HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering

JARI VANHANEN

# Effects of Pair Programming at the Development Team Level: An Experiment

Licentiate thesis submitted for official examination for the degree of Licentiate in Technology.

Espoo, 27.05.2005

Instructor: Casper Lassenius

Supervisor: Casper Lassenius

| | |
|---|---|
| **HELSINKI UNIVERSITY OF TECHNOLOGY**<br>Department of Computer Science and Engineering | **ABSTRACT OF LICENTIATE THESIS** |
| Author<br>Jari Vanhanen | Date<br>27.05.2005<br>Pages<br>48 |
| Title of thesis<br>Effects of Pair Programming at the Development Team Level: An Experiment | |
| Professorship<br>Software Engineering | Professorship Code<br>T-76 |
| Supervisor<br>Casper Lassenius | |
| Instructor<br>Casper Lassenius | |

Pair programming is an intensive form of programmer collaboration where two programmers design, code and test software collaboratively at one computer. Pair programming has been proposed to provide several benefits for software development. This work studied the effects of pair programming for productivity, defects counts, design quality, knowledge transfer, enjoyment of work, and effort estimation accuracy. The research consisted of an experiment with five four-person teams each doing a 400-hour project based on the same specifications, technologies and process guidelines. The only difference between the teams was that three of the teams used pair programming for all development work and the other teams used solo programming.

In the experiment pair programming increased the development effort of the first few use cases, but after that there were almost no differences in the use case efforts compared to solo programming. Surprisingly, higher use case complexity did not favor pair programming. Due to the less productive learning time the pair programming teams had worse overall project productivity. However, in an industrial context the causes for the learning time are usually avoided, because developers already know each other and after the first pair programming project are already familiar with the practice. Even if there still were learning time involved, its effects become negligible in a large project.

The pair programming teams had less pre-delivery but more post-delivery defects. Their attitude towards system testing might have been less careful due to an over-reliance on the peer review taking place during coding. Our efforts at contrasting the resulting design quality were inconclusive.

Pair programming improved both the breadth and depth of knowledge transfer. In the pair programming teams each code package was understood well by more developers than in the solo programming teams. On the average developers in the pair programming teams also understood well more packages than developers in the solo programming teams.

About half of the developers in the pair programming teams favored solo programming over pair programming, but still most developers liked working in the pair programming teams. Thus developers' feelings toward pair programming should not hinder deploying pair programming.

The pair programming teams were slightly better in estimating efforts required for implementing externally specified use cases early in the project. However, when updating the effort estimate just before the implementation of a use case, solo programmers were accurate more often than pairs.

These results certainly shed some more light on the topic, even though this experiment, like all the previous ones, contained many deficiencies such as the small sample size. Based on the results, it seems that the use of pair programming leads to fewer defects in code after coding and better knowledge transfer within the development team without requiring additional effort if the learning time can be avoided. These benefits are likely to decrease the further development costs of the system and increase an organization's productivity due to improved competence of the developers.

Keywords:
pair programming, programming practices, software engineering experiment

| | |
|---|---|
| **TEKNILLINEN KORKEAKOULU**<br>Tietotekniikan osasto | **LISENSIAATINTUTKIMUKSEN TIIVISTELMÄ** |
| Tekijä<br>Jari Vanhanen | Päiväys<br>27.05.2005<br>Sivumäärä<br>48 |
| Työn nimi<br>Koe pariohjelmoinnin vaikutuksista tiimikontekstissa | |
| Professuuri<br>Ohjelmistotuotanto | Professuurin koodi<br>T-76 |
| Työn valvoja<br>Casper Lassenius | |
| Työn ohjaaja<br>Casper Lassenius | |

Pariohjelmointi on ohjelmoijien välistä intensiivistä yhteistyötä, jossa kaksi ohjelmoijaa suunnittelevat, koodaavat ja testaavat ohjelmistoa yhdessä yhden tietokoneen ääressä. Pariohjelmoinnin on esitetty tarjoavan useita hyötyjä ohjelmistokehitykselle. Tämä työ tutki pariohjelmoinnin vaikutuksia tuottavuuteen, virhemääriin, koodin rakenteen laatuun, tiedon siirtymiseen, työtyytyväisyyteen ja työmäärien arviointitarkkuuteen. Tutkimus koostui kokeesta, jossa viisi neljän hengen ryhmää tekivät kukin 400 tunnin projektin perustuen samoihin määrittelyihin, teknologioihin ja prosessiohjeisiin. Ainoa ero asetelmassa ryhmien välillä oli se, että kolme ryhmistä käytti pariohjelmointia kaikkeen kehitystyöhön, kun taas muut ryhmät käyttivät yksinohjelmointia.

Kokeessa pariohjelmointi suurensi kehitykseen käytettyä työmäärää muutamien ensimmäisinä toteutettujen käyttötapauksien kohdalla, mutta sen jälkeen erot käyttötapauksiin käytetyissä työmäärissä verrattuna yksinohjelmointiin olivat mitättömiä. Yllättäen pariohjelmoinnin vaikutukset työmääriin eivät olleet suotuisampia monimutkaisempien käyttötapauksien kohdalla. Pariohjelmoinnin vaatiman tuottavuudeltaan tehottomamman opetteluajan takia pariohjelmointiryhmien projektitason tuottavuus oli huonompi. Kuitenkin teollisessa ympäristössä opetteluaika tavallisesti vältetään, koska kehittäjät tuntevat toisensa ja viimeistään ensimmäisen pariohjelmointiprojektin jälkeen osaavat myös pariohjelmointia. Vaikka opetteluaikaa ei voitaisiinkaan välttää, niin isossa projektissa sen vaikutukset tuottavuuteen ovat olemattomat.

Verrattuna yksinohjelmointiryhmiin pariohjelmointiryhmien ohjelmissa oli vähemmän virheitä ennen toimitusta, mutta toimituksen jälkeen jäljelle jääneiden virheiden määrä oli suurempi. Pariohjelmointiryhmät mahdollisesti luottivat liikaa koodauksen aikana parin toimesta tapahtuvaan koodin katselmointiin ja olivat siksi järjestelmätestauksessaan huolimattomampia kuin yksinohjelmointiryhmät. Mahdollisiin eroihin designin laadussa eivät tuloksemme anna selkeää vastausta.

Noin puolet kehittäjistä pariohjelmointiryhmissä sanoivat pitävänsä enemmän yksinohjelmoinnista kuin pariohjelmoinnista, mutta kuitenkin useimmat kehittäjät pitivät työskentelystä pariohjelmointiryhmissä. Siten kehittäjien tuntemuksien pariohjelmoinnista ei pitäisi olla pariohjelmoinnin käyttöönottoa estävä tekijä.

Pariohjelmointiryhmät olivat hieman parempia arvioimaan ulkopuolisten kuvaamien käyttötapauksien vaatimia työmääriä projektin alussa. Kuitenkin, kun verrattiin juuri ennen toteutusta toteuttajan toimesta tehtyjä päivitettyjä työmääräarvioita, yksinohjelmoijat olivat tarkkoja arvioissaan useammin kuin parit.

