the creation of related documents. Third, the changeability of the module is reduced, helping to overcome the problems related to customer changes described previously. Fourth, all of these benefits are conferred, albeit to a lesser degree, onto other modules in the game that were *not* submitted. This represents the insight gained into the system as a whole via the discussion of one of its parts. When a player submits a multiple, rather than a single-module, build, even greater benefits are reaped, in terms of both the submitted modules and the other modules in the project.

- *Early versions can be used to gather information about how to develop later versions.* As we have discussed already, a partial build submitted to the customer causes some of each module's hidden attributes to be revealed for the first time (if no risk analysis or difficulty analysis was done) or clarified (if these values were previously estimated through risk analysis or difficulty analysis). This represents that incremental versions of the software can provide useful aids for discussion with the customer about their desires for the project, which can help to steer development of subsequent versions to the customer's liking.
- *Risk analysis should be used liberally to shape the process.* As mentioned previously, performing risk analysis on a module causes its value, inflexibility, and changeability to be revealed, all of which are the most important factors in determining which modules should be developed in which order.
- *Frequent iterations should be used to ease the difficulties of changing requirements.* To encourage an incremental approach, this model simulates a customer that makes frequent changes to their concept of the product. This frequency is lessened significantly each time the player submits an intermediate version to the customer (and hence, completes an iteration).
- *Do requirements before design.* Although this is a general best practice of software engineering not specific to incremental models, this simulation model enforces it in a way that demonstrates the role of requirements in incremental processes in particular. First, working on the requirements for a module increases the accuracy of the module, and is the primary means of doing so.

Thus, implementing a module without thoroughly working on its requirements will leave the player with a fully implemented module that most likely does not meet the customer's needs. In the case of a high-inflexibility module, this can be catastrophic. Second, working on requirements for a module reduces the changeability of that module, representing that engaging in requirements discussions with the customer helps to settle any uncertain issues that may be present. On top of these incremental-specific effects, the value of requirements in general is illustrated in that the design of a module will be sped up if its requirements have been specified beforehand.

- *Do design before implementation.* Again, while this is a well-known theme of software engineering in general, our model illustrates it in a way that is unique to an incremental process in which the requirements are likely to change. In particular, the design greatly determines the difficulty of evolving the code. Thus, while a player may be able to implement their code without a design, if the customer makes changes to their desires for the module, adjusting the code to restore its accuracy to the customer's demands will be nearly impossible without a design. In addition, creating a design for a module also increases the ease of implementation and integration of that module.

## 7.4 Extreme Programming Model

After successfully completing one "specific" and two "classic" software process simulation models, we decided to stretch the modeling capabilities of SimSE in a different direction by modeling a modern, agile process: Extreme Programming (XP). XP has numerous facets and dimensions, too many to go inside a single SimSE model.

Hence, we chose a subset of XP lessons that covers some of the most central tenets of the process, resulting in a model that encourages employees working in pairs using test-driven development to create small, frequent releases that are continuously tested, refactored, and integrated. Specifically, this model embodies the following lessons (all taken from [144]):

- *Pair programming delivers more efficient results than programming individually or in small groups.* If the player instructs their employees to "pair up" with each other, development will be much faster. The higher the percentage of team members that are paired up, the faster development will be.
- *System development should consist of small, frequent releases.* The model rewards an approach in which code is developed incrementally and releases are made at increments of about 20% completion (i.e., a release at 20% completion, a release at 40% completion, etc.) Increments significantly larger or smaller than this will decrease the efficiency of testing, refactoring, and integration of the code.
- *Testing, refactoring, and integration should be done frequently.* Testing, refactoring, and integration should all be done (in that order) after the implementation of each increment. Otherwise implementation will become increasingly slower as it progresses, indicating that the system is getting clunky, buggy, and difficult to evolve. The order of test, refactor, integrate is enforced in that if the preceding step (e.g., refactoring) is not completed, the subsequent step (e.g., integration) will be slowed down.

- *Test cases should be created before implementation.* Both implementation and testing will be significantly slowed down if test cases are not created first, demonstrating the value of test cases as a guide for both activities.
- *Development of a rapid prototype should be the first step before any implementation of the actual system.* The more complete the rapid prototype, the faster implementation of the actual system will be.
- *Coding standards should be created and used for development.* If the player has their employees invest time to create coding standards at the start of development, implementation, testing, refactoring, and integration will go much faster, representing that having a standard to follow will help make all development activities more efficient.
- *Provide an open workspace in which there is a central area where pairs can collaborate surrounded by private spaces for encouraging focus.* As can be seen in Figure 46, the office layout for this model fits this description of a workspace conducive to XP. Enclosed, "private" spaces are lined up along the top part of the office, while the bottom part houses the open areas for collaboration.

## 7.5 Rapid Prototyping Model

With our rapid prototyping model we set out to do a number of things differently than we had done in the other models, both to test the different aspects and capabilities of our modeling approach, and to create more variety in our set of models. First, although the rapid prototyping model, like the waterfall and incremental models, depicts an entire software life cycle, we made it much more focused and did not include some of the effects that were included in the other models. For instance, this model ignores budget,

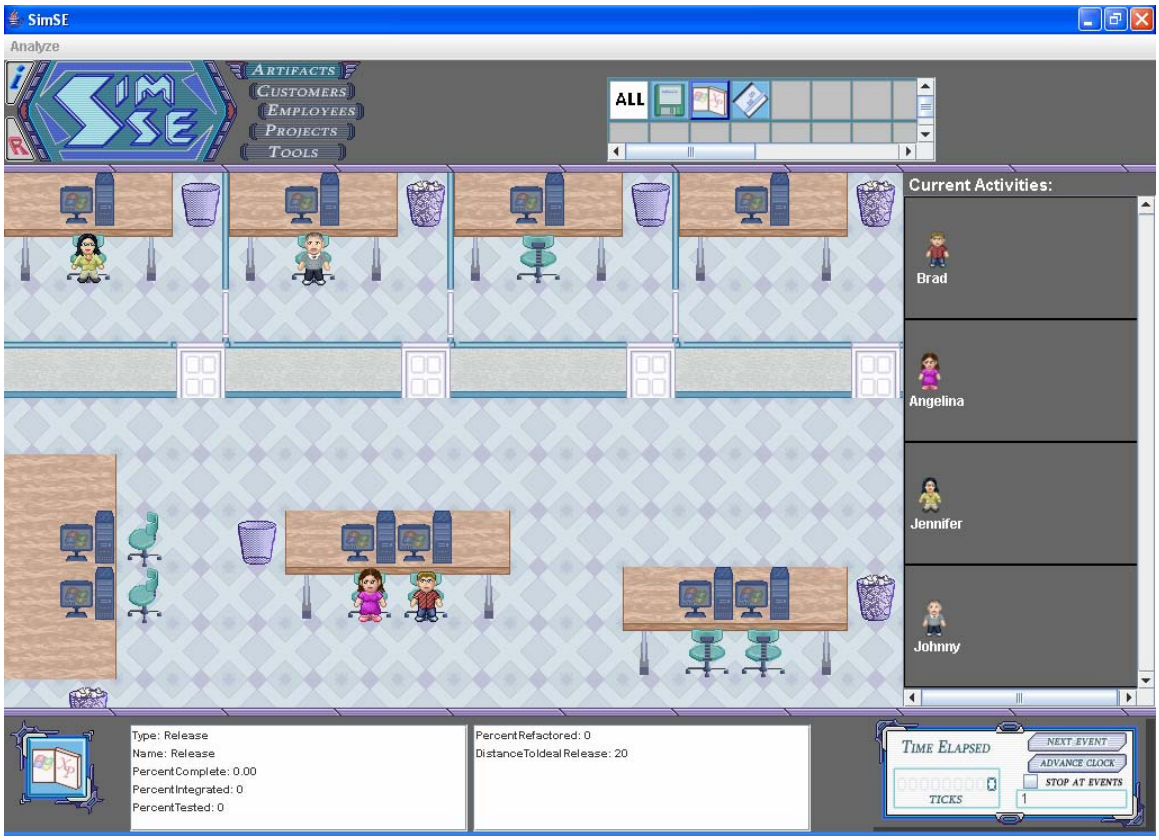


**Figure 46: The Open Workspace Depicted in the Extreme Programming Model.**

considers all employees to be equally skillful at every task, and omits explicit review, inspection, testing, and correction activities (the starting narrative states that these activities are implied in the development of each artifact). These simplifications were made in order to put the focus strongly on the prototyping process and create a model that depicts an entire software life cycle but focuses on one particular aspect of that life cycle.

The second difference in this model concerned the use of the employee speech bubbles. In the first three models, we had mainly used the speech bubbles for making simple statements that gave very little insight, such as, “I’ve finished coding!” or “I’m reviewing the requirements document now!” In this model we attempted to make the statements appearing in these speech bubbles more meaningful by using them to provide

guidance to the player in taking a successful route through the simulation. For instance, when the player has spent too long on the prototype, the employees will announce that the customer has called and complained about not seeing a prototype yet. As another example, once the employees are told to start the initial meeting with the customer to outline the requirements, they declare, “We’re off to meet with the customer to get an idea of what they want for their software. Monitor the requirements document to see how requirements discovery is progressing.” Likewise, when the employees are told to have the customer evaluate the prototype, they state, “We’re off to the customer now to see what they think of the prototype! Monitor the requirements document and the prototype to see the progress of this meeting.” Comments such as these are designed to both provide some more explanation about what the employees are actually doing while completing these tasks and inform the player about how they can find out what the results of the task are. For example, when a player follows the directions to “monitor the requirements document and the prototype to see the progress of [the prototype evaluation] meeting,” they will see that this task results in new requirements being discovered, communicating to them that a prototype can be a good tool for eliciting requirements from the customer.

Finally, in this model we experimented with making the scoring easier. In previous models, players were penalized severely for deviations from an “ideal” approach, making it difficult to obtain a good score. In this model, we made this less severe, making it easier to achieve a good score.

Our rapid prototyping model depicts a “throw-away” prototyping process in which a rapid prototype is iteratively developed with customer input, and then thrown away, after

which a final version of the software is built. (This is in contrast to an evolutionary prototyping process in which the rapid prototype is evolved to become the final version of the software.) The rapid prototype is basically used as a requirements analysis tool in this model, as the process of developing the prototype and subsequently discussing the prototype with the customer is the primary means of discovering the requirements for the system. The basic process that the model enforces and rewards is the following: (1) Outline the requirements for the system with the customer, (2) develop a prototype, (3) have the customer evaluate the prototype, (4) continue re-developing the prototype and having the customer re-evaluate the prototype until the player decides that it is time to move on, and (5) follow the waterfall model for development of the final system. (Although we could have built any one of a number of different life cycle models to follow after the rapid prototyping cycle, we chose the waterfall model because its simplicity allows us to keep this simulation model focused on the rapid prototyping process, in which the model's central challenges lie.) Aside from this overall process, our rapid prototyping simulation model also teaches the following lessons (all taken from [134]):

- *A rapid prototyping approach is appropriate for situations in which the requirements are unclear or not well-known to begin with.* In the starting narrative of this model, after describing what the project is, the player is told, "Your customer is not entirely sure what they want the system to do or to look like, so unearthing their requirements might take a little bit of work and creativity." This piece of text was included specifically to hint to the player that a rapid prototyping approach should be considered in situations such as this one.

When the customer is unsure of their requirements, a rapid prototype can serve as a tangible tool with which to transform vague requirements into concrete ones.

- *A rapid prototype can be an effective means of eliciting requirements from the customer.* As mentioned previously, discussing a prototype with the customer is the primary means of discovering requirements in this model. Each time the developers bring a revised prototype to the customer for evaluation and they spend time engaging with the customer in a discussion about it, new requirements are discovered as the ability to look at and play with an executable prototype of the system gives the customer more ideas about what they want. However, there does come a point when all of the requirements that are going to be discovered are discovered, and in this model that occurs after three rounds of prototype development and customer evaluation.
- *Rapid prototyping can make the rest of development go more smoothly.* The more complete the prototype, the faster requirements specification, design, and implementation will go. This signifies that having a prototype: (1) helps make the requirements clear and (2) gives the developers a head start on design and implementation, as they have already at least experimented with some of the design and implementation issues they will likely encounter in the development of the final system.
- *Rapid prototyping can have a positive impact on the quality of the resulting system.* The completeness of the prototype also has a positive effect on the correctness of the subsequent artifacts (requirements document, design, and

code), representing that prototyping can help to ensure that what the developers are specifying, designing, and implementing matches what the customer wants.

- *Too much or too little prototyping can be detrimental to the project.* Despite a lengthy literature search to determine what the "right amount" of prototyping is, we found no real-world data suggesting a value—only the general consensus that too little prototyping can result in a product that does not fully meet the customer's needs, and too much prototyping can be an unnecessary waste of time, as the return on the investment into prototyping starts to dwindle at a certain point. As a result, we set this "right amount" around 60%, meaning that the player is rewarded for developing the prototype to include about 60% of the total discovered requirements at any given point. We experimented with different numbers and found that this value did well at communicating to the player that there is a balance that must be achieved between too much and too little prototyping, and this balance falls somewhere in the middle region of the spectrum from prototyping no requirements to prototyping all of the requirements. This is enforced in the model in three ways. First, as development of the prototype begins to go past the 60% mark (indicating that they are probably spending too much time on it), the developers will announce to the player, "The customer called – he's anxious for the prototype and wants to know what the hold up is!" This is designed to give the player a hint about what the right stopping point for prototype development is in the game. Second, the factor by which the completeness of the prototype speeds up development of subsequent artifacts (requirements, design, code) is maximized at a prototype

completeness level of about 60% (meaning more than 60% of the total discovered requirements are incorporated into the prototype), and significantly levels off shortly thereafter. This is designed to illustrate that the most important requirements should be prototyped so that they can become well-understood, but if less important requirements that the customer is not too particular about begin to be included, precious time that could be spent implementing those requirements into the final system is being wasted. Third, this effect of diminishing return on investment is also present in the prototype's effect on the correctness of subsequent artifacts. The difference between the correctness of, say, a design for which a prototype is 100% complete and the correctness of one for which a prototype is 60% complete is miniscule—about 1%.

- *Certain programming languages are more appropriate for prototyping than for implementation, some more appropriate for implementation than prototyping, and some are appropriate for both*. The player must choose both a prototyping language and an implementation language from three choices: Visual Basic, Java, and C++. In our model, we have made the generalization that Visual Basic is the one most appropriate for prototyping, C++ is the one most appropriate for implementation, and Java falls somewhere in the middle and can be used for either one. Choosing C++ for prototyping will make development of the prototype go awfully slow. Choosing Visual Basic for implementation will make implementation go fast, but will result in decreased correctness, illustrating that a prototyping language generally does not have the capacity to implement all of the requirements for the system. Choosing Java for either

activity influences the speed and correctness by an amount that is somewhere in between the two other languages.

## 7.6 Rational Unified Process Model

Our second attempt at building a simulation model of a "modern" approach was our Rational Unified Process (RUP) [87] model. We roughly based our model on a non-computerized RUP simulation game that IBM uses to train their employees in the RUP process, but also added and removed some elements to make it more appropriate and fitting for SimSE. For instance, in the IBM game the player could "progress" through development by answering a series of questions about RUP correctly. In SimSE, development progresses by having employees work on artifacts.

Like the rapid prototyping model, the RUP model also makes extensive use of the employee speech bubbles for conveying important information to the player, even more so than in the rapid prototyping game. Most importantly, in this model frequent intermediate feedback is given to the player through the text in the speech bubbles. This is done in two major ways: through phase assessments and through prototype submissions. Before attempting to end a development phase, the player must have their employees assess the phase. When they are finished with their assessment, the employees let the player know what their determination is—whether all of the work that should have been done during the phase was actually completed, and whether they are on track in terms of budget and schedule. At this time they also suggest to the player what their next step should be: either start planning for the next phase, go back and do some more work in the current phase, or, in the most severe circumstances, quit and abandon the project.

Likewise, when the player has their employees submit a prototype to the customer, the employees report what the customer's reaction was through their speech bubbles. There are three possible reactions that the customer may have: (1) They are happy with the prototype because it covers all of the use cases they had hoped to see there, (2) they are unhappy with the prototype because it does not include all of the critical use cases they had hoped to see, or (3) they feel the prototype is too complete and wish it would have been delivered sooner with less functionality.

The RUP model also makes frequent use of the speech bubbles to provide guidance and hints to the player about how to proceed successfully through the simulation. We have already seen, in the phase assessments example, a case in which explicit next steps are suggested. In addition to cases like these, we also used the speech bubbles to give the player more subtle guidance. For instance, after the player begins a new phase, they must first assign employees to a phase and then assign them tasks before any progress can be made. In order to hint at this and try to ensure that the player does not think that starting a new phase automatically causes some activities to occur (in which case they would likely waste time stepping the clock forward with an idle staff), an employee will say something to the effect of, "We are now officially in the Inception phase—which of us do you want to work on this phase?" Likewise, in order to ensure that the player also knows that they must assign a task to an employee after assigning them to a phase, an employee who just got assigned to, say, the Inception phase says, "I am assigned to the Inception phase and am ready to work. Let me know what to do."

In the RUP model we also tried to make more realistic use of random events than we have done in any other model. So far the only model that has any random events is the

waterfall model, which has only relatively simple ones, such as the customer introducing new requirements and employees getting sick. The IBM game that this model is based on included numerous slightly more complex and realistic "surprise" events that either set the player back or move the player ahead in progress. We incorporated some of these into our RUP model. For instance, sometimes the employees will discover a component from a previous project that implements one of the use cases for the current project, so they can reuse it. As a result, that particular use case in the current project gets automatically set to100% completion. In another random event, the player is notified that the customer has changed the contact person for the project—this new contact person decides to rework some of the requirements, which results in the player losing all progress for one of their implemented use cases.

Probably the most distinguishing factor about the RUP model is its use of the prescriptive aspects of SimSE. In particular, a significant portion of what this model teaches is communicated through the allowable actions a player can take at each point in the process. This is partly by necessity: RUP is a highly dense process with numerous prescribed steps. If we were to make them all available at every point throughout the game, the player would undoubtedly be lost and overwhelmed by all the choices. Instead we limit the choices available at each step, depending on what part of the process the player is in. For example, a player can only develop the architecture when in the Elaboration phase and can only develop code when in the Construction phase. As another example, a player may only plan for one phase after they have completed all of the work required in the preceding phase. This is even taken so far so that, at several points throughout the game, only one choice appears on each employee's menu. Thus the player

is steered much more than in other models and given significantly less freedom. While it may seem that a model like this would be too easy, the RUP model actually has quite a few challenges—challenges that lie mainly in other aspects of the game aside from deciding which steps to take next. These challenges will be brought forth as the model is described in the remainder of this section.

There are two major lessons encoded in the RUP model. The first of these concerns the steps and overall flow of RUP which, as already mentioned, are taught through the allowable next steps a player can take at any point in the game. Figure 47 shows a state chart diagram depicting our RUP model's general flow. (A link to an online version of this diagram is given to the player in the starting narrative of the RUP game.) As the player progresses through the four RUP phases of Inception, Elaboration, Construction, and Transition, their course through the game is as follows: The player starts a phase, assigns employees to that phase, starts an iteration, does some development work, and then either starts another iteration or assesses the phase. If the assessment is negative, they must go back, start another iteration, and do more development work. Otherwise, they may plan for the next phase and then end the phase. Going through the prescribed parts of these steps multiple times is designed to ingrain the RUP model in the player's mind. The variability in this aspect lies in the number of iterations done per phase, a decision the player must make for themselves. If too few are done, time is wasted in performing phase assessments multiple times (a phase assessment will result in negative results if the work for the phase is incomplete, requiring that another phase assessment be performed after more work is done). If the player chooses to do too many iterations, their employees will find themselves assigned to work that is already done. The employees

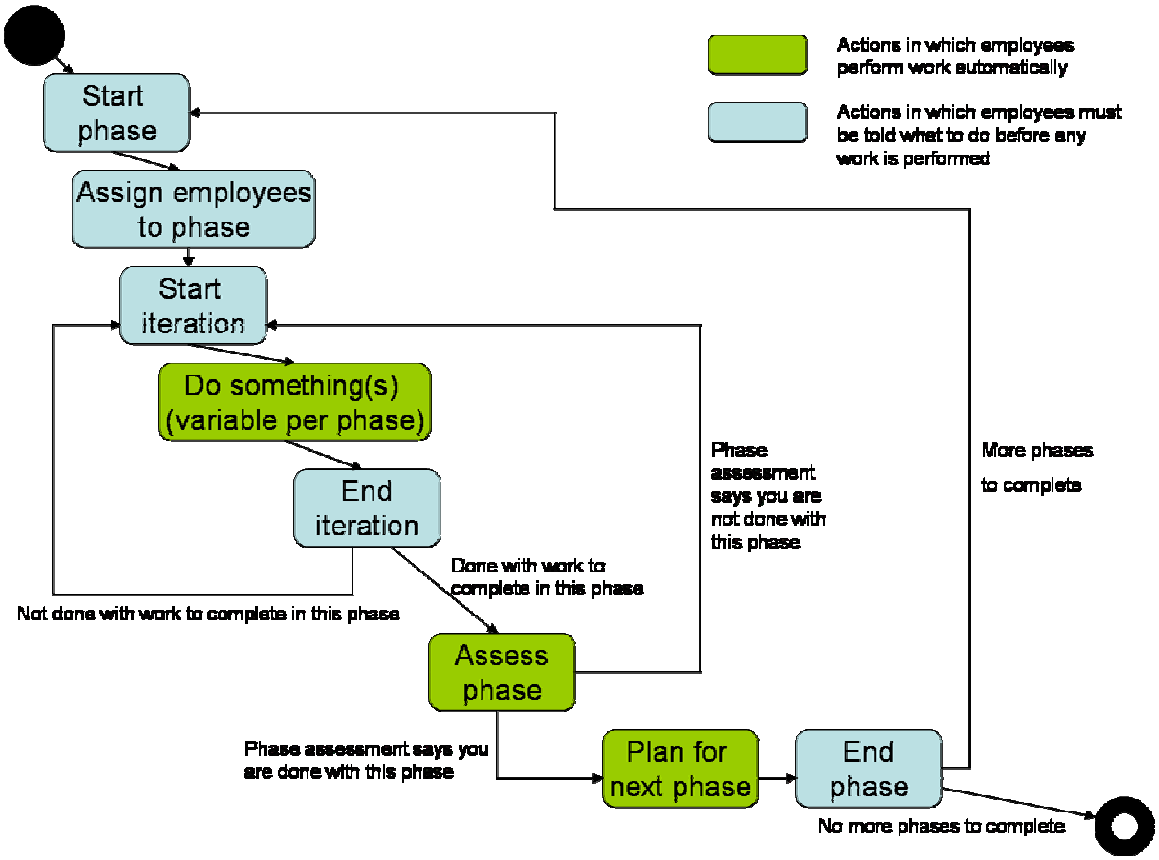


**Figure 47: State Chart Depiction of the SimSE RUP Model's Overall Flow.**

will then have to come back to the player and announce to them that the work is already done. By this time, precious clock ticks will have been wasted in unnecessary communication. Thus, after multiple runs of the game the player will likely be able to come to some conclusions about how many iterations are appropriate in each phase.

The second major lesson taught by the RUP model (and the central challenge of both our model and the IBM game that our model is based on) is efficient allocation of personnel. The player has 10 employees available, each with a skill set and a pay rate. There are five categories of skills: project management, architecture, requirements, design and development, and testing. Each employee has either "low" or "high" skill in each area. In order to enhance realism and add challenge to the model, an employee can

only have, at most, three “high” skills, and their pay rate depends on how many “high” skills they have—those with one skill are paid $300 per clock tick, those with two skills $400, and those with three skills $500. Thus, the player must make careful choices about who to assign to which phase, ensuring that the necessary skills are present to complete the work, but also taking care that they do not assign so many employees that they exceed their budget. Although efficient allocation of manpower is not a lesson unique to RUP, in this game it nevertheless serves as a tool with which to teach about what occurs in the different phases of RUP. Namely, the player must know which activities are performed in each phase so that they can accordingly assign the employees with the appropriate skills to the appropriate phase(s).

In addition to these two major lessons, the RUP model also teaches a secondary lesson about prototyping. At the beginning of the Construction phase, the player is notified that the customer would like to see two intermediate prototypes during the phase. It is up to the player to decide when to submit each prototype—namely, which use cases should be completed and incorporated into each intermediate version. The customer will be “happy” with a first prototype that contains the five to eight most critical use cases (out of 20 use cases for the entire system), and “happy” with a second prototype that contains the eleven to fifteen most critical use cases. For each of these “happy” results, the player will also be rewarded with 5 bonus points (which will be added to their final score). The customer will also be “somewhat happy” with a prototype that contains more than the desired use cases, but will inform the player that they would liked to have seen it sooner with less functionality. These situations will penalize the player by 2 points. If a

prototype does not meet any of these conditions, the customer will be “unhappy”, and the player will lose 5 points.

## 7.7 Discussion

While the models presented in this chapter do teach a number of things about real-world software engineering processes, and accurately represent a number of real-world phenomena, no SimSE model is completely faithful to reality. In fact, due to the educational purpose of these models, we have deliberately designed parts of our models to be *unfaithful* to reality in two primary ways: simplification and exaggeration.

First, we have simplified the real-world processes our models represent by leaving out several details and even some central aspects of the process at times. Including too many details would likely overwhelm the player and distract from the lessons the model is trying to teach. Including too many larger lessons would also confuse the player, as there would be so many interacting factors that they would detract from each other. Thus, for each of our models we have chosen to portray only a subset of the principles comprising the real-world process it represents.

Second, and perhaps more importantly, we have also chosen to exaggerate some of the real-world effects demonstrated by our models. We discovered through experimentation that adhering too closely to reality causes some lessons to be expressed at an imperceptible level—they are not brought out obviously enough in the simulation to be educationally effective. At the expense of some realism, effects often needed to be somewhat obvious and “over the top” in order to effectively illustrate and enforce the concepts being taught. This can be seen, for example, in the sample implementation of the waterfall model we presented in Section 4.2: One rule was discussed that multiplied

by 10 the effect of errors in the requirements document on the number of errors being introduced into the design document. While it is unrealistic that every error in the requirements document would be carried over tenfold into the design document, when testing this rule in the resulting game, the effect was very hard to detect without this amplification. Similar exaggerations have been made in several parts of our six SimSE models.

A potential ill-effect of these exaggerations could be that students will look for these types of amplified effects in the real world whereas in reality, they are often small and difficult to discern. However, the recommended complementary usage of SimSE with typical instructional techniques provides an ideal setting for counteracting this: as students learn through SimSE (and its occasional exaggerations) the effects and consequences of their actions, the lecturer can augment this with statements of caution that things in the real world are often not as readily discerned.

# 8. Explanatory Tool

As mentioned in Section 3.2, one of the key decisions in designing our simulation approach was to include an explanatory tool to aid the player in discovering the rationale behind their score and in gaining insight into the cause and effect relationships underlying the simulated process. In this chapter we describe the explanatory tool we designed for use with SimSE.

## 8.1 User Interface

SimSE's explanatory tool was designed as a feedback mechanism that goes above and beyond a simple numerical score given at the end of a game (and the revealing of previously hidden attributes). In particular, the purpose of the explanatory tool is to give a player deeper insight into where they might have gone right or wrong in the game, specifically by providing them with information such as which rules were triggered when, which events occurred at what times and how long they lasted, and the evolution of various attributes (e.g., correctness of the code) over time. These pieces of information are primarily given in the form of customized graphs, generated by the player. The user interface for creating these graphs—and the main user interface for the explanatory tool—is shown in Figure 48. As can be seen from the user interface, the explanatory tool allows a user to generate three types of graphs: object graphs, action graphs, and composite graphs.

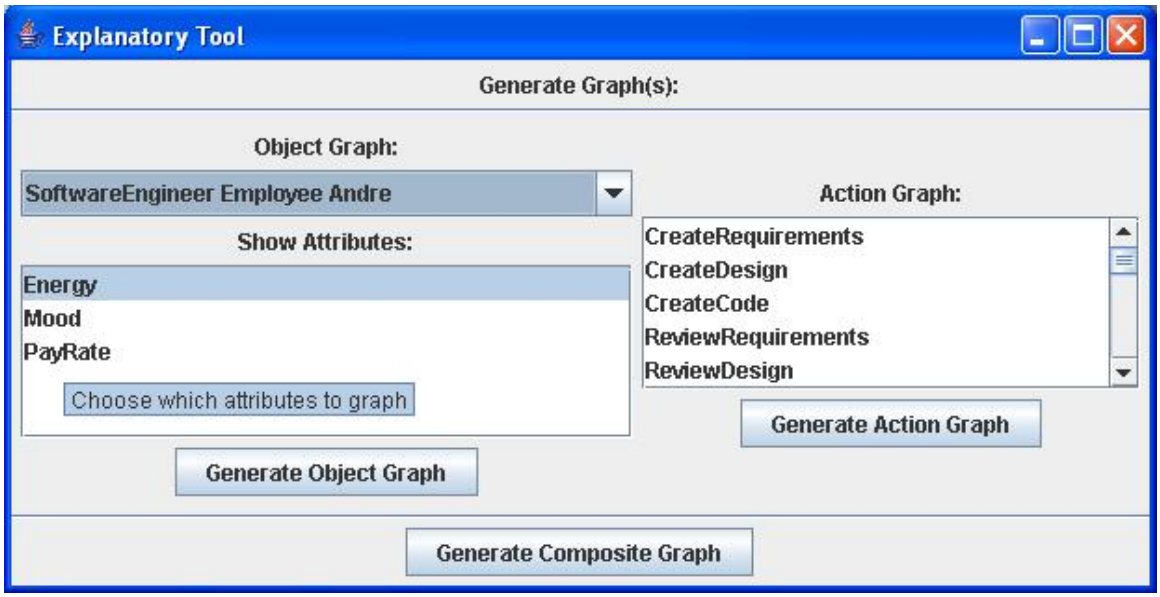


**Figure 48: Explanatory Tool Main User Interface.**

An object graph depicts how an object's attribute values changed over time, and can be generated by choosing an object in the drop-down list labeled, "Object Graph", choosing one or more of that object's attributes in the list marked, "Show Attributes", and then clicking the "Generate Object Graph" button. Figure 49 shows an object graph for an employee's energy and mood. Time is represented by the horizontal axis and attribute value is represented by the vertical axis. The title of the graph indicates which object's attributes are being graphed—in this case, a "SoftwareEngineer" Employee named "Andre." The key below the graph explains which data points correspond to which attributes. Any data point in the graph can be moused over to reveal that point's exact x- and y-values. In Figure 49, the data point for the energy attribute at clock tick 892 is being moused over, at which point the employee's energy was 0.48.

An action graph provides a trace of events or actions that occurred in the simulation, and is also customizable by the user. An action graph can be generated by choosing one or more actions to graph in the "Action Graph" list in the explanatory tool user interface

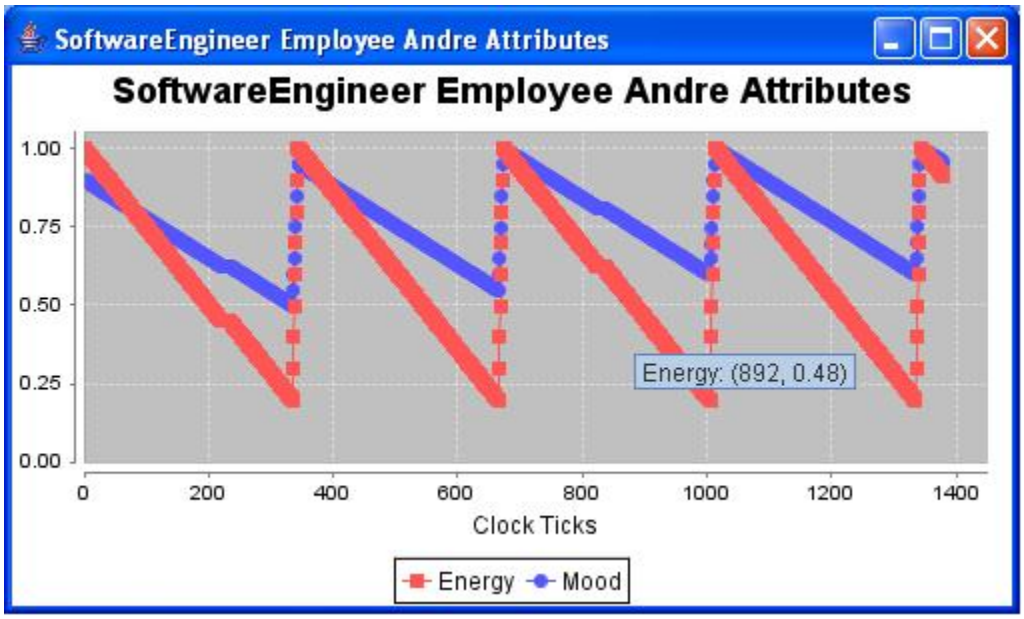


**Figure 49: An Object Graph Generated by the Explanatory Tool.**

(see Figure 48), and then clicking the "Generate Action Graph" button. Figure 50 shows an example of an action graph that includes three different types of actions: "CreateRequirements", "ReviewRequirements", and "CorrectRequirements", with one occurrence of "CreateRequirements" and two occurrences each of "ReviewRequirements" and "CorrectRequirements" (multiple occurrences are indicated by the number at the end of the action label, e.g., "ReviewRequirementsAction-2"). The x-axis indicates time progression, in clock ticks. The y-axis has no semantics, but only serves as a delineator for graphing actions—each action is graphed on a separate gridline on the y-axis. The key below the x-axis indicates which data points correspond to which actions. The data points for an action begin at the time that action was triggered and end at the time that action was destroyed. For example, in Figure 50 the "CreateRequirements" action, represented by the orange (bottom) line, began at clock tick 0 and ended around clock tick 230.

