Engendering an Empathy for Software Engineering

Katherine Shaw and Julian Dermoudy

School of Computing
University of Tasmania
Private Bag 100, Hobart, Tasmania 7001

Julian.Dermoudy@utas.edu.au

Abstract

Students have little empathy for the fundamentals of Software Engineering practice when it is first introduced. The current method of teaching this topic involves the presentation of curriculum material through lectures. Whilst being an effective method of teaching this information, it does not provide students with enough opportunity to develop an interest in, and an understanding of, the subject.

To engage students in this area and to provide them with a deeper understanding of the issues involved in software development, an interactive, web-based, graphical simulation game of the software development process was created. This simulator allows students to take the role of the project manager developing a hypothetical software product in an environment that is both graphical and entertaining.

Keywords: Software Engineering Education, Experiential Learning, Simulation.

1   Introduction

In general, it can be said that students have little empathy for, or affinity with, the fundamentals of Software Engineering practice. This is an important issue to consider. Unless students are motivated to want to *engineer* software, it will be difficult to convince them that the methods and techniques taught can be effective, and they will not utilise them properly once they enter employment (Briggs, 1994).

The current method of exposure is *via* lectures; this does not provide students with enough opportunity to develop an interest in, nor an understanding of, the subject. Although lectures are effective methods of teaching information, they can be quite ineffective for stimulating higher-order thinking and cannot be relied upon to inspire or to change students' attitudes favourably (Biggs, 1999). The educational environment also has limitations that prevent students from experiencing the full range of problems that are encountered in the real world (Dawson, 2000).

Thus it is desirable to investigate methods to engage students in this area and to provide them with a deeper understanding of the issues involved in the software development process. We present a simulation game of two software development life cycles that allows students to gain experience of managing a software development project in an environment that is both graphical and entertaining. The simulator is completely functional and is implemented in Java.

The purpose is to investigate the effects of providing students with experiential learning of the software development process. The investigation aims to determine if the use of an educational software process game will enable students to gain a greater understanding and appreciation of the software development process, and finally, whether students will enjoy learning through the use of this tool.

To present our case, we start by introducing the context for the work: software development life cycles, experiential learning, motivation theory, discrete-event simulation, and related work. We then present the implementation of the simulator and the results of the evaluation. Finally, we conclude and offer some thoughts on future work.

2   Context

2.1   Software Development Life Cycles

The primary purposes of software development models are to define the stages comprising software development — and the order in which these stages should be undertaken — and also to establish the transition criteria that allow progression from one stage to the next (Boehm, 1988). A defined software process also provides other benefits including (Humphrey, 1995):

- enabling effective communication amongst stakeholders;
- facilitating process reuse, evolution and improvement; and
- assisting process management.

There are various software process models and life cycles available to define the software development process. In the simulator introduced here, two life cycle models will be examined: the waterfall life cycle model (which was the first formal model to be created for guiding the software process), and the spiral model (a more contemporary software process model that incorporates the ideas of prototyping and risk management). An overview of each of these will now be presented.

2.1.1 The Waterfall Life Cycle Model

The waterfall life cycle model is so named because the stages of the model are depicted as cascading from one to another — each development stage is completed before the next is commenced (Pfleeger, 1998). For example, in the waterfall model all the design work involved in the project must be undertaken in the early stages of the project, which is then followed by all of the work on coding.

The waterfall life cycle model also incorporates validation (ensuring that the software will meet customer requirements) and verification (making sure that it is functionally correct) into the software development process. The broad stages of the waterfall life cycle model are analysis, design, coding, testing, and integration.

2.1.2 The Spiral Life Cycle Model

The spiral life cycle model was developed by Boehm (1988) and combines development activities with risk management activities in an iterative process, in which each iteration resembles the waterfall life cycle model. Prior to the development of each iterative prototype, risk analysis weighs different alternatives with regard to the requirements and constraints on the project (Pfleeger, 1998).

2.2 Education — Learning, Motivation, and Simulation

2.2.1 Experiential Learning

The foundation of experiential learning lies in the concept that immediate personal experience is the focal point for learning. The theory of experiential learning defines learning as the process whereby knowledge is created through the transformation of experience (Kolb, 1984). The four stages of the experiential learning cycle are (Boud, Keogh, and Walker, 1985):

- abstract conceptualisation;
- active experimentation;
- concrete experience; and
- reflective observation.

Two aspects of this cycle are of particular importance to teaching: the emphasis on concrete and subjective experience as the heart of learning, and the premise that experiences are translated into concepts through observation and reflection. The model illustrates the need to place an emphasis on concrete experiences in education as a significant aspect of learning, and the requirement for promoting reflection to enable students to extract specific learning from the overall experience.

2.2.2 Motivation

Motivation theory states that if a person is to engage in an activity, they need to expect some valued outcome (Kolesnik, 1978). People learn more effectively, and with greater enjoyment, if learning is important to their immediate lives.

The importance of a task arises from four aspects: the value placed on the process, on the product, on what the product earns, and on what other people value. The motivations arising from the importance are: *intrinsic* motivation, *achievement* motivation, *extrinsic* motivation, and *social* motivation respectively (Biggs and Moore, 1993).