Nämä tulokset varmasti valaisevat aihetta hieman lisää, vaikka tämä koe kaikkien aikaisempien kokeiden tapaan sisälsi monia heikkouksia kuten otoksen pieni koko. Tulosten perusteella vaikuttaa siltä, että pariohjelmoinnin käyttö vähentää koodauksen jälkeen koodissa olevia virheitä ja lisää tiedon siirtymistä ryhmän sisällä vaatimatta enempää työtä, jos opetteluun kuluva aika saadaan vältettyä. Nämä hyödyt saattavat vähentää jatkokehityskustannuksia ja lisätä organisaation tuottavuutta kehittäjien parantuneen ymmärryksen myötä.

Avainsanat:
pariohjelmointi, ohjelmointikäytännöt, ohjelmistotuotannon koe

# Acknowledgements

This research was done at the Software Business and Engineering Institute (SoberIT) at the Helsinki University of Technology during the SEMS and SHAPE research projects.

Many people participated in the planning, preparation and execution of the experiment reported in this thesis. Timo Jalonen and Matti Kokkola did great work in the preparations for the course, where the experiment took place. They defined the requirements for the assignment project, developed the core architecture, and gave lectures about J2EE. Mikko Rusama familiarized himself with several development tools needed in the project and lectured about them. Several discussions about pair programming experiments with Asko Seeba and Priit Salumaa at the XP2003 conference were valuable for the initial planning of this experiment. Finally, the experiment could not have been conducted without the students, who participated in the course T-76.633 in the spring of 2004. I'm very grateful to all these people for their contribution.

I also want to thank professor Casper Lassenius and all the other members of the Software Process Research Group at SoberIT for their support during this work. Especially Kristian Rautiainen gave important feedback through reviewing certain parts of the work in progress.

Finally, I want to thank my wife Heidi for the constant love and support she has given me.

Espoo 27.5.2005

Jari Vanhanen

# Contents

# 1 Introduction

## 1.1 Motivation

Pair programming is an intensive form of programmer collaboration where two programmers design, code and test software collaboratively at one computer (Williams and Kessler 2002). Lately it has received lots of publicity being one of the practices of Extreme Programming (XP) (Beck 2000). However, pair programming has been known for a long time before XP, and is by no means limited to be used with XP only.

Based on previous studies which compared pairs with solo programmers (Williams 2000, Nosek 1998), it seems that pairs produce better design with fewer defects in the code, in shorter elapsed time and more enjoyably without using significantly more effort. Benefits for teamwork, knowledge transfer and learning have also been proposed (Cockburn and Williams 2000). More effective learning of the development tools and domain most likely leads to higher productivity, as does higher enjoyment of work. Improved knowledge transfer of the developed system is a way to decrease the risk of having critical persons in the team. It also accelerates new developers in becoming productive in their work. If the increase in the development effort when using pair programming is low or nonexistent, the realization of just some of the proposed advantages would make the overall effect of pair programming positive from the project's and organization's point of view.

However, the results of the previous studies are quite varying and even contradictory in some aspects. The experimental designs have varied between different studies, they have not been described accurately, and they have contained certain deficiencies. Most of the studies have concentrated on studying individuals and pairs in isolation, even though in industry software is typically developed by a team of developers. Due to all these reasons, it is difficult to assess the validity of the results, generalize them to the industrial context or replicate the experiments elsewhere. In order to make the research more reliable and replicable, it is important to make more experiments using improved and more explicitly described experimental designs.

## 1.2 Terminology

*Pair programming* means that two persons work simultaneously on analyzing, designing, coding or testing the same software development task sitting at the same computer continuously collaborating with each other. The person typing at the keyboard or writing down the design is called the driver. The other person is called the navigator, who should continuously look for defects in the solution and on a higher level make sure that the pair is heading to the right direction. These roles should be switched between the developers during the work of a task (Williams and Kessler 2000 p. 3-4, Beck 2000 p. 50-51)

*Solo programming* means in this work programming mostly alone, but being allowed communicate with and ask help from other team members during the development work. This should mean the most typical way of developing software in industry nowadays. Term *individual programming* (Nawrocki and Wojciechowski 2001, Gallis et al. 2003) has also been used in the literature to distinguish the normal way of programming from pair programming.

Concepts related to time often cause confusion when discussing about pair programming. In this work term *elapsed time* is used to denote stop-watch time when a pair or an individual finished a task. Term *effort* is used to denote the total effort related to the work.

For an individual the effort is equal to the elapsed time, but for a pair effort is twice the elapsed time.

## 1.3 Objectives and scope of the research

The research problem of this work is to understand:

> How does the use of pair programming affect product quality, project productivity, knowledge transfer, enjoyment of work, and effort estimation accuracy when small, co-located teams develop software?

Defining unambiguously the desired amount of collaboration within a co-located team is difficult. Therefore this research focuses only on comparing the effects of two quite easily definable, but realistic alternatives. The first is pair programming (PP) applied as described by Williams and Kessler (2002), and used for all development tasks and activities (analysis, design, coding, unit testing) as recommended in extreme programming (Beck 2000). The second is solo programming (SP) as defined in the previous section. Other alternatives would be, e.g., paired developers working with the same task using two computers or developers working alone without any daily communication with other team members.

The research questions of this work aim to answer how pair programming used in a team context affects the following project attributes:

1. project productivity
2. use case implementation effort
3. the number of defects
4. design quality
5. knowledge transfer within the team
6. the enjoyment of work
7. effort estimation accuracy

The first research objective is to analyze all previously made pair programming experiments in order to identify relevant hypotheses related to the benefits of pair programming, but also to identify weaknesses in the experimental designs. The second objective is to execute a well-planned experiment where pair programming is used in a more realistic and industry like context than in the previous experiments and based on the experiment answer the stated research questions.

## 1.4 Structure of the thesis

The thesis consists of six chapters: Introduction, Literature research, Research design, Results and discussion, Evaluation of the experiment, and Conclusions. Chapter 2 summarizes the relevant pair programming literature and focuses especially on analyzing the previous experiments. Chapter 3 presents the research questions, related hypotheses and the experimental design. Chapter 4 contains the results of the experiment and discussion of the results. Chapter 5 evaluates the threats to the validity of the results and chapter 6 contains conclusions and discussion on future work.

# 2 Literature study

This chapter summarizes pair programming literature to the extent that is relevant for this research.

## 2.1 Scope

The first goal of this literature study was to understand what pair programming is and why it may be a beneficial practice. The second goal was to analyze all previous pair programming experiments in order to learn from their research designs for my experiment and to compare my results to the other studies. The last goal was to find tips on how pair programming should be done in practice in order to do pair programming in the most efficient way in my experiment.

The material for the literature study was collected through searches to ACM Digital Library and IEEE Xplore using keywords "pair programming" and "collaborative programming". The papers included for the study were filtered based on their title and abstract. Relevant references found from the included papers were handled recursively with the same process. The amount of papers strictly discussing pair programming or collaborative programming was moderate and they were all read through.