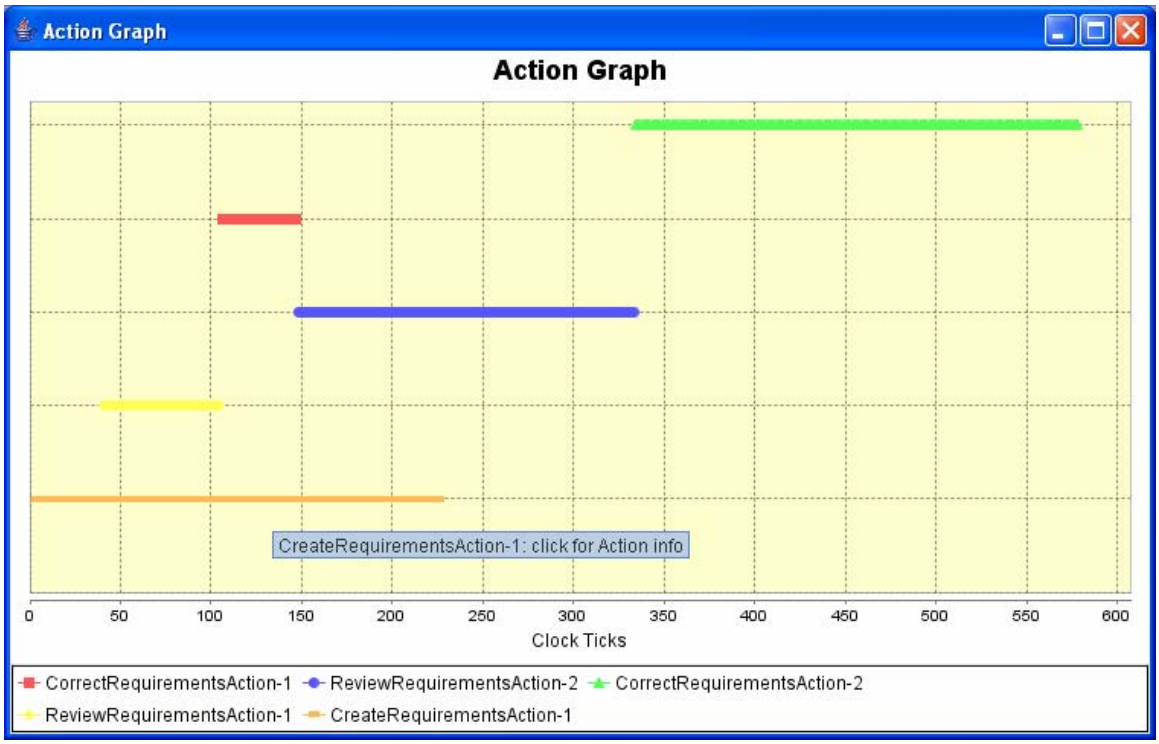


**Figure 50: An Action Graph Generated by the Explanatory Tool.**

Mousing over a data point will display the name of the action and a reminder that the data point can be clicked on for more information, as shown for "CreateRequirementsAction-1" in Figure 50. When a data point in an action graph is clicked on, the details and effects of that action are displayed in the form of the screen shown in Figure 51. There are two tabs in this screen: Action Info and Rule Info. As their names indicate, the Action Info tab contains information about the action and the Rule Info tab contains information about the rules that are attached to that action.

The Action Info tab is divided into three portions, one for each type of information provided about the action. The top portion contains a description of the action, which is specified by the modeler. In the example shown in Figure 51, the description says, "Software engineers review a requirements document to try to find errors." The middle portion displays the participants that were involved in the action during the clock tick of

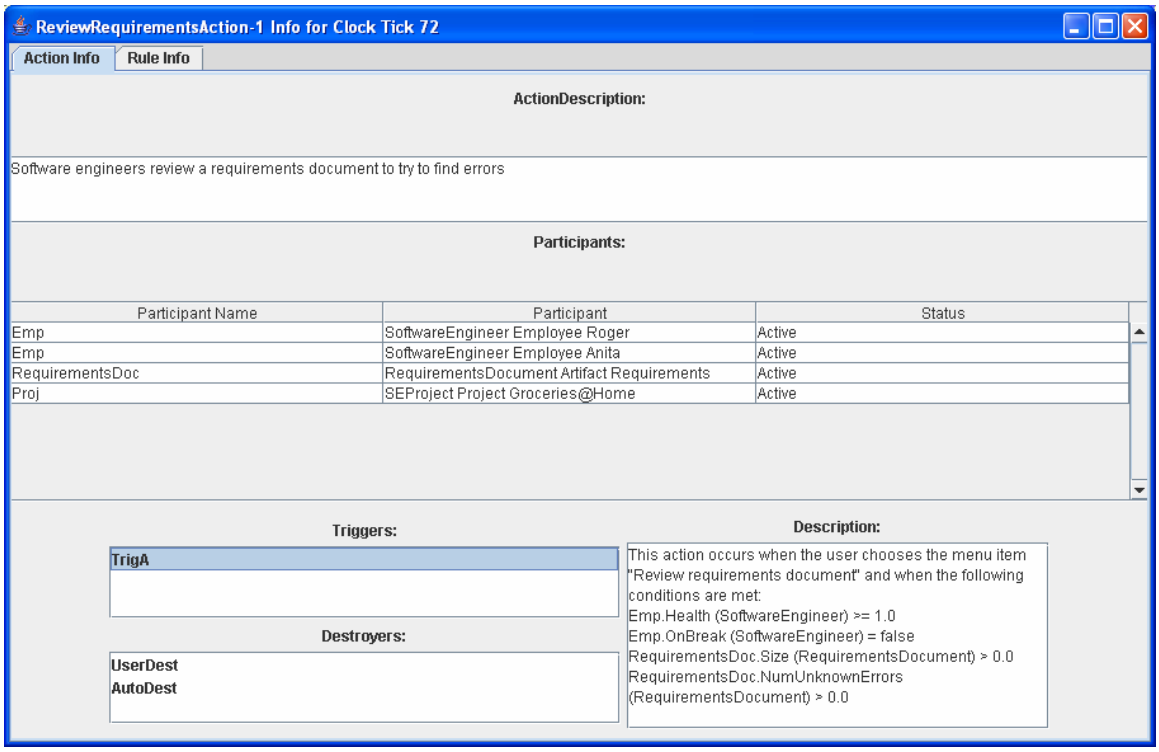


**Figure 51: Detailed Action Information brought up by Clicking on an Action in an Action Graph, with the Action Info Tab in Focus.**

the selected data point (the point that was clicked on to bring up the action information). For instance, the screen shown in Figure 51 corresponds to clock tick 72, as indicated in the title bar of the window. For each participant, it shows which participant role they filled (indicated by the "Participant Name" column), which object filled the role (indicated by the "Participant" column), and whether they were active or inactive during that clock tick (indicated by the "Status" column). The bottom portion of the Action Info tab lists all triggers and destroyers for the action, so that the player can see exactly what could have caused the action to either stop or start. A user can click on any one of these triggers or destroyers to bring up a description in the field to the right. This description is automatically generated based on the type of trigger or destroyer and its conditions. In Figure 51, the trigger for the "ReviewRequirements" action is selected, and the

description for this trigger explains that it will occur when the user selects the menu choice, “Review requirements document” and four conditions are met: the employee participant must be healthy, they must not be on a break, and the requirements document has to have been started and have greater than 0 unknown errors.

The Rule Info tab is shown in Figure 52. On the left are listed all of the rules that were fired during the selected clock tick. For instance, because the example shown in Figure 52 corresponds to clock tick 72, which was neither the beginning nor the end of the action, only the continuous rules (called “intermediate” here) are listed. If the data point that corresponds to the beginning of an action is selected, only the trigger rules are listed. Likewise, if the selected data point corresponds to the end of an action, only the destroyer rules appear. Any one of the rules in the list can be clicked on to bring up a description of that rule (written by the modeler) in the right hand pane. For example, the description for the rule in focus in Figure 52 explains how the unknown errors in the requirements document are decremented and the known errors are incremented—based on the requirements productivity of each employee participant tempered by the number of communication links between participants.

The third and final type of graph that can be generated is a composite graph. A composite graph shows both an object graph and an action graph lined up on the same time axis. The purpose of a composite graph is to help the player discover the reasoning behind attribute behaviors shown in an object graph and, as a result, gain a better understanding of the cause and effect relationships underlying the simulated process. For example, Figure 53 shows a composite graph that contains an object graph for the “RequirementsDocument” artifact, including the two attributes “NumKnownErrors” and

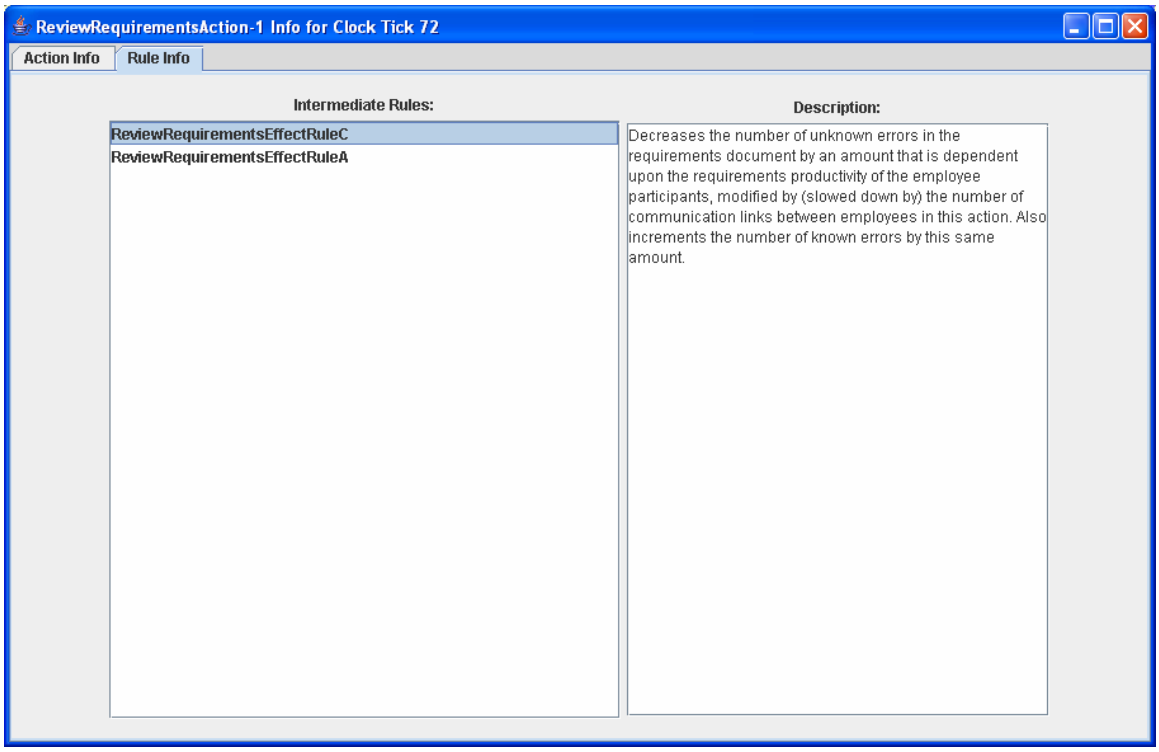


**Figure 52: Rule Info Tab of the Action Information Screen.**

"NumUnknownErrors", and an action graph that includes the same three actions from the previous example in Figure 50 ("CreateRequirements", "ReviewRequirements", and "CorrectRequirements"). Studying these two graphs on the same timeline can explain all of the spikes, dips, and slopes in the object graph. For instance, the number of unknown errors increased steadily as the requirements were being created, and then began to drop dramatically when the requirements document was completed and the only activity occurring was review of the requirements document. If the player clicked on the "ReviewRequirementsAction-2" both before and after requirements creation was complete, they would see that the magnitude of this decrease in unknown errors was due to the fact that they assigned all of the employees who had just finished creating the requirements document to join the requirements review activity, dramatically boosting the productivity of that task.

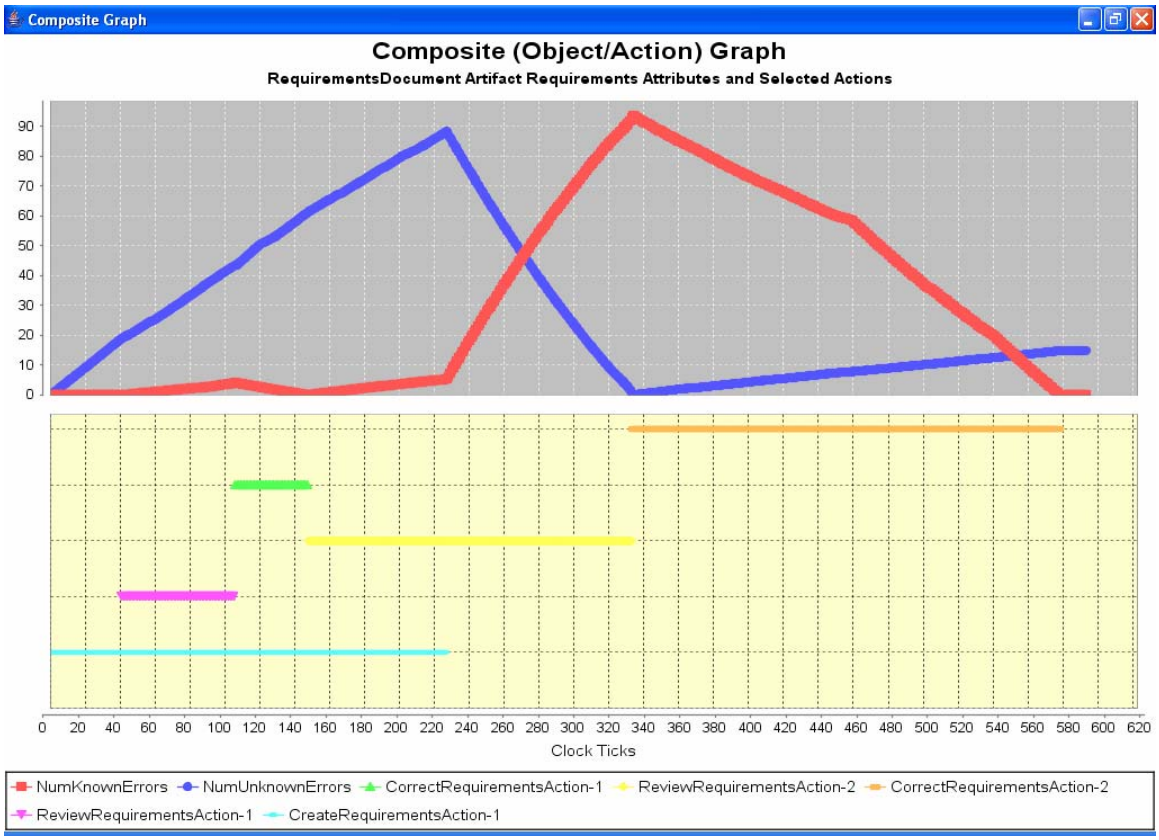


**Figure 53: A Composite Graph Generated by the Explanatory Tool.**

As another example, let us look at the last upward slope in the unknown errors (approximately clock tick 330 to 580), an effect with a less intuitive cause. A player would probably first notice this effect by seeing the hidden attributes revealed at the end of the game and observing that, even though they did a thorough review of the requirements document and corrected all of the known errors, there were still undiscovered errors in the document at the end. Looking at this composite graph would provide them with the reasoning behind this: The final upward slope in unknown errors corresponds exactly to the second "CorrectRequirements" action, indicating that this action caused some more unknown errors to be introduced into the requirements document. Clicking on this action in the graph will then reveal why—employees

correcting requirements will introduce new errors into the document at a rate dependent on their requirements skill. Thus, a player may infer from this that it is just as important to have skilled personnel involved in requirements document correction as it is for requirements document creation, and will likely be more careful in assigning people to this task in the next game.

All graphs can be further customized in terms of appearance—they can be zoomed in or out on, colors can be changed, and labels can be turned on or off. A user can also print a graph or save it as an image if they want to keep it for future reference.

## 8.2 Design and Implementation

The role of the explanatory tool in the overall design of the simulation environment is shown in Figure 54 (a duplicate of Figure 44 with the explanatory tool components highlighted). The link between the explanatory tool component and the rest of the environment lies primarily in the state's logger component. The logger records the state every clock tick (when the state receives the notification to update itself) so that, at the end of the game, it has a full record of the entire simulation that can be used by the explanatory tool component. The explanatory tool component is primarily a user interface component. When a user makes a request to generate a graph, the explanatory tool component fetches the corresponding parts of the simulation record from the logger, and then formats and displays it for the user in the form of the requested type of graph.

For graph generation and formatting, the explanatory tool uses JFreeChart [1], an open-source Java package for displaying charts and graphs in Java applications. The remainder of the explanatory tool component code, not including the JFreeChart

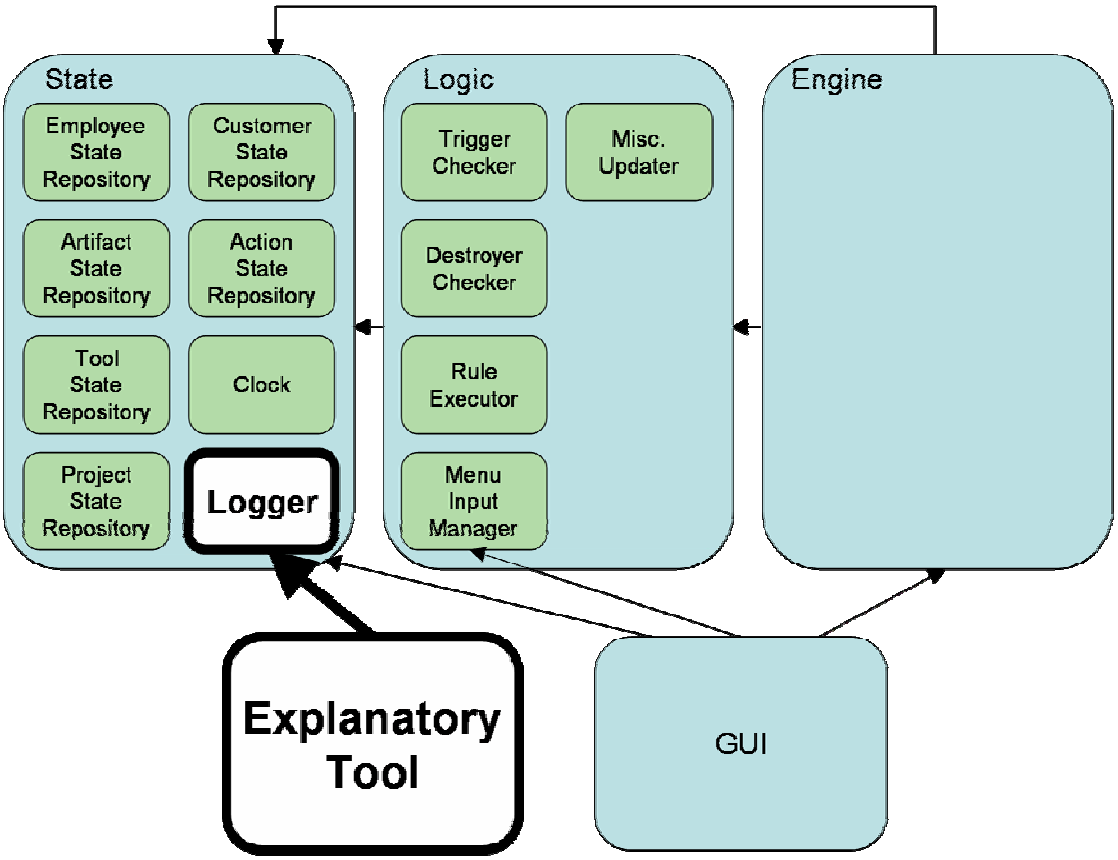


**Figure 54: Place of Explanatory Tool in the Overall Simulation Environment Design.**

packages, ranges from about 2100 lines of code for our smallest model (inspection) to approximately 6500 lines of code for our largest model (RUP).

# 9. Evaluation

To frame the discussion of SimSE's evaluation, let us first revisit the research questions on which SimSE is based (from Chapter 1), and which have driven the design of our evaluation plan:

1. **Can a graphical, interactive, educational, customizable, game-based software engineering simulation environment be built?**
2. **Can students actually learn software process concepts from using such an environment?**
3. **If students can learn software process concepts from using such an environment, how does the environment facilitate the learning of these concepts?**
4. **How can such an environment fit into a software engineering curriculum?**

The first question was conclusively addressed with the successful development of SimSE, hence we will forego discussion of that question here. The other questions can be further broken down into a series of specific evaluation questions, the answers to which can point collectively for or against the usefulness and educational effectiveness of SimSE. Due to the subjective nature of educational research, these research questions could not be conclusively tested, but through a series of evaluations, we have gathered subjective evidence to suggest answers to them:

1. **How do students feel about the learning experience playing SimSE (e.g., is it enjoyable, do they perceive it as an effective method of learning software process concepts)?** Although not a direct indicator of learning, students'

opinions and perceptions are important factors that, along with other data, can suggest how educationally effective SimSE is.

2. **How well does SimSE fit into the traditional software engineering curriculum as a complement to existing methods (which is its intended use)?** Do students learn the concepts taught by the models? What are the implications for instructors? Is there anything difficult about using SimSE in a classroom setting?
3. **How well does SimSE teach the software process concepts that its models are designed to teach?** As the primary goal of SimSE is to teach software process concepts, it is critical to determine how well it accomplishes this goal.
4. **How does SimSE compare to traditional methods of teaching software engineering process concepts such as reading and lectures?** Discovering how they compare in both practical aspects such as time spent and subjective aspects such as student attitudes and motivation can inform decisions about whether SimSE is truly a more useful addition to a course than adding extra traditional assignments such as readings or lectures.
5. **Are the learning theories that SimSE was designed to employ actually being employed by students who play the game, and are there other, unexpected learning theories that are being employed by SimSE?** Answering these questions can provide useful insight into the learning process SimSE (and perhaps educational simulation environments in general) facilitates. These insights can be used both to improve SimSE and to inform the design of other educational simulation environments.

6. **Are the SimSE model-building approach and associated tools adequately expressive?** To promote maximum applicability, a wide variety of educationally effective process models should be able to be built using SimSE.
7. **Does the SimSE explanatory tool help players of the game understand their score and the process better than using the game without the explanatory tool?** Prior research has shown that a student's experience with an educational simulation is significantly enhanced if it includes an explanatory tool to elucidate the cause and effect relationships underlying the simulated process. Thus, it is crucial to determine whether or not SimSE's explanatory tool adequately fulfills this purpose.

The remainder of this chapter describes the experiments we designed and conducted to discover answers to these questions.

## 9.1 Pilot Experiment

### 9.1.1 Setup

For our first experiment with SimSE, our goal was to gain an overall understanding of the thoughts, attitudes, and reactions of students who play SimSE, all for the purpose of making an initial judgment about its potential as an educational tool. In addition, we aimed to determine the strengths and weaknesses of SimSE through the collective feedback of the students who play it.

We recruited 30 undergraduate computer science students to participate in the experiment (although one student did not show up, leaving us with 29 total subjects). This number was chosen because we felt this was an appropriate size for a preliminary

feasibility study—one that would give us statistically meaningful results without overburdening us with an unnecessarily high number of subjects. Although SimSE is designed to be used as a complementary component to a software engineering course, we felt an informal, out-of-class setting was more appropriate for an initial pilot study. However, to ensure that the students had enough background knowledge to be able to understand the game, we required each subject to have passed ICS 52 (the one introductory software engineering course at UC Irvine at the time).

The subjects first received instruction on how to play SimSE, and then played the waterfall model version of the game (see Section 7.1) for approximately two 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 them to write out their responses in free form. The questions from this questionnaire are listed in Appendix C.

The version of SimSE used in this experiment was different in many ways from the version described in Chapter 6. Most notably, it did not include the explanatory tool. On top of that, the graphical user interface was less sophisticated than it is in SimSE's current form: there was no ability to stop the clock, the user could not interact with the simulation while the clock was running, there were no 'i' (information/starting narrative) and 'r' (reset) buttons, and the quality of the graphics was inferior to what they are now.

### 9.1.2 Results

In general, students' feelings about the game were favorable, as summarized in Table 4. On average, students found the game enjoyable to play (3.5 rating out of 5) and relatively easy to play (3.2). They also felt that it was quite successful in reinforcing software engineering process issues taught in the introductory software engineering course they had taken (3.7) and equally successful in teaching software engineering process issues in general (3.6). For the most part, they agreed that SimSE would be helpful to teaching software engineering concepts if incorporated into the introductory software engineering course (3.5).

**Table 4: Questionnaire Results for Pilot Experiment.**

| ***Question*** | ***1*** | ***1.5*** | ***2*** | ***2.5*** | ***3*** | ***3.5*** | ***4*** | ***4.5*** | ***5*** | **Avg** |
|---|---|---|---|---|---|---|---|---|---|---|
| How enjoyable is it to play? (1=least enjoyable, 5=most enjoyable) | 1 | 0 | 1 | 0 | 12 | 2 | 10 | 0 | 3 | **3.5** |
| How difficult/easy is it to play? (1=most difficult, 5=easiest) | 0 | 0 | 7 | 0 | 9 | 1 | 11 | 0 | 1 | **3.2** |
| How well does it reinforce knowledge of SE process taught in class? (1=not at all, 5=definitely) | 0 | 0 | 2 | 1 | 9 | 1 | 8 | 2 | 6 | **3.7** |
| How well does it teach new SE process knowledge? (1=not at all, 5=definitely) | 3 | 0 | 14 | 0 | 6 | 0 | 4 | 0 | 1 | **2.5** |
| How well does it teach the SE process? (1=not at all, 5=very much so) | 0 | 0 | 2 | 0 | 12 | 1 | 10 | 0 | 4 | **3.6** |
| Incorporate it as standard part of SE course? (1=not at all, 5=very much so) | 0 | 0 | 3 | 0 | 12 | 2 | 6 | 1 | 5 | **3.5** |
| As an optional part? (1=not at all, 5=very much so) | 0 | 0 | 6 | 1 | 8 | 1 | 7 | 0 | 6 | **3.4** |
| As a mandatory part? (1=not at all, 5=very much so) | 1 | 0 | 6 | 0 | 9 | 0 | 8 | 1 | 4 | **3.3** |

Students' answers to the open-ended questions also reflected their positive feelings about SimSE. Regarding the enjoyability of the game, some students remarked: *"It does a good job of reinforcing the process in a very fun way!"*, *"[It] makes [software engineering] seem more real and makes it more enjoyable."*, and *"It is a unique way to learn and study software engineering."* Regarding how well the game teaches software

engineering process issues, students wrote: *"[My favorite aspect of the game was] managing a team of employees. It is cool to see how they react to certain environments, and see how the project develops according to the selection of employees for different jobs."*, *"[It taught me] delegating tasks and budgeting. In 52 we learned how to create but not manage."*, *"52 teaches the intellectual level, overall view; the game illustrates this by feel and trial/error."*, and *"[Having to deal with] pay, energy, and mood introduces more complex, real-life issues present in a workspace."*

Although responses were positive for the most part, it was clear from this experiment that some aspects of the game needed to be improved. The most negative response on the numerical questions was that students did not feel that the game taught them much *new* software process knowledge (2.5). While reinforcing the concepts taught in lecture is useful in and of itself, one of the primary goals of SimSE was to also teach new concepts that are either not taught in lectures at all, or do not come across well using other means. It is understandable, however, that this particular model did not teach much new knowledge, since it was based on the waterfall model, which is a well-known and straightforward model that is frequently talked about in lectures. This underscored the need to build more models of different sizes, scopes, and foci, and test them out with students (which we subsequently did, as we describe in the remainder of this chapter).

Many of the students also wished that they were given more explanation as to why they received the score they did and felt that it was sometimes hard to tell where they went wrong. This was part of the impetus for the development of the explanatory tool (see Chapter 8), which was built for the primary purpose of allowing the player to see which of their decisions had good or bad effects. Aside from this, the other aspects of the

game that the students were unhappy with were mainly technical issues, such as the lack of a "stop" button when the clock was ticking, and the quality of the graphics. These issues were addressed in subsequent versions of SimSE.

Interestingly, students' attitudes about the game seemed to be correlated to some pieces of background information that they were asked about on the questionnaire. The first of these is gender. The differences in response between males and females are shown in Figure 55. Surprisingly, females rated nearly every question higher than males. The only issue they rated lower was SimSE's ability to teach new process knowledge, and their perception of SimSE's difficulty was equal to that of males. Because the realm of computer games is notorious for being male-dominated [49], this was definitely an unexpected result that suggests SimSE's potential as an educational tool that is applicable to both genders.

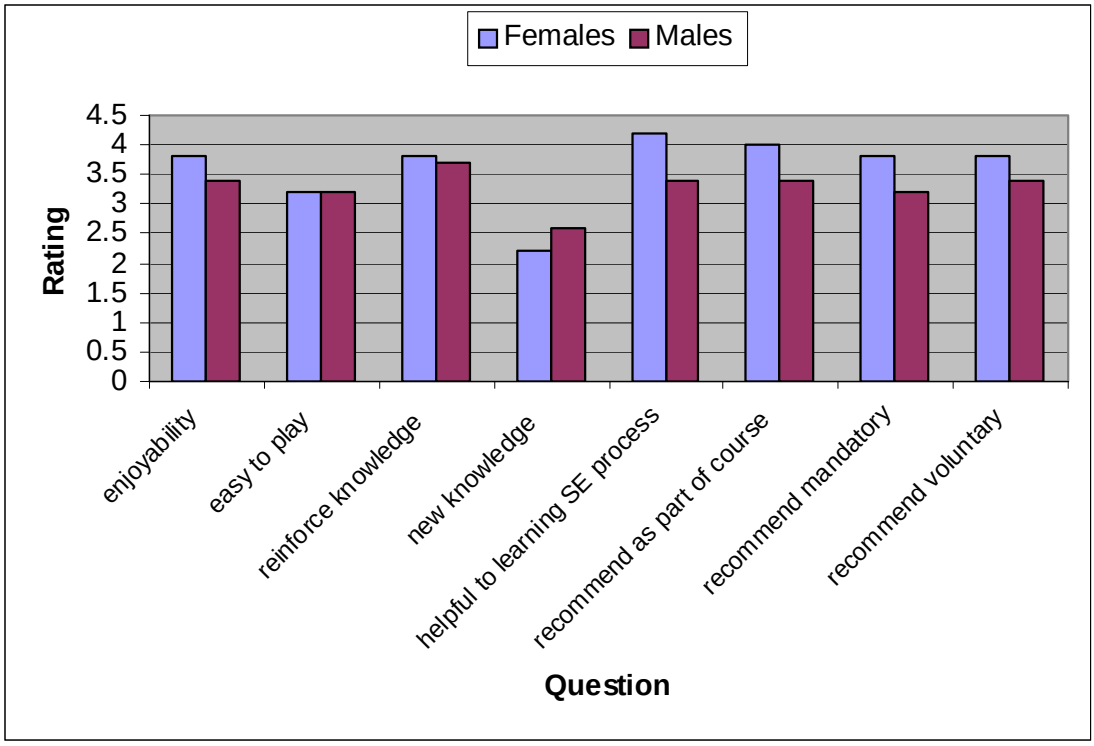


**Figure 55: Gender Differences in SimSE Questionnaire Results for Pilot Experiment.**

There also seemed to be some correlation between the amount of industrial experience a student had and their opinions of SimSE, shown in Figure 56. In all questions except enjoyability and ease of play, students who had industrial experience (which ranged from one to two years) ranked SimSE higher than those who did not. In other words, while they felt it was somewhat more difficult and somewhat less enjoyable than the inexperienced students (though it is unclear why), they were also better able to see its value as an educational tool. Perhaps the real-world experience these students had under their belt gave them more insight into the need for the type of knowledge a tool like SimSE provides. This suggests that, at least in the waterfall model, SimSE is accurate in its portrayal of a realistic software engineering process, as these students who had actually experienced such a process were able to appreciate its value.

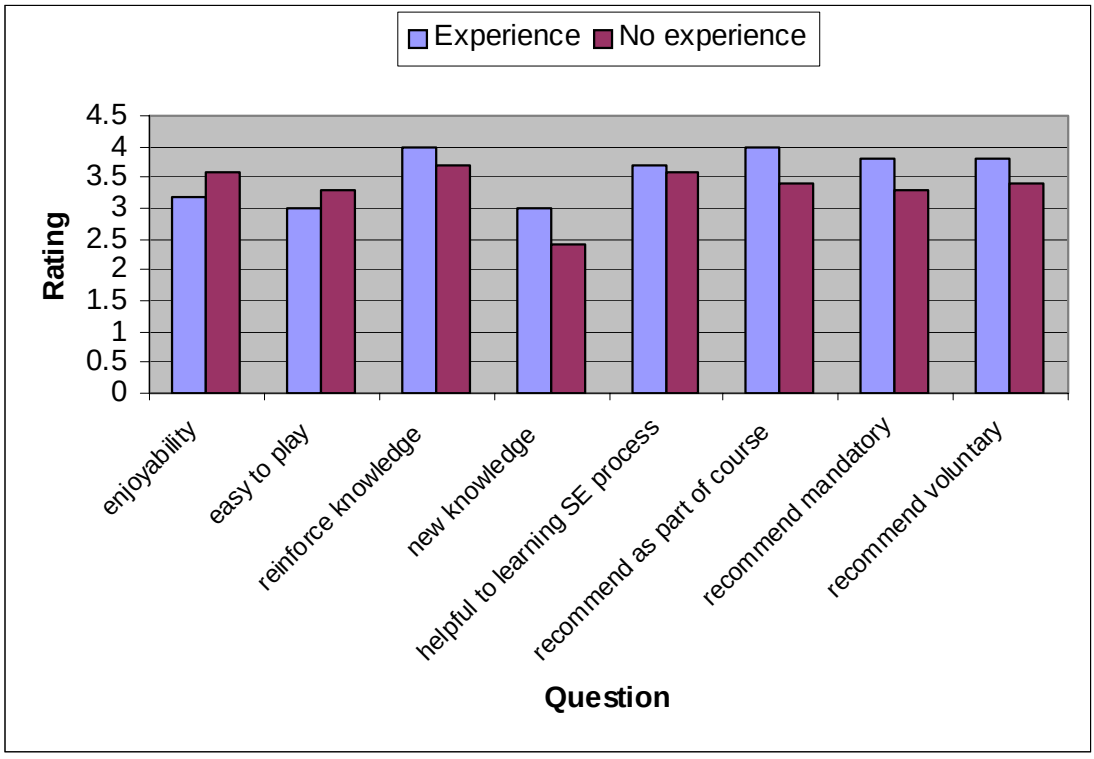


**Figure 56: Industrial Experience Differences in SimSE Questionnaire Results for Pilot Experiment.**

Similarly, those who had additional educational experience (at least one additional software engineering course on top of the introductory one) also seemed to have higher opinions of SimSE, as shown in Figure 57. This is also a positive outcome, again suggesting that those who have some extra software engineering experience to look back on can better see how SimSE can help students learn what they need to know to succeed in more advanced software engineering situations.