Intrinsic motivation comes from within; involvement in a task is derived from interest in the task or activity itself, rather than the outcome of the activity (Biggs and Moore, 1993). Intrinsic motivation can be created through interest, enjoyment or fun, and/or through personal consequences. Rather than influence the students to learn through competition, assessment, or collaborative learning, we are interested in stimulating the individual to learn deeply. Hence, our work attempts to increase intrinsic motivation.

2.2.3 Simulation

The primary motivation behind using simulation is that it allows experimentation with a system that would not be possible in the real world. The likely cost of implementing changes, potential consequences or even danger that could result from experimenting with a real system make simulation an attractive approach (Seila, 1995).

In modeling a system, there are two main paradigms available. Time can be represented either as a continuous variable or as a discrete variable (Seila, 1995). These two approaches are known as *continuous simulation* and *discrete-event simulation* respectively.

Discrete-event simulation models are those in which the state of a system is considered to change only due to the occurrence of events. Therefore a discrete-event simulation is one in which the system state changes at a set of discrete, and possibly random, simulated time points (Schriber and Brunner, 1998). This is appropriate for modeling the development of software artefacts.

2.3 Current Approaches to Teaching Software Development Life Cycles

2.3.1 Introduction

As noted by McCauley and Jackson (1998), the importance of an undergraduate course in the preparation of software engineers has been recognised for many years. In particular, an early and consistent emphasis on software engineering concepts creates students who value the principles and practices of Software Engineering. It is therefore important to establish how the software development process should be taught.

The software development process has a number of characteristics that are difficult to teach with traditional methods (Oh and Van der Hoek, 2001a). These include that:

- software development is non-linear — activities, tasks and phases are repeated and can occur simultaneously;

- software development involves several intermediate steps and continuous choices between multiple, viable alternatives — difficult decisions must be made, tradeoffs must be considered, and unanticipated events and conflicts must be handled;
- software development may exhibit dramatic effects with non-obvious causes — there are several common situations in which the cause is not very apparent;
- software engineering involves multiple stakeholders — decisions are made by many people; and
- software engineering often has multiple, conflicting goals — tradeoffs between aspects such as quality versus cost, timeliness versus thoroughness, or reliability versus performance (Oh and Van der Hoek, 2001a).

In the remainder of this section we consider alternative interactive mechanisms for teaching software development life cycles.

2.3.2 Role-Playing

As a teaching method, role-playing involves the use of scripts to assign specific identities in a system to students, who then interact to perform their designated roles in the system. It is a natural and effective way to introduce and expand upon many concepts in Computer Science.

In the exercise developed by Barrett (1997), a set of fourteen scripts, including those for six end users and six customers, are utilised to provide guidance for students role-playing the clients involved in the requirements gathering of a hypothetical system. It is the responsibility of the student assigned to a particular role to ensure that their system needs, guided by the scripts, are expressed during the session. Other students assume the roles of developers, such as the systems analyst. To allow the students to experience a real-life situation, the scripts have conflicting, ambiguous and incomplete requirements. The group must discuss and resolve problems to produce a reasonable set of requirements for the system. This technique has been used with success in an undergraduate Software Engineering class, both as an educational activity and an enjoyable diversion from the normal teaching routine.

Cope and Horan (1996) utilised a similar method in which the students role-played the systems analysts. The role of the client was assumed by a person outside the course in which the exercise was conducted. To enhance the realism and authenticity of the activity, a (previously completed) real project was utilised as the basis. Initially, the role-play begins with the client explaining the background of the project to all students, and students are divided into teams. Two further opportunities to question the client are presented in subsequent weeks, after which students develop a requirements document. The authors' evaluation of this utilisation of the role-played case determined that it provides students with a learning context conducive to conceptually significant learning (Cope and Horan, 1996).

Although incorporated into a series of larger group projects, Polack-Wahl (1999) also utilised students role-playing as the clients in systems development. This experience enabled the students to gain a valuable first-hand insight into the viewpoint of clients, and in particular their frustration when systems developers did not listen to their requirements.

The experiential learning involved in role-plays provides students with considerable benefits. These experiences can provide students with a deeper understanding of the processes covered (Simsarian, 2003), and seem to have a considerable and lasting effect (Andrianoff and Levine, 2002). It also allows the exploration of possibilities and alternatives that may not be available in the real world — and without the associated consequences. For example, allowing students to role play a client's role enables them to obtain first-hand insight into a client's viewpoint, which would not ordinarily be possible in the real world (Polack-Wahl, 1999).

2.3.3 Live-Thru Case Histories

Live-thru case histories are a teaching method developed by Bernstein and Klappholz (2003) specifically to enhance students' appreciation for the software development process. It was developed as a result of the failure of other methods of teaching. The authors found that discussing failed case histories and having the students read similar case studies only caused the students to recognise the oversights and mistakes of others.

