Problems and Programmers: An Educational Software Engineering Card Game

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

Abstract

Problems and Programmers is an educational card game that we have developed to help teach software engineering. It is based on the observation that students, in a typical software engineering course, gain little practical experience in issues regarding the software process. The underlying problem is time: any course faces the practical constraint of only being able to involve students in at most a few small software development projects. Problems and Programmers overcomes this limitation by providing a simulation of the software process. In playing the game, students become aware of not only general lessons, such as the fact that they must continuously make tradeoffs among multiple potential next steps, but also specific issues such as the fact that inspections improve the quality of code but delay its delivery time. We describe game play of Problems and Programmers, discuss its underlying design, and report on the results of a small experiment in which twenty-eight students played the game.

1. Introduction

In surveying the state of software engineering education as reported in conferences [3, 13, 14] and special journal issues [1, 2, 4], it becomes clear that students gain remarkably little experience in exercising the software process. While the relevant theory certainly is taught in lectures, the practical side of most courses typically guides the students through a linear execution of the waterfall model.

The underlying problem is simple: class projects must be completed within the length of their course. Combined with the need for students to at least once exercise creating a set of project deliverables (e.g., requirements documents, design documents, test cases), little room is left to illustrate the many facets of the software process.

Industry has recognized this problem and over time has requested academia to better prepare its students for their future in the workplace [8, 9, 12, 16]. In response, many attempts have been undertaken to improve software engineering education. Some attempts involve basing an entire course on the latest software development approach, such as extreme programming [17] or PSP [5]. Other attempts involve intentionally applying a series of real-world complications during a class project, such as suddenly changing the requirements while design is in progress [10]. Unfortunately, none of these approaches addresses the core of the issue, and all suffer from being spot solutions that provide glimpses of at most a few difficulties.

As an initial investigation into the feasibility of a rather different approach, we have developed Problems and Programmers, an educational software engineering card game that simulates the software process. As a game-based simulation, Problems and Programmers provides students with an experience that is similar to a class project, but requires no deliverables to be built. It can therefore be played quickly and through repeated use illustrate many different aspects of the software process.

Problems and Programmers is built as a physical card game. Compared to computer-based simulations such as SESAM [11] and OSS [15], this has the distinct advantage of being easy and open to use. The complete game state is present on the table; the results of actions are immediately visible; and, because Problems and Programmers is set up as a competitive game, interactive games ensue in which students learn from each other.

To evaluate the educational value of Problems and Programmers, we conducted an experiment in which twenty-eight undergraduate students played the game. All of the students had already passed a sophomore-level software engineering class and, after playing, were asked to comment on the quality of the game and its value if it had been a standard part of that class. Results are encouraging: although some aspects of game play can certainly be improved, the overall reaction is that use of Problems and Programmers in a software engineering course would enhance students' understanding of the software process.

The remainder of this paper is structured as follows. In Section 2 we first discuss the main objectives of Problems and Programmers. We then discuss its design in Section 3. Section 4 introduces the details of game play. We present the results of our experimental study in Section 5, and conclude in Section 6 with an outlook at future work.

2. Objectives

The first and foremost goal of Problems and Programmers is to enrich a student's understanding of the software process. To create this higher level of understanding, the game makes careful tradeoffs among faithfulness to reality, level of detail, usability, teaching objectives, and fun factors. Our choices in these tradeoffs were guided by the following high-level objectives.

- *The game should advocate proper use of software engineering techniques*. Moreover, it should discourage taking exorbitant risks or cutting corners. For example, while it is technically possible to skip design, this should generally lead to a loss in the game.
- *The game should illustrate both general and specific lessons concerning the software process*. General lessons concern such issues as the non-linearity of the software process and the need to make decisions that may have drastic consequences. Specific lessons include, among others, Brooks' Law [6] and the fact that the use of a configuration management system normally improves the overall development process.
- *The game should provide a student with clear feedback concerning their decisions*. Consider, for example, a student who rushes the implementation phase and at a later stage in the game loses some progress. The game must provide feedback and explain that the loss is due to the fact that hasty code development introduces a higher number of bugs than as compared to normal code development.
- *Game play should be easy and comparatively quick*. Without compromising the level of complexity that is needed to make each round of game play unique, a single game must neither be too difficult nor too long in order to maintain the interest of students and, thus, the game's value as an educational tool.
- *The game should encourage interaction among students*. It is well-known that collaborative learning has significant advantages [7]. Because some lessons in Problems and Programmers are not immediately visible and must be abstracted from repeated game play, interaction typically plays a critical role in uncovering these lessons.