Summaries of the previous pair programming studies have already been made by Gallis et al. (2003) and by Boehm and Turner (2003). The book by Williams and Kessler (2002) covers most areas around pair programming, but even though it contains lots of references to their own and others' studies to back up its claims, it is evident that there are still many topics that are yet lacking scientific studies.

## 2.2 History of pair programming

Pair programming is a very tight form of collaboration between two people developing software. The other end is a solo programmer working alone in her own room. In the middle ground are partner programmers (Gallis et al. 2003) working with their own tasks with their own computers but in the same physical workspace allowing them to easily ask help from each other.

Even though software development has been seen as the work of individuals, some degree of collaboration between developers is a normal way of working and it has certainly been performed since the early days of software development. A member of a software development team most certainly asks help to a problem with a development task from a colleague, at least after struggling with the task long enough. Some tasks such as software design and specifying interfaces between modules developed by different people are quite naturally made by two or more developers together. Actually, De Marco and Lister (1999) refer to an old study claiming that software developers spend only 30% of their time working alone.

Due to the naturalness of collaboration between developers naming a certain point of time as the beginning of pair programming is difficult. Probably the most striking characteristic separating pair programming from a casual collaboration is the requirement for working together at the same computer for long periods of time, e.g., for designing, programming and testing a whole software feature. It is something that probably does not just emerge without active encouragement of such collaboration.

The first statement in the literature about pair programming kind of collaboration is probably Larry Constantine's observation during his visit to Whitesmith Ltd. in the late 1970s[1]. He saw developers sitting paired, two at each terminal. The pairs were discussing about programming related topics and frequently switching places for allowing the other partner to sit at the keyboard. Constantine called these pairs dynamic duos. (Constantine 1995) Randall W. Jensen recalls an experiment, which he made in 1975 in a large software organization. Based on his good experiences from using teamwork during his undergraduate electrical engineering studies, he executed an experiment in the organization in order to find ways to improve programmer productivity. In the experiment a 50 000 LOC system was developed by a 10-person team split into five pairs who worked at the same terminal. Roles in the pairs were changed daily. (Jensen 2003) In 1993 Wilson et al. made an experiment with students comparing individuals and pairs writing a small program (Wilson et al. 1993). John Nosek, who participated in the previous study, made later a similar study in industrial environment (Nosek 1998). James Coplien published in 1995 the "Developing in Pairs" pattern, reasoning that two persons can solve problems better than an individual. An extended description of the pattern has been published in a later book (Coplien and Harrison 2004).

The rise of pair programming to the knowledge of the larger audience happened together with the hype around extreme programming (XP) (Beck, 2000) since 2000. Pair programming is one of the twelve practices of XP. XP dictates using pair programming for developing all code to be released in an XP project. In the academic front Williams made in 1999 a larger pair programming experiment than that of Nosek's. Her doctoral dissertation (Williams 2000) and several papers discussed the results of the experiment (Williams et al. 2000, Cockburn and Williams 2000) and pair programming in general (Williams and Kessler 2000). Both the increased publicity of XP and Williams's positive results of the benefits of pair programming were probably the reasons for the increase of the deployment and research about pair programming. In 2001-2002 Cusumano et al. (2003) made a global survey in several companies charting which practices are used in software projects. They reported that pair programming was used by 35.3% of the projects in their sample.

Critique against pair programming has also appeared. XP as a controversial development methodology with its fanatic supporters has aroused hard criticism against XP, and the pair programming practice has also received its share of the critique (Stephens and Rosenberg 2003, Keefer 2003). However, their critique against pair programming focuses mostly on the requirement in XP to use pair programming for all development tasks by everyone in an XP team, not on the pair programming practice itself. Williams and Kessler (2002) disprove several typical negative myths about pair programming based on their reasoning and experiences. Due to the lack of conclusive studies on most aspects of pair programming the debate certainly continues.

## 2.3 Proposed benefits of pair programming

Pair programming has been said to benefit software development in many ways. Williams and Kessler (2002) mention benefits to quality, time, morale, trust and teamwork, knowledge transfer, and enhanced learning. Their list of benefits mostly covers the

[1] Constantine does not mention the year of the visit, but says he visited the company soon after J.P. Plauger had started it. According to http://www.plauger.com/resume.html the company was started 1978.

proposed benefits from all other sources discussed in this thesis, but additional references are included in the discussion below.

### 2.3.1 Factors behind the benefits

Williams and Kessler (2002 p. 21-31) and Williams (2000) discuss several behaviors which should contribute to the achievement of the proposed benefits of pair programming. They say that these behaviors tend to happen naturally with pairs, i.e., the pairs work in a different, better way than individuals. The behaviors are:

- Pair pressure. People work harder in order not to disappoint their partner and in order to finish the task within the limited time allocated to the pair programming session.
- Pair negotiation. Williams and Kessler refer to many studies and conclude that solving a problem by more than one person leads to better solutions and allows solving harder problems.
- Pair courage. If both partners have the same opinion of something, they have courage to do things they might avoid doing alone. If neither of the partners understands something, they ask help from someone else sooner than a person working alone.
- Pair reviews. Reviews are an efficient way to find defects, but are often neglected in practice, because developers don't like them. Pair programming is a continuous review done immediately when the code is written.
- Pair debugging. Williams and Kessler refer to several anecdotes from software developers, who advocate explaining a problem in a program aloud in order to solve the problem. When doing pair programming there is always someone who can comment on the explanation.
- Pair learning. Knowledge related to programming language, tools etc. passes constantly between partners. When partners are changed the knowledge spreads efficiently among the whole team.
- Pair trust. Developers learn to know and trust other team members, which is beneficial for the team.

Cao and Xu (2005) studied activity patterns between pair programmers and found that pair programmers engaged in activities such as asking for opinions, requesting explanations, critiquing partner's approach and summarizing current status. They conclude that these activities lead to a deeper level of thinking and thus to better learning. They suggest that these patterns can help explain the benefits or pair programming.

Bryant (2004) notes that thinking aloud is natural when doing pair programming and that this verbalization has been proposed to change programmers' behavior by forcing them to be more reflective.

Domino et al. (2003) propose based on previous research that task conflicts, i.e., occurrences of incompatible preferred outcomes between two parties, enhance performance, when the conflicts appear in low to moderate levels.

Next we will discuss the proposed benefits in more detail, i.e., what aspects are improved and why the improvement should happen. The evidence supporting or refuting the claims will be discussed in section 2.5.

### 2.3.2 Quality

Improvements to software quality have been discussed in several sources. Constantine (1992) noticed that pairs produced nearly 100% bug free code, which was also better, tighter and more effective. He proposes that the reason for the improvements was the increased work visibility. Wilson et al. (1993) read about studies done with children proposing that collaboration improved problem solving success. By generalizing those results, they hypothesize that student programmers working in pairs would produce more readable and functional solutions to programming problems. Beck (2000 pp. 101) proposes that people are more disciplined in following agreed on practices, e.g., writing unit tests, when working with a pair. Better discipline should lead to improved quality, if the selected practices are effective for improving code quality. According to Williams and Kessler (2002) continuous review of the code by the navigator finds many defects in the code early. They also propose that the pairs end up in better solutions to problems because together the pair evaluates more alternative solutions and picks the best one. Van Deursen (2001) suggests that pair programming benefits program comprehension due to better code and due to learning the code and program understanding strategies from the partners.