One of the main advantages of live-thru case histories is that it is a very powerful experiential learning tool (Bernstein and Klappholz, 2001). Students internalise the need for the software process and their attitudes towards customer interaction and the developer's responsibility towards customers, the success of the software product and requirements engineering change dramatically (Bernstein and Klappholz, 2003).

Live-thru case histories, however, have several limitations. Regardless of the implementation chosen, live-thru case histories are extremely time consuming. Also, despite the fact that they can be scaled down to accommodate less experienced students, this method requires the development of actual software process artefacts and therefore may still be unsuitable for first year students.

2.3.4 Coursework

In addition to the inclusion of Software Engineering concepts into laboratory exercises as undertaken by Robergé and Suriano (1994), another way of integrating software process principles and practices into a curriculum is through their use in coursework tools. Morell and Middleton (2001) utilise a web-based environment to reinforce the importance of software development concepts. This tool, named *SELF* (Software Engineering Learning Facility), is made up of three main parts, one of which is the process component. This part guides students through the waterfall model of software development.

Incorporating software process activities into a competitive situation, such as assignments, also provides

a high degree of achievement motivation to students. It can also provide significant motivation to weaker students. For example, when requiring students to develop test sets as part of an assignment, Goldwasser (2002) found that even students who were struggling with their own implementation enjoyed and felt fully included in developing their own test sets.

2.3.5   Games and Simulators

Adventure games enjoy enormous popularity amongst Computer Science students, and therefore are a natural way to motivate them. Players enjoy the games' stories and interfaces, are challenged by the tasks and can also learn from gaming (Ju and Wagner, 1997). Therefore it is not surprising that they are being incorporated into Computer Science courses as learning environments.

The other main category of computer game suitable for educational purposes is simulation. Sharp and Hall (2000) simulated a software house called Open Software Solutions (OSS) to provide students with interactive software engineering case studies. The simulation environment consists of the OSS office building, with each project having one floor; the student 'joins' the company as a member of one of these project teams. Each project office contains everything the user will need to complete the project tasks: the simulated project manager (who provides guidance and feedback), items such as books and videos (which can be opened and played respectively), and other resources including prototype systems, simulations, and access to meetings (Sharp and Hall, 2000).

From informal feedback, the authors note that the environment appears to be viewed favourably by students, who find it both engaging and easy to use.

Drappa and Ludewig (2000) developed a simulation entitled *SESAM* (Software Engineering Simulation by Animated Models) to teach Software Engineering principles and practices. The student takes on the role of the project manager, and controls the simulation through a textual interface. To manage the simulated project, the player is able to hire team members, assign tasks to the team members, control the progress of the project, and utilise other management functions (Mandl-Striegnitz, 2001).

The player's aim in this simulation game is to complete the software project successfully. When the project is completed, the student is able to analyse his or her performance using an analysis tool, which displays the internal variables of the simulation in a graphical format. This enables students to understand the overall project results as a consequence of their actions and management decisions (Mandl-Striegnitz, Drappa and Lichter, 1998).

The authors report that the SESAM project has been very successful so far: analysing the improvement in the performance of students from their first and second simulations shows that most are learning from their mistakes.

The main limitation of computer simulations and adventure games is that there have to be considerable

tradeoffs between the fun and the educational value provided by the game (Oh and Van der Hoek, 2001b). It is also vital to achieve a harmony between the level of challenge offered and the level of skill required: low challenge, high skill games result in boredom, and high challenge, low skill games in anxiety (Carswell and Benyon, 1996).

Drappa and Ludewig (2000) also noted that in some cases, whilst the simulation game enhanced students' motivation it did not improve either their learning or their skills. This is likely to have been due to the fact that the simulation did not provide students with sufficient feedback to allow them to reflect on, and learn from their experiences. As with role-playing, this method also requires the active participation of students to succeed.

3   Implementation

3.1   Scope

The goal of our simulator, entitled *SimjavaSP,* is for the student, acting as the project manager, to develop a software project within the required time and budget, and of acceptable quality. This will require students to optimise the three factors of time, expenditure and quality in parallel.

In *SimjavaSP*, there is no tutor or help functions to support either project planning or decision making. This design ensures that educational objectives are achieved. As noted by Mandl-Streignitz, Drappa and Lichter (1998), the educational success of a training environment strongly depends upon the fact that the student is not guided by the system. Rather, as in real software projects, the project manager is entirely responsible for planning, staffing, directing, and controlling. In this way students are forced to manage the project on their own, and will therefore perceive the project outcomes as a direct result of their own decisions.

This allows the focus to be taken off the project deliverables and to be placed on the process. By allowing the player to manage all aspects of the project, they will be gaining valuable first-hand experience of project management without the need to develop actual project deliverables. This makes it innately suitable for first-year students.

3.2   Design

3.2.1   Functionality

Developers are the core of the software development model created for *SimjavaSP*. Like real software developers, they have individual personalities and abilities, which influence the way that they work on software development tasks.

Projects in the simulation game are modeled as being a collection of activities necessary to develop a software product. Projects define the attributes of the software product and the process by which it is produced.

Properties of software development projects that should be measured in a simulation include cost, defect level, duration, and size (Wickenberg and Davidsson, 2002).