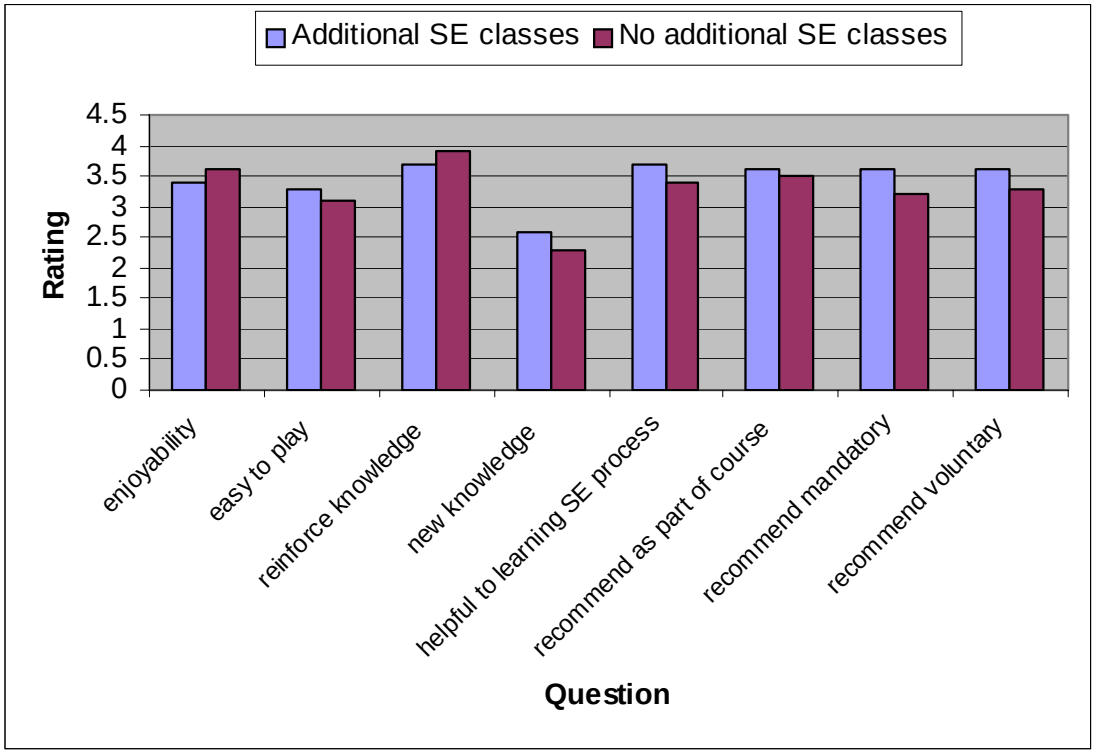


**Figure 57: Educational Experience Differences in SimSE Questionnaire Results for Pilot Experiment.**

In sum, the following lessons were learned from this pilot experiment:

- *SimSE has the potential to be an educationally effective tool in teaching students software process concepts.* The students who played SimSE in this experiment, especially those with significant industrial and/or educational experience, viewed it as a positive and reasonably educationally effective

experience and recommended its addition to an introductory software engineering course.

- *SimSE has applicability to females as well as males.* The difficulty of getting females interested in computer science (and computer games) is well known. However, in this experiment females rated SimSE higher than males in most categories, suggesting that SimSE has the potential to help students of both genders learn software process concepts.
- *An explanatory tool is needed to provide students with more insight into their final score.* A frequent complaint of students in this experiment was the lack of feedback given about their performance in the game, an issue that can be directly addressed with the addition of an explanatory tool.

## 9.2 In-Class Use

### 9.2.1 Setup

The pilot experiment established that SimSE did indeed seem to have potential to be a useful tool in teaching software engineering. On top of that, the feedback given by the subjects in the pilot experiment also gave us valuable guidance about ways SimSE needed to be refined and enhanced to make it more effective. After addressing these issues and developing two more simulation models, the next step was to try SimSE in the setting for which it was designed: in conjunction with a software engineering course. Our chief goal in doing so was to assess how SimSE fits into such a setting, including whether the students actually learn from the experience, how they perceive and feel about the

experience in the context of a course, and how it fits in logistically as a course component.

Because this was the first time SimSE was being used in the context of a course and we were unsure about how it would work in such a setting, we thought it appropriate to make it an extra-credit rather than compulsory exercise. Moreover, in the pilot experiment the students rated its inclusion as an optional exercise higher than its inclusion as a mandatory exercise. Hence, we made it a moderate to minimal extra-credit assignment, worth 7.5% of the final grade.

The course in which we used SimSE was ICS 52 / Informatics 43, a one-quarter introductory software engineering course at UC Irvine. We used SimSE over two subsequent offerings of the course. In the beginning weeks of the quarter, we presented a short five to ten minute tutorial about how to play SimSE, and gave the students the assignment: by the end of the quarter, play three SimSE models (waterfall, incremental, and inspection) and answer a set of questions concerning the concepts the models are designed to teach (although partial credit could be given for partial completion of the assignment). In addition to the questions about the models, students were asked to complete a questionnaire about their experience with SimSE (the questions of which are listed in Appendix D), similar to the one used in the pilot experiment.

The questions about the models the students were assigned are listed in Appendix E, along with the correct answers. The students were assigned five questions per model, and these questions were carefully designed to cover both concepts presented in the course and those that were only present in the models. This was done so that we could detect if students performed better on one of these types of questions over the other. Although we

could not ask questions about every single lesson built into each model, the questions we included were specifically chosen to cover both some major and some minor ones from each model. This would allow us to generalize from the results of these questions and come to some conclusions about how well SimSE as a whole communicates to its players the lessons its models are designed to teach.

The questions were also specifically written in such a way that the students had to play the game in order to find out the answer. Some of these questions instructed the student to take a particular route through the simulation and note the results. For example, one of the waterfall model questions asked, “How is the outcome of the game affected if you fire André right at the beginning?” (a question designed to make the student aware of how crucial a skilled software designer is to a project). As another example, an incremental model question said, “Try skipping one or more of the documentation phases (requirements/design) on one or more modules. What effect does this have?” There were also questions that were more straightforward, but still required that the students play the game in order to find the answer, such as, “What is the ideal size of an inspection team?” and “Is it worth it to purchase software engineering tools?” Inclusion of these types of questions not only ensures that they put forth legitimate effort for their extra credit, but it also ensures that they get a thorough and meaningful experience of SimSE by playing the models thoughtfully, carefully, and, in all likelihood, multiple times to find out the answers to the different questions.

Based on the results of the pilot experiment, the version of SimSE used in these instances included a new and improved user interface—the version presented in Chapter 6, minus the explanatory tool.

### 9.2.2 Results

In the end, 50% of the first class and 66% of the second class attempted the SimSE extra-credit assignment, so the interest in the opportunity was significant, although the possibility of getting extra credit was likely a large part of the draw. The students' scores on the assigned questions were quite high—the high scores were 99% and 98%, the average was 80%, and the low scores were 15%, 50%, 54%, and 55%, with a large jump after that. This seems to suggest that the majority of students were able to learn most of the concepts the models were designed to teach.

In looking at the answers given by the low-scoring students, it is clear that most of their scores could be attributed to the students simply not spending the time playing that is needed to answer the questions—for example, many of these students simply skipped all of the questions pertaining to a particular model, suggesting that they probably did not even attempt to play that particular model at all. It was not often that a student who obviously at least attempted each model could not get a decent score (about 75% or above) on the questions. This seems to suggest that when enough time and effort are put forth, most students do learn the concepts the SimSE models are designed to teach.

As mentioned previously, the assigned questions were carefully devised to cover both concepts presented in the course and concepts that were only present in the models. Students scored equally well on both types of questions. Hence, it is clear that students were not only reinforced in the knowledge they gained in their courses, but also did learn a number of lessons that were not present in their lectures. For instance, the students were asked to study the incremental model, an approach that involves incremental delivery of code to a customer. While the lecture covered one aspect of why one would want to

follow this approach (customer buy-in), students learned from playing the model that another reason is changeability: by delivering code early and often, code changes less frequently and the program core becomes stable faster. As other examples, students learned the ideal size of an inspection team as well as the reasons for this size (tradeoff among finding more bugs but having slower discussions), they understood that intrinsic factors such as not putting an employee on too many tasks was important, and discovered that inspection meetings lose their benefit if they run long. None of these items were explicitly covered in the lectures, all were encoded in the models, and all were discovered by the majority of the students during game play.

We examined each student's SimSE assignment scores in relation to their final grade in the class and found that there was no correlation between the two. In other words, both students who were doing well and those who were not doing well on other assignments in the class were able to provide quality answers on the SimSE assignments. This is important for two reasons: First, it suggests that SimSE is applicable across a broad range of students with different levels of academic performance, without biasing one group over another. Second, it suggests that SimSE has a strong potential to help students who are not doing well academically, because apparently this method of instruction is one that they can grasp. Of course, there is also the possibility that those who were not doing well in the class simply tried harder or spent more time on the assignment since they needed the extra credit more. Because we did not collect data about the time put forth by each student, however, we are unfortunately not able to investigate this possibility further.

Looking at the students' responses on the questionnaires provides even more insight. The results of the questionnaires are shown in Table 5, with the averages for each

**Table 5: Questionnaire Results from Class Use of SimSE, with Averages Compared to Pilot Experiment.**

| *Question* | *1* | *1.5* | *2* | *2.5* | *3* | *3.5* | *4* | *4.5* | *5* | Avg | Pilot |
|---|---|---|---|---|---|---|---|---|---|---|---|
| How enjoyable? (1=least enjoyable, 5=most enjoyable) | 5 | 1 | 19 | 0 | 14 | 1 | 5 | 0 | 1 | **2.5** | **3.5** |
| How difficult/easy? (1=most difficult, 5=easiest) | 2 | 0 | 7 | 2 | 16 | 0 | 14 | 0 | 5 | **3.3** | **3.2** |
| Reinforces material taught in class? (1=no, 5=definitely) | 2 | 0 | 7 | 0 | 20 | 0 | 13 | 1 | 3 | **3.2** | **3.7** |
| Teaches new process knowledge? (1=no, 5=definitely) | 12 | 1 | 13 | 0 | 12 | 0 | 7 | 0 | 5 | **2.4** | **2.5** |
| Teaches SE process in general? (1=no, 5=definitely) | 4 | 0 | 9 | 1 | 18 | 1 | 8 | 0 | 4 | **3.0** | **3.6** |
| Helps understand lecture concepts? (1=no, 5=definitely) | 4 | 0 | 13 | 0 | 15 | 1 | 10 | 0 | 2 | **2.9** | **N/A** |
| Helpful as extra-credit? (1=no, 5=definitely) | 1 | 0 | 8 | 0 | 13 | 0 | 6 | 0 | 18 | **3.7** | **N/A** |
| Helpful as required part? (1=no, 5=definitely) | 1 | 0 | 15 | 0 | 13 | 1 | 8 | 0 | 3 | **2.8** | **N/A** |
| Incorporate into SE course? (1=no, 5=definitely) | 9 | 0 | 9 | 2 | 8 | 0 | 12 | 0 | 4 | **2.8** | **3.5** |
| As a mandatory part? (1=no, 5=definitely) | 16 | 0 | 12 | 0 | 13 | 0 | 5 | 0 | 0 | **2.2** | **3.3** |
| As an extra-credit part? (1=no, 5=definitely) | 0 | 0 | 5 | 0 | 3 | 1 | 8 | 0 | 29 | **4.3** | **N/A** |
| As a voluntary part? (1=no, 5=definitely) | 7 | 0 | 3 | 0 | 7 | 0 | 12 | 1 | 16 | **3.6** | **3.4** |

question compared with the pilot experiment's average when applicable (some of the questions on the in-class questionnaire were not asked in the pilot experiment). First we note that the students who used SimSE in class rated it somewhat lower overall than those in the controlled, out-of-class setting of the pilot experiment. This was particularly true in the enjoyability aspect of the game: students using it in class ranked it an entire point lower (2.5) than those in the pilot experiment (3.5). We hypothesize that this can be attributed to two factors: First, course use of SimSE involved the added pressure to earn extra credit, required that they play three models instead of one, and required that they play enough to find answers to the assigned questions, all of which resulted in significantly more time invested than the two or three hours involved in the pilot experiment. These circumstances all made the experience decidedly less fun than participating in a novel experiment for a few hours through which they earn money. This was particularly noticeable when comparing the free-form answers on the questionnaires between the two groups—there was definitely a more positive attitude on the part of the

pilot experiment subjects, while the in-class students seemed to feel more pressure that this was something they "had to do".

We believe the second contributing factor to the lower in-class scores was the repetitive nature of playing models over and over again in order to master the game and discover the answers to the assigned questions—this was the most frequent complaint of the students on their questionnaires. Many reported that it was a frustrating, cumbersome experience to try to figure out how to succeed in the game, and suggested that SimSE provide more feedback about their performance in the game, help tips, and/or a more extensive manual. Although repetitiveness can be beneficial educationally, the extent to which it was necessary to get to the answers seemed to become a seriously detracting factor. As described in Chapter 8, the explanatory tool was specifically designed to remove this hurdle (see Section 9.4 for a description of the experiment conducted to determine the effectiveness of the explanatory tool in achieving this goal). However, on top of the addition of the explanatory tool, this experience may suggest that the way SimSE is introduced to students in class should be re-examined, as will be discussed in Section 9.6.

There were only three questions on which the in-class group ranked approximately the same as the pilot experiment group: First, both groups believed that SimSE was equally difficult/easy to play (3.3 for the in-class group and 3.2 for the pilot experiment group). Second, both groups gave SimSE's ability to teach new process knowledge a mediocre rating (2.4 for in-class, 2.5 for pilot). Finally, both groups also had similar feelings about SimSE being a voluntary part of the course (3.6 for the in-class group and 3.4 for the pilot experiment group). The in-class students particularly liked the fact that it

was extra-credit, judging from the fact that they rated the question about whether it should be incorporated as an extra-credit exercise higher than any other question (4.3). This is not surprising—students are always going to appreciate the opportunity for extra points. However, this is telling for an instructor's perspective: use of SimSE in the classroom must be rewarded with some form of credit, either as an integral assignment or as an extra-credit assignment (again, not surprising, since students are notorious for not doing optional elements in any kind of class).

The students' answers to the free form questions revealed even more insights, sometimes in conflict with their numerical ratings. Interestingly, although the most frequent complaint was that it took them too long to be able to "master" a game, when asked specifically about the appropriateness of the length of the game, only about half said that it lasted too long, while the majority of the rest said the length was just right. Also surprising were the answers to the question about whether SimSE taught them any concepts better than the lectures did: approximately half of them were able to come up with at least one thing they learned better in the game, despite their mediocre ratings of SimSE's ability to help them understand the concepts taught in lecture (2.9) and its ability to teach new process knowledge (2.4).

We were unable to correlate assignment scores to any other student data given in the questionnaire responses (such as male versus female) because the questionnaires were anonymous in this experiment and were in no way tied to the respondent's assignment score. However, we were able to make correlations among different pieces of data given within the questionnaire. First, we compared the responses of students with industrial software engineering experience versus those with no experience. The results are shown

in Figure 58. The overall trend is that those without experience rated SimSE higher in all categories except enjoyability, for which both groups rated it equal. This was almost the complete opposite result from the pilot experiment, in which those with experience rated SimSE higher. Without further discussion with students in both groups, it is unclear why this is, but we can hypothesize that it must have something to do with the different environments in which each group was exposed to SimSE. In any case, the differences in ratings between the experienced and inexperienced students were not that great. If we take these results into consideration with the data from the pilot experiment in which the trends were opposite, we can conclude that there probably is no significant difference between those with industrial experience and those without, suggesting that SimSE is applicable for both types of students.

The second trend we were able to notice in the questionnaire responses involved the differences between males and females, shown in Figure 59. Males ranked SimSE higher in all categories except difficulty (females thought it was slightly easier to play), its ability to teach new knowledge (equal to females) and its helpfulness to learning process concepts (also equal to females). Again, this is in direct conflict to the results in the pilot experiment, in which females rated SimSE higher in almost every category. Although we would ideally like SimSE to be equally applicable to both genders, these results from in-class use are not surprising, as it is well-known that computer games are substantially more popular with males than females [49]. Nevertheless, the difference in ratings between males and females in this case was not that large compared to the known difference between the two groups in their affinity for computer games, suggesting that

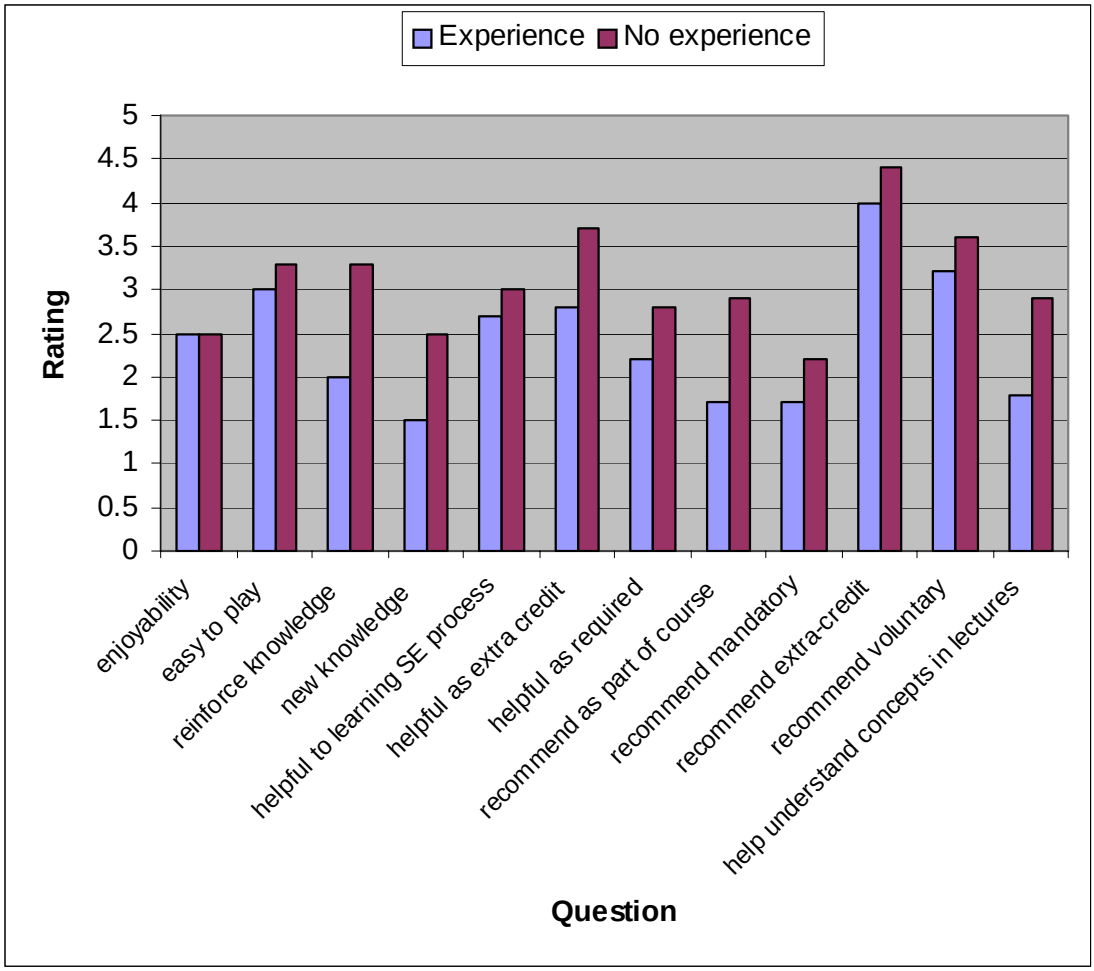


**Figure 58: Industrial Experience Differences in SimSE Questionnaire Results for Class Use.**

SimSE is still a promising way to educate both males and females nearly equally well in software process concepts.

To summarize, our experience with class use of SimSE revealed the following insights:

- *Students who play SimSE in parallel with taking a software engineering course are able to learn from the game most of the concepts the models are designed to teach.* These include both new concepts and reinforcement of concepts taught in lectures. When given a set of questions concerning these concepts, students are generally successful in finding the answers to them by playing SimSE.

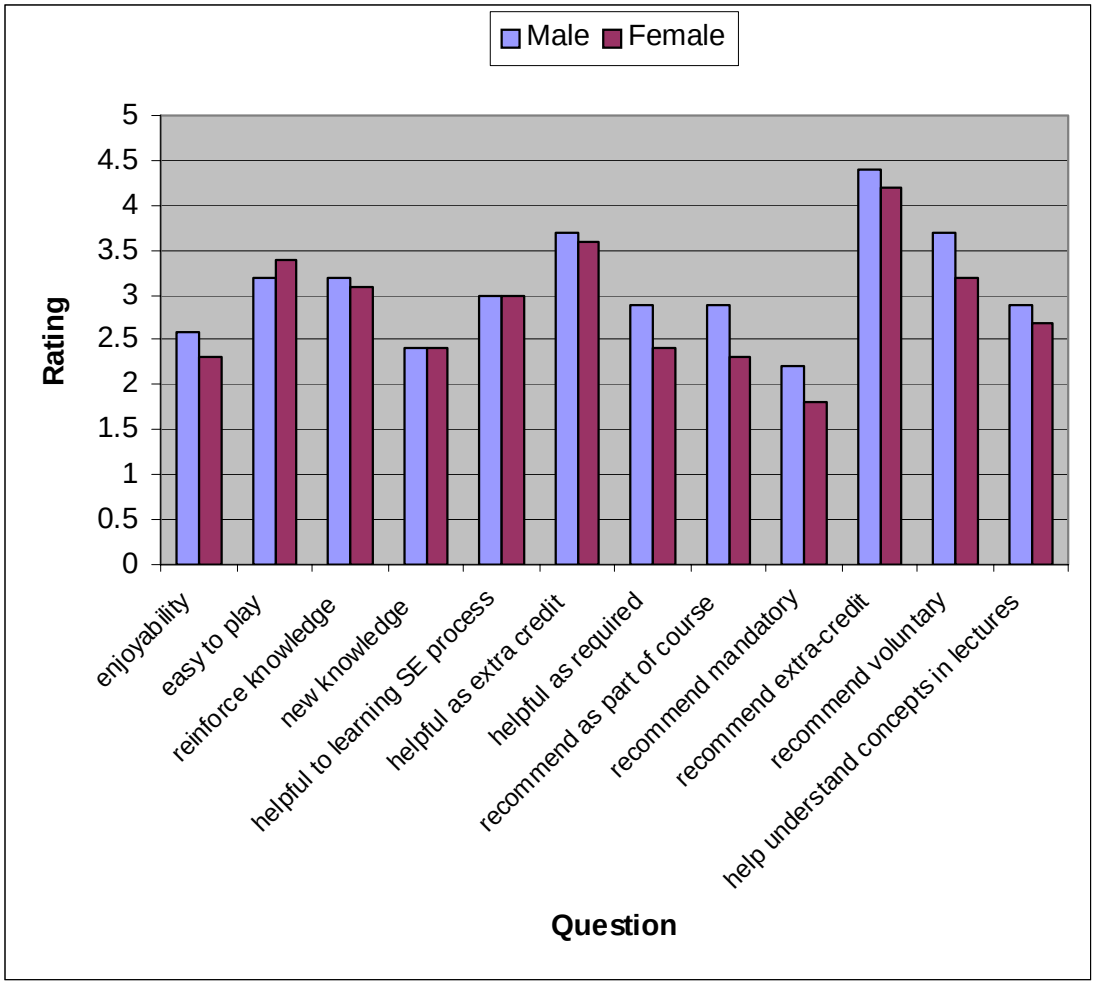


**Figure 59: Gender Differences in SimSE Questionnaire Results for Class Use.**

- *Use of SimSE in the classroom should at least be rewarded with some form of credit.* Offering SimSE as an extra-credit exercise seemed to work well, as many students attempted the assignment and felt favorably about the opportunity for extra credit. Of course, the next step, then, is to experiment with incorporating SimSE as a compulsory component, which we plan to do as part of our future work (see Chapter 12).
- *SimSE is applicable to a broad range of students along the academic performance spectrum.* Both students who did well on other assignments and

those who did not were able to succeed in the SimSE extra-credit assignment, suggesting that it can be a useful tool for students of varying abilities.

- *SimSE is equally applicable for both males and females in terms of their attitudes and perceptions about the game.* While males (not surprisingly) ranked SimSE slightly higher than females, the difference was quite minor, and not as great as one would expect, given the known attraction of males to computer games. Taken with the pilot experiment results in which females' rankings were higher, we can conclude that there is probably no difference between the two groups' opinions about SimSE.
- *Additional guidance and feedback in a SimSE game is needed to make the experience less frustrating, more enjoyable, and potentially more educationally effective.* Because the main complaint of students was the lack of guidance and feedback, it is clear that more help is needed in order to create a more positive experience.

## 9.3 Comparative Experiment

### 9.3.1 Experiment Setup

After establishing in the first two experiments that SimSE does indeed have significant potential as a teaching tool and that students who play it do seem to learn the concepts the models are designed to teach, the next step was to try and discover how it compares to traditional teaching methods (reading from a textbook and hearing lectures). In particular, we aimed to compare the effectiveness of each method in teaching a specific set of software process concepts, as well as other aspects underlying the learning process—both

practical aspects such as time spent and subjective aspects such as student attitudes and motivation. With this comparison we would be able to make some informed judgments about whether SimSE would truly be a useful addition to a course—an addition more useful than one that included extra traditional assignments such as readings or lectures.

For this experiment we recruited 30 undergraduate students, 15 who had passed either ICS 52 or Informatics 43, and 15 who had not taken either of these courses (however, only 19 subjects total ended up completing the experiment, as is discussed in the next section). This particular mix of educational experience was chosen for the following reason: SimSE is meant to be used as a complement to existing teaching methods, so it assumes some background knowledge of basic software engineering concepts, and hence, the target population is those students who have taken at least one introductory software engineering course. However, students who have taken a software engineering course will have already been taught (through textbooks and lectures) much of the material that was taught in this experiment using textbooks and lectures. Hence, creating an equal mix of students from the two different experience levels creates a balance addressing both of these concerns, as well as helps provide some insight into how SimSE does as a teaching tool for those who have no software engineering experience. The number of subjects (30) was chosen based on the desire to have a high enough number of people in each of three treatment groups so that statistically significant statements could be made about the results. The students were randomly divided into three groups (SimSE, reading, and lecture) of approximately equal size, with the condition that in each group approximately half of the people had passed either ICS 52 or Informatics 43 while the other half had not.

The SimSE group was given the same version of SimSE that was used in the in-class experiments, along with three SimSE models to play, specifically, the waterfall, incremental, and inspection models. These were the three SimSE models that were most stable and had already been shown in the previous experiments to be potentially useful and effective in teaching the software process concepts they are designed to teach. The subjects in this group were instructed to play each model enough to be able to obtain a "good" score in each game (85 or above). This instruction was given to try to ensure that they would play each game enough to learn most of the concepts the games are designed to teach.

The reading group was instructed to complete a set of readings that covered the software process concepts embodied in the SimSE models played by the SimSE group. The readings were taken from Ian Sommerville's textbook, *Software Engineering* [134], since this is the most widely-used software engineering textbook, and the specific topics covered in the book matched well with the lessons in the SimSE models.

The lecture group was required to attend two 50-minute lectures about the same software process concepts that were taught to the SimSE group (through SimSE) and the reading group (through readings). The slides used for the lectures were those that were created by Ian Sommerville to accompany his textbook [134]. A graduate student who was experienced in teaching software engineering classes gave the lectures.

The experiment ran over a duration of five days. On day one, all subjects were given a pre-test (the questions of which are listed in Appendix F) that measured their knowledge in the software process concepts that were to be taught using the three methods. At the completion of the test, all subjects were then randomly assigned to a

treatment group and given instructions about the assignment to complete (SimSE, reading, or lectures). At this time, all subjects were also notified that on day five they would be given another test on the concepts they were to be taught in the learning exercise during the week. For the next four days, the SimSE group was expected to play SimSE, the reading group was expected to complete the readings, and the lecture group was expected to attend their two lectures, which took place on days two and four.

On day five, the subjects were given a post-test (the questions of which are listed in Appendix G) which contained some of the same questions as those in the pre-test, but also included some different questions. This mix of questions was designed to both ensure some consistency between the two tests and, at the same time, mitigate the possible bias of students knowing the questions that will be asked ahead of time (in which case they might have looked up answers, or prepared for them in some other way). The pre- and post-tests, which were anonymized so that the grader did not know which group each subject was in, were then graded and each subject received a score on each test.

The questions on the pre- and post-tests were of three main types: Specific questions asked students to recite explicit pieces of software process knowledge that were taught in the learning exercise. Insight questions asked students to abstract away general concepts from the material, and make comparisons between various concepts in the material. Application questions required students to apply their software process knowledge to a hypothetical real-world problem. These three different types of questions were included for three main reasons: (1) In order to cover a broad range of different types of knowledge; (2) To reflect the different types of questions that are normally asked on the

tests in ICS 52 / Informatics 43, (an existing standard designed to test software engineering knowledge); and (3) To provide insight into whether there is any difference in the types of questions that students from each group score high or low on, which may suggest something about what types of knowledge each method is better or worse at teaching than the others.

Some of these questions were also designed with a deliberate bias toward either SimSE or the readings and lectures. (The readings and lectures taught exactly the same material, while some of the SimSE concepts were different. It was not possible to find a set of readings and lectures that matched perfectly the knowledge taught in the SimSE models, since the SimSE models were built using a variety of knowledge sources.) These questions asked about concepts that were not overlapping between the different treatment groups. For example, a SimSE-biased question asked about knowledge that was only taught through one of the SimSE models, but not through either the readings or the lectures. These few, select biased questions were included for another comparison point between the three methods, namely, in order to compare how well each method enables students to remember the specific knowledge taught, as well as how well each group can infer knowledge that was not taught in their treatment group.

At the end of the experiment all subjects were also given a questionnaire (see Appendix H) that asked them to state their thoughts and feelings about the instructional method in which they participated. This questionnaire contained two main types of questions: The first type of question asked them about the teaching/learning method used, including how much time they spent on the exercise, how much they enjoyed it, how effective they felt it was, and whether they prefer it over other methods. These questions

allowed us to compare both practical aspects of the methods (such as time spent), as well as students' attitudes about the various methods. The second type of question asked them to provide some background information, which allowed us to detect any correlations between such variables as experience level or gender and the subject's performance on the learning exercise. Overall, the purpose of the questionnaire was to gain insight into how SimSE compares with the traditional teaching methods of reading and lectures in some of the fundamental aspects of learning such as students' attitude and motivation.

### 9.3.2 Experiment Results

Due to the unfortunate facts that (1) several subjects who were scheduled for the experiment did not show up, and (2) some dropped out between day 1 and day 5 of the experiment, the number of subjects in each group and in the experiment as a whole was fewer than planned. The experiment ended up with only 19 subjects versus the 30 that were planned for, with seven in the SimSE group, six in the reading group, and six in the lecture group. As a result, there were very few trends in the data that can be considered truly statistically significant (all are $p>0.05$ unless otherwise stated). However, as a pilot comparative study, the data still hints at the probability of several trends that warrant further investigation in future studies, and points out critical issues that must be considered when conducting these studies.