### 2.3.3 Elapsed time

When talking about the elapsed time in conjunction with pair programming one must note that two person's effort is used. If a pair finishes a task in half the elapsed time compared to an individual, the total effort is equal.

Nosek (1998) found that pairs used shorter elapsed time when he made an experiment about the effect of pair programming to software quality. He proposes that this decrease in elapsed time could be utilized to speed up the development, and would sometimes be valuable even if the total effort was higher. Williams and Kessler even claim that the shortening is so remarkable that the total effort between pairs and individuals is equal, i.e., pairs finish their work in half the elapsed time used by individuals. They discuss several factors that contribute to the decrease in the elapsed time. People work harder when pairing and spend less time on unproductive tasks. People cut down time spent on phone calls and e-mail during the pair programming session. Two people are also less likely to hit a dead end and are able to find a new alternative solution faster. (Williams and Kessler, 2002) Pair programmers may also be more predictable what comes to the effort spent (Nawrocki and Wojciechowski, 2001).

Williams et al. (2004) have examined the relationship of pair programming and Brook's law (Brooks 1975) saying that *adding man power to a late software project makes it later*. They use a mathematical model of Brooks' law with values for the model's parameters obtained from their survey to practicing pair programmers. They conclude that adding manpower to a late project will yield productivity gains to the team more quickly if the team employs the pair programming technique.

### 2.3.4 Human factors

Several human factors are affected by the use of pair programming. Wilson et al. (1993) hypothesize that pair programming increases developer's confidence for their solutions and enjoyment of work. Being a happier programmer should improve morale and decrease the likelihood of leaving the job (Williams and Kessler 2002). Pair programming makes people familiar with each other and thus builds trust and improves teamwork (Williams and Kessler 2002).

### 2.3.5 Knowledge transfer

Improved knowledge transfer is a consequence of improved communication. If people pair with many different people, they learn more about the system under development. This decreases the negative consequences of a key person leaving the team. The partners exchange knowledge of the used tools and programming tips. All people have different skill sets, so everyone can learn something from anyone. If an expert and a novice pair with each other, the expert may get a better understanding of a topic when she explains it to the novice. An important consequence of improved learning is that new developers can be made productive faster. (Williams and Kessler 2002)

### 2.3.6 Summary of the proposed benefits

Table 1 summarizes the proposed benefits classified as quality, elapsed time, human factors and knowledge transfer.

**Table 1** The proposed benefits of pair programming.

| Category | Effect |
|---|---|
| Quality | • decreases the number of defects<br>• leads to better solutions to the problems, e.g., to better design<br>• improves the comprehensibility of code and design |
| Elapsed time | • a pair finishes a task faster than an individual<br>• a project may be finished sooner by adding more people<br>• facilitates making more accurate effort estimates |
| Human factors | • higher satisfaction and confidence to own work<br>• improves trust and team work<br>• adds more discipline to work |
| Knowledge transfer | • people learn more from each other<br>• the number of critical experts decreases<br>• new employees become productive faster |

## 2.4 Costs of pair programming

The most notable cost of pair programming is that it may add some effort for the development tasks. Different studies have reported figures for the effort increase from 15% (Williams 2000) to 100% (Nawrocki and Wojciechowski 2001). The effort increase is higher when starting to do pair programming (Williams 2000). Padberg and Müller (2004) propose two reasons for this effect. The reasons are learning to do pair programming and learning to work with a new person. Bryant (2004) has found a difference between more and less experienced pair programmers. When two experienced pair programmers work together, no matter who is acting as the driver/navigator the driver/navigator behaves in a similar way, i.e. the partners switch behavior when they switch roles. Two less experi-enced partners also change behavior when they switch roles, but the partners do not have a similar behavior to their partner in the same role. Bryant does not draw any conclusions whether this difference could be the reason why the learning time is less productive. Lui and Khan (2003) report that when two persons with different skill level work together, they finish the task as quickly as the more skillful of the partners would have done alone, meaning that only the effort of the less skillful programmer is lost. Thus the decrease in productivity equals to what the less skillful programmer would have been able to do alone with the same effort. However, when working together the worse programmer may learn something that increases her productivity later.

Some less quantifiable disadvantages of pair programming are related to human factors. Some people don't want to pair and some people don't get along with each other (Beck 2000). Williams and Kessler (2002) mention that pair programming is efficient because people work harder, but they also admit that is too intense for many to do all day continuously.

Two economic models for evaluating the feasibility of pair programming have been proposed (Williams and Erdogmus 2002, Padberg and Müller 2003). Both models consider variables such as pair speed advantage, i.e. shorter elapsed time, defect advantage, personnel costs and economic value of shorter time to market. By assigning context specific values to the model variables it should be possible to evaluate whether pair programming is beneficial in a certain context or not. Padberg and Muller have later refined their model by considering the improvement in the efficiency of pair programming after a learning time (Padberg and Müller 2004).

## 2.5 Studies on pair programming

The amount of pair programming research has increased steeply after Williams published the results of her pair programming experiment (Williams 2000). Below I discuss the experimental designs and results of the previous pair programming studies put into four categories. The first two categories contain experiments focusing on studying the differences in effort and quality when using pair programming vs. some other way of developing software. The experiments in the first category compare individuals and pairs in isolation, i.e., not within a team. The second category contains studies where pair programming is performed within a development team. The third category contains experiments studying the effects of pair programming from the viewpoint of learning programming in the context of introductory programming courses. The fourth category contains experiments studying some detailed aspects of pair programming.

### 2.5.1 Overview

Some analyzes and summaries of the previous studies have already been made. Gallis et al. (2003) summarize the experimental designs and results of six pair programming studies. They identified apparent contradictions in the results of the experiments and therefore propose an initial framework for supporting better research on pair programming. They identify and discuss the independent variable (level of collaboration), dependent variables (time, cost, quality, information and knowledge transfer, trust and morale, risk) and countless context variables from a research viewpoint. Boehm and Turner (2003 p. 230-233) summarize shortly the results of seven pair programming experiments regarding effort, quality and developer's satisfaction.

There are many common challenges in making good experiments comparing pair programming and solo programming. Defining exactly what is meant by pair programming is not trivial. How should partners behave during a pair programming session? How much true pair programming should be used and are some looser forms of collaboration between partners allowed? How should pair programming be taught to the subjects? Should the partners be familiar with each other? Are the subjects allowed to choose themselves whether they are using pair programming or the other way?

In all of the studies discussed here the participants have done pair programming as a part time assignment some hours a week, which means that they may have been fresher for the pair programming sessions than a person who does considerable amounts of pair programming every day.