Finally, we note that Problems and Programmers is explicitly meant to complement existing course materials in software engineering. Instead of replacing those materials, Problems and Programmers is best used in a role that builds upon the theory presented in lectures and in turn reinforces its lessons with actual experiences in subsequent (or parallel) project assignments.

3. Overall Design

Problems and Programmers is set up as a multiplayer game in which two or more players each attempt to be the first to complete a hypothetical software project. Different players usually take different strategies as determined by their own preferences and decisions, yet calibrated for the particular cards they draw. One player, for instance, may play cautiously by establishing solid requirements, carefully designing the system structure, and testing all code immediately. Another player may be more optimistic, only partially establish all requirements up front, rush the development of their code, and completely skip testing until integration time. In this scenario, the cautious player runs the risk of losing the game due to slow progress, whereas the optimistic player runs the risk of losing the game due to delays incurred by, for example, integration problems.

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

As shown in Figure 1, game play in Problems and Programmers follows the steps of the waterfall model. We considered incorporating other life cycle models, but two reasons prevent us from doing so. First, physically visualizing different life cycles creates non-uniform card layouts that are difficult to compare and interpret. Second, the need to establish different rules for different life cycles violates our objective of game play being easy. Players would be so burdened by the various rules as they apply in different situations, that they would lose the focus on actually understanding the software process and its issues.

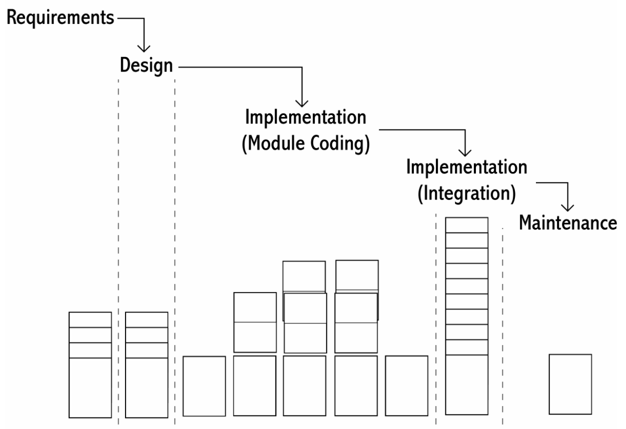

Figure 1. Different phases in Problems and Programmers.

By no means must players complete one phase of the waterfall before moving on to the next. Returning to previous phases in the life cycle is allowed (although it incurs a penalty to simulate the extra effort involved). As a result, players gain insight into two important, high-level lessons regarding the software process: it is not a linear sequence of actions and generally involves choosing among multiple viable alternatives as to what to do next.

The particular project to be completed is determined at the beginning of the game, when a single project card is drawn and placed in the middle amongst all players. As shown in Figure 2 with an example project card for a Payroll Controller system, a project card specifies a project in terms of its complexity, length, quality, and budget. The complexity and length are parameters defining the nature of the application to be developed. Complexity influences the amount of progress each player can make during the implementation phase (the higher the complexity, the less code each programmer can produce per turn) and length defines the size of the application to be developed as the number of code cards that must eventually be integrated to form the application.

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

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

Figure 2 shows three additional kinds of cards used in Problems and Programmers. First, programmer cards are used during the implementation phase. Each programmer is defined by a salary, skill, personality, and brief description. The salary defines the cost to the overall project of hiring a programmer; the sum of all programmer salaries may not exceed the project budget. The skill and personality of programmers define their behavior in terms of how productive they are in developing code and how cooperative they are within the organization, respectively. Finally, the particular values for salary, skill, and personality are explained with a brief textual description of the specific traits of the programmer.

Problem cards are at the heart of game play in Problems and Programmers. They, in fact, create many of the uncertainties that make winning the game difficult. Problem cards are drawn by one player and played upon another player. If that other player matches the condition on the problem card, they are penalized. The example problem card in Figure 2 illustrates this process. If played on a player who has more than one "unclear requirements" (e.g., a requirements card containing a problem that has not been resolved as of yet), that player must concede two code cards and one design card of their choice. Clearly, this particular problem card addresses the need to first establish unambiguous and clear requirements before code is developed. Many other problem cards exist that highlight all kinds of problems. Of note is that feedback is immediately present on the card: it explains the condition, names the problem, and describes the penalty.