The overall results for the pre- and post-test scores are shown in Figure 60. In terms of measured gain in software process knowledge, while all groups improved somewhat, the reading group improved the most (5.08), followed by the lecture group (4.04), followed by the SimSE group (1.21). In the end, the reading group also seemed to end up with the greatest amount of software process knowledge, as the post-test scores followed

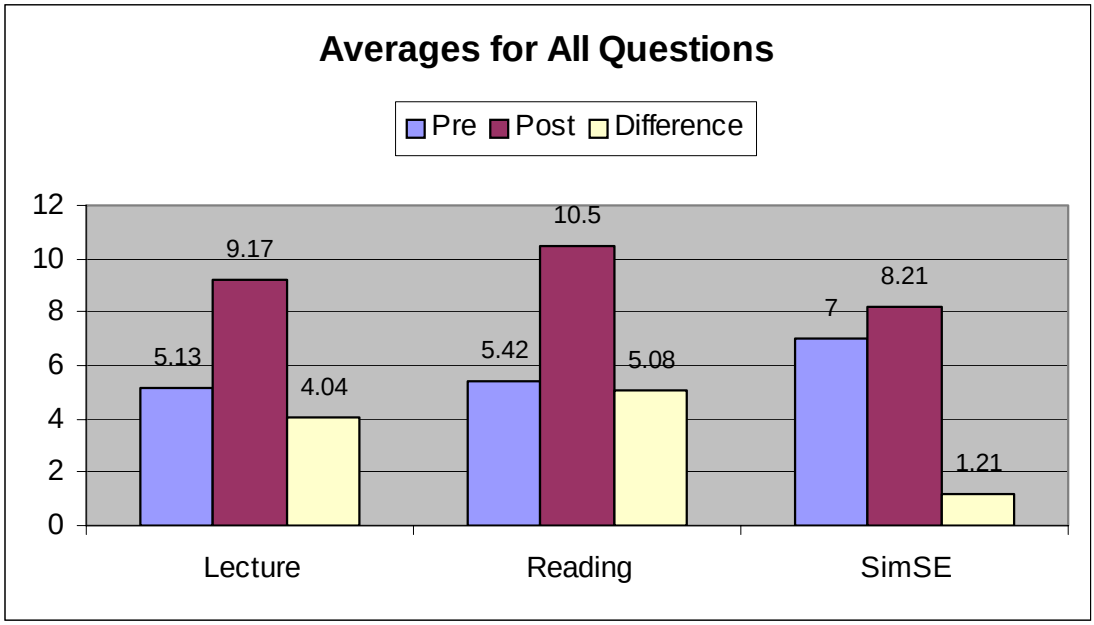


**Figure 60: Test Score Results for All Questions Divided by Treatment Group.**

this same trend (reading group highest, followed by the lecture group, followed by the SimSE group). However, this data also shows that the SimSE group had significantly higher pre-test scores to begin with. This could be partially due to the fact that the SimSE group, because of the random distribution of experiment drop-outs, by chance happened to end up with the highest percentage of students who had taken ICS 52 / Informatics 43 (5 out of 7), compared to the reading (3 out of 6) and lecture (4 out of 6) groups. (For simplicity, from here on we will refer to students who have taken ICS 52 / Informatics 43 as "52/43 students", and those who have not as "non-52/43 students".)

If we divide the subjects not only by treatment group, but also by whether or not they are 52/43 students (as shown in Figure 61), we can gain even more insight. First, it is probable that, as we hypothesized, the high pre-test score average of the SimSE group can be attributed to its 52/43 students, as these students' pre-test scores were significantly higher than any other group (8.9). Their gain in knowledge, however, was significantly lower than any other group (0.6). One possible reason for this (and there are others, as

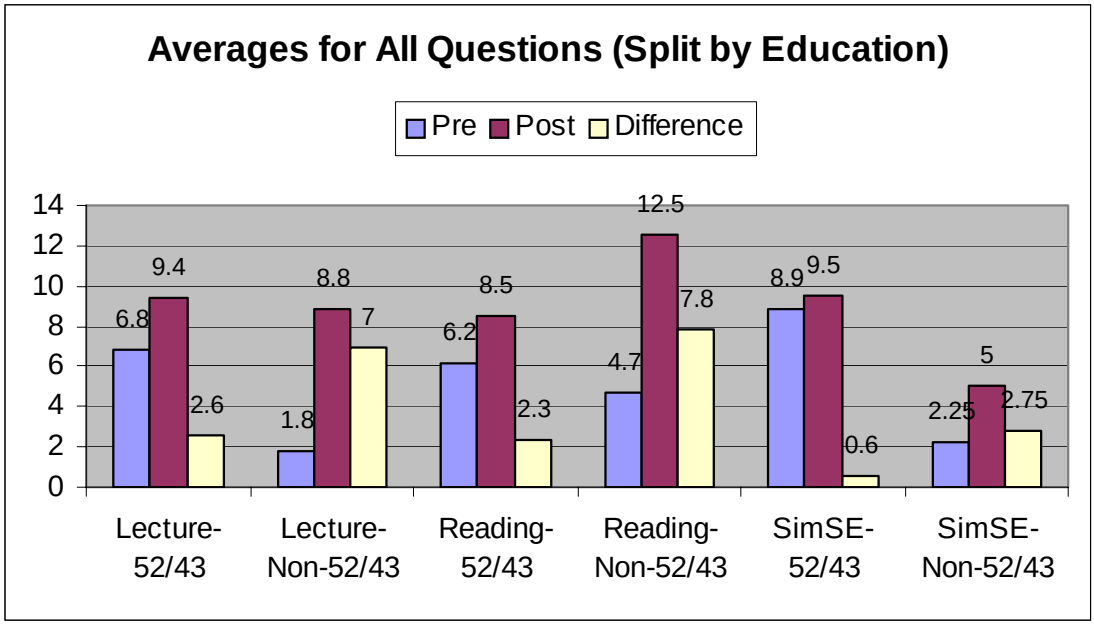


**Figure 61: Test Score Results for All Questions Divided by Treatment Group and Educational Experience.**

will be discussed later) is that, since the SimSE group had the strongest students to begin with, they did not have much room for improvement—that is, most of the knowledge they stood to gain from the exercise they already possessed.

If we look at the post-test scores in Figure 61, we can see that, although overall the SimSE group scored lowest, in actuality the SimSE 52/43 students had the second-highest post-test score average of any group (9.5), and it was the especially poor performance of the non-52/43 SimSE students (5) that brought the overall group's average down. In fact, the non-52/43 SimSE students had the lowest post-test scores of any group (although they still improved by 2.75 points on average). This suggests that using SimSE with students who have no background knowledge in software engineering is not very effective (as compared to reading and lectures, which seem to be equally, if not more effective for students with no background knowledge). From this data, it is hard to say whether it is effective for students who do have background knowledge, since the

52/43 SimSE students happened to be so well-versed in software process to begin with. However, it does show that even these students who knew a great deal already were able to improve somewhat (albeit modestly) by using SimSE.

Looking at the data in this way also shows an unexpected trend that occurred in the reading group: On the post-tests, the non-52/43 students scored an entire three points higher than the 52/43 students on average. In all other groups across the board, the 52/43 students scored noticeably higher on the post-tests than the non-52/43 students. This trend, however, can be easily explained by what many of the 52/43 reading students wrote on their questionnaires when asked if they did less than assigned, and why: they began reading the material, noticed that it was familiar to them since they had been exposed to it in class before, and, as a result, decided to either skip it or just skim it. The non-52/43 students, on the other hand, read the material more thoroughly, since they had not seen it before. To corroborate this, the 52/43 students also reported spending less time on the reading exercise (1.3 hours on average) than the non-52/43 students (1.7 hours on average).

In general, the rest of the trends when split up by education level (in Figure 61) were as expected: in each group, the 52/43 students scored higher than the non-52/43 students on the pre-tests (suggesting that the pre-tests measured software process knowledge accurately); all students in all groups improved between the pre- and post-test; and the non-52/43 students improved more than the 52/43 students in all groups (since they had more to learn).

We can also look at the data in terms of each groups' scores on the type of question, whether specific, insight, or application. The trends for specific questions are shown in

Figure 62, and for the most part follow the same trends that we have seen already. Although for these questions the reading and lecture groups showed equal improvement (0.3), the SimSE group was, again, significantly behind the others (0.036).

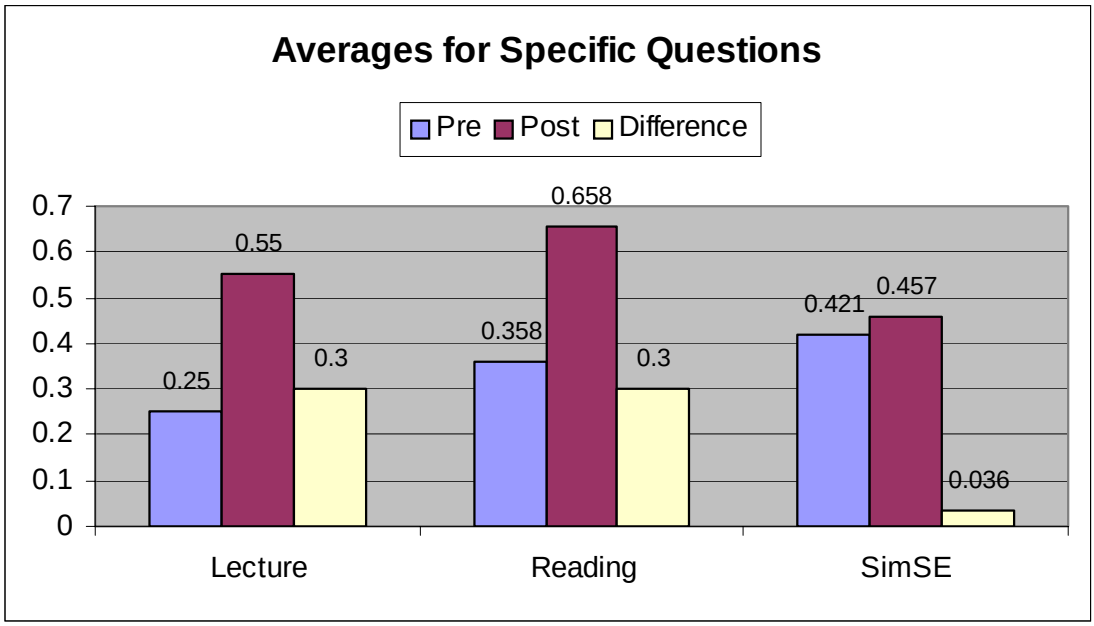


**Figure 62: Test Score Results for Specific Questions Divided by Treatment Group.**

If we look at the data for insight questions, however, there is an interesting trend to notice (see Figure 63). This was the only time that the SimSE group actually scored highest on the post-test. (Again, of course, their starting point was also higher.) Looking at the data for the insight questions split into 52/43 and non-52/43 students (see Figure 64) shows that this was primarily due to the high scores of the 52/43 students, not the non-52/43 students. Perhaps this suggests that, for students with sufficient background knowledge, SimSE is more useful than reading or lectures for teaching the kind of skills needed to answer this type of question—specifically, the skills to abstract away general concepts from the material, and make comparisons between various concepts in the material.

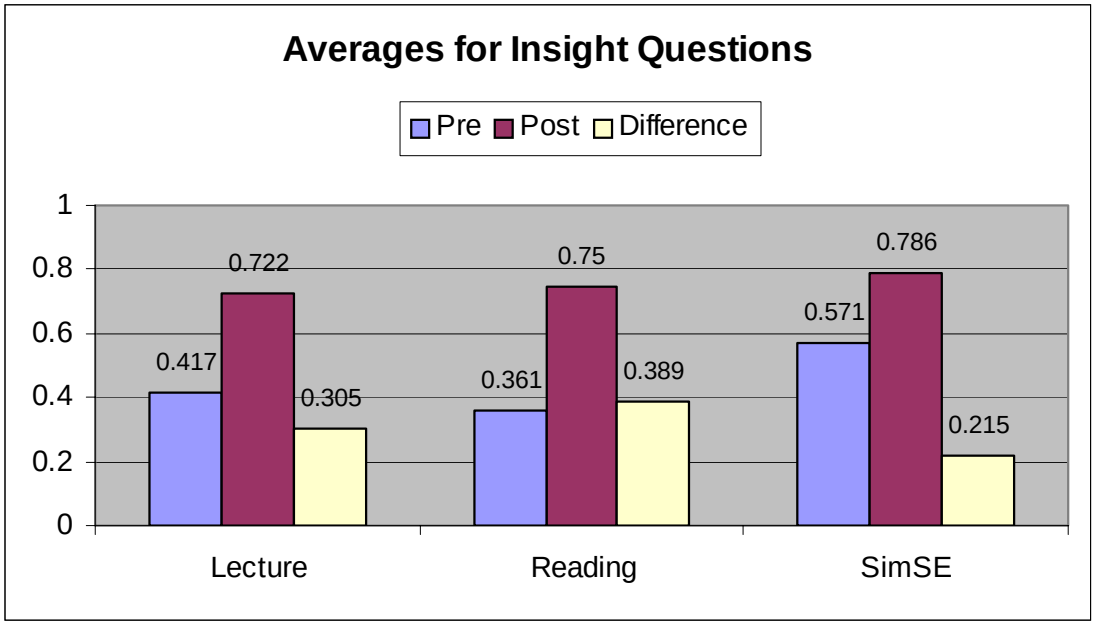


**Figure 63: Test Score Results for Insight Questions Divided by Treatment Group.**

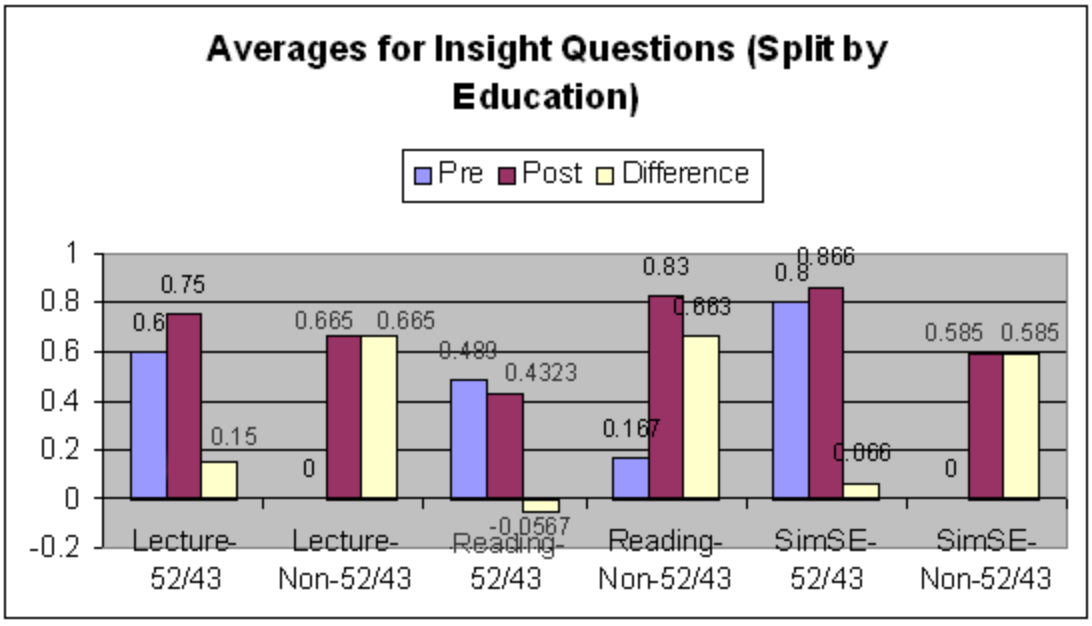


**Figure 64: Test Score Results for Insight Questions Divided by Treatment Group and Educational Experience.**

The averages for the application questions seem to follow the overall trends, as is seen in Figure 65. However, splitting each group into 52/43 and non-52/43 students, shown in Figure 66, yields a notable trend: Of the 52/43 students in all groups, the 52/43

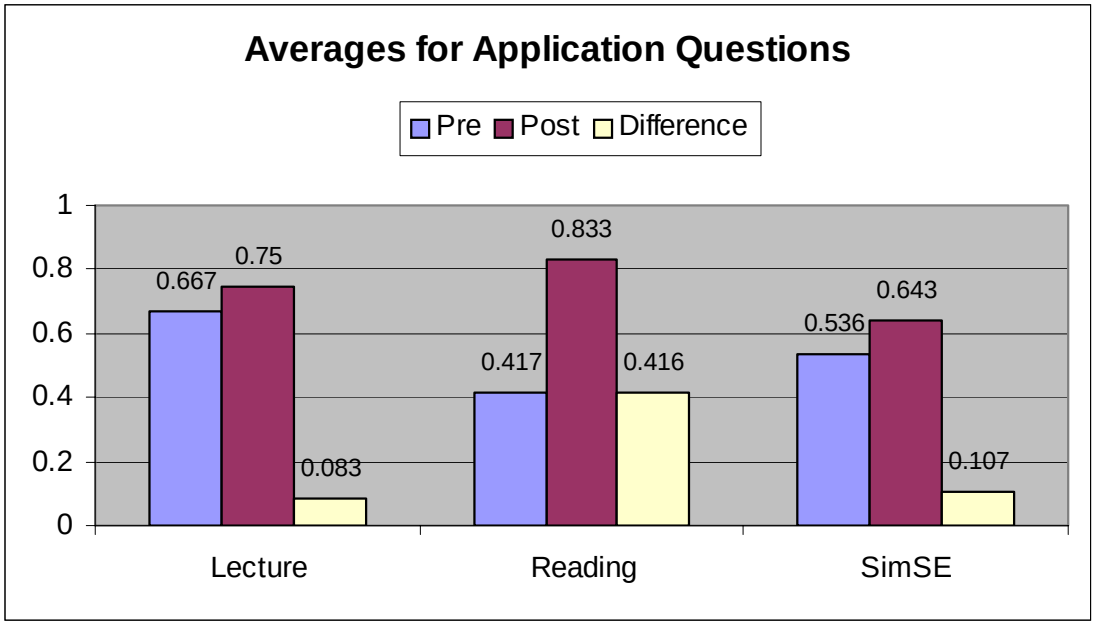


**Figure 65: Test Score Results for Application Questions Divided by Treatment Group.**

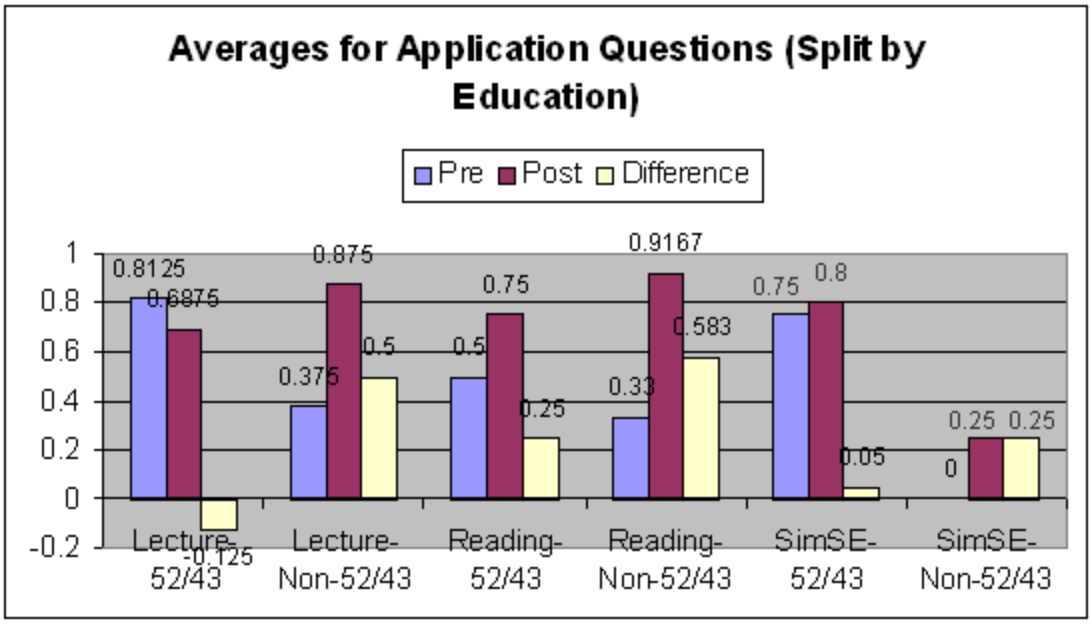


**Figure 66: Test Score Results for Application Questions Divided by Treatment Group and Educational Experience.**

students in the SimSE group had the highest post-test scores for application questions. This may suggest that SimSE is also especially effective in teaching the type of skills needed to answer these types of questions, namely, skills in applying software process

knowledge to real-world problems (specifically the type of skills that SimSE is designed to teach), but, again, with the caveat that the students must first have sufficient background knowledge in software engineering. Of course, this trend is small and only shows that SimSE helped these 52/43 students perform better on these questions by a small amount. (This is not surprising since this group's test scores started out high already.)

Now let us take a look at the biased questions. Surprisingly, the SimSE group was only the second-most improved group in SimSE-biased questions, as shown in Figure 67. This apparently means that either the SimSE-biased questions were not truly SimSE-biased, or else the SimSE group did not really learn the concepts meant to be taught by SimSE as well as they were expected to. The latter is probably true, judging from the rest of the data that indicates the SimSE students did not gain a great amount of new knowledge.

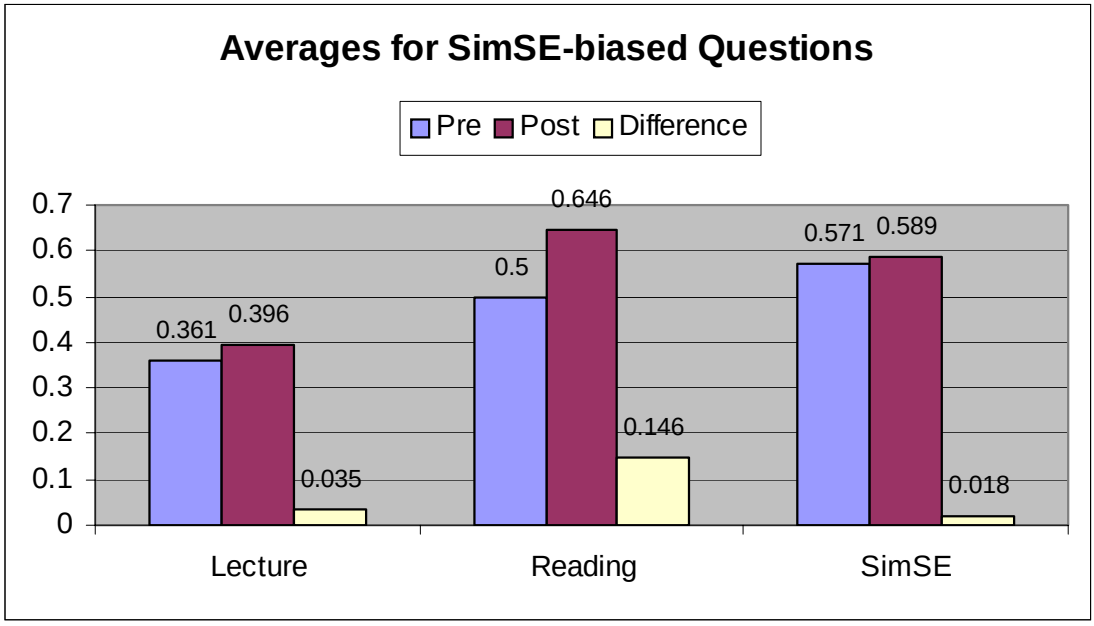


**Figure 67: Test Score Results for SimSE-Biased Questions Divided by Treatment Group.**

Breaking down the groups into 52/43 and non-52/43 students for the SimSE-biased questions reveals more interesting insights (see Figure 68): First, the 52/43 SimSE group scored much higher than the non-52/43 SimSE students on both the post-test (0.675 versus 0.375), and in the difference between the pre- and post-test (0.183 versus -0.0435). In fact, the non-52/43 SimSE students actually worsened in their performance on SimSE-biased questions from the pre- to the post-test. This was the only instance in the SimSE group in which students did not improve from pre-test to post-test (aside from the 52/43 students with the reading/lecture-biased questions, probably simply because SimSE did not teach these particular concepts). Moreover, this was the only instance throughout the whole experiment in which the 52/43 students in a group improved more than non-52/43 students (with the exception of the lecture students with the SimSE-biased questions, however, the haphazardness of the data and the fact that the lectures do not really address the SimSE-biased questions suggests that the lecture students may have been guessing on the questions). Again, these trends may indicate that in order for students to learn the concepts SimSE is designed to teach, it must be used only with students who have sufficient background knowledge in software engineering.

Finally, the trends for the reading/lecture-biased questions are as expected: the reading and lecture groups improved about equally as well (0.5 and 0.48, respectively), and the SimSE group's improvement was almost zero (0.018). This data is reflected in Figure 69.

The students' answers to the questionnaires provide us with more insight about the trends seen here, by capturing their attitudes, thoughts, and perceptions of the particular learning exercise in which they were involved. The first part of the questionnaire asked

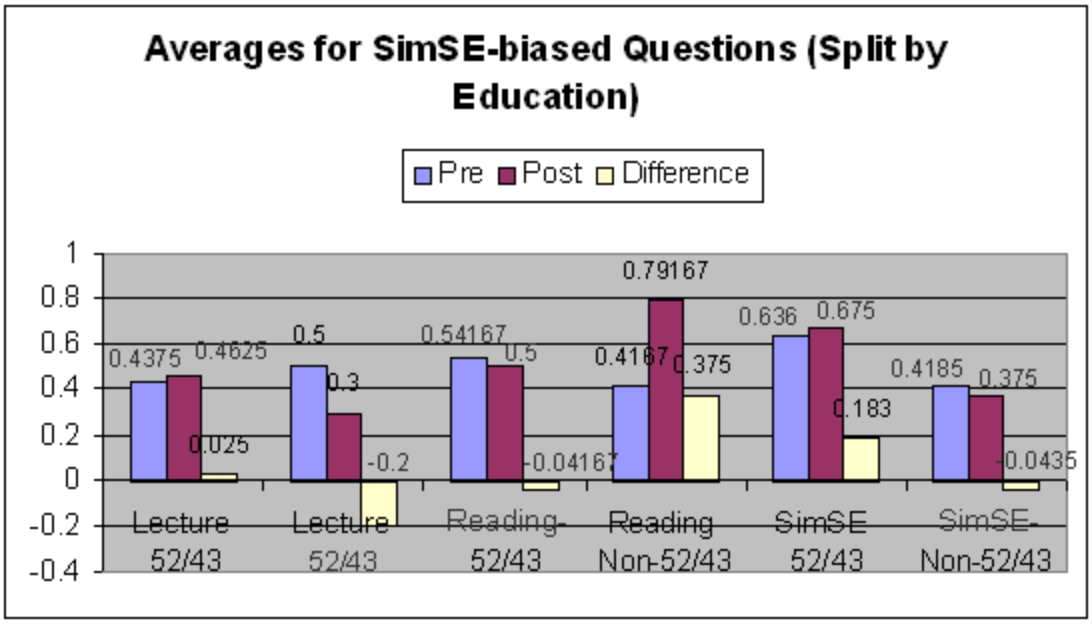


**Figure 68: Test Score Results for SimSE-Biased Questions Divided by Treatment Group and Educational Experience.**

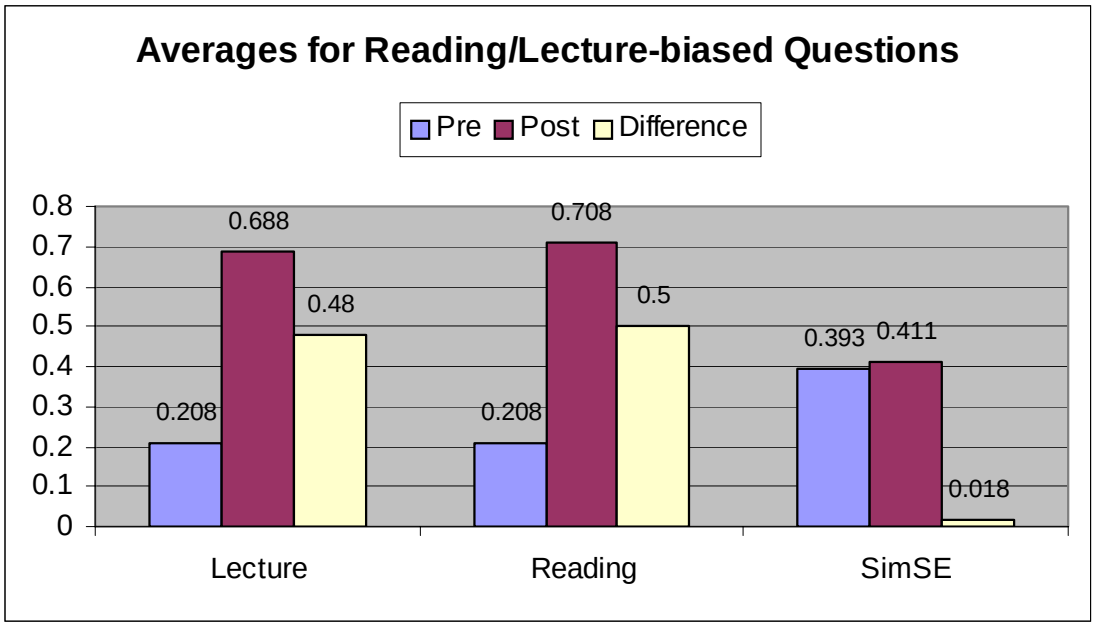


**Figure 69: Test Score Results for SimSE-Biased Questions Divided by Treatment Group.**

the students to rate and report on various aspects of their experience. These are summarized in Table 6.

**Table 6: Summary of Rating/Reporting Questions on Questionnaire.**

| Group | Time spent (average) | More / less than / equal to assigned? (number of subjects reporting) | Enjoyable? (average) | Engaging? (average). | Helpful in learning process concepts? (average) | Effective lecturer? (average) |
|---|---|---|---|---|---|---|
| **Lecture** | **2 hours** | **6 all** | **3.7** | **3.9** | **3.9** | **4.3** |
| **Reading** | **1.5 hours** | **1 more, 4 less, 1 all** | **2.0** | **2.0** | **2.8** | **N/A** |
| **SimSE** | **4.6 hours** | **7 less** | **3.7** | **4.0** | **3.2** | **N/A** |

The SimSE group spent significantly longer on the exercise (4.6 hours) than either the lecture group (2 hours) or the reading group (1.5 hours). However, every subject in the SimSE group also played less than they were assigned (they did not play each model enough to get a score of 85 or above). When asked why, the answers of every SimSE subject indicated that the game was frustrating for them because it was too hard to get a good score, so they gave up. Some of them did not even attempt all of the models even once (one student only played the waterfall model, and never even started the other two models). All of them stated that they needed more guidance and/or background information in order to be able to succeed in the game.

This was probably the biggest factor behind the SimSE group's comparatively low test score improvement–if they did not complete the exercise, they are obviously not going to learn all of the lessons the exercise was meant to teach. Plotting the time spent on the learning exercise versus improvement from pre- to post-test (see Figure 70) underscores this. Although the reading group showed no correlation between time spent and improvement (and this analysis was irrelevant for lectures since all subjects spent the same amount of time), the SimSE group showed a strong and highly significant

correlation between the time spent and improvement (Pearson r=0.81, p<0.001)[1].

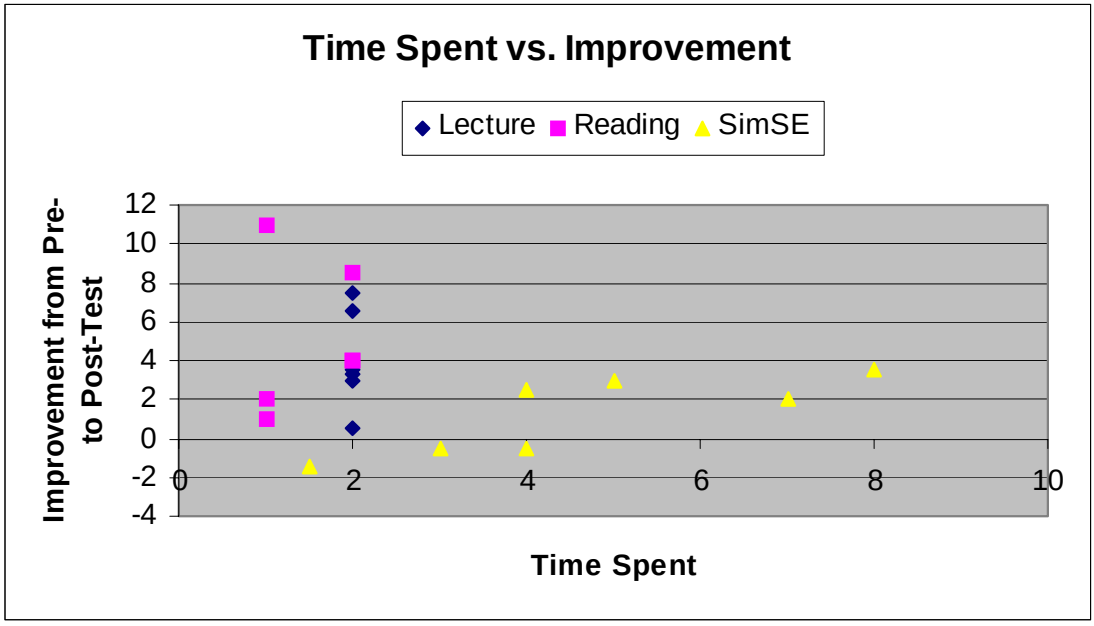


**Figure 70: Time Spent on Learning Exercise Versus Improvement from Pre- to Post-Test.**

This suggests that even though the way in which SimSE was delivered in this case was less-than-ideal, the students were still learning something as they played, and it is likely that they would have learned more had they continued playing and not given up when they did. It is also evident from this data that the cost in time for using SimSE effectively is high. This is a potential drawback of SimSE, as it requires significantly more time invested on average than readings or lectures covering roughly equivalent material.

All of this is more evidence that SimSE needs to be used in conjunction with other teaching methods, but, since the 52/43 students also complained that they did not have enough guidance to succeed in the game, it is clear that more guidance needs to be given with the game, even with students who have background knowledge in software engineering. This corroborates the data from in-class use, in which students also

[1] To be thorough, we also used the two main ordinal measures of association (Gamma and Spearman rho), and the results were similar (Gamma=0.789, p<0.001; Spearman rho=0.845, p<0.001).

expressed this same frustration at the lack of direction given with the game (although not with the same frequency or severity as in this experiment).

Even though the SimSE students found the experience frustrating, they still gave it surprisingly high scores in enjoyability (3.7 out of 5, tied for first place with the lecture group) and level of engagement (4.0 out of 5, higher than both the lecture group (3.9) and the reading group (2.0) rated their experiences). They also still felt that SimSE, although frustrating, was helpful in teaching software process concepts (3.2 out of 5, compared with 3.9 for the lecture group and 2.8 for the reading group).