Activities can be defined as software development tasks in which documents (or code) are produced or improved (Drappa and Ludewig, 2000). In a typical software development process, tasks that must be performed include requirements analysis, architecture development, detailed design, implementation (code and unit testing), integration, and system testing (Rus and Collofello, 1999).

There are many problems and difficult situations that students may encounter in the real world of software development. Amongst the 'dirty tricks' that Dawson (2000) recommends for use in student software project developments, the following adverse external events have been implemented in the simulation game:

- crashed hardware or software — effectively destroying the hardware or software of one developer so that more must be purchased;
- disrupted files — deleting some of the project work;
- changed requirements or new feature requests — simulating the client asking for more features after the project has commenced;
- developer quits;
- developer becomes ill — simulating a developer becoming prevented from working for a period of time; and
- schedule moved back — the deadline for the project is shortened.

Additionally, the following beneficial events have been implemented:

- relevant experience — simulating the developers having experience from working on similar projects, increasing their productivity;
- effective working environment — simulating a working environment in which developers will spend more time working and less time drinking coffee;
- case tools or reusable code — allowing a portion of the code to be generated automatically;
- increase in budget — simulating an increase in the amount of money available to develop the product; and
- extended schedule — providing the project manager with more time to complete the project.

3.2.2 Feedback

One of the limitations of previous software process simulators is that they have provided little feedback to students whilst the simulation is running. For example, an issue with SESAM noted by its developers Drappa and Ludewig (2000) is that it appears that students who fail at the project development are unable to understand the reasons why.

The interface for *SimjavaSP* has a combination of graphical and textual feedback. This makes it easier for students to assess cause-and-effect in the game, as they

receive continual feedback as the project progresses. The continual feedback includes an animated 'company office' (see Figure 1), which allows the student to observe their developers at work, and therefore allows both short and long-term cause-and-effect to be more easily illustrated in the simulation (Oh, 2002).

3.3 Choices

3.3.1 Discrete-Event Simulation

It was decided to implement *SimjavaSP* using the discrete-event paradigm for several reasons. Firstly, discrete-event simulation is able to capture details and complex interactions of systems, which makes it inherently suitable for modelling the complex process of software development. For instance, as stated above, the software development process is non-linear, and can involve random factors such as human behaviour and technological advances (Oh and Van der Hoek, 2001a). It would be extremely difficult to model such behaviour in a continuous system using differential equations, and therefore discrete-event simulation with its ability to represent individual behaviour and random occurrences is more suitable.

It is also important to include the commercial realities of software development in a simulation of the software development process. This may include problems and difficult situations such as the client changing the requirements after commencing the project, or project work being lost due to a software or hardware crash (Dawson, 2000). Stochastic events such as these are only able to be represented in a simulation that uses the discrete-event paradigm.
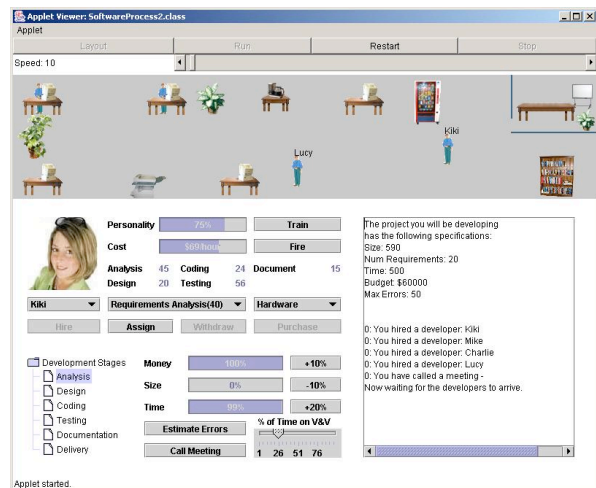

Figure 1. The Graphical User Interface of *SimjavaSP* Showing the Start of a Typical Game.

Finally, the decision to make *SimjavaSP* a simulation game implies that it will allow players to interact with the simulation whilst it is running. This leads to the conclusion that players will be influencing the operation of the simulation through their actions. These are effectively external stochastic events. Once again, the need to be able to represent discrete, 'random' events leads to the choice of discrete-event simulation to implement *SimjavaSP*.

3.3.2 Simjava

Simjava (Howell and McNab, 1998) is a process-based discrete-event simulation package for Java. It is based on the HASE++ simulation package for C++, and was developed at the University of Edinburgh.

It makes use of entities each running in their own thread to run the simulation. The entities send and receive events to communicate with one another. A central system class is responsible for controlling all the entities, advancing the simulation time and delivering events (Howell and McNab, 1998).

Simjava was selected over other evaluated simulation packages as the basis for *SimjavaSP* due to the fact that it possesses features that make it more suitable as a base for creating a game. In particular, Simjava includes facilities for representing simulation objects as animated icons, and therefore provides the basis for graphics in a game (Howell and McNab, 1998). The animation produced has the further benefit of being created in a Java applet, which can be easily integrated into a web page for use on the Internet.