Concept cards play the opposite role of problem cards: they represent benefits to a project. For example, the card shown in Figure 2 illustrates how a walkthrough helps to reduce unclear requirements or design. A few concepts will incur a cost. For example, one concept card increases the motivation of each programmer with a bonus, resulting in higher productivity. This card can only be played if its cost, $30K, can be absorbed by the project budget.

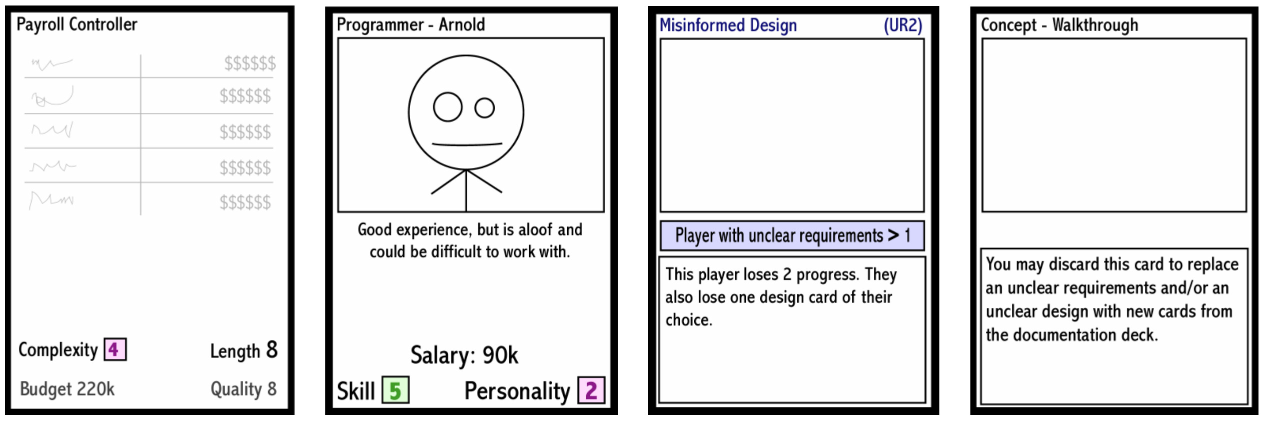
Payroll Controller
$$$$$
$$$$$
$$$$$
$$$$$
$$$$$
Complexity 4
Length 8
Budget 220k
Quality 8
Programmer - Arnold
Good experience, but is aloof and could be difficult to work with.
Salary: 90k
Skill 5
Personality 2
Misinformed Design
(UR2)
Player with unclear requirements > 1
This player loses 2 progress. They also lose one design card of their choice.
Concept - Walkthrough
You may discard this card to replace an unclear requirements and/or an unclear design with new cards from the documentation deck.

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

4. Game Play

Game play in Problems and Programmers is straightforward. Each turn, a player takes the following six steps:

1. *Decide to move to the next phase of the life cycle, the previous phase, or remain in the current phase.*
2. *Allow each opponent the chance to play one problem card on them.*
3. *Draw cards.*
4. *Take actions, as allowed in the respective phase.*
5. *Play any programmer and concept cards.*
6. *Discard any extra cards.*

Play starts in the requirements phase, and ends when one player successfully completes their acceptance test.

Throughout the game, a player continues to draw cards at each turn that may include programmer, problem, and concept cards. These cards can be put into play after the normal actions have been taken, and effectively create a dynamic simulation in which different people slowly may take the upper hand at different stages of the game.

A small number of rules exist that govern "exceptional game states" (e.g., moving backwards to a previous phase in the life cycle; replacing one programmer with another; firing programmers that do not perform very well). Typically, certain negative consequences are associated with these actions that parallel the effects in the real world. Space limits us to further discuss these rules and the interested reader is referred to the web site of Problems and Programmers: *www.problemsandprogrammers.com*. Here, we focus on the actions that normally take place in each phase of the life cycle.

4.1 Requirements

Creating a requirements document constitutes the first phase of game play. Over a number of turns, a player must create a single column of requirements cards, the length of which simulates the size of the requirements document. Based on the project's complexity and length, a player must decide when the document is "complete" for their purposes. Some players will intentionally move on to the design phase rather quickly, others will take a more cautious approach and first build up a significantly sized requirements document.

Requirements cards are either clear or unclear. The presence of unclear requirements cards is problematic: certain problem cards trigger unwanted negative progress if too many requirements cards are unclear. Fortunately, players may forgo some or all of their creation of new requirements to rework existing unclear sections: at each turn a player may choose to draw two new requirements cards, draw one new and one replacement requirements card, or draw two replacement requirements cards.