### 2.5.2 Individuals vs. pairs in isolation

The most relevant experiments comparing pairs working themselves to individuals working alone are presented in Table 2. These experiments have been done in a context where the pairs are forced to pair most of the time and individuals are not allowed to collaborate or ask help from anyone at all. It is important to do this kind of experiments in order to understand the inner workings of pair programming, but one must be very careful in generalizing their results to a team context. The context is far from the typical industrial way of developing software where you mostly write code alone, but when facing a problem can ask help from your colleagues. The developed programs have also been quite small. Typically, any larger software system is developed by a team containing more than one or two persons. Working as a part of a team doing a whole software project is a different situation from doing small, separated programming tasks alone or with one single partner. If the tasks in the experiments had been larger, it would have been more realistic to compare pairs to pairs, because then it would have been necessary for the solo programmers to divide and integrate their work as was done by Heiberg et al. (2003).

**Table 2** Pair programming experiments with experienced programmers writing small programs alone vs. with a pair.

| | **(Nosek 1998)** | **(Williams 2000)** | **(Nawrocki and Wojciechowski 2001)** |
|---|---|---|---|
| **Comparison** | PP vs. solo | CSP (pairs) vs. PSP (solo) | XP2 vs. XP1 (=no PP) |
| **N** | 15 (= 5*2 + 5*1) | 41 (= 14*2 + 13*1) | 15 (= 5*2 + 5*1) |
| **Randomization** | yes | yes, with same avg. GPA | yes, with same avg. GPA |
| **Subjects** | professional programmers, but a new problem for all | advanced students, 2-3 years of C++ experience | 4th year students |
| **Context** | at a company, familiar tools | a course at Univ. of Utah | a course at Univ. of Poznan |
| **PP training** | not mentioned | effective pp was taught | not mentioned |
| **Type of project** | fixed scope | fixed scope | fixed scope |
| **Type of assignment** | a script for database consistency check | four small programs | the four PSP assignments |
| **Language** | a script language | C++ | C or C++ allowed |
| **Programming effort** | less than an hour | a few hours per program | a few hours per program, 120-420LOC (avg./prog.) |
| **RESULTS (PP vs. the other way)** | | | |
| **Effort** | 42% | 60%, 15%, 15% (prog. 1-3) | ~100% (prog. 1-4) |
| **Quality** | better readability and functionality | 90 vs. 75% test cases passed, results more consistent, LOC smaller | same number of re-submissions, LOC smaller, st. dev. of LOC smaller |

#### 2.5.2.1 *Evaluation of the experiments*

Nosek's (1998) study is interesting because it was made with professional programmers in corporate environment using tools familiar for the programmers. However, the number of programmers was rather low (fifteen) and they did only one tiny program (max. 60 minutes of effort). Despite of the small sample Nosek was able to find statistically significant differences between the performance of pairs and individuals.

Williams (2000) had 41 computer science students, who developed four rather small programs[2]. Students were asked for preferences for working alone or with a pair and with whom they would like to work and with whom not. In addition students' GPAs were used ensure that there were the same amount of high, average, and low performers among solo and pair programmers. Individuals used a modified personal software process (PSP) and pairs used the same process adapted for pair programming. One threat for the validity of the results is that the pairs developed an additional program after each program developed by all participants in order to balance the effort required. This may have given them an advantage of learning more about, e.g., the used technologies.

Nawrocki and Wojciechowski (2001) had 21 computer science students, who developed four small programs. Based on student's GPAs, they divided them in three equally skilled groups, one using a modified XP process with pair programming, one using the same XP process without pair programming, and one using the personal software process (PSP) without pair programming,. Studying the effects of pair programming is justified here only between both forms of XP, because PSP differs so much from XP.

#### 2.5.2.2 *Effects to the quality*

The results of all these experiments indicate some improvements in quality when using pair programming. The differences are statistically significant in Nosek's and Williams' studies. Nosek found that the functionality of the developed scripts evaluated by an objective evaluator was better for pair programmers (mean 5.60 vs. 4.20 on scale 0-6). Williams found that the average number of passed instructor's test cases was higher in pairs' programs. Individuals' programs passed 70-79% test cases (avg. for each program) whereas pairs' programs passed 86-94% of test cases. The difference was about 15 percentage units for each program, i.e., pairs made about 60% less bugs (about 10% vs. 25% failed test cases). Williams also reports superior high level designs from pairs in the form of better exploitation of object oriented programming constructs, e.g., encapsulation and better class-responsibility alignment, but the only hard data she reports is about 20% lower number of LOC in pairs' programs.

Nawrocki and Wojciechowski (2001) assessed quality by calculating the number of re-submissions of corrected programs to acceptance testing until all the tests passed. He found almost no difference in the number of re-submissions between pair programmers and solo programmers. Nawrocki and Wojciechowski found that pair programmers implemented the same programs with less lines of code (LOC) and also the standard deviation of LOC was smaller for pairs' programs than for individuals' programs.

Wilson et al. (1993) made a similar experiment with students as Nosek made later in the industry. They found that pairs wrote more readable programs (mean 1.75 vs. 1.29 on scale 0-2) and the difference was statistically significant ($p<.1$). The degree to which the

[2] The only published measure of the size is the summed development effort for programs 2 and 3 (Williams 2000, pp. 65). The median value was about 9 hours for both pairs and individuals.

programs solved the problem was also higher in pairs' programs (mean 4.20 vs. 3.03, scale unknown[3]), but this difference was not statistically significant.

Arisholm (2002) had industrial programmers doing the same programming task by 123 individuals and 4 pairs. Each programmer was classified by her experience as junior, intermediate and senior. Each pair contained a junior and a senior. The correctness of pairs' programs was 74% compared to 62% of those from intermediate individuals. Thus pairs made 32% less defects (26% vs. 38%).

#### *2.5.2.3 Effects to the effort*

Nosek (1998) found that the pairs spent on the average 42% more effort than individuals but the difference was not statistically significant. Williams (2000) analyzed the effort differences for the separate programs. For the first program pairs spent 60% more effort, but for programs 2 and 3 pairs spent only 15% more effort. For programs 2 and 3 the effort differences were not statistically significant.

Nawrocki and Wojciechowski (2001) found that pairs spent 100% more effort for each of the four programs in their experiment. An interesting finding was that the standard deviation of the pairs' efforts was only half of that of individuals proposing that pair programming is more predictable than solo programming.

Rostaher and Hericko (2002) compared pair programmers and individuals in an industrial environment, where people had already used pair programming before. He found that pairs and individuals used almost the same amount of elapsed time (358 vs. 362 minutes), which indicates 98% increase in effort when using pair programming.

In the experiment by Arisholm (2002) pairs needed 98% more effort than individuals (118min vs. 60min).

Of all these experiments Nawrocki and Wojciechowski (2001) is the only one where a similar level of quality was ensured by the experimental design. One must take into account that requiring the same level of quality for all programs would have required extra effort from those developers having more defects at the first attempt.

### 2.5.3 Individuals vs. pairs in a team context

Table 3 summarizes the experiments where pair programming was studied in an environment more similar to real software development projects. Some of the proposed benefits of pair programming, e.g., knowledge transfer within the team and improvement of trust and team work between the team members, are related to using it in a team context. If these benefits are true, it can be assumed that they lead to better quality of work and decreased total effort spent for a project carried out by a team. These effects cannot be studied in experiments with isolated individuals or pairs.