4 The Students' Experience

4.1 Interactivity and Observation

In addition to an animated 'company office' that allows the player to observe their developers at work, *SimjavaSP* continuously shows the project's status. This allows both short and long-term cause-and-effect to be more easily illustrated in the simulation. For instance, a student might notice that the project is behind schedule and decide to correct this by hiring as many developers as possible. This will have the immediate effect of causing the project budget to decrease faster, as well as causing increased communication between developers trying to coordinate their development efforts. In the long term, this may in fact adversely impact the project. It is likely that the large number of developers may cause all the money to be spent too soon. Additionally the drop in productivity, due to the increase in communication overheads and introduction of developers who are new to the project, may in fact cause the project to be late.

The animated view of the developers and continuously monitored project variables provide immediate feedback to the students, which should allow them to see the cause-and-effect of their actions more easily. In a more simple example, the animated company office allows the student to see which of their employees are doing useful work, as those that are idle will spend their time away from their desks at the coffee machine.

The simulator has also been created so that the student can take a very active role as the project manager, to improve on the passive simulation created by Merrill and Collofello (1997). Throughout the project development, the student has absolute control over the developers, and can perform tasks such as hiring or firing them at any time. The student is also responsible for assigning developers to software development tasks, monitoring the project's size, budget and time, calling progress meetings, and ensuring that the project is of high enough quality.

4.2 Fun

Malone's (1980) guidelines for making instructional computer games fun have been incorporated into the implementation of *SimjavaSP*. Firstly, *SimjavaSP* achieves the quality of challenge by providing the student with the goal of developing an imaginary software project. The successful attainment of this goal is by no means certain.

Intrinsic fantasy occurs in *SimjavaSP* through the nature of the game itself: the player is engaged in the fantasy of managing the development of an imaginary software product. This fantasy is intrinsic as the problems are presented in the terms of the elements of the fantasy world: the imaginary software office and the tasks the player is asked to perform both affect each other.

Finally, in *SimjavaSP* sensory curiosity is provided through the animated 'company office'. Cognitive curiosity on the other hand is stimulated by not providing complete information to the user about certain aspects of the software project. This is limited, however, to instances where incomplete information is available to project managers in the real world, such as in trying to determine the number of errors present in a software product.

4.3 Failure

The goal of the simulator is for the student, in their role as the project manager, to develop a software project within the required time and budget, and of reasonable quality. Balancing the project drivers of budget and delivery time with product quality is one of the most difficult tasks for a project manager in the real world (Boehm, 1996 in Rus and Collofello, 1999).

This goal is enforced by ending the game when the project is 100% complete, or when the player runs out of either money or time.

The main way in which students are expected to fail is through the creation of a project that is of unacceptable quality. As the number of defects in the imaginary software product is only made available to players through a rough estimation — and even then only if such an estimation is requested — it is expected that many students will fail to detect and correct enough defects to pass the quality threshold. Secondly, the amounts of money and time provided to students have been set at a low level so that they must be balanced carefully in order for the project to succeed. The combination of these factors should ensure that most students will fail on their first attempt at developing a software product in the simulation game. It is hoped that this will increase not only their desire to 'master' the process, but also to understand its requirements and the degree to which these may be influenced.

4.4   The Employees

One of the student's tasks is to manage their development team. Each developer is implemented as an autonomous entity within the simulation.

A developer's name, their skills, personality, and cost are all set at the beginning of the simulation. These properties have been assigned so that each of the developers in the simulation has strengths in different areas. For example, one developer may have high skills in the areas of analysis and design, but have poor coding and testing abilities.

Developers start with no experience. Their initial location is that of home, their state is set to being idle — and although they are yet to be employed, their hardware and software is functional.

One of the main abilities of a developer is the ability to decide what task to perform next — from a basic choice of work, communicate, or drink coffee. Initially, a process of elimination is conducted to determine if there is the possibility that the developer should do work. If their current state predisposes activities other than work, such as if they are not assigned to a task, then the developer will simply go and drink coffee or visit the drinks machine.

Alternatively, if work is possible, a stochastic factor is calculated to determine the developer's next task. This is calculated so that the higher the number of developers working on the project, the greater chance there is that the developer will need to communicate with another developer. Outside of this possibility, the developer will either choose to work on their current software development activity, or based on their personality factor may choose to go and drink coffee rather than do work.

Regardless of the outcome of the decision, once a developer's next task has been determined, operations must exist so that they can carry out the desired activity. For this purpose, developers have several activites that require the student to interact with them. These include allowing the student to hire, fire, train, and assign the developer to software development tasks. Training a developer results in raising their skills by a small stochastic amount, at a cost of time and money to the project.

When working on their currently assigned task, the amount of time that they work is determined stochastically. The amount of work that they get done in this time, however, is determined directly by their appropriate skills and experience. For example, assuming neither developer has accumulated any experience yet, one developer with a skill level of 50% may accomplish 6 units of work in 12 hours, whilst another with a skill level of only 25% will most likely accomplish only 3 units of work. Experience has a similar effect but is valued more lowly than skills in the calculation of the quantity of work that a developer will accomplish. This leads to the effect that when a developer starts work on the project, their skills are the dominating factor, whilst later on in the project their experience has a greater influence on the amount of work they can achieve.