4.2 Design

The design phase is played much like the requirements phase. Players create a—separate—stack for the design document, and at each turn can choose to add new cards, rework existing cards, or mix the two activities. Again, a player has to decide (based upon experience with repeated game play for projects with different complexities and sizes) upon the desired number of cards that will constitute a "good" design document for this project.

4.3 Implementation

Implementation is the most important phase in Problems and Programmers. Per turn, each programmer that a player has hired can take four different actions: program good code (cost equal to project complexity), program rush code (cost equal to project complexity divided by two), inspect code (cost one), and fix a bug (cost one or more). Total cost cannot exceed the skill-level of a programmer. A programmer of skill-level four working on a project of complexity two could choose to program two good code, one good code and two rush code, inspect four code cards, or any other combination totaling four or less.

Code is programmed by drawing code cards from the code deck and placing the cards face down in a column above the programmer. Good code is distinguished from rush code by the card color that is put at the top (blue and red, respectively). Inspection is not mandatory, but gives a player insight into their code quality by turning cards over and revealing whether the code has bugs. Three types of bugs exist: simple, normal, and nasty. Code containing a simple bug can be replaced with another code card when a player chooses to fix the bug. Of note is that the new code card is placed face down in the stack to simulate the fact that bug fixes can contain bugs themselves.

Normal bugs are fixed by swapping them, one-by-one, with the code cards above them until they are at the top of the column and can be gotten rid of like a simple bug. This process simulates that other code often must be modified as well if it builds on buggy code and emphasizes the importance of early bug detection and resolution.

Finally, nasty bugs are what they are called: they wipe out all progress above it in the column, and a player has to redevelop that code.

Inspecting code clearly has the benefit of uncovering bugs early in the process. Just like in the real world, however, improving code quality bears the cost of increased time-pressures that exist to get coding completed.

Note that when a player has reached the implementation phase, problem cards may be played upon that player. Some players may hold off on playing their problem cards until later in the process to wreak even more havoc (mimicking the fact that later changes typically are costlier).

4.4 Integration testing

During the integration phase, a player can take all of the code developed by a particular programmer and move it into the integrated code column. Code that is known to be buggy after inspection and code that has not yet been inspected can both be integrated, but doing so exposes the player to the risks described in the next section. Code is integrated programmer by programmer, illustrating trade-offs between having more programmers, which reduces the time needed to build the overall system, and fewer programmers, which makes integration quicker.

Note that once code has been integrated, problem cards no longer have an effect on it. Note also that a player does not have to wait until all programmers have finished coding in order to start the integration process.

4.5 Maintenance / acceptance testing

This is the final phase of the game and can only be entered after the length of the integrated code column equals or exceeds the length of the project. At that point, a player has to decide whether to continue integrating code or attempt to win the game. In the latter case, all integrated code is placed face down, shuffled, and a number of cards equaling the project quality (as stated on the project card) are pulled from the top of the pile. If no bugs appear, the player has won the game. If any bugs appear, however, the penalty is severe: all code cards must be placed above a single programmer of choice with the bugs that were discovered at the bottom of the column. Play continues: the bugs must be fixed, additional code can be inspected, and the player is once again vulnerable to problem cards.

5. Experiment

To evaluate Problems and Programmers, we conducted an experiment in which twenty-eight undergraduate students played the game. Teamed in pairs, each student was first introduced to the game, its objectives, and its mechanics of game play (30 minutes), then played twice against the other person (45 minutes per game), and was asked to fill out a questionnaire regarding their particular experiences (10 minutes). Our primary objective was not to observe or analyze the students' play, but rather to simply let them play and report on whether they considered the game helpful in learning about the software process.

To promote objectivity, the experiment was carried out independent of any particular software engineering course. Furthermore, while all students had passed the same software engineering course as background material (ICS 52, Introduction to Software Engineering), they did so in different quarters with different instructors. Finally, we randomly chose subjects from a full pool of applicants.

The questionnaire consisted of questions regarding the experience of playing the game and potential for the game to be used as a teaching aid. Two kinds of questions were asked: numerical score questions and open-ended questions. Table 1 presents the results for the numerical score questions, with 1 indicating a strongly negative answer and 5 indicating a strongly positive answer. Overall, the results are very encouraging. With the exception of question 4, which had an average score of only 2.6, all other questions indicate positive feedback with average values ranging from 3.3 (question 8) to 4.1 (question 1). Perhaps most indicative is the 3.7 average score for question 6, which is a clear vote of confidence in the game and its merits by the students who participated in the experiment.