The questionnaires also asked the students to state which method of learning about software process concepts they would choose if given a choice (along with a mention that playing the game would take twice as long as reading or hearing a lecture). The answers to these questions are summarized in Table 7. Again, even though the students found SimSE frustrating, the majority of them would still choose to learn software process concepts through SimSE instead of reading (57%) and instead of lectures (86%). And for those who had never been exposed to SimSE before, just the idea of a software engineering game is intriguing and attractive—100% of the reading group would choose a game over reading, and 50% of the lecture group would choose a game over lectures. This was also evidenced by the students' observable attitudes during the experiment: on the first day of the experiment when they were assigned to their treatment groups, most of the students assigned to the reading and lecture groups were noticeably disappointed, and even angry, as they expected to get to play a game as part of the experiment! On top of that, several of these students asked for information about how to get a copy of the game, so they could play it on their own time after the experiment was over.

**Table 7: Summary of Learning Method Choice Questions on Questionnaire.**

| Group | Reading or lectures? | Reading or game? | Lectures or game? |
|---|---|---|---|
| **Lecture** | **100% lectures** | **N/A** | **50% lectures, 50% game** |
| **Reading** | **60% reading, 40% lectures** | **100% game** | **N/A** |
| **SimSE** | **N/A** | **43% reading, 57% game** | **14% lectures, 86% game** |

What is interesting is that while the difficulty of figuring out how to get a good score was repeatedly listed as the most discouraging part of SimSE, it was also listed many times as one of the most enjoyable and attention-grabbing aspects of the exercise. Although the challenge posed by SimSE might have been too large in this particular setting, they still enjoyed the process of trying to tackle it. Other aspects of the game the students listed as most enjoyable were: the "gaming" aspects such as graphics and interactivity, the "fun" of being in control, managing employees, and getting to experience a hands-on approach to software engineering.

All of these high ratings in an out-of-class context with little guidance suggest that, if used in the proper context (in conjunction with a software engineering course) and with an adequate amount of guidance, SimSE has tremendous potential to be a highly enjoyable, engaging, and effective method of teaching software process concepts in which students are excited to participate.

Other questions on the questionnaire asked about each students' amount of industrial experience in software engineering, how many software engineering classes they had taken, and whether they were male or female. However, since only one person had industrial experience, only one person had taken more than one class in software engineering, and there were only three females in the experiment (two in the lecture group and one in the SimSE group), there were no detectable trends involving this data.

To sum up, this experiment revealed the following insights about SimSE:

- *The idea of playing a game to learn software process concepts is intriguing and attractive to students.* Both the fact that the SimSE group was noticeably the most desirable group to be in on day one of the experiment, and their stated preference on the questionnaire for learning software process concepts through a game over other teaching methods attest to this.
- *SimSE should only be used complementary to other teaching methods, and more guiding information than was given in this experiment must be provided when giving an assignment to play SimSE.* This was suggested again and again in the data: The 52/43 SimSE students performed overwhelmingly better than the non-52/43 students on the post-tests; the non-52/43 students performed overwhelmingly worse than any other group on the post-test, and only improved modestly between pre- and post-test; the 52/43 SimSE students seemed to learn the SimSE-biased concepts much better than the non-52/43 SimSE students; and all SimSE students repeatedly expressed that they needed more information and guidance to be able to succeed in the game.
- *The longer a student plays SimSE, the more they learn.* The one strongly significant effect that was detected in this experiment was the positive correlation between time spent playing SimSE and the magnitude of improvement between pre- and post-test. Thus, proper investment of time is a critical factor in using SimSE effectively.
- *It requires significantly more time to play and learn from SimSE than it does to attend lectures or complete a reading assignment covering roughly the same*

*concepts.* This high time commitment no doubt added to the frustration felt by SimSE players in this experiment. Although it is possible that the extra time actually pays off in additional learning that does not take place through readings or lectures, this was not suggested by the data from this experiment.

- *SimSE has tremendous potential to be an effective, engaging, and enjoyable tool for teaching software process concepts—if used in the context of a software engineering course, and if adequate instruction and guidance is provided to the students playing SimSE.* Even without adequate background knowledge and guidance, students who played SimSE rated their experience remarkably high in several different areas. Moreover, even though none of them fulfilled the assignment to completion, they still improved between pre- and post-tests, indicating that they did learn something, and the data indicates that the more they played, the more they learned.

## 9.4 Observational Study

### 9.4.1 Setup

Although the first three experiments provided us with much valuable data about SimSE and its ability to help students learn, the insight gained into an individual student's learning process was limited to questionnaires and test results, due to the design of these experiments. Thus, for our final experiment we conducted an in-depth observational study in which we observed students playing SimSE and interviewed them about their experience. The primary purpose of this study was to investigate the learning processes students go through when playing SimSE—namely, *how* SimSE helps people learn

software engineering process concepts. We designed SimSE with a number of learning theories in mind (in particular, Learning by Doing, Situated Learning, Keller's ARCS, Discovery Learning, and Learning through Failure), and student responses from the first three experiments hinted that some of these were being employed. Because these experiments focused on other aspects besides the in-depth learning process, these learning theories were not looked into any further. In this experiment, therefore, we specifically set out to detect which of these (and other) learning theories actually come into play in the learning process of a SimSE player. In so doing, we aimed to gain further insight into the way SimSE helps students learn, which can inform future work both in educational simulation in software engineering, and educational simulation environments in general. Moreover, this data can serve to help validate whether or not the learning theories simulation environments are thought to embody are actually employed by students who use them.

The secondary purpose of this experiment was to evaluate how well the explanatory tool achieves its goals of aiding students in understanding their score, helping them recognize where they went wrong and/or right in the approach they took, and assisting them in planning a successful approach to the next run of the game. This was done by having some students play SimSE with the explanatory tool and some without, and noting the differences in their attitudes and opinions about the game, as well as their behavior in playing the game.

For this experiment, we recruited 15 undergraduate computer science students who had passed either ICS 52 or Informatics 43 to participate (although only 11 actually completed the experiment—four students either cancelled or missed their appointment).

As in previous experiments, the requirement of passing either ICS 52 or Informatics 43 was put in place because of the intended audience for SimSE: those who have some prior knowledge of basic software engineering concepts. The number of subjects (15) was chosen because this was a highly focused study that required a significant amount of time spent with each subject. Therefore, the focus was on getting an in-depth look at a few subjects, rather than an overall, shallower view of a larger number of students.

This experiment occurred in a one-on-one setting—one subject and one observer. Each subject was first given approximately 10 to 15 minutes of instruction on how to play SimSE. They were then observed playing SimSE for around 2.5 hours. Eight subjects played with the explanatory tool and three played without. While they were playing, their game play and behavior were observed and noted. Following this, the subject was interviewed about their experience for about 30 minutes, and the audio of the interview was recorded. In addition to any spontaneous questions the observer formulated based on a particular subject's actions or behavior during game play, all subjects were asked a set of standard questions. Several of these questions were designed to specifically detect the presence of one or more learning theories in the subject's learning process. Some questions did not target a particular theory or set of theories, but were instead meant to evoke insightful comments from the subject from which various learning theories could be detected, and from which general insight into the learning process could be discovered. The standard set of questions is listed here, with the targeted learning theory (or theories) listed in parentheses afterwards when applicable.

1. *How would you summarize what happened in game 1/2/x?*

2. *How did your score change each time you played (did it improve, worsen, fluctuate, remain constant)?* (Discovery Learning, Learning through Failure)
3. *To what do you attribute the change (or lack of) (improvement, worsening, fluctuation, steady state) of your score with each game?* (Discovery Learning, Learning through Failure)
4. *How many times did you feel you "won", or were successful at the game? What did you learn from each of these games?* (Discovery Learning, Learning through Failure)
5. *How many times did you feel you "lost", or were unsuccessful at the game? What did you learn from each of these games?* (Discovery Learning, Learning through Failure)
6. *Do you feel you learned more when you "won" or when you "lost"? Why?* (Discovery Learning, Learning through Failure)
7. *When you lost, did you feel motivated to try again or not? Why?* (Learning through Failure)
8. *On a scale of 1 to 5, how much did playing SimSE engage your attention? Why?* (Keller's ARCS)
9. *How relevant do you feel this experience will be to your future experiences in software engineering? Why?* (Keller's ARCS)
10. *How much has your level of confidence changed in the learning material since completing this exercise?* (Keller's ARCS)
11. *How satisfied do you feel with your experience playing SimSE?* (Keller's ARCS)

12. *Did you feel that you learned any new software process concepts from playing SimSE that you did not know before? If so, which ones?*
13. *If you feel you learned from SimSE, what do you believe it is about SimSE that facilitated your learning?*

The next three questions were primarily designed for comparison between the subjects who used the explanatory tool and those who did not. These questions aim to discover how the player went about figuring out the reasoning behind their scores, as well as how well they understood this reasoning.

14. *Where do you think you went wrong in game 1/2/x?*
15. *Please describe the process that you followed to figure out the reasoning behind your score, or where you went wrong/right.*
16. *How would you alter your approach in the next game based on this information?*

The final four questions were only asked of those who used the explanatory tool, and were designed to determine how well the explanatory tool achieves its purpose.

17. *What was your strategy for using the explanatory tool to figure out where you went wrong/right?*
18. *How helpful did you feel the explanatory tool was to figuring out where you went wrong, the reasoning behind your score, and how you could improve in the next game?*
19. *Was there anything confusing about the explanatory tool? If so, what?*

20. *What changes would you make to the explanatory tool to make it more helpful for figuring out where you went wrong, the reasoning behind your score, and how to improve in the next game?*

Following the experiment, the interviewer's observations and interview notes were analyzed to try to discover which learning theories were employed, and how, as well as to discover any other insights about SimSE as a teaching tool that could be gained from this data. We used different techniques for detecting different learning theories. Learning by Doing and Situated Learning are theories that are more difficult to detect than some of the others—any associations between the act of "doing" or realistic factors in the learning environment and the process of learning are not obvious through observation, and interview questions targeting these theories would be too suggestive (e.g., "Was it the act of *doing* something that helped you learn?") Rather, we wanted to ask more general questions that would allow the subject to state their opinions and comments honestly and freely, without any subtle suggestions about what the "right" answer was (e.g., "If you feel you learned from SimSE, what do you believe it is about SimSE that facilitated your learning?") We mainly used the subjects' answers to questions like these, as well as any other relevant comments, to detect these two theories. Specifically, anything they said that indicated the usage of one of these theories was noted. For example, "SimSE helped me learn because I could actually put into practice what I learned in class" would be considered a comment indicative of Learning by Doing. An example of a comment hinting at the Situated Learning theory might be, "SimSE helped me learn because I could experience a software engineering process in a realistic setting."

To measure the utilization of the Keller's ARCS learning theory, we primarily looked at each subject's answers to questions that specifically asked about their attention, (perceptions of) relevance, confidence, and satisfaction in relation to SimSE. In addition to this, we also used observations of their behavior during game play, as well as any other relevant comments they made, to make conclusions about the presence of this theory in their learning process. For example, we noted whenever a subject behaved in a way that suggested their attention was or was not engaged (e.g., leaning forward with an expectant look on their face, or letting out a sigh of boredom), or made a comment relating to attention, relevance, confidence, or satisfaction (e.g., "It was fun", "It was repetitive", or "It was frustrating").

The presence of the Learning through Failure theory was detected in a manner similar to that of Keller's ARCS. Some of the interview questions were specifically targeted to discover how often subjects felt they failed and how much they learned through those failures. We analyzed answers to these questions, as well as other relevant comments and behavior (e.g., appearing defeated after a low score) to evaluate the utilization of this theory.

We looked for the presence of the Discovery Learning theory by analyzing several parts of the interview, as well as observations of game play, to determine what each subject learned and how they learned it (i.e., through independent discovery or some other means).

We also sought to detect if any other learning theories that we did not anticipate were employed by analyzing interviews and subject behavior to see if any additional theories became evident. Finally, we compared the answers and behaviors of those who used the

explanatory tool to those who did not, noting any differences that would suggest how well the explanatory tool achieves its purposes.

The version of SimSE used in this experiment was the same as the one used in the previous two experiments, with the addition of the explanatory tool for eight of the 11 subjects. To ensure that the results could be generalized for SimSE as a whole, and not for a particular simulation model, a variety of models were used—four subjects played the RUP model (three with the explanatory tool and one without), one subject played the waterfall model (with the explanatory tool), and six subjects played both the rapid prototyping and the inspection models (four with the explanatory tool and two without). The rapid prototype and inspection models did not take as long to play as the others, so they were always played together. The waterfall model was only played by one subject because it was deemed less appropriate for this experiment than the other models, as will be explained in Section 9.4.2. Two of the subjects had played SimSE before, so to make sure they did not have any prior experience with the model played, they were given the RUP model, which was newly built and not yet released at the time.

### 9.4.2 Results

**General Learning.** First and foremost, as corroborating with the previous experiments, it appears that all subjects in this experiment learned, at least to some degree, from playing SimSE. All subjects were able to recount software process lessons that they learned from SimSE, nine of the 11 subjects reported that their confidence in the subject matter (software process) had increased at least somewhat, and, for the most part, subjects tended to improve their scores from game to game as they successfully implemented the learned lessons in their game play. However, we found that scores alone are not accurate

indicators of learning—even subjects who were never able to improve their score reported that they still learned, and were able to list a number of specific lessons they could take away from the experience. This can partly be attributed to too-harsh scoring in some models (which will be discussed later in this section), but we also discovered through our observations that fluctuating scores can result from the way most subjects set about tackling the challenges of each model: isolate aspects of the process and experiment with them individually (or in small sets), while keeping the others constant. Thus, once they have mastered one aspect, they move on to another aspect, with their scores fluctuating with each round of experimentation as they likely attempt a few incorrect strategies before discovering a correct one. In the end however, with the exception of the model we determined used too-harsh scoring, nearly every subject was able to achieve their best score with each model the last time they played that model. This, along with each subject's ability to describe lessons they learned, suggests that through the experience they gained a decent understanding of the lessons taught.

**Learning Theories.** The learning theory that was most clearly implemented by every subject was Discovery Learning. All subjects were able to recount at least a few lessons they learned from SimSE, and none of these lessons were ever told to them explicitly during their experience. Rather, they discovered them independently through exploration and experimentation within the game. Interestingly, although all subjects that played a model seemed to discover the same lessons (for the most part), no two subjects discovered them in the same way. Every subject approached the game with a different strategy, but came away with similar new knowledge, suggesting that SimSE, and perhaps educational simulation in general, can be applicable to a wide range of students

that come from different backgrounds with different ideas. This is a central aspect of a learner-focused theory like Discovery Learning. Since learning depends primarily on the learner and not the instructor, the learner is free to use their own style and ideas in discovering the knowledge, rather than being forced to adhere to a rigid style of instruction.

Learning through Failure also seemed to be widely utilized. As mentioned previously, every subject seemed to take a "divide and conquer" approach to playing SimSE, isolating aspects of the model and tackling them individually (or a few at a time). When subjects described the progression of their games in the interviews, it was clear that the way they conquered each aspect was by going through at least one or two rounds of failure in which they discovered what *not* to do, and from this discovering a correct approach that lead to success. When asked explicitly about learning through failure, every subject stated that they learned when they failed, but the amount of learning they reported varied. Five subjects said they learned more from failure than success, two subjects said they learned more when they succeeded, and four subjects said they learned equally as much from failure and success. All but one subject said that they were motivated to try again after they failed. This motivation was also evident in the behavior of several subjects, as some, after the completion of one failed game, hurriedly and eagerly started a new one. One subject even tried to start a new game when the time for the game play portion of the experiment was up and he was already informed that it would be the last game.

Overall, the challenge of receiving a "failing" score and trying to improve it seemed to be a significant avenue of learning and a strong motivating factor of SimSE. We can

abstract away from this a broader lesson for educational simulation environments in general: Simulation models should be made challenging enough that students are set up to fail at times. It is these failures that provide some of the greatest opportunities for learning.

The Learning by Doing theory seemed to be employed by most of the subjects. Eight out of the 11 subjects made comments about their experience playing SimSE that hinted at their usage of Learning by Doing. Some of their comments included:

- "[SimSE helped me learn because it] *puts you in charge of things. It's a good way of applying your knowledge.*"
- "[SimSE helped me learn because it is] *interactive, not just sitting down and listening to something.*"
- "[SimSE helped me learn because] *you're actually engaged in doing something.*"
- "[SimSE is] *a good way of putting concepts into practice.*"

As can be seen, several of these comments mentioned the ability to put previously learned knowledge into practice as a learning-facilitating characteristic of SimSE. This again reinforces the principle that simulation should be used complementary to other teaching methods, so that it can fulfill this important role of being an avenue through which students can employ Learning by Doing as they apply concepts learned in class.

Comments indicative of Situated Learning were also rather frequent, mentioned by seven out of the 11 subjects. Some of these included:

- "[SimSE helped me learn because] *it was very realistic and helped me learn a lot of realistic elements of software engineering, such as employees, budget, time, and surprising events.*"
- "[One of the learning-facilitating characteristics of SimSE was] *seeing a real-life project in action with realistic factors like employee backgrounds and dialogues.*"
- "[One of the learning-facilitating characteristics of SimSE was] *the real-life scenarios.*"
- "[SimSE is helpful to learning because] *it would be good for students to apply what they learn in a pseudo-realistic setting.*"

The realistic elements in SimSE seem to add significantly to its educational effectiveness. Thus, it is important to include elements of the real world in any educational simulation, in order to situate students' knowledge in a realistic environment.

Keller's ARCS Motivation Theory seemed to also be employed by the subjects, although certain aspects of the theory came out stronger than others. To explain, let us look at the four aspects of the theory (attention, relevance, confidence, and satisfaction) individually.

First, the attention of the subjects seemed to be quite engaged with SimSE. This was evident in their body language, the comments made both during game play and the interview, and their ratings of SimSE's level of engagement. Many of them spent the majority of their time during game play sitting on the edge of their seats, leaning forward and fixing their eyes on the screen. There were head nods, chuckles in response to random events and character descriptions, shouts of "Woo hoo!" after achieving a high

score in a game, shaking of the head when things were not going so well for a player, and requests of, "Can I try this one more time?" when the experiment's allotted time for game play was coming to an end. Words some subjects used to describe SimSE in the interview were "challenging", "fun", "interesting", "addictive", and "amusing." When explicitly asked how much SimSE engaged their attention, the students rated it quite high—4.1 on average out of five.

Second, relevance was rated moderately high, but not as high as level of engagement. Five of the subjects rated SimSE's relevance to their future experiences as "pretty relevant" or "very relevant", five described it as "somewhat" or "partially" relevant and one said it was not relevant at all. Some of the positive comments about relevance included:

- "*It will definitely help in decision-making.*"
- "*It will be very relevant for my ICS 121 midterm next week.*"
- "*What it's simulating I expect I'll be doing eventually*."
- "*It will be pretty relevant because I kind of want to do some software engineering in the future if I get a job in that area*."

Some of the subjects who rated relevance less positively had the following comments:

- "*It didn't help that much compared to what I already know*."
- "*I definitely don't want to go into software engineering so it's probably not too relevant for the future, but for classes it could be useful*."
- "[I do not consider it relevant to my future experiences because] *I don't really see myself as the type of person who would govern those processes, I see myself as the guy that follows the orders*."

Although not explicitly asked about SimSE's relevance to their past experiences, nearly all of the subjects mentioned that they used some of the knowledge they had learned in software engineering courses to come up with their strategies for playing the game, suggesting that there is also a relevance between their past experiences (learning the concepts in class) and their learning experience with SimSE.

Third, most subjects felt their level of confidence in the learning material had increased at least somewhat since playing SimSE. Four subjects reported their level of confidence had changed "a lot" or "very much", five said it had changed "somewhat", and two said it had not changed at all. Some of their comments included:

- "[I now have] *a better understanding of how* [the processes] *work.*"
- "*It enhanced my level of knowledge of the process.*"

Interestingly, subjects' confidence ratings seemed to be unassociated with their performance in the game. For instance, several people who never improved their score still reported that their confidence in the subject matter had increased as a result of playing SimSE. This suggests, again, that game scores alone are not an accurate indicator of learning. It is the experience of going through the simulated process, rather than the eventual result, that seems to be the central avenue of learning.

Fourth, satisfaction was rated quite high by the subjects. Nine out of the 11 subjects reported that they were "quite satisfied", "very satisfied", "fully satisfied", or "pretty satisfied", and three subjects stated they were "somewhat satisfied." Most of the reported factors that contributed to a feeling of satisfaction pertained to a subject's increasing success from game to game, although some also mentioned that the fun and challenge of SimSE contributed to their satisfaction as well.

In reviewing and analyzing the interview transcripts, one unanticipated learning theory became evident: Constructivism [25]. The basis of this theory is that learners construct new concepts or ideas based on their past knowledge and current experience. As already mentioned, when asked how they came up with their strategies for playing SimSE, nearly all of the subjects reported that it was a combination of knowledge they had learned in their software engineering course(s) and the experience of playing the game to figure out how to succeed. This is another piece of evidence suggesting that simulation should be used complementary to other teaching methods, so that learners can employ Constructivism as their new knowledge is built and framed on their existing knowledge.

**Explanatory Tool.** Most of the subjects that had access to the explanatory tool did make use of it, using it for, on average, five to 25 minutes after most games. It was obvious that the subjects who did not have the explanatory tool (to whom we will henceforth refer as "non-explanatory subjects") were significantly more confused and unconfident about the reasoning behind their scores than those who did have the explanatory tool (to whom we will henceforth refer as "explanatory subjects"). All of the non-explanatory subjects expressed this, while only one explanatory subject stated such an opinion. The following are some of the comments made by the non-explanatory subjects:

- "*I still don't really understand what the score is based on*."
- "*I'm not really sure exactly what the scoring criteria are*."
- "*I was trying to guess what I was doing wrong, so I probably chose the wrong areas that I was doing wrong, and then I tried to switch back to my original way*

*and then I kind of forgot what that was and once I started trying to improve it, all of my little details started changing and I didn't know what parts were causing my score to go lower*."

- "*I felt like I knew, oh, that's where I went wrong sometimes, like I should spend a little less time there, but a lot of times I was wrong about where it was I went wrong*."
- "*I thought maybe afterwards* [SimSE should] *kind of give you a description of here's where you went wrong, or a little hint or something, not exactly the actual solution, or little warning signs like you forgot to do this*."
- "[I wish SimSE had] *more descriptions of what each task does*."

Interestingly, the last two comments even seem to describe some aspects of the explanatory tool, indicating that the addition of this tool fills a real need of SimSE.

There was no noticeable difference in the other aspects of each subject's experience (such as learning theories employed, ratings of SimSE, game scores, etc.) between the two groups, suggesting that even when a player doesn't fully understand the reasoning behind their score, they can still have an overall successful learning experience. And again, while scoring does play an important part, it is not *the* most important part—it is the overall experience of going through game play that seems to be the most influential factor.

The helpfulness of the explanatory tool as expressed by the explanatory subjects was only moderate. Of the eight explanatory subjects, three said it was "very helpful" or "pretty helpful", two said it was "somewhat helpful", and three said it was not helpful at all. What is interesting, however, is that these ratings of helpfulness were strongly

correlated to whether or not the subject made use of the rule descriptions in the explanatory tool (which are brought up by clicking on an action graph to find out more information about the action). Four of the eight explanatory subjects read the rule descriptions, and four did not. Of those that read the rule descriptions, three of them rated the explanatory tool as either "very helpful" or "pretty helpful", and one rated it "somewhat helpful." This is in stark contrast to the four subjects who did not read the rule descriptions: three of them said the explanatory tool was not helpful, and one said it was only somewhat helpful. Furthermore, most of the positive comments made about the explanatory tool pertained to the rules in some way:

- "*Rules were a major help*."
- "[What was helpful about SimSE was that] *it's a combination of being able to read the rules and apply them and go through the process*."
- "*The rules are really helpful—even if someone doesn't know anything about software engineering I think the rules can teach you how to play the game*."

Only two of the eight explanatory subjects reported that they got any useful information out of the graphs. Thus, it seems that the usefulness of the explanatory tool as it currently stands lies primarily in the rule descriptions.

Even when subjects did use the graphs, very few of them used the composite graphs, tending to focus mainly on the object and action graphs. This was surprising, as we anticipated that the composite graphs would be the most useful part of the explanatory tool. However, based on our observations it seems that this lack of use can be attributed to the difficulty of formulating a meaningful object and action graph combination that will produce an insightful composite graph. Based on the number of possible

combinations, this seems to be too overwhelming a task for the average student. To address this, we plan to add functionality that will point the user to useful composite graphs for each model. Whether this is something that will be specifiable in the model builder, or something that can be automatically detected by the explanatory tool per individual game remains to be seen. In our future work we will experiment with both options to determine which is most feasible and effective.

An additional way to make the graphing mechanism of the explanatory tool more useful would be to add some attributes to each model that are meant specifically for explanatory graphing purposes. For example, in the RUP model we could add project attributes representing suggested budget for each phase and suggested time for each phase. (These attributes would be hidden in the game interface but visible in the explanatory tool.) The player could graph these attributes against the actual budget or time for each phase to see where they need to adjust. As another example, the inspection model could include a "meeting productivity" attribute that shows how productive the inspection meeting as a whole was over time, so the player could see, in one attribute, how effective their approach was at each point in the game. In our future work we plan to add explanatory attributes such as these to each model (see Chapter 12).

The overwhelming importance of the rule descriptions leads us to a critical question: If the rule descriptions were so useful, why did only half of the explanatory subjects use them? We specifically asked those who did not use them why they did not use them and for all of them the answer was the same: they forgot they were there. After subject #1 failed to use the rule descriptions, we started being more careful about emphasizing their presence when instructing the students on how to play SimSE and use the explanatory

tool. However, subject #2 also did not look at the rule descriptions. We continued to emphasize the rule descriptions more and more in our instructions, including showing specific examples of how they can be useful, along with reminding subjects that "this is one of the most useful parts of the explanatory tool and everyone forgets to look at them!" Finally, subject #4 was the first to read the rules. The remainder of the explanatory subjects after subject #4 (with the exception of #5) also used the rule descriptions. Although placing strong emphasis on rule descriptions in the instructions seemed to eventually help, there is obviously more that needs to be done to get students to take advantage of this valuable resource. We anticipate that making the rule descriptions more accessible will help significantly. At the moment, in order to get to the rule descriptions one has to first generate either an action or a composite graph, click on a point on the graph, and then click on the Rule Info tab. This is a somewhat cumbersome and non-intuitive process to go through. Some of the subjects, even though they remembered that the rules were there somewhere, had to ask to be reminded of how to access them. We plan to experiment with making rule descriptions directly accessible from the main explanatory tool user interface to see if this increases their visibility and thus, their usage. This could take the form of an added drop-down list of actions from which the player could choose to automatically bring up the rule descriptions for that action.

One additional insight discovered from this experiment was that students wanted the explanatory tool accessible during the game. Some of them even assumed it was accessible during the game and asked how to access it. As mentioned in Chapter 8, this is something that we plan to do. Whether or not having it accessible during the game will

"give too much away" and take away too much of the challenge remains to be seen. Additional experiments after this change is made will be necessary to determine this.

**The importance of instruction.** As we already saw in the way subjects tended to forget about the rule descriptions, the instruction one receives in playing SimSE is crucial. The explanatory tool instructions were one example, but it was equally apparent that the instructions given about how to play the game in general make an enormous difference as well. The first subject failed to take advantage of several informational resources in SimSE that are designed to guide a player and help them succeed in the game. For example, the subject only skimmed over the starting narrative, seemed to ignore the text in the speech bubbles, and failed to monitor the status of any artifacts during development (even though these features were pointed out during the instruction period). This subject's opinions of SimSE and the experience in general were lower than average, perhaps as a direct result of these oversights. After subject #1, therefore, we altered the instructions given to place more emphasis on these overlooked sources of information, including giving specific examples of why and how they can be helpful. As the experiment went on, we discovered more aspects of SimSE that could be helpful to players, but that were not being taken advantage of, and we accordingly altered the instructions to emphasize these as well. By about midway through the experiment, subjects were giving most of these aspects the proper attention, and their overall opinions of the experience seemed to be significantly more positive as a result.

The obvious lesson we can learn from this is that the instructions given to a player of SimSE must include certain specific pieces of information about components they must pay attention to in order to promote a maximally effective educational experience. It is

not safe to assume that students will figure these things out on their own. Our first step in addressing this issue will be to rewrite SimSE's instruction manual (included electronically with a download of a SimSE game) to include these commonly overlooked features. However, given that users are notorious for not reading instruction manuals, it is necessary to take this a step further, especially for in-class usage of SimSE. Students could be given paper-based handouts along with the electronic version, and the instructor could emphasize the importance of reading them carefully. Even more effective would be holding a training session in class under the leadership of a teaching assistant or instructor, in which students are also given verbal instructions, with live examples, to underscore and illustrate the information provided in the textual instructions.

Another issue that needs to be explored is whether SimSE can be altered so that a player's success is less dependent on their attention to these details, and more on the integral game play. Perhaps some of the crucial information contained in textual components such as the starting narrative and speech bubbles can be incorporated into game play in a non-textual way. It is unclear how this could be done, but it is definitely an avenue that warrants investigation. Another possible way to address this is by making the models simpler so that less attention to detail is needed. However, this would take some of the challenge of SimSE away, so this is something that also must be carefully experimented with.

**Models.** The data revealed a number of insights about the SimSE models used in this experiment, both individually and as a whole. One of these insights was the average time it takes to play each model. These averages are shown in Table 8. The inspection model, being our only model in the "specific" category (see Chapter 7), was the one that took the

**Table 8: Average Time Taken to Play Different SimSE Models.**

| Model | Average Time to Play (in Minutes) with Explanatory Tool | Average Time to Play (in Minutes) without Explanatory Tool |
|---|---|---|
| **Inspection** | **7** | **2** |
| **Rapid Prototyping** | **13** | **8** |
| **Waterfall** | **34** | **N/A** |
| **RUP** | **47** | **21** |

shortest amount of time to play. The rapid prototyping model took approximately twice as long to play, and the waterfall model was almost three times as long as the rapid prototyping model. The RUP model was the most time-consuming model to play.

From this data we were also able to compare the relative difficulty of each game in terms of scores subjects were able to achieve. Table 9 shows two types of average scores for each model: the average score for all times that model was played ("average overall score"), and the average high score for each subject who played that model ("average high score"). Subjects had the easiest time achieving a high score in the rapid prototyping model, and a somewhat more difficult time mastering the inspection model. The waterfall model was the next most difficult in terms of scoring, and the RUP model was by far the most difficult of all the models.

**Table 9: Average Scores Achieved for Different SimSE Models.**

| Model | Average Overall Score | Average High Score |
|---|---|---|
| **Rapid Prototyping** | **78** | **96** |
| **Inspection** | **54** | **90** |
| **Waterfall** | **35** | **68** |
| **RUP** | **8** | **32** |

As mentioned previously, we purposely designed the rapid prototyping model with more lenient scoring than the other models. Our observations of subjects who played this model suggest that the scoring is perhaps *too* lenient. For instance, one subject went through the model with only one round of prototyping and received a score of 85 with a

resulting system that was 13% erroneous and implemented only 70% of the customer's requirements. The subject felt satisfied with the score of 85 and assumed they were not going to play that model anymore since they had "mastered" it. As a result, the subject did not even know that their resulting system lacked in these areas since they did not bother to look at any artifact statistics to try to find out why 15 points were deducted (until the observer stated that they would be playing the model again to try to get a higher score). This is obviously a dangerous situation—a student could come away from playing this model thinking that one round of prototyping is sufficient for completing a successful rapid prototyping approach. Accordingly, we plan to adjust the scoring for this model to make the penalization for such situations harsher.

The RUP model fell on the other end of the scoring spectrum—it seemed to be too harsh. 24 RUP games total were played in this experiment, and only four of them resulted in non-zero scores. Three of the five subjects who played RUP never achieved a score greater than zero, even though their performance was improving from game to game. Therefore we also plan to adjust the RUP model scoring to make it more lenient.

Although the scoring for the inspection model did not seem to be overly harsh, it was clear from interviewing subjects who played it that the majority of them missed some of its most central lessons. Even when a subject figured out an approach that would lead to a high score, they would sometimes translate it incorrectly into real-world concepts. For instance, a number of subjects thought that the size of the code and the size of the checklist should correlate to each other (e.g., a small checklist should accompany a small piece of code, a large checklist should accompany a large piece of code, etc.), whereas the model is actually trying to teach that there is a certain size of checklist (approximately

one page) and a certain sized piece of code (less than or equal to 200 lines) that are ideal for all code inspection situations (see Section 7.2). A complicating factor that likely detracted from this lesson is the fact that, with three different pieces of code and three different checklists, there are nine possible combinations that a player could choose, and only one of them is maximally rewarded by the model. (This is in addition to the numerous combinations of employees that can also be chosen.) Players often tended to stumble upon the correct combination of checklist and piece of code only by luck.

We can address these problematic issues by both simplifying the search process and simultaneously providing more guidance to the player in finding the correct combination and inferring the correct real-world lessons. To do this, we will first remove the smallest checklist choice and the largest piece of code choice (or vice-versa) so that it will be more obvious that there is no dependency between the two. At the same time, this will reduce the search space that the player must go through. Additionally, we will include with the inspection model carefully-worded questions for the player to answer that suggest the proper real-world translations (e.g., "What is the ideal size of checklist (in number of pages) that should be used in a code inspection?") This is precisely what we did in the in-class usage of SimSE with the questions that each student had to answer in order to receive their extra-credit points (see Section 9.2). The students that played the inspection model in class, with the questions, answered them correctly for the most part, which seems to indicate that they did make the proper real-world interpretations (although it would require actually interviewing these students to determine whether this is actually true). This observational experiment has suggested that these questions may be

necessary to always include with certain models (such as inspection) that have lessons which sometimes tend to get interpreted incorrectly.