4.5   Managing Employees

4.5.1   Jurisdiction

Students, as the project managers, interact with many other aspects of the simulation. These include the ability to:

- hire, train, and fire developers;
- change the percentage of time that is allocated to validation and verification activities;
- call a meeting;
- assign a developer to a software development task or to estimate the errors in a project;
- withdraw a task;
- extend the completion date, reduce the size or increase the budget of the project; and
- purchase hardware or software for one of their developers.

4.5.2   Control Panels

Interaction occurs *via* control panels which occupy the lower half of the simulation screen (see Figure 4).

The project control panel of *SimjavaSP*, illustrated in Figure 2, is designed to display the relevant process and product attributes to the user. It consists of a development stages tree, progress bars for time, size and budget, general project action buttons and a slider that allows the user to allocate effort to validation and verification activities.

Figure 2. The Project Control Panel from *SimjavaSP*

As shown in Figure 3, the developer control panel allows the developers in *SimjavaSP* to be investigated and controlled by the student. The functions offered by the simulation game are intended to correspond with the actions that a project manager could take in a real software project. Therefore the interface allows the player, as the project manager, to be entirely responsible for planning, staffing, directing, and controlling.
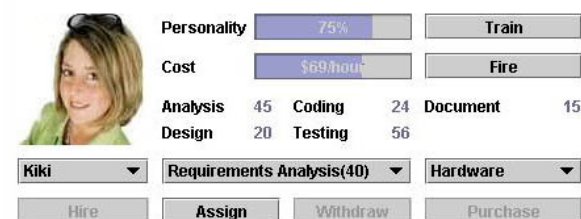

Figure 3. The Developer Control Panel from *SimjavaSP*

The developer control panel works in combination with the project control panel to allow the player access to a range of project management actions. An example of such actions is provided by Abdel-Hamid and Madnick

(1989), who note that management actions when a project is behind schedule include revising the completion date, or hiring more staff. A player in *SimjavaSP* is able to carry out both of these actions — the project control panel allows the project deadline to be extended, and use of the developer control panel allows more staff to be hired.

5 Evaluation

5.1 Investigation

In order to investigate students' opinions of the software process simulation game — as well its value as a teaching tool — an experiment was conducted in which participants played *SimjavaSP* for a period of time and then completed a twenty-question questionnaire.

The questions in the survey were devised to address each item of interest through two or three questions. The survey questions can be divided into the following categories:

- participant demographics;
- opinions on the simulator itself;
- opinions on software development life cycles;
- achievement of learning and knowledge acquired through the simulation; and
- the usefulness of the simulator as a teaching tool, particularly for the *KXA154 Software Process* unit.

5.2 Results

One of the aims of the experiment was to determine the students' opinions of *SimjavaSP*. To this purpose a series of questions were included in the survey about the simulator itself.
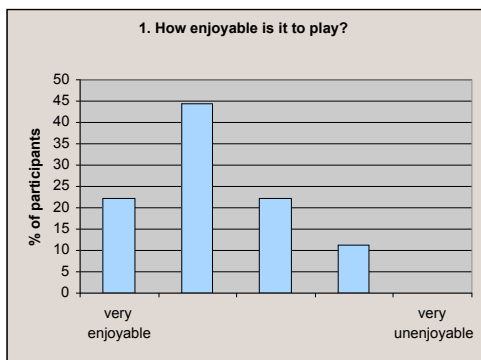

Figure 4. Graph Showing Responses to Question 1

The answers to one such question (Question 1) are summarised in the graph of Figure 4. A significant number of respondents found the simulation game enjoyable to play, with over 20% stating that it was "very enjoyable", and only 11% rating it as unenjoyable. This is a very positive indication of the game's ability to interest students, and support the researcher's observations that students for the most part seem to find *SimjavaSP* intrinsically motivating.

The second question in the survey aimed to find out how difficult or easy *SimjavaSP* is to play. As illustrated in Figure 5, a mixed response was received with most participants indicating that the game was somewhat easy to play, but with a significant amount reporting that it was difficult.
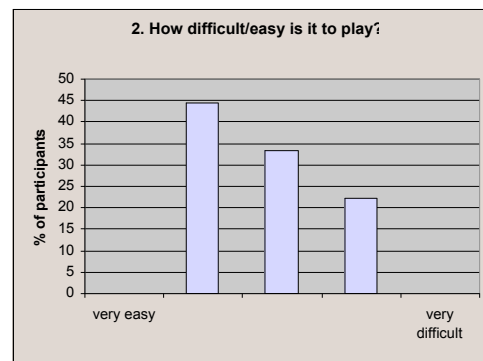

Figure 5. Graph Showing Responses to Question 2

This diverse response is most likely due to a combination of several factors. Firstly, it is probable that the participants ranged in ability and background, and therefore what is an easy game for some was difficult for others. It is also possible that a lack of clarity in the question is responsible for the variety of answers. The question may have been interpreted as:

- asking how difficult the simulator is to use (which was the intent of the question); or
- asking how difficult it is to successfully complete a project.