Table 1. Questionnaire numerical scores.

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

Also informative were the answers given to the open-ended questions. Designed to present us with feedback to assist in further improving the game, questions focused on favorite and least-favorite parts, clarity, problematic parts, lessons learned, suggestions for improvement, and overall impressions. Again, results are encouraging: *"I learn that even though doing requirements and design take a lot of time, but if we don't do well on these parts of the project might actually take longer to finish!"*, *"It reinforces the ideas taught in ICS 52 with a fun way to learn it."*, *"The game illustrates in 1 hour or 2 almost half of the material in ICS 52."*, *"I think I learned that having many programmers = more time in integrations."*, and *"Tells me WHY it is important to create quality code."* Most importantly, a significant number of respondents considered the competition aspects of the game enjoyable and helpful.

At the same time, the responses brought to light some issues with the game, e.g.: *"Requirements and design are boring."*, *"Add different life cycles."*, *"It was unclear the amount of time should be spent on requirements and design."* The majority of these comments were focused on the early phases of the game and we will take these and the other comments provided and examine whether we can improve the game accordingly. In particular, we intend to make the requirements and design phases more varied in order to introduce both more alternative strategies and more depth into the game.

6. Conclusions

Problems and Programmers represents a first attempt at using a physical card game for educating students in the many difficult issues inherent to the software process. While it certainly is not designed to teach the "right" answer to each particular situation (such an endeavor is impossible), repeatedly playing Problems and Programmers prepares a student for the breadth of issues they may face in their future careers, reinforces in a practical manner many of the theoretical lessons presented in lectures, and builds a student's understanding of their actions and role in the overall software development process.

Compared to existing automated simulations, our card-based simulation has the advantage of being highly visual, being easy and fun to play, engaging students in collaborative learning, and providing almost instantaneous feedback on the actions that students take. While we had to make some compromises to achieve this level of functionality and therefore were unable to include as many aspects of the software process as we would have liked, we believe the current incarnation of the game achieves a good balance among the objectives stated in Section 2.

The results of our experiments confirm our intuitions: students largely believe that use of the game in their software engineering class would have helped them in their studies. Our research, however, does not end here. First, we plan to actually use Problems and Programmers in our introductory software engineering class. Using an improved version based on the feedback received, we hope to further build our understanding of how the game helps in teaching the software process. Planned collaborations with three other institutions will provide us a broad sample size in this process. Second, we will develop an automated version of the game. Although losing the physical aspects of the game, automation facilitates easy distribution and allows us to build a version of the game that incorporates additional software engineering principles and life cycle models—the advantage being that the program, and not the player, enforces the rules of game play.

Acknowledgments

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

This research is supported by the National Science Foundation under Grant Number CCR-0093489. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement numbers F30602-00-2-0599 and F30602-00-2-0607. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

References

1. *IEEE Software, Special Issue on Educating Software Professionals*. Vol. 19 (5), September/October. 2002: IEEE.
2. *The Journal of Systems and Software*. Vol. 49. 1999: Elsevier Science Inc.
3. *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002: IEEE.
4. Balci, O., *Annals of Software Engineering*. Vol. 6. 1998: Baltzer Science Publishers.
5. Borstler, J., et al., *Teaching PSP: Challenges and Lessons Learned.* IEEE Software, 2002. **19**(5): p. 42-48.
6. Brooks, F.P., *The Mythical Man-Month: Essays on Software Engineering*. 2 ed. 1995: Addison-Wesley. 336.
7. Bruffee, K.A., *Collaborative Learning: Higher Education, Interdependence, and the Authority of Knowledge*. 1983: John Hopkins University Press.
8. Callahan, D. and B. Pedigo, *Educating Experienced IT Professionals by Addressing Industry's Needs.* IEEE Software, 2002. **19**(5): p. 57-62.
9. Conn, R., *Developing Software Engineers at the C-130J Software Factory.* IEEE Software, 2002. **19**(5): p. 25-29.
10. Dawson, R., *Twenty Dirty Tricks to Train Software Engineers*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 209-218.
11. 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.
12. 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.
13. Mengel, S.A. and P.J. Knoke, *Proceedings of the Thirteenth Conference on Software Engineering Education & Training*. 2000: IEEE Computer Society.
14. Ramsey, D., P. Bourque, and R. Dupuis, *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001: IEEE Computer Society.
15. 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.
16. Shaw, M., *Software Engineering Education: A Roadmap*, in *The Future of Software Engineering*, A. Finkelstein, Editor. 2000, ACM. p. 373-380.
17. 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.