[3] Scale is documented as 0-2, but this is impossible based on the mean values.

**Table 3** Pair programming experiments with experienced programmers developing a larger system as a team.

| | **(Williams 2000)** | **(Ciolkowski and Schlemmer 2002)** | **(Baheti et al. 2002)** |
|---|---|---|---|
| **Comparison** | CSP (pairs) vs. PSP (solo) | PP vs. unsystematic collaboration | PP vs. solo |
| **N** | 7 + 3 teams, 4 persons/team | 3 + 3 teams, 6persons/team | 9 + 16 teams, 2-4 persons/team |
| **Randomization** | same GPA in CSP and PSP groups | not mentioned | no, self selected members and type of working |
| **Context** | advanced students, 2-3 years of C++ experience on a course at Univ. of Utah, 1999 | 2nd year students on a course at Univ. of Kaiserlauten, 2002 | graduate class, O-O Languages and Systems course at North Carolina State University |
| **PP training** | effective pp was taught | material given for all teams | not mentioned |
| **Project type** | not mentioned | fixed scope | not mentioned |
| **Language** | C++ | Java | Smalltalk/Java[4] |
| **Assignment type** | a project | extend and modify an existing web quiz system | teams made different projects |
| **Program size** | not mentioned | ~4000LOC | not mentioned |
| **Project effort** | not mentioned | ~700 hours, of which programming ~40h/person | not mentioned |
| **Duration** | 4 weeks | 5 weeks[5] | 5 weeks |
| **RESULTS (PP vs. the other way)** | | | |
| **Effort** | -28% | 9% (coding & testing only) | 3% |
| **Quality** | -2% (passed test cases) | LOC smaller, coupling factor smaller | pairs received 1% better (93.6 vs. 92.4 of 110) grade |

The experiment by Ciolkowski and Schlemmer (2002) used quite a large project, where teams of six persons spent together about 700 hours of effort. However, only about 40 hours of effort per person was spent on programming work. The researchers were not able to evaluate software quality using defects metrics, but used LOC and coupling factor measures instead. Both of these were slightly smaller for pair programming teams indicating slightly better design quality. The difference in effort was evaluated based on data of the programming tasks (including writing test case) for two consecutive iterations lasting 5 weeks in total. In both iterations pair programmers spent about 9% more effort. No statistical analysis was made for evaluating the significance of the results.

Williams (2000 p. 77-78, 100-101) continued her experiment described in the previous section by assigning the students to a four-week project in four-person teams. She does not mention the total effort used or the size or type of the software developed. She found that on the average pair programming teams spent 28% less effort, but solo programming teams passed 2% more test cases. Neither of the differences was statistically significant. In this experiment the students using pair programming were experienced pair programmers after having used it in the previous experiment.

[4] The course taught Smalltalk and Java, nothing is mentioned about the project.

[5] The whole project lasted 13 weeks, but programming phase only 5 weeks

Baheti et al. (2002) studied pair programming with 132 students working in self-selected teams of two to four. Each team selected themselves whether they were going to use pair programming or solo programming and whether they were going to work co-located or physically distributed. We discuss here only the differences between the 25 co-located teams. Each team made one of several possible assignments during a 30 days long project. The co-located pairs had slightly worse productivity than the co-located individuals (14.8 LOC/h vs. 15.2 LOC/h), i.e., pairs needed 3% more effort for writing the same amount of LOC. The quality of the systems was evaluated by a teaching assistant on scale from 0 to 110 based on a 30 minute demo. The pairs received slightly better grades (mean 93.6 vs. 92.4). Neither the productivity nor quality differences were statistically significant. The experimental design can be criticized for not using randomized design either for forming the teams or selecting the type of programming, having different projects made by different teams and using vague quality metric.

When comparing these experiments to those performed with isolated pairs and developers doing small tasks, the effort increase incurred by pair programming was smaller or even negative. Comparing quality is hard due to the different metrics used, but it seems that the quality improvements were smaller in the team context. However, it may be that the smaller effort difference explains decreases in quality.

### 2.5.4 Pair programming and learning to program

The use of pair programming on introductory programming classes has been studied with hundreds of students in North Carolina State University (McDowell et al. 2002; McDowell et al. 2003a; McDowell et al. 2003b) and University of California Santa Cruz (Nagappan et al. 2003). All these experiments have compared students working as pairs to students working individually. The results of these experiments are summarized in (Williams et al. 2003). These studies provide strong support for certain benefits of pair programming. Students working as pairs got same or higher percentage of good grades, performed at least similarly in the exam, produced better programs, were not hampered in future solo programming courses, and were more likely to pursue computer science related majors one year later (Williams et al. 2003).

Hanks et al. (2004) analyzed the programs developed at UC Santa Cruz more carefully regarding their external and internal quality. They found that pairs made more functional programs measured by the amount of features implemented and the number of remaining defects. There were no differences in the internal quality evaluated using source code size and complexity metrics in the first two programs, but the third programs from the pairs were longer and more complex. McDowell et al. (2003a) analyzed the elapsed time between pairs and individuals and found that the pairs were 25% faster, i.e. they spent 50% more effort on the average, but the difference in the elapsed time was not statistically significant.

Heiberg et al. (2003) studied the productivity of pair programming with 110 first year computer science students at University of Tarto. The pairs developed one program as far as they could during four 90 minutes lab sessions. The special aspect of this experiment was that it compared two partners working together to two partners working with the same task but alone, i.e. the effects of coordinating work between two people working alone with the same tasks were taken into account. The students were given automated acceptance tests to test their code during development. The amount of functionality developed was measured by calculating the number of passed tests, effectively combining the measurement of quality and scope. In this setting Heiberg et al.

did not found a statistically significant difference in the effort usage between the two types of pairs.

### 2.5.5 Other studies

Müller (2004a) compared pair programming to solo programming with code review using 27 experienced students on a programming course at University of Karlsruhe as the subjects. He discusses a typical problem of pair programming (and other) experiments, i.e., both the quality and effort being outcome variables because a program is considered ready when its developer thinks so. Müller solved the problem by forcing programmers fix bugs until the programs passed at least 95% of the acceptance tests. He analyzed separately time spent on implementation and on quality assurance. The quality assurance included bug fixing starting from the first execution of the acceptance tests. After the implementation in the first run of the acceptance tests the pairs passed more tests than the individuals (mean 57% vs. 44%), but spent 15% more effort for the implementation (mean 409min vs. 357min). Neither of the differences was statistically significant. The quality assurance phase ended when the participants delivered a program passing at least 95% of the tests, so that the quality of all programs was similar. When including the effort spent on quality assurance the pairs spent only 7% (mean 490min vs. 467min) more effort. The difference was not statistically significant. This study suggests that reviews produce the same code quality with slightly smaller cost when compared to pair programming. It must be noted that the reviews were made by persons who had just been solving the same programming task themselves, which may increase the efficiency of the review. However, Müller notes that the reviews did not create many comments, which may indicate that the bare existence of a review makes a programmer write better code. In another paper Müller and Padberg (2004) re-analyzed data from the experiment described above and from its replication (Müller 2004b) considering also programming experience and feelgood factor, i.e., how comfortably the developers feel in a pair session. They found no correlation between programming experience and performance, but they found a positive correlation between feeldgood factor and performance. Even though existence of a correlation does not reveal which factor drives which, they propose feelgood factor as a candidate driver for the performance of a pair.