Another model that seems to necessitate the inclusion of a set of guiding questions is the waterfall model. As mentioned previously, only one subject in this experiment (subject #2) played this model. This is because it became clear from observing and interviewing this subject that the waterfall model was too large and complex for the setting of this experiment. The subject seemed somewhat lost and confused, was unable to achieve a good score, and only reported one new concept that they had learned from playing SimSE. We believe this can be attributed to the fact that the waterfall model contains too many variables, interactions between these variables, and possible actions a player can take at any given time (this model steers the player very little, allowing them to perform almost any action at any time). On top of the waterfall activities in the model, there are also several non-software engineering specific aspects—employees have energy and mood levels (in addition to their experience levels and pay rates), and they can get sick, take breaks, and quit their jobs. A player can fire an employee, give them bonuses, and give them pay raises (aside from assigning them regular software engineering tasks). Because of this complexity, it is hard for a player to isolate and experiment with variables to find a successful approach to the game. If given a set of guiding questions, however, we expect that the lessons contained in the model will be more readily noticed and learned by a player, as this seemed to be the case with the in-class usage of the waterfall model (see Section 9.2).

The overarching lesson this experiment taught us about models is that it is difficult to create good game-based educational simulation models. There are a number of crucial

choices that must be made to develop an educationally effective model. Namely, the following critical issues must be carefully considered:

- **The number of lessons/variables.** As we saw with the waterfall model, including too many effects results in an overly complex model that students find difficult to play and learn from. Including too few effects would likely make a model that is not challenging enough to keep the student engaged.
- **How lessons are communicated.** There are numerous different ways a lesson can be taught through a SimSE model. Sometimes it becomes apparent that a lesson is not getting picked up on (as in the inspection model), indicating that something about the way it is communicated must be changed. Alternatively, a set of guiding questions can be made to accompany the model, to point the player in the direction of lessons that are difficult to pick up on.
- **Explicit versus implicit information.** A modeler can put all of the information a player needs to know in the instructions, starting narrative, speech bubbles, and rule descriptions of the model, but there is no guarantee the player will actually read these sources of information. Therefore, removing the need for this information by making the model simpler or using other, non-textual ways to communicate this information should be explored.
- **Scoring.** Although students who play a model with overly-harsh scoring can still learn from the experience (as we saw with the RUP model), it is still a frustrating experience to be unable to achieve a high score. A greater danger, as we saw with the rapid prototyping model, is overly-lenient scoring, which can

lead to the player coming away from the experience with the wrong lessons being learned. A careful balance between the two must be achieved.

- **User testing.** In our experience, often the only way to discover the weaknesses of a model is through user testing. As with any software, the developer always holds misconceptions in their minds about such things as what will be obvious to players versus what must be pointed out to them, how people are going to play the model, and how difficult a model will be, among others. These misconceptions will only be brought to light by allowing others to play a model and collecting their feedback.

From our own experience, it seems that the most effective way to learn the proper balance of all these factors and create good models is through practice. This experiment revealed that our later models (rapid prototyping and RUP) are noticeably better than our earlier models (waterfall and inspection) at getting their lessons across effectively. (Despite the scoring issues with rapid prototyping and RUP, players nevertheless seemed to learn a significant amount from these models, based on their interviews.) We plan to include this lesson, plus the critical considerations mentioned above, in our model builder "tips and tricks" guide (see Appendix B).

**Implications for Class Use.** Two of the subjects in this experiment had played SimSE previously, one in the pilot experiment and one in class. Both subjects were asked if they learned more playing SimSE during this experiment or during their previous time(s) playing it, and both reported they learned more during this experiment. They also provided the same reason for this, which is best summed up by a direct quote from one of these subjects: "When you have somebody watching and checking up on you, you work

harder and I guess, in the end learn more." Because the presence of an observer seems to have a positive effect on learning in SimSE, it would be ideal if students using it in conjunction with a class could be observed one-on-one, although this is obviously infeasible. However, a possible way to simulate this "observer presence" would be to instrument SimSE with a logging mechanism that records traces of the games and sends this information to the instructor in a format that can be quickly and easily viewed and assessed. (The students would, of course, be told that this information is being sent so that they feel the added pressure of an observer's presence.) Another option is to use a "pair programming" approach in which students play SimSE in groups of two, so that each can be the observer of their partner. Whether or not these options would take too much fun out of the experience and obviate the extra motivation that seems to come from an observer presence would need to be determined through actual experimentation.

**Applicability for Varying Academic Abilities.** With any instructional method, there will always be some students who "just don't get it." There was one subject in this experiment that seemed to fit this description with SimSE. This subject was unable to make much progress in either of the two models he played, mainly because he missed some things that were very obvious to all of the other subjects (e.g., more than one round of prototyping should be done). The subject also tended to simply repeat the same approaches over and over even though they continually resulted in less-than-ideal scores. Surprisingly, however, this subject still seemed to learn a significant amount (although probably somewhat less than other subjects who "got it"), judging from the interview. This corroborates the findings of our in-class use that suggested SimSE is equally applicable for both students with high and low academic performance levels. From this

experiment, however, we can sum this up in a slightly different way: even students who seem to largely "miss the mark" when playing SimSE can still learn from the experience.

**Summary.** To summarize, this observational study revealed the following insights about SimSE:

- *Discovery Learning, Learning through Failure, and Constructivism are the learning theories most central to SimSE, being employed by all subjects. Learning by Doing and Situated Learning were employed by most subjects, but not all. Keller's ARCS theory was moderately evident, as some of its aspects (attention and satisfaction) were more seen more strongly than others (relevance and confidence).* All of the theories we used in the design of SimSE (plus one unanticipated theory—Constructivism) were observed to be employed by the subjects, although some to a greater extent than others. Thus, educational simulations should be designed with these theories in mind, aiming to maximize the characteristics that are known to promote each one.
- *SimSE's explanatory tool is a useful resource for helping players understand their score, but its value lays primarily in its rule descriptions.* To make the graph generation feature more helpful, the explanatory tool and/or the models will need to be enhanced to provide a larger set of useful graphs, along with ways to point the player to these graphs. In addition, the rule descriptions, which are currently somewhat hidden in the user interface, must be made more directly accessible to the player.
- *The instruction one receives in playing SimSE is crucial.* Subjects tend to miss important information if it is not adequately emphasized in the instructions.

Thus, instruction must be a carefully and deliberately planned part of SimSE use, either with paper-based handouts, training sessions, or some other means.

- *It is difficult to create educationally effective SimSE models.* A modeler must make a careful balance of such aspects as achieving the proper scope, giving the player adequate guidance, communicating the model's lessons in an effective way, and making scoring neither too difficult nor too hard. Achieving this balance requires both practice in building models and collection of user feedback.
- *Models that are unusually large and models containing lessons that are difficult for students to translate into real-world concepts require the accompaniment of a set of guiding questions to adequately communicate these lessons to the player.* There are certain lessons that almost all players picked up on, but others that seemed to be either hidden among other lessons, or difficult to pick up on for some other reason. Based on the fact that students who used SimSE in class and were given a set of questions to answer about the material seemed to pick up on these lessons, this approach should always be used with models containing these less perceptible lessons.
- *An observer presence can be educationally beneficial to players of SimSE.* Students who played SimSE both with and without the presence of a one-on-one observer reported that they learned significantly more when being observed. Thus, use of SimSE in class may be more effective if an observer presence is simulated either through automatic logging and reporting of students' games, or playing SimSE in pairs.

- *Even students who have unusual difficulty succeeding in SimSE can still learn from the experience.* The one subject who seemed to miss many of the lessons picked up on easily by other subjects still seemed to employ several learning theories and was able to report several things he had learned from the experience.

## 9.5 Model Builder and Modeling Approach Evaluation

We informally evaluated SimSE's model builder tool and associated modeling approach in terms of its expressiveness, or its ability to model a wide variety of different software processes of different scales, purposes, and teaching objectives. As evidenced by the six models we built spanning the three different categories (classic, modern, and specific), overall, SimSE seems to have achieved a relatively high level of expressivity. These models vary rather widely in several different aspects such as scope, scoring difficulty, intermediate feedback, and guidance, but all of the ones we have used with students (five out of 6—we have not used the XP model with students) appear to help students learn the concepts they are designed to teach.

We have already mentioned that building a successful SimSE model is a difficult task. This was especially evident in the performance of undergraduate students we recruited to build models. One of these students spent three quarters trying to build an inspection model, the efforts of which ended in failure—the resulting model consisted of a static, linear set of steps of an inspection process. Another student spent two quarters trying to build a RUP model, and this also resulted in an unusable model with very little dynamics. (Both the inspection and RUP models were then rebuilt, resulting in the ones described in Chapter 7.) Our third attempt at having an undergraduate build a SimSE

model was slightly more successful, resulting in the XP model described in Section 7.4. However, although this model is playable, it is flawed in some ways. Its most significant problem is that it tries to teach all of its lessons through the same effect—the slow-down of activities. Specifically, failing to follow any of the XP practices taught by the model (e.g., pair-programming, frequent releases, rapid prototyping, using coding standards) all result in the same consequence—slow-down of development. Therefore, because so many factors contribute to the same effect, it would be quite difficult for a player to detect which one(s) are responsible for the effect. Thus, part of our future work will entail rebuilding this model to use different effects to illustrate different consequences, in order to make the lessons clearer.

The only successful model that we did not build ourselves was the incremental model. This model was built by a graduate student well-versed in software process and game development. It took him approximately one week to build this model, and in our class use it appeared to be effective at communicating the lessons it contains. Thus, it seems a certain level of knowledge is required to be able to build an effective SimSE model, a level normally not possessed by undergraduate computer science students.

The one-week development time of this graduate student seemed to be the standard for our model development as well. All of our models took, on average, one week (7 days) of full-time work to develop, with the last day or two usually being devoted to play testing and adjustment. Larger models, such as the RUP model, took longer than this, while shorter models, such as inspection, took less than a week.

SimSE's model building process was unexpectedly enhanced by the addition of the explanatory tool. Because this tool provides direct insight into a model's internal

workings, it has proven to be a useful aid in building models. An illustrative example is the following: in the RUP model, one of the published "rules" of this process is that the four phases of the process (Inception, Elaboration, Construction and Transition) take approximately 10%, 30%, 50%, and 10% of the total cycle time, respectively [87]. To test the implementation of this rule in a model prior to the inclusion of the explanatory tool, one would have to write down the time it takes for each phase and then calculate the relative percentages. With the explanatory tool, however, a quick glance at a graph like the one shown in Figure 71 will yield the same results.

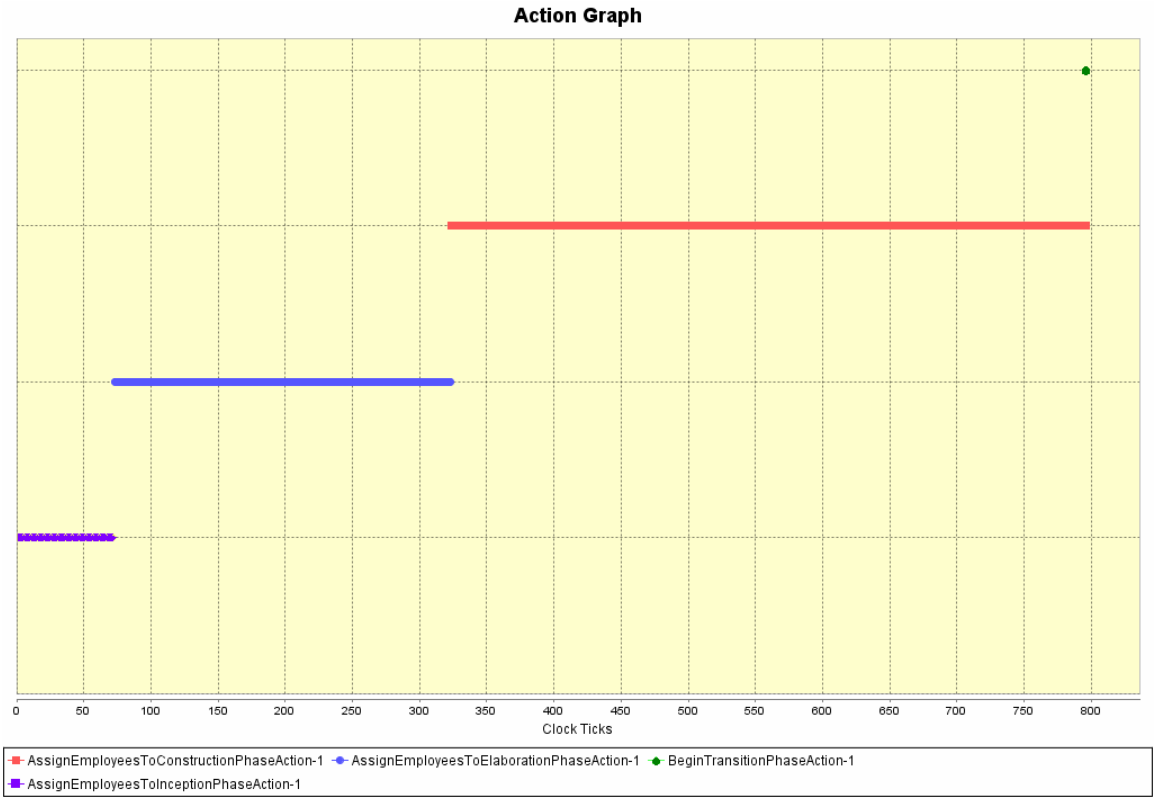


**Figure 71: A Graph Generated by the Explanatory Tool that Depicts the Relative Lengths of Rational Unified Process Phases.**

## 9.6 Summary

This chapter has described the five different parts of our approach's evaluation. Each of these was designed to assess a different aspect of SimSE—the pilot experiment focused on SimSE's initial potential as a teaching tool, the in-class use focused on how SimSE could fit into a software engineering course, the comparative experiment focused on discovering the differences between SimSE and traditional instructional approaches, the observational study focused on the learning process SimSE promotes, and the model builder evaluation focused on the expressiveness of SimSE's modeling approach. The collective results from these can be distilled into a summative list of valuable lessons and insights about our approach. The first three lessons pertain to the effectiveness of SimSE in helping students learn software process concepts:

- *Students who play SimSE seem to successfully learn the concepts it is designed to teach.* We have seen this clearly, in students' ability to answer questions correctly about these concepts (in class), the strong correlation between time spent playing SimSE and increase in software process knowledge (in the comparative experiment), and players' ability to recount learned concepts and improve their game scores (in the observational experiment).
- *Students find playing SimSE a relatively enjoyable experience.* Students in all experiments enjoyed playing SimSE for the most part, although the enjoyment of those who used it in class was noticeably lower than the others (likely due to the added pressure to perform for extra credit, and perhaps the absence of the explanatory tool).

- *Students find SimSE repetitive when played for extended periods of time.* Although it was clear from the comparative experiment that the longer a student plays SimSE, the more they learn, both the comparative experiment and the in-class usage revealed that a longer playing time also contributes to a feeling of repetitiveness. Because the version of SimSE used in these experiments included neither the explanatory tool nor adequate instructions, it is anticipated that the addition of these two factors will lessen the need for so many repetitions of the same model when used in classes in the future.
- *Students learn through playing SimSE by employing the theories of Discovery Learning, Learning through Failure, Constructivism, Learning by Doing, Situated Learning, and Keller's ARCS.* Educational simulations should therefore be designed with these theories in mind, aiming to maximize the characteristics that are known to promote each one.
- *SimSE is most educationally effective when used as a complementary component to other teaching methods.* All four experiments strongly suggested that a certain level of existing software process knowledge must be possessed by a student in order for maximal learning to be promoted. Thus, SimSE should be used with other teaching methods that provide this required knowledge, and not be used as a standalone tool.

The next set of lessons concern in-class usage of SimSE, and the critical considerations that must be made when such an approach is taken:

- *Provide students with adequate and proper instruction in playing SimSE.* This was clearly evident in the frustration and confusion felt by the subjects in the in-

class usage and the comparative experiment, who did not feel they received enough guidance to succeed in SimSE. The results from the observational experiment corroborated this, as it was observed that subjects tended to miss important information if it was not sufficiently emphasized in the instructions. Thus, instruction must be a carefully planned part of SimSE's use, and should include such measures as holding training sessions and/or providing paper-based handouts.

- *Students should be assigned a set of questions to answer about each model they play*. Comparing the in-class usage and the observational experiment results revealed that such questions help guide the student in discovering less discernable lessons. Moreover, questions such as these provide the instructor with a way to assess how much the student learned from the exercise.
- *In-class usage of SimSE may benefit from the addition of an observer presence.* As we saw in the observational experiment, the presence of an observer seemed to motivate students to more effective learning. This could be simulated in classroom usage by either instrumenting SimSE with mechanisms for automatic logging of simulation runs and reporting of these runs back to the instructor, or having students play SimSE in pairs.

We also gained important insights about SimSE's applicability to different types of students:

- *SimSE has applicability to females as well as males.* The opinions of females in these experiments were comparable to those of males. In the pilot experiment female opinions were even higher, on average, than those of males. Thus,

SimSE has the potential to help students of both genders learn software process concepts.

- *SimSE has applicability for students of varying abilities.* We saw in our in-class usage of SimSE that both students who did well on other assignments and those who did poorly were able to succeed in the SimSE exercise. Both the pilot and in-class experiments showed that a student's amount of industrial experience also does not seem to have an effect on SimSE's applicability to them. The observational study revealed that students who have unusual difficulty succeeding in SimSE can nonetheless come away from the experience having learned several lessons. Together, these results suggest that SimSE can be an effective teaching tool for students of different backgrounds and aptitudes.

The results of our experiments also revealed important lessons about the role and effectiveness of SimSE's explanatory tool:

- *The explanatory tool is a needed and useful part of SimSE that helps players understand the reasoning behind their score.* The most frequent complaint of the students who played SimSE without the explanatory tool (in all four experiments) was the lack of feedback given about their performance in the game. Students who played SimSE with the explanatory tool (in the observational experiment) overall found it to be a helpful resource for understanding their score and the simulated process.
- *The value of SimSE's explanatory tool as it currently stands lies primarily in its rule descriptions.* Students who used the explanatory tool found rule descriptions to be the most useful part and the graphs to be only marginally

useful. Mechanisms for providing more useful graphs (and pointing players to them) should be added to the explanatory tool and/or the models.

Finally, our experiences also taught the following overarching lesson about SimSE's model builder and modeling approach:

- *SimSE's model builder and modeling approach are adequately expressive for creating a wide variety of software process simulation models, but designing these models in such a way for them to be maximally educationally effective is a difficult task.* We were able to build a representative set of simulation models that differed in several fundamental aspects and seemed to communicate their software process lessons effectively. However, our experience with building models and using them with students revealed that the task of creating good models is nontrivial, requiring critical and difficult choices to be made about such issues as scope, guidance, lessons, and scoring. Making the proper choices about these issues can only be learned through practice and user testing.

If we revisit the evaluation questions posed at the beginning of this chapter, we can see that the results of the experiments described in this chapter have provided answers to each one:

1. **How do students feel about the learning experience playing SimSE (e.g., is it enjoyable, do they perceive it as an effective method of learning software process concepts)?** Students enjoy and get excited about playing SimSE, although when it is not used in the context for which it was designed (as a complement to a software engineering course) and/or not used with the explanatory tool, students at times find it frustrating. For the most part, students

feel that it is a reasonably effective tool for learning software process concepts. These opinions seem to be shared by a wide range of students, including males and females, high-achieving and low-achieving students, and students with and without industrial experience.

2. **How well does SimSE fit into the traditional software engineering curriculum as a complement to existing methods (which is its intended use)?** SimSE has been shown to integrate relatively well as an optional extra-credit assignment in a course that provides the background knowledge required to understand the simulation models. In our experience with this type of setting, the majority of students chose to complete the assignment, and seemed to learn the concepts the models are designed to teach. However, even though they were learning some of the same concepts in class lectures and readings, many of them still felt that their experience with SimSE was frustrating, and felt that it would have been significantly improved had more guidance and background information about the concepts embodied in the models been given.

3. **How well does SimSE teach the software process concepts that its models are designed to teach?** If given adequate instruction and background knowledge, students who play SimSE do seem to glean from the simulation models the concepts they are created to teach, regardless of gender or academic performance.

4. **How does SimSE compare to traditional methods of teaching software engineering process concepts such as reading and lectures?** In a setting that used SimSE as a standalone teaching tool rather than a complementary one,

SimSE was enjoyed as much as lectures and more than reading, perceived to be more educationally effective than reading but less than lectures, and measured (using pre- and post-tests) to be less effective than both reading and lectures. The time investment required to play and learn from SimSE was significantly higher than both reading and lectures. Use of SimSE in this setting revealed that the proper amount of guidance and instruction must accompany SimSE's use, and it must be used complementary to other teaching methods, in order for it to fulfill its educational potential.

5. **Are the learning theories that SimSE was designed to employ actually being employed by students who play the game, and are there other, unexpected learning theories that are being employed by SimSE?** Discovery Learning, Learning through Failure, and Constructivism (an unanticipated theory) are the learning theories most often seen to be employed by players of SimSE. Learning by Doing and Situated Learning seem to be employed by most players of SimSE. Keller's ARCS theory is a moderately employed theory of SimSE, and some of its aspects—attention and satisfaction—are exhibited more strongly than others—relevance and confidence.

6. **Are the SimSE model-building approach and associated tools adequately expressive?** The tools and approach were found to be adequate in expressing a wide range of different software process models. However, building an effective SimSE model is a difficult endeavor that requires the careful balance of several critical issues. This task can be made less difficult through practice and user testing.

7. **Does the SimSE explanatory tool help players of the game understand their score and the process better than using the game without the explanatory tool?** The explanatory tool does seem to help players understand their score, but it primarily does so through its rule descriptions. The explanatory tool can likely be made to fulfill its purpose even more effectively if more useful graphs are created and highlighted to the user, and the rule description feature is made more accessible.

Though our experiment results have provided answered to these questions, they have also raised new questions—questions that can only be answered through further experimentation. Our evaluations showed that there are a number of adjustments and enhancements to our approach that need to be experimented with. Specifically, the following future evaluations must be conducted:

- **In-class usage with modifications.** Four modifications must be made to our approach for further in-class usage: First, we will make SimSE a mandatory, rather than optional, exercise. Second, we will use SimSE with the explanatory tool in class, as the only version used in class to date has not included the explanatory tool. Third, we will increase the level of instruction students receive in learning to play SimSE, by providing them with paper-based handouts that contains detailed instructions, and requiring them to attend a training session in which an instructor illustrates these instructions and shows them how to play SimSE through live examples. Fourth, to try to further motivate students through an observer presence, we will add an automatic logging and reporting mechanism to SimSE that records a student's game and sends a trace of the

game back to the instructor. We will also place students in groups of pairs to play SimSE. The perceptions, opinions, and learning of students who use SimSE in class with these modifications will be carefully studied and compared to previous in-class usage to try to determine the effects these modifications have on the effectiveness of our approach. Of particular interest will be whether or not these alterations reduce the repetitiveness of SimSE reported by students in the comparative and in-class experiments.

- **Observational experiments with new and revised models.** Our observational experiment proved invaluable for revealing flaws in our existing models. Thus, we plan to continue these types of experiments with models we will build in the future, as well as with revisions of existing models (which will be revised based on the results of our observational experiment).
- **Observational experiments with a revised explanatory tool.** Our observational experiment also revealed the need for more useful graphing mechanisms and more accessible rule descriptions in the explanatory tool. We plan to make these enhancements and then assess them with further observational experiments.

Overall, our evaluations revealed that SimSE can be an effective, engaging, and enjoyable tool for teaching software process concepts when used correctly with the proper critical considerations taken into account. However, some hurdles remain. The enhancements and evaluations described here are designed specifically to address these hurdles in order to help SimSE achieve its full educational potential.

# 10. Related Work

One notable piece of work that has used similar principles to ours in developing their approach is represented by AgentSheets [114], an educational simulation environment focused on the simulation-building activity as the primary learning experience. AgentSheets has been used at multiple educational levels, and has been shown in numerous evaluations to be very effective. AgentSheets is relevant to our approach in that it concerns simulation and roots itself in learning theories, both from design and evaluation standpoints. Our approach has aspired to achieve the same kinds of favorable results, but in a different domain with somewhat different concerns. The first difference is that our approach models only software engineering processes, while AgentSheets is a general purpose simulation environment that can simulate a wide variety of different processes. Our modeling and simulation approach was deliberately designed to be less flexible than AgentSheets, focusing specifically on software engineering processes. As a result, SimSE is more powerful and appropriate for modeling software engineering processes, and not able to model other types of processes.

The second difference between our approach and AgentSheets is that AgentSheets focuses on the simulation-building activity as the primarily learning experience, while our approach instead focuses on the simulation-playing aspect. Because our model-building process is geared toward the software engineering instructor rather than the student, building a model in SimSE is not as straightforward as in AgentSheets, and therefore we have chosen not to focus on the model building process as a learning activity (although it is certainly possible to use it as a learning exercise for advanced students, and our approach has been used successfully in such a situation [13]). Despite

these differences in focus, AgentSheets and other simulations like it have provided us with examples of rigorous, learning theory-centric evaluation methodologies that we have adopted in our evaluation approach.

In addition to general educational simulations such as AgentSheets, there also exist a number of other educational simulations that focus specifically on the domain of software engineering. Because these educational software engineering simulations relate directly to our approach, we will focus on making direct comparisons to them in the remainder of this chapter. As described in Section 2.1.3, these approaches fall into three main categories: industrial simulation brought to the classroom, group process simulation, and game-based simulation. Our approach falls into the game-based simulation category, which shares the same general focus of the industrial simulation category—the overall processes of software engineering. Because group process simulations have a different focus—namely, group discussion and interaction processes [103, 136]—we will omit this category of approach from this discussion.

As described in Section 2.1.3, industrial software engineering simulations brought to the classroom involve the use of highly-realistic simulators to illustrate to students, using real-world data, the overall life cycle and project planning phenomena of software engineering [35, 106]. These approaches differ from ours in four major ways.

First, because the original purpose of industrial simulations is prediction, their simulation models are based strictly on empirical data. SimSE's primary focus is on education, not prediction. Accordingly, some portions of our simulation models are deliberately unfaithful to reality to make them more appropriate for educational purposes (see Section 7.7).

Second, industrial simulations have a low level of interactivity, generally running in the following overall manner: Obtain a set of inputs from the user (e.g., project complexity, time allocated to inspections, person power), run the simulation, and output a set of results (e.g., cost, time, defects). In contrast to this, because SimSE is designed as an educational game, we aimed to make its game play as interactive as possible. We designed SimSE to operate on a clock-tick basis to give the student an active role in the simulation and allow them to drive the simulation continuously throughout the game, making adjustments and steering the process as necessary.

Third, industrial simulations are strongly focused on prediction (as this is their primary purpose), but not prescription—specifying the allowable next steps a user can take at any given point in the process. Because of our focus on interactivity, engagement, and educational effectiveness, SimSE makes ample use of both predictive and prescriptive aspects in its game play in order to maximally promote these qualities (see Section 4.3).

Fourth, in contrast to SimSE's fully graphical user interface, industrial simulations have non-graphical user interfaces that generally display a set of gauges, graphs, and meters rather than characters and realistic surroundings. Again, this is motivated by the purpose of industrial simulations—tools meant to be used in industrial environments for the purpose of prediction do not necessitate entertaining graphical user interfaces.

Finally, industrial simulations are non-customizable. Because they are typically created to predict the effects of process changes on a particular real-world process, they are built upon a precise model of that real-world process with no need for simulating

other processes. SimSE's educational purposes, on the other hand, require the ability to demonstrate a wide variety of software processes, hence its customizability.

There have also been a handful of approaches that, like SimSE, fall into the game-based educational software engineering simulation category. Unlike industrial simulations, these game-based simulations share the same underlying purpose of our approach: allowing students to practice "virtual" software engineering processes in an interactive, fun environment that engages the student, making learning more effective. However, the existing approaches differ from our approach in some fundamental ways.

OSS [129] is a game-based software engineering simulation environment that allows a user to take a "virtual tour" of a software engineering company. Although OSS includes audio, animations, and more extensive graphics than SimSE, the user's role is rather limited in comparison—the player takes more of a passive "observer" role rather than that of an active participant in a software engineering process. The user can look at sample documents, "listen in" on meetings, and hear explanations of tasks, but they cannot actually effect change on the state of the simulation. Thus, OSS is adequate as more of a software engineering tutorial program than an actual interactive game. Moreover, it is static, containing only one underlying model, without any facilities for customization.

The Incredible Manager [44] is a simulation game designed specifically to train software project managers. Consequently, its focus is different from SimSE's, and is concentrated more on project management than on software processes. In essence, it is much like an industrial simulator with an added graphical, game-like user interface. The interactivity of the game is similar to industrial simulations in that the player creates a

project plan, runs the simulation, and receives a result (but can intermediately stop the simulation, make adjustments, and restart again). Rather than viewing only a series of gauges, graphs, and meters, however, they can see employees working, getting tired, going home for the evening, and coming back the next day. The Incredible Manager also allows for customization of its simulation models through a textual interface, but requires that these models be built on a system dynamics paradigm—a paradigm that is generally used by real-world industrial simulation models.

SimVBSE [78] is a game-based simulation specifically designed to teach students the theory of value-based software engineering [16]. SimVBSE has a relatively high level of interactivity—players can visit different "rooms" in a software engineering company where they can perform such activities as changing project parameters, obtaining feedback from stakeholders, undergoing tutorials on relevant topics, and analyzing project metrics, risks, and investments. The user interface is fully graphical and includes animations and audio. The simulation portrays one real-world case study, and does not include facilities for customization. Thus, the primary difference between SimVBSE and SimSE is that SimVBSE focuses only on value-based software engineering while SimSE instead focuses on simulating a variety of different software engineering processes.

Problems and Programmers [9] is also a game-based simulation, but it is a card game rather than a computer game. It is a two player game designed to simulate a waterfall software development process from conception to completion. Players in the game compete against each other to finish their projects while avoiding the potential pitfalls of software engineering. Being a competitive, multi-player card game, Problems and Programmers is highly interactive. However, it only simulates one process (waterfall)

and, being a simulation that involves physical objects (cards), it is significantly more difficult to customize than a computer-based simulation.

SESAM [47] is the approach that is perhaps most similar to SimSE. It is a game-based simulation environment that allows for the modeling and simulation of different software engineering processes. It also operates on a clock tick basis, allowing the student to drive the simulation throughout the game by performing such actions as hiring and firing employees, assigning them tasks, and asking them about their progress and state of the project. It also includes an explanatory tool that is similar to ours, and is the only approach besides SimSE that does so. However, SESAM differs from our approach in three major ways. First, it lacks a visually interesting graphical user interface, which is considered essential to any successful educational simulation [51]. Players must type in commands textually, and can only "view" the process through the form of textual feedback. The second difference lies in the modeling language. SESAM represents a first example of a software process modeling language that is prescriptive, predictive, and interactive (but not graphical). It is also a highly flexible and expressive language, but its model building process is learning- and labor-intensive and requires writing code in a text editor. There has only been one SESAM model developed to date, which does not give an instructor many examples with which to work when trying to build a new model, and is also perhaps evidence that, despite SESAM's powerful language, the need to actually textually program a model is a significant challenge that few wish to tackle. Third, SESAM has only been evaluated in one small out-of-class experiment. We build on SESAM's approach in four major ways: First, we simplify the modeling process by providing our model builder tool, eliminating the need for writing source code in an

explicit modeling language. Second, we provide support for including graphics in the simulation models. Third, we have chosen to sacrifice some of the flexibility and expressivity that SESAM has by making a number of simplifications to our modeling approach (e.g., limiting all objects to five meta-types). Fourth, we make evaluation and actual class use an integral part of our approach, both so that we can make conclusions about SimSE's effectiveness that are thoroughly rooted in actual experience, and provide insights about educational software engineering simulations and educational simulations in general that can be used by others in the research community.

To summarize, we can generalize four fundamental differences between our approach and the existing educational software engineering simulations:

- *Existing software engineering simulations are not adequately flexible.* Judging from the wide variety of software processes that exist, it is obvious that educational software engineering simulations must be easily configurable to model different processes. Although two of the existing approaches (The Incredible Manager [44] and SESAM [47]) are configurable, SimSE has gone above and beyond their level of configurability through two major features: its graphical model builder tool that removes some of the difficulties of an explicit process modeling language; and a set of pre-existing models that can be easily used off-the-shelf, and/or configured to fit different educational goals.
- *Existing software engineering simulations have not been adequately used and evaluated in a classroom setting.* As mentioned in Chapter 3, one of the guidelines for a successful educational simulation is that it is used complementary to the other components of a course. Although a few of the

existing simulations have been used in conjunction with a class [28, 35, 129], these instances have only been anecdotally observed and reported on. Other approaches that have performed more formal studies have done so with out-of-class experiments [47, 106]. Although useful as initial evaluations, neither approach gives much thorough insight into how simulation can effectively be incorporated into an existing course. One of the fundamental components of our approach is carefully planned in-class use with objective measurements of students' learning, opinions, and attitudes.

- *Existing software engineering simulations have not been robustly verified.* Either in-class or out-of-class, there have been relatively few studies that have definitively affirmed the effectiveness of simulation in software engineering education. Again, with the exception of [106], all of the other experiments involving educational software engineering simulations, although mostly favorable, have been preliminary and informal in nature. Our set of four experiments was a central component of our approach, and these experiments were carefully designed to provide a thorough, well-rounded assessment of SimSE's value as an educational tool.
- *Existing software engineering simulations do not adhere to well-known principles for educational simulations.* The guidelines for successful educational simulations that our approach has been built on (see Chapter 3) have not all been followed in any of these approaches: several of them are only minimally engaging and challenging, many are not used complementary to other teaching methods, and most do not provide feedback and/or explanatory tools.