A third indication of *SimjavaSP*'s ability to teach was obtained by asking students for their opinions on how well the simulator teaches the process of software development. The answers are summarised in the graph of Figure 6.
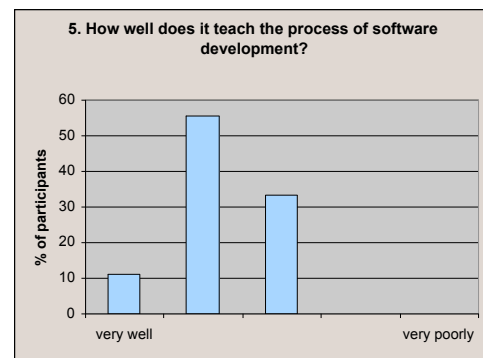

Figure 6. Graph Showing Responses Received to Question 5

Whilst 67% of students indicated that they thought that *SimjavaSP* was able to teach the software development life cycles well, other students were not as positive. Again this indicates an area in which the simulation game could be improved.

6 Conclusions and Further Work

6.1 Conclusions

There is a clear qualitative indication that students enjoy learning through playing this simulation game. Students reported that playing the game was entertaining, and

therefore it can be said to be providing them with intrinsic motivation. This finding was strengthened by one of the researcher's observations of students playing the game.

Secondly, it has been shown that students are able to benefit from the experience of playing the simulation game. Students were successfully able to identify the reasons why their simulated project had failed or succeeded, and in the case of failure determine a strategy that would allow them to avoid this in the future. Learning from experience in this way shows that the simulation is providing students with enough feedback for them to achieve experiential learning of the software development process.

The results also indicate that students are able to acquire software process knowledge from *SimjavaSP*. Its main strength in this area is in reinforcing knowledge of the software development process taught in the *KXA154 Software Process* unit. The results indicate that it is also able to teach new knowledge reasonably well.

Finally, there is also a clear qualitative indication that *SimjavaSP* is suitable as a software development process teaching tool, particularly at the introductory level. Overall, the reaction to the game as a teaching tool was very positive, with students indicating that whilst they did not think it should be a mandatory part, it should definitely be incorporated into the *KXA154 Software Process* unit as an optional part or as an alternative to one of the current tutorials. The reader is referred to (Shaw, 2003) for more information.

6.2   Further Work

The simulator could be modfied to allow now-hidden calculations (such as defect calculations) to be seen by the students. The simulation could then be used as an introduction in a tutorial, for example, in which students use *SimjavaSP* in order to observe how the number of defects present is calculated at each stage of the project, before attempting to perform the calculations themselves. Not only would this provide an interesting activity for students, it would also provide an example of the application of theory to the real world.

A second enhancement that could be made to *SimjavaSP* is the extension to include more software development process models. Furthermore, the educational value of the game could be increased by extending it to incorporate multiplayer simulation, as this could provide academic and social motivation through competitive and collaborative play respectively.

7   References

Abdel-Hamid, T. and Madnick S. (1989): Lessons learned from modeling the dynamics of software development. *Communications of the ACM*, ACM Press, **32**(12): pp. 1426–1438.

Andrianoff, S. and Levine, D. (2002): Role Playing in an Object-Oriented World. *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education,* Kentucky, USA. ACM Press, pp. 121–125.

Barrett, M. L. (1997): Simulating Requirements Gathering. *Proceedings of the 28th SIGSCE Technical Symposium on Computer Science Education*, San Jose, California, USA, pp. 310–314.

Bernstein, L. and Klappholz, D. (2001): Getting Software Engineering Into Our Guts. *CrossTalk: The Journal of Defense Software Engineering* July 2001: pp. 25–26.

Bernstein, L. and Klappholz, D. (2003): Personal communication, 8th April 2003.

Biggs, J. (1999): *Teaching for Quality Learning at University*. Buckingham, Open University Press.

Biggs, J. and Moore, P. (1993): *The Process of Learning*. New Jersey, Prentice Hall.

Boehm, B. (1988): A Spiral Model of Software Development and Enhancement. *Computer* **11**(4): pp. 61–72.

Boud, D., Keogh, R., and Walker, D. (1985): *Reflection: Turning Experience Into Learning*. London, Kogan Page.

Briggs, J. (1994): Do Students Want to Engineer Software? *Software Engineering in Higher Education*. In King, G., Brebbia, C., Ross, M., and Staples, G. (Eds.). Southampton, Computational Mechanics Publications.

Carswell, L. and Benyon, D. (1996): An Adventure Game Approach to Multimedia Distance Education. *Proceedings of the 1st Conference on Integrating Technology into Computer Science Education*, Barcelona, Spain, ACM Press, pp. 122–124.

Cope, C. and Horan, P. (1996): The Role Played Case: An Experiential Approach to Teaching Introductory Information Systems Development. *Journal of Information Systems Education On-line* 8(2): pp. 33–39.