Parrish et al. (2004) analyzed the performance of programming pairs having high vs. low amounts of collaboration when developing the same software module in a context of a large industrial project. Here the collaboration did not mean pair programming in the normal sense, but just that two persons were somehow working on the same day with the same module. They found that high amount of collaboration led to clearly worse productivity, and propose that pairs working together aren't naturally productive, but the role-based protocol provided by true pair programming combats the productivity loss. This finding indicates that the way how two persons collaborate has a considerable effect on productivity.

Lui et al. (2003) made two experiments regarding problem solving by pairs. In the first experiment fifteen industrial programmers were divided to five pairs and five individuals who solved algorithmic problems by answering multiple choice questions. After delivering the results, participants were given told the amount of errors left until all their answers were correct. At the first attempt pairs spent 20.9% more effort (26.6 min vs. 22 min), but when all errors had been corrected pairs had spend 4.2% less total effort (36.4 min vs. 38 min). In the second experiment all subjects solved deduction problems as both individuals and pairs in shift. Again at the first attempt pairs used 73% more effort (130 min vs. 75 min), but the work done by pairs had already 85% correctness compared to 51% for the individual work. After correcting the errors the pairs had spent 5.3% less total effort.

Canfora et al. (2004) made an experiment with forty-five 3rd year students in order to test the effect of pair designing on knowledge building. The knowledge building was measured using a query before and after each of the three phases of a design assignment. In all phases the pairs increased their knowledge more than solo designers, but the difference was statistically significant ($p<0.05$) only in the first phase.

Williams et al. (2004) made a survey of how quickly a new employee becomes productive when mentoring is done using vs. not using pair programming. The averages were 12 vs. 27 days and the required mentoring efforts 26% vs. 37% of mentors work time showing clear benefits for using pair programming.

## 2.6 Advice for practicing pair programming

There are not yet many published studies discussing the details on how to do pair programming in the most efficient way. Instead the studies have so far focused on comparing solo programming to pair programming in some but not very formally defined way. Some advice have been proposed in the literature but they are mostly based on the personal observations and reasoning of the corresponding authors.

Cao and Xu (2005) observed 23 students doing pair programming and found that pairing two highly skilled developers worked best for both knowledge generation and quality improvement, but pairing two medium developers did not show interesting benefits in terms of either knowledge generation or quality improvement. Jensen (2003) has reported similar findings from industry, proposing that partners with different skill levels is a more beneficial situation for pair programming than equally skilled partners.

Williams and Kessler analyze what kind of developers match together considering developer's skill level, personality (extrovert-introvert), gender and cultural issues. People with excess ego cause problems with everyone else and large difference in skills requires mentoring attitude from the more skillful partner, but other types of pairs work well. Most people first resist pair programming, but after trying it, most people prefer it over solo programming. However, no one should be forced to pair. (Williams and Kessler 2002)

Williams and Kessler propose rotating pairs regularly, e.g., on a daily basis so that developers end up pairing with many different partners. Pairs should be formed very casually, e.g., as a part of a daily scrum meeting or just by letting a developer ask an appropriate person to pair with her. Each task should have an owner, who recruits a partner to work with her for the various parts of the task. Williams and Kessler consider workspace needs proposing common and personal areas in the office for having some privacy and protecting solo programmers from noise. Suitable tables are needed so that both can see the display and switch the keyboard easily, or alternatively two display, keyboards and mice can be provided. Standard development environment including a common coding standard is recommended. Williams and Kessler discuss certain behaviors during a pairing session mostly related to being a good communicator in all respects. Pairing is most important for analysis and design activities, and especially when working with more complex tasks. (Williams and Kessler 2002)

Coplien and Harrison (2004, p. 165-167) emphasize forming pairs by self selection so that pairs work well together. They also discourage dictating the style of doing pair programming, e.g., that no code may be written unless both are at the keyboard.

Beck (2000) proposes in XP pair programming should be used for all production code. Williams and Kessler (2002 p. 14-15) made a survey on several practicing pair programmers and found that 22% of them spent more than 75% of their work day doing pair programming and 30% less than half of the day. Williams and Kessler (2002 p.15) propose that because pair programming is very intense form of work and there are schedule constraints, it should be used only for the most complex tasks if people don't want to use it all the time. In that case they recommend that there are dedicated pair programming hours when everyone pairs and no one interrupts the work.

## 2.7 Summary

Collaboration between developers is not a new thing, but pair programming as a very tight form of collaboration has gained more publicity only during the last few years. It has been proposed that the use of pair programming has many benefits related to quality, elapsed time, human factors and knowledge transfer. Several studies comparing pair programming to solo programming have been made. Software quality has been better for the pairs or similar in most of the experiments. The results about the required effort have varied quite a lot. The elapsed time has always been the same or shorter for the pairs, but the total effort has been -28% - 100% larger for the pairs. Pair programming has also been studied in the context of learning programming in introductory university courses. The results have been very positive. Students working as pairs got same or higher percentage of good grades, performed at least similarly in the exam, produced better programs, were not hampered in future solo programming courses, and were more likely to pursue computer science related majors one year later. There are not yet many published studies discussing the details on how to do pair programming in the most efficient way. Some advice have been proposed in the literature but they are mostly based on the personal observations and reasoning of the corresponding authors. It may be that the details have not been considered very important, e.g., many sources propose being very casual when forming of pairs.

# 3 Research design

This chapter deals with the experimental context and design. Preliminary guidelines for empirical research in software engineering have been proposed by Kitchenham et al. (2002). Their guidelines for the empirical context and empirical design have been considered when planning and reporting this study. Section 3.1 describes the research questions and hypotheses of the study. A motivation is given for each hypothesis and practical issues related to measuring the intended attributes are discussed. Section 3.2 describes the pair programming experiment, which was executed in order to answer the research questions. In the experiment several teams made a similar project, half of the teams used pair programming and the other half solo programming. Section 3.3 describes how certain realities affected the experimental design.

## 3.1 Research questions and hypotheses

Each research question is described below with some motivation and background information. Related to each question, one or several hypotheses are derived based on literature and what I have personally learnt from discussions with industrial developers who have used pair programming in their work. Finally the challenges of measuring the related phenomena are discussed. The context for all research questions and hypotheses is comparing the performance of pair programming (PP) teams to solo programming (SP) teams when four-person teams develop software as described in section 3.2.

### 3.1.1 Productivity

The key question when deciding on the use of pair programming or any other practice is how it affects the overall productivity of the organization. Productivity, i.e., the amount of work results divided by the effort used, can be analyzed on several levels. We concentrate on analyzing productivity in connection with a release project and the implementation of a use case.