# 11. Conclusions

This dissertation has presented a new approach to educating students in software process concepts—an approach consisting of three parts: (1) an implementation of a graphical, interactive, educational, customizable, game-based simulation environment for simulating software processes (SimSE), (2) a set of simulation models to be used in seeding the environment, and (3) evaluation of the environment and models, both in actual software engineering courses and in out-of-class experiments.

Our experience with SimSE has provided a number of important contributions to both the field of software engineering education and education in general. The most tangible contribution is the implementation of SimSE, along with its set of simulation models, which have been put through both in-class use and out-of-class formal evaluations.

We have also established through our experience the insight that a graphical, interactive, educational, customizable, game-based simulation environment such as SimSE can be beneficial to software engineering process education. Students who play SimSE tend to learn the intended concepts, and find it a relatively enjoyable experience. These statements apply to students of different genders, academic performance levels, and industrial experience backgrounds. However, in order for SimSE to be used in the most effective way possible, we have demonstrated that it is crucial that it be used complementary to other educational techniques and accompanied by an adequate amount of direction and guidance given to the student.

Our experience has also provided as a contributed insight the role and potential of an explanatory tool in an educational simulation, as well as an implementation of such a tool. In particular, despite the needed enhancements of our explanatory tool, we have

found that it is a much needed and useful part of our simulation approach that significantly aids students in understanding the underlying simulated process and their performance in the simulation.

Our evaluations strongly suggest that SimSE is a useful and educationally effective approach that has the potential to be even more effective if certain modifications are made to its implementation and usage. As it currently stands, some difficulties with our approach exist, most notably the feeling of frustration frequently reported by students who played SimSE in class, the minimal usefulness of the graph generation feature in the explanatory tool, and a certain amount of awkwardness in our modeling approach. We have plans for addressing each of these difficulties in our future work (see Chapter 12). Beyond these observed hurdles, we have also identified a number of promising directions for future research that will potentially add to the effectiveness of SimSE and, in turn, provide even more insights that the research community can utilize. These future research plans are also discussed in the next chapter.

In sum, this dissertation has contributed an approach to addressing some of the difficulties with software engineering education—particularly software *process* education—by allowing students to practice, through SimSE, the activity of managing different kinds of quasi-realistic software engineering processes. Our usage and evaluation of SimSE has demonstrated that this approach does help students learn software process concepts, and has highlighted the crucial considerations that must be made when using such an approach. It is our hope that the lessons learned from our experience can be utilized by the larger research community and eventually contribute to a new generation of software engineers that are better versed in software processes.

# 12. Future Work

Our experience with SimSE, both in its development and its usage, have highlighted a number of areas that can be improved, enhanced, and/or modified to help SimSE better fulfill its goal of providing an engaging, interactive, and effective way for students to learn software process concepts.

Some of these concern features of the environment itself. First, we want to reduce some of the difficulties in our modeling approach that at times require non-intuitive, roundabout solutions (described in Section 4.3). To do this, we will explore ways of adding new constructs to our modeling approach that achieve the needed expressiveness without causing it to degenerate into a full-fledged process modeling language. For instance, we will add the ability to specify in an effect rule specific actions to activate or deactivate, rather than the "all or nothing" approach that currently exists.

We also plan to modify the explanatory tool to address the deficiencies brought forth in the observational experiment—the marginal usefulness of the graph generation feature and the inaccessibility of the rule descriptions (see Section 9.4). To make the graph generation feature more useful, we will augment the simulation models with attributes that are expressly for explanatory graphing purposes (e.g., "suggested budget for phase X" and "actual budget for phase X" that can be graphed against each other). We will also experiment with either adding functionality to the model builder that allows a modeler to specify potentially useful graphs that can be generated for that model, or adding functionality to the explanatory tool that automatically makes graph generation suggestions based on a particular simulation run. To increase the accessibility of the rule descriptions, we plan to add a component to the main explanatory tool user interface

through which they can be viewed. We will also make the explanatory tool accessible during a simulation run, rather than only at the end of one, and conduct further experiments to determine how helpful to learning this may or may not be.

Because a frequent request of students who played SimSE was for better graphics, we will also attempt to enhance the game's graphical sophistication to make it more appealing and engaging. One of the main ways we plan to do this is by adding some simple animation capabilities to the model builder. Specifically, we will add functionality that will allow a modeler to specify different graphics for different states of an object (e.g., an employee with low energy will appear to be sleeping; a highly erroneous piece of code will appear red and flashing).

We also plan to enhance SimSE's graphics by adding semantics to the layout of the office. Currently, the position of an employee is meaningless and their surrounding images are merely for decoration. We will experiment with allowing both employee position and surrounding graphical components to come in to play when specifying effects. For instance, the productivity of an employee working in an XP process could be increased if they are in close proximity to another employee with whom they are pair programming. As another example, an employee's mood could be raised if they have their own large, nicely-decorated, corner office near a window, or lowered if they are stuck in a tiny, dark cubicle with three other people. Including such graphical semantics will also require that we add more standard office surrounding images, such as windows, plants, and pictures.

A more semantically-enhanced map may also require a larger map size, to take full advantage of these enhancements. Currently, the map is limited to 16 x 10 tiles. We will

experiment with making the map size customizable per model to see if this extra flexibility will increase the graphical attractiveness and interaction of SimSE in any way.

In addition to these environment enhancements, we also plan to enhance our repertoire of simulation models by developing a number of new models. In particular, we plan to build a Personal Software Process [74] model, a Team Software Process [75] model, and a model of a component-based software engineering process. We will also explore the possibility of building "mixed" models that illustrate relative strengths and weaknesses of different models, and focus on honing students' skills in recognizing situations in which one approach is better than another, and vice-versa. For instance, we will attempt to build a model that teaches the balance between unit, integration, and acceptance testing, and another model that illustrates the tradeoffs between choosing a particular high-level process approach such as XP or waterfall. We also plan to experiment with building more models of varying complexity. One of the principles for successful educational simulations presented in Chapter 3 states that simulation must start with simple tasks and gradually move towards more difficult ones. Our model-building work to date has been focused on testing and demonstrating the feasibility and applicability of our modeling approach, and has therefore resulted in a comprehensive set of mostly large models. To better apply this principle of moving from the simple to the more complex, we will attempt to create scaled-down versions of our existing models that can be used for introducing students to SimSE before they tackle the more complex models.

As described in Section 9.6, there are also three additional types of experiments with SimSE that need to be conducted. The first of these is further class use with three

modifications: incorporation of SimSE as a mandatory (rather than optional) exercise, class use of SimSE with the explanatory tool, and either an added automatic game logging and reporting mechanism, or placement of students in pairs to play SimSE. The other two types of experiments are both observational in nature: one assessing the modified explanatory tool, and another set of experiments evaluating the future simulation models we will build and the modified versions of our existing ones.

Finally, we will use all of our experience and lessons learned to create SimSE course modules that will help guide instructors in adopting SimSE in their courses. These course modules will include such things as the understandings and/or skills that the module intends to teach, the time it will take, lecture-wise, discussion-wise, and homework-wise, the relevant simulation models to be used, class materials for the instructor to present and discuss, instructions for the students, guidelines on how to hold a SimSE training session for the students, and test questions to be answered with the corresponding correct answers.

## References


1. *JFreeChart*, http://www.jfree.org/jfreechart.
2. Abdel-Hamid, T. and S.E. Madnick, *Software Project Dynamics: an Integrated Approach*. 1991, Upper Saddle River, NJ: Prentice-Hall, Inc.
3. Abernethy, K. and J. Kelly, *Technology Transfer Issues for Formal Methods of Software Specification*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 23-31.
4. ACM Committee on Computers and Public Policy, *RISKS-FORUM Digest*, http://catless.ncl.ac.uk/Risks.
5. Alessi, S.M. and S.R. Trollip, *Multimedia for Learning*. 2001, Needham Heights, MA, USA: Allyn & Bacon.
6. Anderson, J.R., et al., *Cognitive Tutors: Lessons Learned.* The Journal of the Learning Sciences, 1995. **4**(2): p. 167-207.
7. Andrews, J.H. and H.L. Lutfiyya, *Experience Report: A Software Maintenance Project Course*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 132-139.
8. Angehrn, A.A., *Advanced Social Simulations: Innovating the Way we Learn how to Manage Change in Organizations.* International Journal of Information Technology Education, 2004 (to appear).
9. Baker, A., E.O. Navarro, and A. van der Hoek, *Problems and Programmers: An Educational Software Engineering Card Game*, in *Proceedings of the 2003 International Conference on Software Engineering*. 2003: Portland, Oregon. p. 614-619.
10. Beck, K., *Extreme Programming Explained: Embrace Change*. 2000, Reading, MA: Addison-Wesley.
11. Beckman, K., et al., *Collaborations: Closing the Industry-Academia Gap.* IEEE Software, 1997. **14**(6): p. 49-57.
12. Bernstein, L. and D. Klappholz, *Eliminating Aversion to Software Process in Computer Science Students and Measuring the Results*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY, USA. p. 90-99.
13. Birkhoelzer, T. and E.O. Navarro, *Teaching by Modeling instead of by Models*, in *Proceedings of the 6th International Workshop on Software Process Simulation and Modeling*. 2005: St. Louis, MO, USA.
14. Birtwistle, G.M., *Discrete Event Modelling on Simula*. 1979, Houndmills, Basingstoke, Hampshire: MacMillan Education Ltd.
15. Blake, B.M., *A Student-Enacted Simulation Approach to Software Engineering Education.* IEEE Transactions on Education, 2003. **46**(1): p. 124-132.
16. Boehm, B., *Value-Based Software Engineering: Overview and Agenda*, in *Value-Based Software Engineering*, S. Biffl, et al., Editors. 2005, Springer Verlag.
17. 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.

18. Boehm, B.W., *Software Engineering Economics*. 1981, Upper Saddle River, NJ: Prentice Hall, Inc.
19. Boehm, B.W., *A Spiral Model of Software Development and Enhancement.* IEEE Computer, 1988. **21**(5): p. 61-72.
20. Bransford, J.D., et al., *Anchored Instruction: Why we Need it and how Technology can Help*, in *Cognition, Education, and Multimedia: Exploring Ideas in High Technology*, D. Nix and R. Spiro, Editors. 1990, Lawrence Erlbaum: Hillsdale, NJ. p. 115-141.
21. Brereton, O.P., et al., *Student Group Working Across Universities: A Case Study in Software Engineering.* IEEE Transactions on Education, 2000. **43**(4): p. 394-399.
22. Brooks, F.P., *The Mythical Man-Month: Essays on Software Engineering*. 2 ed. 1995, Boston, MA: Addison-Wesley. 336.
23. Brown, J.S., A. Collins, and P. Duguid, *Situated Cognition and the Culture of Learning.* Educational Researcher, 1989. **18**(1): p. 32-42.
24. Brown, S.M., *A Software Maintenance Process Architecture*, in *Proceedings of the Ninth Conference on Software Engineering Education and Training*. 1996, IEEE: Daytona Beach, FL, USA. p. 130-141.
25. Bruner, J., *Acts of Meaning*. 1990, Cambridge, MA, USA: Harvard University Press.
26. Bryan, G.E., *Not All Programmers are Created Equal*, in *Software Engineering Project Management*, R.H. Thayer, Editor. 1997, IEEE Computer Society: Los Alamitos, CA. p. 346-355.
27. Callahan, D. and B. Pedigo, *Educating Experienced IT Professionals by Addressing Industry's Needs.* IEEE Software, 2002. **19**(5): p. 57-62.
28. Carrington, D., A. Baker, and A. van der Hoek, *It's All in the Game: Teaching Software Process Concepts*, in *Proceedings of the 2005 Frontiers in Education Conference*. 2005: Indianapolis, IN. p. T1A-1 - T1A-6.
29. Carswell, L. and D.R. Benyon, *An Adventure Game Approach to Multimedia Distance Education*, in *Proceedings of the 1996 Integrating Technology into Computer Science Education Conference*. 1996: Barcelona, Spain.
30. 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.
31. Cheswick, W.R. and S.M. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*. 2nd ed. 2003: Addison-Wesley.
32. Chi, M.T.H., et al., *Eliciting Self-Explanations Improves Understanding.* Cognitive Science, 1994. **18**: p. 439-477.
33. Chua, Y.S. and C. Winton, *A Simulation Tool for Teaching CPU Design and Microprogramming Concepts*, in *Conference Proceedings on APL as a Tool of Thought*. 1989, ACM. p. 94-100.
34. Collins, A., *Cognitive Apprenticeship and Instructional Technology*, in *Educational Values and Cognitive Instruction: Implications for Reform*, L. Idol and B.F. Jones, Editors. 1991, Erlbaum: Hillsdale, NJ.
35. 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.
36. Conn, R., *Developing Software Engineers at the C-130J Software Factory.* IEEE Software, 2002. **19**(5): p. 25-29.
37. Conway, M.E., *How Do Committees Invent?* Datamation, 1968. **14**(4): p. 28-31.
38. Cook, J., *The Role of Dialogue in Computer-based Learning and Observing Learning: an Evolutionary Approach to Theory.* Journal of Interactive Media in Education, 2002. **5**.
39. Cowling, A.J., *The Crossover Project as an Introduction to Software Engineering*, in *Proceedings of the Seventeenth Conference on Software Engineering Education and Training*. 2004, IEEE: Norfolk, VA, USA. p. 12-17.
40. Crnkovic, I., R. Land, and A. Sjogren, *Is Software Engineering Training Enough for Software Engineers?* in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain.
41. Cronbach, L. and R. Snow, *Aptitudes and Instructional Methods: A Handbook for Research on Interactions*. 1977, New York, NY, USA: Irvington.
42. Curtis, B., H. Krasner, and N. Iscoe, *A Field Study of the Software Design Process for Large Systems.* Communications of the ACM, 1998. **31**(11): p. 1268-1287.
43. Dalcher, D. and M. Woodman, *Together We Stand: Group Projects for Integrating Software Engineering in the Curriculum*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain.
44. Dantas, A.R., M.O. Barros, and C.M.L. Werner, *A Simulation-Based Game for Project Management Experiential Learning*, in *Proceedings of the 2004 International Conference on Software Engineering and Knowledge Engineering*. 2004: Banff, Alberta, Canada.
45. Dawson, R., *Twenty Dirty Tricks to Train Software Engineers*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 209-218.
46. DeBono, E., *NewThink: The Use of Lateral Thinking in the Generation of New Ideas*. 1967, New York, NY, USA: Basic Books.
47. 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.
48. Emmerich, W. and V. Gruhn, *FUNSOFT Nets: A Petri-Net Based Software Process Modeling Language*, in *Proceedings of the Sixth International Workshop on Software Specification and Design*. 1991, IEEE Computer Society. p. 175-184.
49. Entertainment Software Association, *Essential Facts about the Computer and Video Game Industry*, http://www.theesa.com/archives/files/Essential%20Facts%202006.pdf.
50. Favela, J. and F. Pena-Mora, *An Experience in Collaborative Software Engineering Education.* IEEE Software, 2001. **18**(2): p. 47-53.
51. Ferrari, M., R. Taylor, and K. VanLehn, *Adapting Work Simulations for Schools.* The Journal of Educational Computing Research, 1999. **21**(1): p. 25-53.

52. Festinger, L., *A Theory of Cognitive Dissonance*. 1957, Evanston, IL: Row Peterson.
53. Flor, N.V., F.J. Lerch, and S. Hong, *A Market-driven Approach to Teaching Software Components Engineering.* Annals of Software Engineering, 1998. **6**: p. 223-251.
54. Gamble, R.F. and L.A. Davis, *A Framework for Interaction in Software Development Training.* Journal of Information and Technology Education, 2002. **1**(4).
55. Gardner, H., *Art, Mind and Brain*. 1982, New York, NY, USA: Basic Books.
56. Gee, J.P., *What Video Games Have to Teach Us About Literacy and Learning*. 2003, New York, NY, USA: Palgrave Macmillan.
57. Gehrke, M., et al., *Reporting about Industrial Strength Software Engineering Courses for Undergraduates*, in *Proceedings of the 24th International Conference on Software Engineering*. 2002, IEEE: Orlando, FL, USA. p. 395-405.
58. Glib, T., *Evolutionary Delivery versus the Waterfall Model.* ACM SIGSOFT Software Engineering Notes, 1985: p. 49-61.
59. Glib, T., *Principles of Software Engineering Management*. 1988: Addison-Wesley.
60. Gnatz, M., et al., *A Practical Approach of Teaching Software Engineering*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 120-128.
61. Godwins., Boode., and Dickenson., *National Survey of Life Stage Needs.* Medical Benefits, 1996.
62. Goold, A. and P. Horan, *Foundation Software Engineering Practices for Capstone Projects and Beyond*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY, USA. p. 140-146.
63. Groth, D.P. and E.L. Robertson, *It's All About Process: Project-Oriented Teaching of Software Engineering*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 7-17.
64. Halling, M., et al., *Teaching the Unified Process to Undergraduate Students*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY, USA. p. 148-159.
65. Harrison, J.V., *Enhancing Software Development Project Courses Via Industry Participation*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA.
66. 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.
67. Hazzan, O. and Y. Dubinsky, *Teaching a Software Development Methodology: The Case of Extreme Programming*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 176-184.
68. Hazzan, O. and J.E. Tomayko, *Reflection Processes in the Teaching and and Learning of Human Aspects of Software Engineering*, in *Proceedings of the*

*Seventeenth Conference on Software Engineering Education and Training*. 2004, IEEE: Norfolk, VA, USA. p. 32-38.
69. Hilburn, T., *PSP Metrics in Support of Software Engineering Education*, in *Proceedings of the Twelfth Conference on Software Engineering Education and Training*. 1999, IEEE: New Orleans, LA, USA. p. 135-136.
70. Hirai, K., *Micro-Process Based Software Metrics in the Training*, in *Proceedings of the Twelfth Conference on Software Engineering Education and Training*. 1999, IEEE: New Orleans, LA, USA. p. 132-134.
71. Howell, F. and R. McNab, *simjava: a Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling*, in *Proceedings of the First International Conference on Web-based Modelling and Simulation*. 1998, Society for Computer Simulation: San Diego, CA.
72. Humphrey, W.S., *Managing the Software Process*. 1990: Addison-Wesley.
73. Humphrey, W.S., *A Discipline for Software Engineering*. 1995: Addison-Wesley.
74. Humphrey, W.S., *Introducing the Personal Software Process.* Annals of Software Engineering, 1995. **1**: p. 311-25.
75. Humphrey, W.S., *TSP: Coaching Development Teams*. 2006: Addison-Wesley.
76. Inkpen, K., et al., *We Have Never Forgetful Flowers in Our Garden: Girls' Responses to Electronic Games.* Journal of Computers in Math and Science Teaching, 1994. **13**(4): p. 383-403.
77. Jaccheri, M.L. and P. Lago, *Applying Software Process Modeling and Improvement in Academic Setting*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA. p. 13-27.
78. Jain, A. and B. Boehm, *SimVBSE: Developing a Game for Value-Based Software Engineering*, in *Proceedings of the Nineteenth Conference on Software Engineering Education and Training*. 2006, IEEE: Turtle Bay, HI, USA. p. 103-111.
79. Jones, C., *Software Assessments, Benchmarks, and Best Practices*. 2000, Boston, MA: Addison-Wesley. 659.
80. Kaiser, G.E., S.S. Popovich, and I.Z. Ben-Shaul, *A Bi-level Language for Software Process Modeling*, in *Proceedings of the 15th International Conference on Software Engineering*. 1993, ACM. p. 132-143.
81. Keller, J.M. and K. Suzuki, *Use of the ARCS Motivation Model in Courseware Design*, in *Instructional Designs for Microcomputer Courseware*, D.H. Jonassen, Editor. 1988, Lawrence Erlbaum: Hillsdale, NJ, USA.
82. Kessler, R.R. and L.A. Williams, *"If This is What It's Really Like, Maybe I Better Major in English": Integrating Realism into a Sophomore Software Engineering Course*, in *Proceedings of the 1999 Frontiers in Education Conference*. 1999, IEEE: San Juan, Puerto Rico.
83. Kolb, D.A., *Experiential Learning: Experiences as the Source of Learning and Development*. 1984, Englewood Cliffs, NJ, USA: Prentice-Hall International, Inc.
84. Kornecki, A.J., *Real-Time Computing in Software Engineering Education*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 197-198.

85. Kornecki, A.J., S. Khajenoori, and D. Gluch, *On a Partnership between Software Industry and Academia*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 60-69.
86. Kornecki, A.J., J. Zalewski, and D. Eyassu, *Learning Real-Time Programming Concepts through VxWorks Lab Experiments*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 294-301.
87. Kruchten, P., *The Rational Unified Process: An Introduction (2nd Edition)*. 2000: Addison-Wesley.
88. Lakey, P., *A Hybrid Software Process Simulation Model for Project Management*, in *Proceedings of the 6th Process Simulation Modeling Workshop (ProSim 2003)*. 2003: Portland, Oregon, USA.
89. Laman, C. and V. Basili, *Iterative and Incremental Development: A Brief History*. IEEE Computer, 2003. **36**(6): p. 47-56.
90. Law, A.M. and W.D. Kelton, *Simulation Modeling and Analysis*. 3 ed. 2000: McGraw-Hill Companies, Inc.
91. Levary, R.R. and C.Y. Lin, *Modelling the Software Development Process Using an Expert Simulation System Having Fuzzy Logic*. Software -- Practice and Experience, 1991. **21**(2): p. 133-148.
92. Lindheim, R. and W. Swartout, *Forging a New Simulation Technology at the ICT*. IEEE Computer, 2001. **34**(1): p. 72-79.
93. Malone, T.W., *Heuristics for Designing Enjoyable User Interfaces: Lessons from Computer Games*, in *Human Factors in Computer Systems*. 1982: Gaithersburg, MA. p. 63-68.
94. MAPICS Inc., *AweSim*, http://www.pritsker.com/awesim.asp.
95. McGraw, G., *Software Security*. 2006: Addison-Wesley.
96. McKendree, J., *Effective Feedback Content for Tutoring Complex Skills*. Human-Computer Interaction, 1990. **5**: p. 381-413.
97. McKim, J.C. and H.J.C. Ellis, *Using a Multiple Term Project to Teach Object-Oriented Programming and Design*, in *Proceedings of the Seventeenth Conference on Software Engineering Education and Training*. 2004, IEEE: Norfolk, VA. p. 59-64.
98. 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.
99. Navarro, E.O., *A Survey of Software Engineering Educational Delivery Methods and Associated Learning Theories*, UCI-ISR-05-5, 2005, University of California, Irvine: Irvine, CA, USA.
100. Navarro, E.O. and A. van der Hoek, *Scaling Up: How Thirty-two Students Collaborated and Succeeded in Developing a Prototype Software Design Environment*, in *Proceedings of the Eighteenth Conference on Software Engineering Education and Training*. 2004, IEEE: Ottawa, Canada (to appear).
101. Navarro, E.O. and A. van der Hoek, *Design and Evaluation of an Education Software Process Simulation Environment and Associated Model*, in *Proceedings*

*of the Eighteenth Conference on Software Engineering Education and Training*. 2005, IEEE: Ottawa, Canada.
102. Noll, J. and W. Scacchi, *Specifying Process-Oriented Hypertext for Organizational Computing.* Journal of Network and Computer Applications, 2001. **24**(1): p. 39-61.
103. Nulden, U. and H. Scheepers, *Understanding and Learning about Escalation: Simulation in Action*, in *Proceedings of the 3rd Process Simulation Modeling Workshop (ProSim 2000)*. 2000: London, United Kingdom.
104. Ohlsson, L. and C. Johansson, *A Practice Driven Approach to Software Engineering Education.* IEEE Transactions on Education, 1995. **38**(3): p. 291-295.
105. Parrish, A., et al., *A Case Study Approach to Teaching Component Based Software Engineering*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 140-147.
106. Pfahl, D., et al., *Evaluating the Learning Effectiveness of Using Simulations in Software Project Management Education: Results From a Twice Replicated Experiment.* Information and Software Technology, 2004. **46**: p. 81-147.
107. Pierce, K.R., *Teaching Software Engineering Principles Using Maintenance-Based Projects*, in *Proceedings of the 10th Conference on Software Engineering Education and Training*. 1997, IEEE Computer Society: Virginia Beach, VA, USA. p. 53-60.
108. Poole, W.G., *The Softer Side of Custom Software Development: Working with the Other Players*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 14-21.
109. Postema, M., J. Miller, and M. Dick, *Including Practical Software Evolution in Software Engineering Education*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 127-135.
110. Prensky, M., *Digital Game-Based Learning*. 2001, New York, NY: McGraw-Hill.
111. Pressman, R.S., *Software Engineering -- A Practitioner's Approach*. 4 ed. 1997, New York, NY: McGraw-Hill.
112. 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.
113. Reigeluth, C.M. and C.A. Rodgers, *The Elaboration Theory of Instruction: Prescriptions for Task Analysis and Design.* NSPI Journal, 1980. **19**: p. 16-26.
114. Repenning, A., A. Ioannidou, and J. Zola, *AgentSheets: End-User Programmable Simulations.* Journal of Artificial Societies and Social Simulation, 2000. **3**(3).
115. Resnick, L., *Learning in School and Out.* Educational Researcher, 1987. **16**(9): p. 13-20.
116. Robillard, P.N., *Measuring Team Activities in a Process-Oriented Software Engineering Course*, in *Proceedings of the Eleventh Conference on Software Engineering Education and Training*. 1998, IEEE: Atlanta, GA, USA. p. 90-101.
117. Rogers, C.R., *Freedom to Learn*. 1969, Columbus, OH, USA: Merrill.
118. Rolfe, J.M., *Flight Simulation (Cambridge Aerospace Series)*. 1988, Cambridge, UK: Cambridge University Press.

119. Rost, J., *Software Engineering Theory in Practice*. IEEE Software, 2005. **22**(2): p. 96-95.
120. Royce, W., *TRW's Ada Process Model for Incremental Development of Large Software Systems*, in *Proceedings of the 12th International Conference on Software Engineering*. 1990. p. 2-11.
121. Sackman, H., W.J. Erikson, and E.E. Grant, *Exploratory Experimental Studies Comparing Online and Offline Programming Performance*. Communications of the ACM, 1968. **11**(1): p. 3-11.
122. Scacchi, W., *Process Models in Software Engineering*, in *Encyclopedia of Software Engineering*, J. Marciniak, Editor. 2001, Wiley.
123. Schank, R.C., *Virtual Learning*. 1997, New York, NY, USA: McGraw-Hill.
124. Schank, R.C. and C. Cleary, *Engines for Education*. 1995, Hillsdale, NJ, USA: Lawrence Erlbaum Associates, Inc.
125. Schlimmer, J.C., J.B. Fletcher, and L.A. Hermens, *Team-Oriented Software Practicum*. IEEE Transactions on Education, 1994. **37**(2): p. 212-220.
126. Schlimmer, J.C. and J.R. Hagemeister, *Utilizing Corporate Models in a Software Engineering Studio*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA.
127. Schön, D., *Educating the Reflective Practitioner*. 1987, San Francisco, CA, USA: Jossey-Bass.
128. 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.
129. 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.
130. Shaw, M., *Software Engineering Education: A Roadmap*, in *The Future of Software Engineering*, A. Finkelstein, Editor. 2000, ACM. p. 373-380.
131. Shukla, A. and L. Williams, *Adapting Extreme Programming for a Core Software Engineering Course*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE. p. 184-191.
132. Sindre, G., et al., *The Cross-Course Software Engineering Project at the NTNU: Four Years of Experience*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 251-258.
133. Slimick, J., *An Undergraduate Course in Software Maintenance and Enhancement*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA. p. 61-73.
134. Sommerville, I., *Software Engineering*. 6th ed. 2001: Addison-Wesley.
135. Sternberg, R.J., R.K. Wagner, and L. Okagaki, *Practical Intelligence: The Nature and Role of Tacit Knowledge in Work and at School*, in *Mechanisms of Everyday Cognition*, J.M. Puckett and H.W. Reese, Editors. 1993, Lawrence Erlbaum Associates: Hillsdale, NJ. p. 205-227.
136. Stevens, S.M., *Intelligent Interactive Video Simulation of a Code Inspection.* Communications of the ACM, 1989. **32**(7): p. 832-843.

137. Sticht, T.G., *Applications of the Audread Model to Reading Evaluation and Instruction*, in *Theory of Practice and Early Reading*, L. Resnick and P. Weaver, Editors. 1975, Erlbaum: Hillsdale, NJ.
138. Suri, D. and M.J. Sebern, *Incorporating Software Process in an Undergraduate Software Engineering Curriculum: Challenges and Rewards*, in *Proceedings of the Seventeenth Conference on Software Engineering Education and Training*. 2004, IEEE: Norfolk, VA, USA. p. 18-23.
139. Tang, J.C., *Findings from Observational Studies of Collaborative Work*, in *Readings in Groupware and Computer-Supported Cooperative Work*, R.M. Baecker, Editor. 1990, Morgan Kaufmann: San Mateo, CA. p. 251-259.
140. Tomayko, J.E., *Carnegie Mellon's Software Development Studio: a Five Year Retrospective*, in *Proceedings of the Ninth Conference on Software Engineering Education and Training*. 1996, IEEE: Daytona Beach, FL, USA. p. 119-129.
141. Tvedt, J.D., *An Extensible Model for Evaluating the Impact of Process Improvements on Software Development Cycle Time*. 1996, Ph.D. Dissertation, Arizona State University.
142. van der Veer, G. and H. van Vliet, *The Human-Computer Interface is the System; A Plea for a Poor Man's HCI Component in Software Engineering Curricula*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 276-286.
143. Wahl, N.J., *Student-Run Usability Testing*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 123-131.
144. Wake, W.C., *Extreme Programming Explored*. 2002, Boston, MA: Addison-Wesley.
145. Weller, E.F., *Lessons from Three Years of Inspection Data*. IEEE Software, 1993. **10**(5): p. 38-45.
146. Wilde, N., et al., *Some Experiences With Evolution and Process-Focused Projects*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 242-250.
147. 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.

# Appendix A: "The Fundamental Rules of Software Engineering"

1. If you don't do a system architectural design with well-defined interfaces, integration will be a big mess [134].
2. Design before coding [134].
3. If a project is late and you add more people, the project will be even later [22].
4. Team members that are new to a project are less productive (1/3 to 2/3 less) than the adequately trained people [18].
5. The average newly hired employee is about half as productive as an experienced employee [2].
6. Two factors that affect productivity are work force experience level and level of project familiarity due to learning-curve effects [2].
7. Developers' productivity varies greatly depending on their individual skills (experience concerning a development activity, knowledge of the tools, methods, and notations used, etc.) [18, 26, 121].
8. Using better and fewer people is more productive than using more less qualified people [18].
9. The greater the number of developers working on a task simultaneously, the faster that task is finished, but more overall effort is required due to the growing need for communication among developers. Thus, the productivity of the individual developer decreases [22].
10. The earlier problems are discovered, the less the overall cost will be [47].

11. The error detection effectiveness of reviews depends greatly on the qualifications and preparations of the reviewers and the completeness and correctness of the documents used as a reference [145].
12. Reviews of non-technical documents (e.g., requirements specification, user manual) are more effective if the customer is involved [111].
13. Develop tests before doing the coding [10].
14. Extreme time pressure leads to decreased productivity [47].
15. Extreme time pressure leads to a faster rate at which errors are made, which leads to a further delay in the completion date [91].
16. Error correction is most efficiently done by the document's author(s) [47].
17. The more errors a document from a previous phase contains, the more errors will be passed on to the next document [47].
18. Always test everything [134].
19. Talk to users, not to customers to verify the prototype [134].
20. Inspection is the most cost-effective measure of finding problems in software [134].
21. Software inspections find a high percentage of errors early in the development life cycle [141].
22. The use of inspections can lead to defect prevention, because developers get early feedback with respect to the types of mistakes they are making [141].
23. Every group has one programmer that is 10 times more productive than everyone else [121].
24. If you disable Internet surfing, productivity will go down [141].

25. The structure of the software reflects the structure of the organization that developed it [37].
26. Changing requirements are inevitable. Anticipating change with open architectures, adaptable designs, and flexible planning can help to mediate some of the ill effects of these changes [45].
27. Design for change/variability [45].
28. Use defensive programming [31].
29. Configuration management is good [134].
30. Successful software is designed by people who understand the application of the software (e.g., a well-designed missile control program was designed by someone who understood missiles) [72].
31. Software development requires a substantial time commitment to learning the application domain [42].
32. Broad application knowledge is acquired more through relevant experience than through training [42].
33. The more bugs you find, the more buggy the rest of your program will likely be [95].
34. Tests reveal errors in the code. The better a test is prepared for, the higher amount of detected errors [134].
35. Sticking with a too-tight schedule increases cost due to a large work force [2].
36. Motivation is increased through monetary incentives (profit sharing, pay for performance, merit pay, work measurement with incentives, and morale measurement), creating a positive frame of mind at work (employee

involvement in wellness programs and creating fun at work), encouraging a feeling of commitment and responsibility (worker participation in decision-making, getting employees to think like owners, self-managing work teams, commitment to productivity breakthroughs, and providing an environment with more freedom and less restrictions), and increasing schedule pressure (using visible milestones and setting individual goals.) Increased motivation leads to increased productivity which reduces cycle time [141].