Dawson, R. (2000): Twenty Dirty Tricks to Train Software Engineers. *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, ACM Press, pp. 209–208.

Drappa, A. and Ludewig, J. (2000): Simulation in Software Engineering Education. *Proceedings of the 22nd International Conference on Software Engineering,* Limerick, Ireland, ACM Press, pp. 199–208.

Goldwasser, M. (2002): A Gimmick to Integrate Software Testing Throughout the Curriculum. *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, Kentucky, USA, ACM Press, pp. 271–275.

Howell, F. and McNab, R. (1998): A Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling. *Proceedings of the First International Conference on Web-based Modelling and Simulation*, San Diego, USA.

Humphrey, W. (1995): *A Discipline for Software Engineering*. Massachusetts, Addison-Wesley.

Ju, E. and Wagner, C. (1997): Adventure Games: Their Structure, Principles, and Applicability for Training. *The Database for Advances in Information Systems* **28**(2): pp. 78–92.

Kolb, D. (1984): *Experiential Learning*. New Jersey, Prentice-Hall Inc.

Kolesnik, W. (1978): *Motivation*. Boston, Allyn and Bacon.

Malone, T. (1980): What makes things fun to learn? Heuristics for Designing Instructional Computer Games. *Proceedings of the 3rd ACM SIGSMALL Symposium and the First SIGPC Symposium on Small Systems*, Palo Alto, California, USA, ACM Press, pp. 162–169.

Mandl-Striegnitz, P. (2001): How to Successfully Use Software Project Simulation for Educating Software Project Managers. *Proceedings of the 31st Frontiers in Education Conference*, Nevada, USA.

Mandl-Striegnitz, P., Drappa, A., and Lichter, H. (1998): Simulating Software Projects — An Approach for Teaching Project Management. *Proceedings of the INSPIRE III: Process Improvement Through Training and Education*, London, UK, pp. 87–98.

McCauley, R., and Jackson, U. (1998): Teaching Software Engineering Early — Experiences and Results. *Proceedings of the Frontiers in Education Conference*, Arizona, USA, IEEE Computer Society, pp. 800–804.

Merrill, D., and Collofello, J. (1997): Improving Software Project Management Skills Using a Software Project Simulator. *Proceedings of the 1997 Frontiers in Education Conferece*, Pittsburgh, USA, pp. 1361–1366.

Morell, L., and Middleton, D. (2001): The Software Engineering Learning Facility. *The Journal of Computing in Small Colleges* **16**(3): pp. 299–307.

Oh, E. (2002): Teaching Software Engineering Through Simulation. *Proceedings of the International Conference on Software Engineering*, Orlando, Florida, USA.

Oh, E., and Van der Hoek, A. (2001a): Adapting Game Technology to Support Individual and Organisational Learning. *Proceedings of the 13th International Conference on Software Engineering and Knowledge Engineering*, Buenos Aires, Argentina.

Oh, E., and Van der Hoek, A. (2001b): Challenges in Using an Economic Cost Model for Software Engineering Simulation. *Proceedings of the 3rd International Workshop on Economics-Driven Software Engineering Research*, Toronto, Canada.

Pfleeger, S. (1998): *Software Engineering: Theory and Practice*. London, Prentice-Hall International.

Polack-Wahl, J. (1999): Incorporating the Client's Role in a Software Engineering Course. *Proceedings of the 30th SIGSCE Technical Symposium on Computer Science Education*, Los Angeles, USA, ACM Press, pp. 73–77.

Robergé, J., and Suriano, C. (1994): Using Laboratories to Teach Software Engineering Principles in the Introductory Computer Science Curriculum. *Proceedings of the 25th SIGSCE Symposium on Computer Science Education*, Arizona, ACM Press, pp. 106–110.

Rus, I., and Collofello, J. (1999): Software Process Simulation for Reliability Strategy Assessment. *Journal of Systems and Software*, **46**(2): pp. 173–182.

Schriber, T., and Brunner, D. (1998): Inside Discrete-Event Simulation Software: How It Works and Why It Matters. *Proceedings of the 1998 Winter Simulation Conference*, Washington DC, USA, pp. 77–86.

Seila, A. (1995): An Introduction to Simulation. *Proceedings of the 1995 Winter Simulation Conference*, December 3–6, 1995, Arlington, VA, USA, ACM, pp. 7–15.

Sharp, H., and Hall, P. (2000): An Interactive Multimedia Software House Simulation for Postgraduate Software Engineers. *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, ACM Press, pp. 688–691.

Shaw, K. (2003): *Experiential Learning of the Software Development Process through a Web-Based Simulation Game*. Honours Thesis. School of Computing, University of Tasmania, Australia.

Simsarian (2003): Take it to the Next Stage: The Roles of Role Playing in the Design Process. *Proceedings of the Conference on Human Factors and Computing Systems*, Florida, USA, ACM Press, pp. 1012–1013.

Wickenberg, T. and Davidsson, P. (2002): On Multi Agent Based Simulation of Software Development Processes. *Proceedings of the 2nd Conference on Multi-Agent Based Simulation*, Bologna, Italy, pp. 171–180.