37. Improving the work environment is done by making ergonomic considerations, giving employees enclosed offices to reduce background noise and interruptions, and giving employees access to required resources, such as computers, software tools, support staff, and information. Improving the work environment leads to increased productivity, which reduces cycle time [141].
38. Getting the most out of employees can be done by utilizing experts, employee training, skills assessment and job matching, and reducing turnover. Getting the most out of employees leads to increased productivity, which leads to decreased cycle time [141].
39. Improving the software development process can be done by formalizing the process, controlling quality, and taking advantage of tools. Improving the software process increases employees' motivation, which also increases their productivity [141].
40. Rework is usually due to customer requirements, product flaws, and communication breakdown between project members. Improving the process to reduce rework can be done by using prototyping and evolutionary development

and by using formal specification methods, modern programming practices, and inspections. Reducing rework increases productivity [141].

41. Design complexity can be reduced by using object-oriented design techniques. Reducing design complexity reduces product complexity, which increases productivity [141].
42. Code complexity can be reduced by using modularization and object-oriented programming techniques. Reducing code complexity reduces product complexity, which increases productivity [141].
43. Cognitive complexity can be reduced by modularization, multiple levels of abstraction, simulation, and prototyping. Reducing cognitive complexity reduces product complexity, which increases productivity [141].
44. Test complexity can be reduced by using testing tools, building the product with testing in mind, and testing for the type of environment a product will be used in. Reducing test complexity reduces product complexity, which increases productivity [141].
45. Management complexity can be reduced by using project management planning tools and methods. Reducing management complexity reduces product complexity, which increases productivity [141].
46. Tasks can be eliminated or simplified by using automation of tasks (e.g., code generators, automated testing) and eliminating non-value added activities and low-priority tasks. This leads to increased productivity [141].
47. Nine ways to reduce cycle time are: increase productivity, reduce rework, maximize software reuse, reduce product complexity, eliminate or simplify

tasks, maximize task concurrency, reduce undiscovered work, reduce risk, and use process models aimed at cycle time reduction [141].

48. Productivity is increased by increasing motivation, improving the work environment, getting the best people for the job, improving the process, and maximizing reuse [141].
49. Product complexity can be reduced by reducing code complexity, design complexity, cognitive complexity, test complexity, and management complexity [141].
50. Decisions made in the upstream portion of the software development process (requirements and design) impact productivity, quality, and costs throughout the life cycle more than the other portions [42].
51. The thin spread of application domain knowledge is a major phenomenon that greatly reduces software productivity and quality [42].
52. Specification mistakes often occur when designers do not have sufficient application knowledge to interpret the customer's intentions from the requirements document [42].
53. Requirements will appear to fluctuate when the development team lacks application knowledge and performs an incomplete analysis of the requirements [42].
54. Coordinating understanding of an application and its environment requires constant communication between customers and developers [42].
55. Specifications are almost always incomplete and fraught with ambiguities. Constant contact with the customer is required to obtain the correct

requirements. Without this communication, the developers tend to make incorrect assumptions about what the customer wants [45].

56. Fluctuating and conflicting requirements is a major phenomenon that greatly reduces software productivity and quality [42].
57. Communication and coordination breakdown is a major phenomenon that greatly reduces software productivity and quality [42].
58. Truly exceptional designers that are extremely familiar with the application domain, skilled at communicating their technical vision to other project members, possess an exceptional ability to map between the behavior required of the application system and the computational structures that implement the behavior, and are recognized as the "intellectual core" of the project are a scarce resource [42].
59. New requirements frequently emerge during development since they could not be identified until portions of the system had been designed or implemented [42].
60. Besides a developer's ability to design and implement programs, his skills in resolving conflicting requirements, negotiating with the customer, ensuring that the development staff shares a consistent understanding of the design, and providing communications between two contending groups are crucial to project performance [42].
61. Undiscovered work (work that was not considered in initial planning estimates) can be reduced by using formal methods, analysis of PERT sizing metrics, the

Spiral life cycle model, and prototyping. Reducing undiscovered work leads to increased productivity [141].

62. Risk can be reduced by using risk management techniques. Reducing risk leads to increased productivity [141].
63. Inspections should be thought of as part of the development process, and time must be set aside accordingly. Once this is done, inspections can have a significant improvement in the development organization's ability to meet internal schedules [141].
64. Proper use of inspections can even shorten life cycle [141].
65. Participants in the inspection team get a high degree of product knowledge, which leads to higher productivity [141].
66. Slower programmers show a great deal of improvement when using inspections [141].
67. A new project assignee does not become productive until six months to a year into the project [42].
68. Collaborators use hand gestures to uniquely communicate significant information [139].
69. Employers often limit the number of hours employees can work, resulting in further pressure to finish a project as quickly as possible [45].
70. The customer often changes deadlines to be earlier than originally agreed-upon, requiring negotiation with the customer for either allowing some deliverables to be delivered at the earlier date, with the rest being delivered later, or dropping some deliverables or requirements altogether [45].

71. Code comments and documentation are often produced at the end of a project, creating major problems when a team member is lost at short notice, leaving others to continue their work. This can be alleviated by having quality auditors require inspections at very short notice [45].
72. Teams often change during projects (members are added and/or removed.) [45].
73. Sometimes the software used for development is upgraded to a new version during development, and despite claims that it is fully backward-compatible and won't affect their work, it usually introduces new problems [45].
74. Hardware crashes, and customers are often unsympathetic to this kind of delay [45].
75. When a project is in its later stages of development, the development hardware and software tend to be under the greatest demand, and performance starts to suffer with lengthy compilations, builds, and test runs [45].
76. Matching the tasks to the skills and motivation of the people available increases productivity [18].
77. Employee motivation is the strongest influence of productivity [18].
78. Above a certain threshold, work conditions are not a powerful motivator, but below that threshold, they are a powerful de-motivator [18].
79. The training of new employees is usually done by the "old-timers," which results in a reduced level of productivity on the "old-timer's" part. Specifically, on the average, each new employee consumes in training overhead 20% of an experienced employee's time for the duration of the training or assimilation period [2].

80. The average assimilation delay, the period of time it takes for a new employee to become fully productive, is 80 days [2].
81. As schedule pressure increases, quality assurance activities (especially walk-throughs and inspections) are often relaxed or suspended altogether [2].
82. In the absence of schedule pressure, a full-time employee allocates, on average, 60% of his working hours to the project (the rest is slack time: reading mail, personal activities, non-project related company business, etc.) [2].
83. Under schedule pressure, people tend to increase their percentage of working hours spent on the project by as much as 100%, due to spending less time on off-project activities, such as personal business and non-project communication, and/or working overtime [2].
84. The three "resource-type" variables that have the greatest impact on programmer productivity are the availability of programming tools, the availability of programming practices, and programmer experience [2].
85. The two "task-type" variables that have the greatest impact on programmer productivity are the programming language and the quality of external documentation [2].
86. The average full-time employee misses 13 – 15 days of work per year (not counting vacation time). Reasons are broken down in Table A.1.

**Table A.1: Average Number of Workdays Missed Per Year (Taken from [61]).**

| Reasons for Missed Work Day | All Employees | Employees with Dependents |
|---|---|---|
| Stress | 1.1 | 1.1 |
| Personal Matters | 1.4 | 1.5 |
| Sick Child | 1.2 | 2.1 |
| Day Care Availability Issue | 0.4 | 0.8 |
| Elder Parent Care | 0.6 | 0.9 |
| Other Family Matters | 4.4 | 4.6 |
| Sick/Illness | 4.5 | 4.2 |
| Total Annual Downtime (days) | 13.6 | 15.2 |

# Appendix B: Model Builder "Tips and Tricks" Guide

This guide will provide solutions to some common problems people have run into while using the SimSE model builder. Specifically, these are problems that exist because there are phenomena that people want to model but think the model builder does not support it, when in actuality it does, but in a non-intuitive way. If you have such a problem that is not addressed here, please send an email to emilyo@ics.uci.edu. Furthermore, if you have solved such a problem yourself, or found a solution to one of these problems that is different from the ones listed here, please also let us know.

## B.1 Starting a Model

Getting started building a model can be difficult, but it helps if you first list out the specific lessons that you want your model to teach, and work on each of them incrementally, specifically thinking of how you want to penalize the player for violating the lesson and/or reward them for adhering to it. For instance, in the waterfall model, some of the lessons are:

- *Do requirements, followed by design, followed by implementation, followed by integration, followed by testing.*
- *At the end of each phase, perform quality assurance activities (e.g., reviews, inspections), followed by correction of any discovered errors*
- *If you do not create a high quality design, integration will be slower and many more integration errors will be introduced.*
- *Software inspections are more effective the earlier they are performed.*
- *The better a test is prepared for, the higher the amount of detected errors.*

- *The use of software engineering tools leads to increased productivity.*

Each of these lessons were isolated and built into the model one by one, and the model generated and tested after each lesson was added. Let us take, for example, the last lesson, "the use of software engineering tools leads to increased productivity." What we would first need to do is add some tool object types, such as a requirements tool, a design tool, and a coding tool. We would then instantiate these as start state objects. Then, in order to allow players to purchase tools, we would create a "purchase tools" action that would set each tool's "purchased" attribute to true. Then, in order to increase productivity, we would need to add some of these tools as participants to the actions of creating requirements, creating design, creating code, etc. (Specifically, we would make them optional participants (quantity = at most one), and have a trigger condition that purchased must equal true.) Following this, in order to create the effect of increasing productivity, we would need to modify one of the rules involved in these actions, namely, the one that increases the size of the artifact (e.g., requirements document) while that action (e.g., creating requirements) is active. We would modify the rule by multiplying the amount by which the size is increased by some factor that is dependent on the tool used – this could be a tool attribute called, for example, productivity increase factor.

## B.2 Finishing a Model

Finishing a model can be a task that is more time-consuming than expected. This is because much play-testing is required to ensure that the model teaches what you designed it to teach. We have found that the best way to do this is to isolate each lesson that you defined when starting to build the model (as discussed in Section B.1), and test each one separately, then collectively with others. What this entails is deliberately violating each

lesson during gameplay (do it the "wrong" way), and see if the outcome is appropriate. For instance, to test the lesson "The use of software engineering tools leads to increased productivity", play the game without using software engineering tools and see what the penalty is (probably a lower score and/or in slower development). Do this for all of the lessons and make a table that lists each approach (i.e., which lesson is violated) and the score (and optionally, any other effects perceived). Then combine some of these approaches/violations and note the additive effects to see if they are appropriate. Continue this process, making adjustments as necessary, until you are confident that all of the lessons are effectively communicated.

## B.3 Getting Around the Lack of If-Else Statements

A common programming language construct is the if-else statement. In SimSE, no such construct explicitly exists. However, a similar effect can be achieved using trigger conditions and rules. As an example, take the following simple if statement:

```
1 if (employee.sleeping == true)
2   employee.productivity = 0;
```

This same effect can be modeled in SimSE using an action, an autonomous trigger, and an effect rule. The action we must create is one that is responsible for executing the statement under the if-clause (`employee.productivity = 0;`). Let's call this "employeeSleepingModProductivity." We transfer the if-statement predicate (`employee.sleeping == true`) directly to this action's trigger—we will make an autonomous trigger with one condition: `employee.sleeping == true`. Finally, in order to execute the statement (`employee.productivity = 0;`), we will attach an effect rule to this new action that simply sets productivity of the employee to 0.

Although this is a simple example, this same technique can usually be applied to more complicated ones. However, in some more complicated cases, using this technique might result in too many different actions and rules. To describe another workaround, we use the following example that enforces the principle in the waterfall model that says you must have the requirements document at least as complete as the design document, or else productivity when working on the design will suffer (in this case, it will be half as productive). Suppose this statement is executed every clock tick during the CreateDesign action:

```
1 if (designDocument.completeness <= requirementsDocument.completeness)
2   designDocument.size = designDocument.size + (some_factor * 2);
3 else // requirements document is less complete than design
4   designDocument.size = designDocument.size + some_factor
```

Of course, because the effect rules do not support if-else statements, this cannot be directly put into an effect rule. However, we can do the following:

1. Add an attribute to the designDocument object type called "completenessDiffReqDoc" that is an integer attribute with minimum value 0 and maximum value 1. Eventually, this attribute will be 0 if the requirements document is less complete than the design document, or 1 otherwise.
2. Create the following two effect rules, attached to the CreateDesign action (these rules **must** be executed in the following order):
    a. designDocument.completenessDiffReqDoc = (((requirementsDocument.percentComplete – designDocument.percentComplete) / 100) + .001) * 100000
    b. designDocument.size = designDocument.size + (some_factor * (1 + designDocument.completenessDiffReqDoc)

What the first rule does is set the attribute's value correctly. Note that it takes adding a very small number to ensure that there is no division by 0, and multiplying by a very large number to ensure that the value will be large enough to be rounded up to 1 if that is the case. This example is taken directly from the CreateDesign action in the waterfall model.

## B.4 Modeling Error Detection Activities

A common activity in software engineering is, of course, detecting errors in an artifact, either through reviews, tests, inspections, or some other method. Although this seems like a relatively straightforward activity, it is actually non-trivial to model in SimSE.

The main idea is to take errors that are unknown to the player and make them known. This is done by subtracting errors from an artifact's (e.g.,) numUnknownErrors attribute and adding them to its numKnownErrors attribute. However, this cycle is not so straightforward in SimSE. The first thing that must be done is the artifact must be given a hidden attribute that holds a temporary value during the activity. We will call this attribute numUnknownErrorsTemp. The following three effect rules must be executed, **in the following order**, to achieve the effect of error detection:

1. artifact.numUnknownErrorsTemp = artifact.numUnknownErrors
2. artifact.numUnknownErrors = artifact.numUnknownErrors – (whatever_factor_in_your_model_affects_how_quickly_errors_are_detected)
3. artifact.numKnownErrors = artifact.numKnownErrors + (artifact.numUnknownErrorsTemp – artifact.numUnknownErrors)

What this sequence does is, in effect, take some errors from the unknown errors and adds them to the known errors. See the waterfall model's ReviewRequirements action and associated rules for a good example of this.

## B.5 Calculating and Assigning a Score

All SimSE games end by giving the player a score. Although any attribute of any object can be designated as the score, for simplicity one of the easiest things to do is to make an explicit attribute called "score" attached to your project object. Assigning a score can then be done in the following way:

1. Designate the trigger(s) or destroyer(s) you want to end the game as game-ending trigger(s)/destroyer(s) (see Section 3.1.1 of the model builder documentation).
2. Designate the attribute (e.g., "project.score") that represents the final score.
3. Attach an effect rule to the action for the trigger/destroyer in step 1, e.g., "Calculate Score." In this rule, set the project.score attribute to the correct value. The timing of this rule will depend on whether this is a game-ending trigger (in which case you would make it a trigger rule) or a game-ending destroyer (in which case you would make it a destroyer rule).

For a good example of this, see the action DeliverProduct and its associated rules in the waterfall model.

## B.6 Using Boolean Attributes in Numerical Calculations

As was seen in Section B.1 with the designDocument.completenessDiffReqDoc attribute, a Boolean attribute can be assigned a numerical value so it can be used in numerical

calculations by making it an integer attribute with a minimum value of 0 and a maximum value of 1. This attribute can then be set correctly using mathematical manipulations in an effect rule, as in the example in Section B.1.

## B.7 Revealing Hidden Information During Gameplay

There may come a time where you want to model some aspect of a project that is hidden from the player at the beginning of the game, but that can be revealed when the player takes a certain action, or under certain conditions. For example, the number of bugs in a piece of software might be hidden, but the player might have the option to discover this number via an inspection action. SimSE allows you to have hidden attributes and reveal them at the end of the game, but if you wish to reveal values mid-game, there is a trick you can use.

First, make two attributes, one that is hidden and one that is not hidden. These might be called “bugs_actual” and “bugs_known”. To start the game, the known value is set to a default value, such as 0, representing the fact that this value is not yet known. Then, when the player takes the necessary action to reveal the value, an event can be used to copy the actual value to the known value, revealing the value to the player.

An interesting offshoot of this trick is that it can be used to represent uncertainty in values, by introducing a random variance to the operation that copies the value from the hidden to the known field. For example, you might create a formula that reads bugs_known = bugs_actual - 10 + random(0,20), introducing a potentially inaccurate value to the player. The player then has a rough idea of the actual value, but cannot know exactly how the random factor has thrown off the result. This trick allows for time/accuracy tradeoffs to be considered. For example, you might create a quick

inspection option which includes a random variation, as well as a thorough inspection option which does not.

## B.8 Taming Random Periodic Events

SimSE allows for there to be a chance that a random event occurs on any given tick. Thus, if you wish for an event to happen every 100 ticks, allowing it to have a 1% chance to happen on each tick will roughly do the job. But what if you are modeling an effect and do not want to take the chance that it could occur in very rapid succession, or otherwise want to smooth out the distribution of the event?

One way is to create a counter, which has a random chance to be incremented, as well as an event that triggers when the event reaches a certain total. So, suppose you want to ensure that it is extremely statistically unlikely that customer changes occur too often, but that they tend to occur about once every 100 ticks. You could create a hidden value called "change_counter", which is initialized to 0. Then you have a random event that has a 10% chance of occurring, and which increments change_counter by 1. Finally, an event would be created which causes a customer change to occur when change_counter reaches 10. This final event would handle the customer change event and would reset the counter to 0. In this way, you can help ensure that overly frequent random events do not throw off your simulation.

## B.9 Alternative Action Theming

Only employees may perform actions, but if you are willing to force your player to bend their metaphor for the system a little bit, you can allow things that are not strictly employees to act as if they were. For example, if you have several customer stakeholders

and wish to allow the player to question each of them, you could create one employee action for questioning each stakeholder. In some cases though, it may be more appropriate to create a stakeholder employee type, place stakeholders in the SimSE environment and allow the customer to run actions "on" them. This would mean that the customer could click on the stakeholder they wished to interact with and perform an action involving them. If you think that it is important enough, and if you think it will be worth the potential confusion of calling such objects "employees", you could even create documents, tools or abstract concepts as employee subtypes and allow the player to interact through them.

This is a trick you should use at your discretion, but may help you to simulate certain types of behavior when performing an action "on" a given entity makes more sense by clicking on it, rather than selecting the option from an employee's menu.

## B.10 Making Customers "Speak"

Although in SimSE the employees are the only objects that can "speak" to the player through pop-up bubbles over their heads, sometimes it is desirable to have customers give their input as well. While it is not possible to do this directly, one way to get around this is to have the employees "say" things for the customer, in the form of a report on something the customer has done or said (e.g., "The customer says he is very unhappy right now.") In order to do this, you can simply add an employee participant to whatever action you want this trigger or destroyer text to be attached to, and then specify this text as the trigger or destroyer's overhead text. You can give the employee participant a quantity of exactly one if you want only one employee to say the message, or give it no

maximum and a minimum of one if you want all employees to say the message simultaneously.

# Appendix C: Questionnaire Used in Pilot Experiment

## C.1 Game Play Questions

1. On a scale of 1-5, how enjoyable is playing SimSE (1 least enjoyable, 5 most enjoyable)?
2. On a scale of 1-5, is it difficult or easy to play SimSE (1 most difficult, 5 easiest)?
3. Concerning the length of game play, do you think playing one game lasts too long, too short, or just right?
4. What is your most favorite part/aspect of the game? Why?
5. What is your least favorite part/aspect of the game? Why?
6. Is there anything confusing about the game? If so, what (more than one suggestion allowed)?
7. What changes would you make to improve the game?

## C.2 Software Engineering Education Questions

8. On a scale of 1-5, did playing SimSE reinforce your knowledge of the software engineering process as taught in ICS 52 (1 not at all, 5 definitely)?
9. On a scale of 1-5, did playing SimSE teach you some **new** knowledge regarding the software engineering process that you did not learn in ICS 52 (1 not at all, 5 definitely)?
10. Are there any software engineering *process* issues you feel you learned better in this game than in ICS 52? If so, which ones?

11. Are there any software engineering *process* issues you feel you learned better in ICS 52 than in this game? If so, which ones?
12. On a scale of 1-5, how helpful do you feel SimSE is to learning software engineering *process* issues (1 not at all, 5 very much so)?
13. On a scale of 1-5, how helpful to learning software engineering concepts do you think it would have been if you had been given the opportunity to *voluntarily* play SimSE while taking ICS 52 (1 not at all, 5 very much so)?
14. On a scale of 1-5, how helpful to learning software engineering concepts do you think it would have been if you had been required to play SimSE while taking ICS 52 (1 not at all, 5 very much so)?
15. On a scale of 1-5, would you recommend incorporating SimSE as a standard part of the teaching materials of ICS 52 (1 not at all, 5 very much so)?

## C.3 Background Information

16. What was your score in game 1?
17. What was your score in game 2?
18. In addition to ICS 52, did you take any other software engineering classes?
19. Have you practiced software engineering in an industrial (outside of ICS) setting? If so, for how many years?
20. Are you male or female?

# Appendix D: Questionnaire Used for In-Class Experiments

## D.1 Use of the SimSE Game

1. Did you or did you not play SimSE? If so, why? If not, why not? (If you answered "yes" to this question, proceed to the rest of the questionnaire. Otherwise, you should not answer any more questions.)

## D.2 Game Play Questions

2. On a scale of 1-5, how enjoyable is playing SimSE (1 least enjoyable, 5 most enjoyable)?
3. On a scale of 1-5, is it difficult or easy to play SimSE (1 most difficult, 5 easiest)?
4. Concerning the length of game play, do you think playing one game lasts too long, too short, or just right?
5. What is your most favorite part/aspect of the game? Why?
6. What is your least favorite part/aspect of the game? Why?
7. Is there anything confusing about the game? If so, what (more than one suggestion allowed)?
8. What changes would you make to improve the game?

## D.3 Software Engineering Education Questions

9. On a scale of 1-5, did playing SimSE reinforce your knowledge of software engineering process concepts as taught in the lectures of Informatics 43 (1 not at all, 5 definitely)?
10. On a scale of 1-5, did playing SimSE teach you some **new** knowledge regarding software engineering process concepts that you did not learn in the lectures of Informatics 43 (1 not at all, 5 definitely)?
11. Are there any software engineering *process* issues you feel you learned better in this game than in the lectures of Informatics 43? If so, which ones?
12. Are there any software engineering *process* issues you feel you learned better in Informatics 43 lectures than in this game? If so, which ones?
13. On a scale of 1-5, how helpful do you feel SimSE is to learning software engineering *process* issues (1 not at all, 5 very much so)?
14. On a scale of 1-5, how helpful to learning software engineering concepts do you think it has been to be able to have the opportunity play SimSE as an *extra-credit* assignment while taking Informatics 43 (1 not at all, 5 very much so)?
15. On a scale of 1-5, how helpful to learning software engineering concepts do you think it would have been if you had been *required* to play SimSE while taking Informatics 43 (1 not at all, 5 very much so)?
16. On a scale of 1-5, would you recommend incorporating SimSE as a standard part of the teaching materials of Informatics 43 (1 not at all, 5 very much so)?
17. On a scale of 1-5, would you recommend incorporating SimSE as a *mandatory exercise* in Informatics 43 (1 not at all, 5 very much so)?

18. On a scale of 1-5, would you recommend incorporating SimSE as an *extra-credit* exercise in Informatics 43 (1 not at all, 5 very much so)?
19. On a scale of 1-5, would you recommend incorporating SimSE as a *voluntary* exercise in Informatics 43 (1 not at all, 5 very much so)?
20. On a scale of 1-5, how well did playing SimSE help you understand the material that was taught in the lectures of Informatics 43?
21. On a scale of 1-5, how well did playing SimSE help you answer questions on the final exam?

## D.4 Background Information

22. Have you practiced software engineering in an industrial (outside of ICS) setting? If so, for how many years?
23. Are you male or female?

# Appendix E: Assigned Questions (With Answers) for In-Class Experiments

## E.1 Inspection Model Questions

1. What seems to be the ideal size of an inspection team? **4 people**
2. How long should an inspection typically last? **2 hours**
3. What is the ideal size(s) of checklist that should be used in an inspection? **1 page**
4. What is the ideal size(s) of code that should be inspected? **150 lines**
5. What are the effects of putting more as opposed to fewer people on an inspection team? **They find bugs faster but take longer to discuss, which delays them in moving on to finding more bugs.**

## E.2 Waterfall Model Questions

1. Describe in detail the process (in terms of the sequence of possible steps that you can take in the game) that this game rewards. **Create requirements; review requirements; correct requirements; create design; review design; correct design; create code; inspect code; correct code; integrate code; create, review & correct system test plan (although this step can come anywhere after requirements are done or partway done); do system test, correct code, deliver product.**
2. What is the effect of giving an employee a bonus? **A short-term increase in productivity/mood.**

3. What is the effect of giving an employee a pay raise? **A longer-term increase in productivity/mood.**
4. Is it worth it to purchase tools? **Yes.**
5. How is the outcome of the game affected if you fire Andre right at the beginning? **This is going to severely hinder the game, because Andre is probably the best all-around employee, good at requirements, design, and coding. Without him you are left with only 2 good designers, and 3 good coders, which is not enough to get a good score. Without him, you are forced to either use too few people on these tasks, or use people who aren't very good at these tasks, which will slow things down and introduce more errors.**

## E.3 Incremental Model Questions

1. Which artifact attribute seemed to be most important and most strongly affect the outcome of the game (e.g., inflexibility, difficulty, changeability, etc.)? **There are multiple possible answers here, though the most obviously correct answer is its changeability, which determines how often customer changes occur, since customer changes are so damaging. It could be argued that the difficulty was the most important, since that determines how long implementation takes, but it would be sort of missing the point. Anything well-justified should be accepted, but if they just write "inflexibility" that is insufficient for full credit.**
2. Try skipping one or more of the documentation phases (requirements/design) on one or more modules. What effect does this have? **If the requirements phase is**

**skipped, a very low accuracy rating will result, which will lower the overall score or require a great deal of redesign. If the design phase is skipped, they will have a very hard time redesigning any changes, and implementation will be slowed. Skipping both will mean an inaccurate module, impossibly hard redesign, and a terrible score, in most cases. This is, once again, a reasonably open-ended question. Most any well-justified answer that demonstrates they actually explored each approach should be accepted.**

3. How does the early submission of a partially complete project affect your work on the remainder of the project? **This will reduce changeability, and the difficulty of some tasks, for each module, especially the submitted module. In general, this means there will be less customer changes, and an easier project lifespan. Also, some of the module's hidden attributes are revealed.**

4. Describe your approach to the game in terms of the lifecycle models we discussed in class. In what ways did you follow a given lifecycle model? **Possible answers include:**

   - **Following the waterfall model by performing requirements on each module, design on each module, etc.**
   - **Following the spiral model by doing risk analysis of each module, implementing one, re-analyzing, working on more modules, etc.**
   - **Following the rapid prototyping model by quickly building one module and submitting it.**

- **Following the XP model by forgoing most documentation and implementing modules quickly, reworking them as necessary.**
- **In general, incremental approaches can be followed by early submission of modules.**

5. Is there any situation where it might be valuable to use the "start over" action?

**The most obvious case is after you have submitted a module without doing any requirements work on it, to obtain the partial submission benefits. Sometimes it is better to start over than to try to fix that module. Also, if you have done a complete requirements document but over time your module accuracy has fallen, it is often quite hard to get that accuracy back up. Starting over may be necessary. There could be other well-reasoned answers revolving around having skimped on documentation and needing the chance to do it right. Again, any that are well-justified should be accepted.**

# Appendix F: Pre-Test for Comparative Experiment

1. Describe three of the major principles behind the waterfall model of software engineering. **(specific, non-biased)**
2. Name two effective ways to increase a software engineer's motivation/productivity. **(specific, SimSE-biased)**
3. Describe three of the major principles behind iterative/incremental software development models. **(specific, non-biased)**
4. In an iterative software process, how does the early submission to the customer of a partially complete project affect your work on the remainder of the project? (**specific, SimSE-biased)**
5. In an incremental/iterative model of software development, how are increments planned (i.e., what is the criteria for determining which features/modules go into which increment)? **(specific, reading/lecture-biased)**
6. What is the purpose of a code inspection? **(specific, non-biased)**
7. What is the maximum amount of time that should be spent on a code inspection (in hours/minutes)? **(specific, non-biased)**
8. What is the ideal size of a code inspection team? **(specific, non-biased)**
9. What is the purpose of a checklist in a code inspection? **(specific, reading/lecture-biased)**
10. What is the ideal size of checklist (in number of pages) that should be used in a code inspection? **(specific, SimSE-biased)**
11. Name two strengths of the incremental life cycle model of software engineering. **(insight, non-biased)**

12. Name two strengths of the waterfall life cycle model of software engineering. **(insight, non-biased)**

13. Discuss the difference, in terms of the software life cycle, between the waterfall model and incremental/iterative models. **(insight, non-biased)**

14. Suppose you encounter a situation in which you really would like to have "the best of both worlds" by combining the incremental and waterfall life cycle models of software engineering. Draw the resulting model, and discuss the strengths and weaknesses of this particular combination. **(application, non-biased)**

15. You have just been named the chief executive officer of a newly established software company. Your first customer is NASA, who has contracted your company to build the software that will launch their newest space shuttle, which has recently been built. Currently, the entire infrastructure for launch is in place except for your launching software. Which software life cycle model will you choose to build this product, and why? **(application, non-biased)**

# Appendix G: Post-Test for Comparative Experiment

1. List the phases of the waterfall model of software development. **(specific, non-biased)**
2. Name two effective ways to increase a software engineer's motivation/productivity. **(specific, SimSE-biased)**
3. Explain the role of risk analysis in the software process. **(specific, non-biased)**
4. In an iterative software process, how does the early submission to the customer of a partially complete project affect your work on the remainder of the project? **(specific, SimSE-biased)**
5. In an incremental/iterative model of software development, how are increments planned (i.e., what is the criteria for determining which features/modules go into which increment)? **(specific, reading/lecture-biased)**
6. What is the purpose of a code inspection? **(specific, non-biased)**
7. Name and describe three of the typical steps in a code inspection process. **(specific, reading/lecture-biased)**
8. What is the maximum amount of time that should be spent on a code inspection (in hours/minutes)? **(specific, non-biased)**
9. What is the purpose of a checklist in a code inspection? **(specific, reading/lecture-biased)**
10. What is the ideal size of checklist (in number of pages) that should be used in a code inspection? **(specific, SimSE-biased)**
11. Describe the pros and cons of a software life cycle model in which increasingly complete versions of the product are delivered each week, versus a model in

which only one, fully complete product is delivered at the end. **(insight, non-biased)**

12. What is the biggest weakness of the waterfall model of software development? **(insight, reading/lecture-biased)**
13. What are the effects of putting more as opposed to fewer people on a code inspection team? **(insight, SimSE-biased)**
14. Suppose you encounter a situation in which you really would like to have "the best of both worlds" by combining the incremental and waterfall life cycle models of software engineering. Draw the resulting model, and discuss the strengths and weaknesses of this particular combination. **(application, non-biased)**
15. You have just been named the chief executive officer of a newly established software company. Your first customer is Disney, who has contracted your company to build the "coolest new kids' computer game" based on their latest animated feature film. Beyond this, they are unsure what they want the game to do or look like, but one of their top priorities is to release the game quickly, by the time the film comes out on DVD. Which software life cycle model will you choose to build this product, and why? **(application, non-biased)**

# Appendix H: Questionnaire Used for Comparative Experiment

## H.1 Learning Experience Questions

1. Which learning exercise did you participate in (SimSE, reading, or lectures)?
2. Approximately how much total time did you spend on the learning exercise?
3. Did you spend any time looking up any further information about the concepts being taught in the learning exercise? If so, why did you do this, how much time did you spend doing this, and which resources did you use to look them up?
4. On a scale of 1-5, how enjoyable was the learning exercise (1 least enjoyable, 5 most enjoyable)?
5. What were the ***most*** enjoyable aspects of the learning exercise?
6. What were the ***least*** enjoyable aspects of the learning exercise?
7. On a scale of 1 to 5, how much did the learning exercise engage your attention (1 least engaging, 5 most engaging)?
8. What were the ***most*** attention-grabbing aspects of the exercise?
9. What were the ***least*** attention-grabbing aspects of the exercise?
10. On a scale of 1-5, how effective did you feel the learning exercise was in helping you learn software process concepts (1 least effective, 5 most effective)?
11. In your opinion, which characteristics of the learning exercise were ***most*** helpful to learning software process concepts?

12. In your opinion, which characteristics of the learning exercise were ***least*** helpful to learning software process concepts?

## H.2 Background Information Questions

13. Have you practiced software engineering in an industrial (outside of ICS) setting? If so, for how many years?
14. Which software engineering classes (if any) have you taken?
15. Are you male or female?

## H.3 Lecture Group Questions

16. On a scale of 1-5, how effective did you feel the lecturer was in teaching the concepts to you (1 least effective, 5 most effective)?
17. Did you spend any time reviewing the slides outside of the lecture sessions? If so, why did you do this, and how much time did you spend?
18. If you could choose between learning software process concepts through spending **two** hours hearing lectures about the subject versus spending **two** hours reading about the subject, which would you choose and why?
19. If you could choose between learning software process concepts through spending **two** hours hearing lectures about the subject versus spending **four** hours playing a simulation game that teaches the same concepts, which would you choose and why?

## H.4 Reading Group Questions

20. Did you read any more or less than what was assigned? (Please be honest – you will get paid regardless of your answer!) If you read more, what extra did you read? If you read less, why didn't you read all that was assigned?
21. If you could choose between learning software process concepts through spending **two** hours reading about the subject versus spending **two** hours hearing lectures about the subject, which would you choose and why?
22. If you could choose between learning software process concepts through spending **two** hours reading about the subject versus spending **four** hours playing a simulation game that teaches the same concepts, which would you choose and why?

## H.5 SimSE Group Questions

23. Did you play each game less than, as much as, or more than you were instructed (in order to get a score of 85 or above)? (Please be honest – you will get paid regardless of your answer!) Why did you play less than, as much as, or more than you were instructed?
24. Would you prefer to learn software process concepts through **four** hours of playing SimSE, or through **two** hours of reading about software process concepts? Why?
25. Would you prefer to learn software process concepts through **four** hours of playing SimSE, or through **two** hours of listening to lectures about software process concepts? Why?