**Research question 1**: Does pair programming affect project productivity?

I assume that even though pair programming may increase the effort for the development work related to individual programming tasks, it may decrease the overall project effort due to all the proposed benefits of pair programming described in section 2.3.

In similar experiments (Ciolkowski and Schlemmer 2002; Williams 2000; Baheti et al. 2002) the difference in project effort when comparing pair programming to solo programming has been between -28% and 9%. The largest figure 9% was reported by Ciolkowski and Schlemmer, but they analyzed only effort spent for programming and writing tests instead of the whole project. This may have disregarded some of the benefits of pair programming on the project level.

**Hypothesis 1.1**: The PP teams have higher project productivity than the SP teams.

In the experiment the total effort spent is the same for all teams, which means that the amount of work results varies between the teams. Measuring project effort means just summing up all hours spent on the project, e.g., on design, programming, unit testing, system testing, bug fixing, meetings, and studying the existing system and new technologies and tools.

Measuring the amount of work results could be done easily using lines of code (LOC). However, LOC is not a reliable measure because different designs and coding styles may lead to different LOC for the same amount of functionality implemented. What really matters related to the amount of work results is what the software provides for the end users. Therefore the amount of work results is measured as the amount of use cases implemented in the software. All teams must implement the use cases in the same order. The total amount of implemented use cases per team puts the teams reliably in productivity order.

Based on the amount of implemented use cases it is not possible to say how many percentages the difference in productivity between two projects is because the use cases are not of equal size. Function points could be used to calculate the absolute amount of functionality in each use case, but their have several limitations (Fenton and Pfleeger 1996 p. 262) and are not used here. I estimate the size of each use case by taking the average of the efforts different teams spend on implementing the use case. The productivity differences are estimated by summing up the sizes of the use cases each team implemented.

When comparing productivity based on the amount of work results, the quality of the work results must be of similar level in order for the comparison to make sense. Therefore the goal of aiming for high and thus similar quality is emphasized to the participants. Systems must be tested afterwards to ensure that use cases are implemented successfully and with similar quality. If major bugs are found, the related use cases are not counted as implemented.

**Research question 2**: Does pair programming affect use case implementation effort?

The implementation effort of a use case covers designing, coding, unit testing, correcting found bugs, and documenting the code. Use case productivity is the inverse of the use case implementation effort.

Previous research (Nosek 1998; Williams 2000; Nawrocki and Wojciechowski 2001) has observed an increase of 15%-100% in the programming effort when pair programming is used in the context of small separated tasks and without a surrounding development team. Some proposed benefits of pair programming, e.g., knowledge transfer and better design, are likely to become more relevant in a typical project context, where several related task are performed by a team. The team context may somewhat decrease the use case implementation effort of the PP teams in this experiment compared to the results of the above-mentioned experiments. However, I believe that there is still some increase in the use case implementation effort when using pair programming.

The productivity of pair programming may be related to the type of task under work. Williams and Kessler (2002) propose using pair programming at least for complex tasks. I have heard many practitioners reporting that pair programming is quite useless for trivial tasks and more efficient when used for complex tasks.

**Hypothesis 2.1**: The PP teams have lower use case productivity than the SP teams.

**Hypothesis 2.2**: Higher use case complexity favors PP teams as measured on use case productivity.

Measuring the effort per use case is not as easy as measuring project effort, because sometimes development effort may be related to several use cases, and different developers may report the effort to different use cases.

The quality criterion in this experiment is based on the opinion of the developers, i.e. they believe the code is ready and works. Other quality aspects of the code (comprehensibility, exact correctness etc.) are not taken into account here. These aspects may affect the effort required for modifying the code later, and the effects can partially be seen in project productivity.

Evaluating the difficulty of implementing a use case is done subjectively. A scale from one to five (1=very easy, 3=moderate, 5=very difficult) is used. In the end of the project, the developers evaluate the complexity of each use case they implemented. The evaluations of pair programmers are omitted, because pair programming may affect the perceived complexity in some cases and here we are interested in the complexity as perceived in the traditional way of working, i.e., solo programming. The complexity of a use case is defined as the average of the evaluations by solo programmers from different teams who implemented the use case.

### 3.1.2 Defects

A lower number of defects in the code after the code has been written and unit tested improves productivity as less time is required for fixing defects found during system testing or at worst by the end users.

**Research question 3**: Does pair programming affect the number of defects?

Several experiments mention better functionality or higher test case pass rate when using pair programming (Wilson et al. 1993; Nosek 1998; Williams 2000; Arisholm 2002). The improvements in quality have been used to justify the economic efficiency of pair programming compensating the costs of increased effort during the development work (Williams and Erdogmus 2002).

**Hypothesis 3.1**: After coding and unit testing the PP teams have fewer defects than the SP teams.

**Hypothesis 3.2**: After system testing and bug fixing the PP teams have fewer defects than the SP teams.

The developers report all defects found in the finished code. The defects in any unfinished code are not interesting for this study. Defects can be found, e.g., during the development of other use cases or during the system testing before a delivery. A researcher tests all the systems after their final delivery in order to assess the quality of the final systems and the quality of testing done by the teams.

### 3.1.3 Design quality

Defining good design quality is difficult. From a practical point of view good design supports efficient maintenance or further development of the product. This goal can be achieved, e.g., if the code is easy to understand, modify and test.

**Research question 4**: Does pair programming affect design quality?

Previous studies have proposed that pair programming improves design quality. Previous studies have used smaller LOC as an indication of better design quality (Williams 2000; Nawrocki and Wojciechowski 2001), but LOC can hardly be justified as a good measure of design quality. Ciolkowski and Schlemmer (2002) found a slightly smaller coupling factor in the designs of the PP teams. I have personally heard from many pair programmers a subjective opinion that pair programming improves design quality.

**Hypothesis 4.1**: The PP teams create better software design than the SP teams.

A practical indirect measure of design quality would be the effort required for modifying the code preferably by someone else than the original developers. Unfortunately in this experiment this cannot be arranged. Another approach is to use some code metrics for evaluating the design quality. Lots of metrics for this purpose have been proposed, e.g. by Fenton and Pfleeger (1996 p. 279-335).

There are many challenges in using code metrics for design quality evaluation in this experiment. The teams receive core architecture for the system, which largely defines the higher level design. The sizes of the final systems vary, because the teams are likely to implement different numbers of use cases. Varying size may affect many code metrics, even if the metrics did not directly measure software size. Alternatively intermediate versions of the systems each having the same set of use cases could be compared, but in this case the snapshots would originate from a different phase of the project, e.g., early or late in an iteration. Because the design quality may be affected by, e.g., system testing, bug fixing, refactoring and delivery preparation activities occurring typically in certain phases of the iteration, the latter approach is not reliable either.

I concentrate on analyzing the design quality on the method level using the following code metrics. LOC per method is used to analyze the size of a method, a reasonably small value being better for a good design. McCabe's cyclomatic complexity per method (McCabe 1996) counts the number of flows through the method, a smaller value being an indication of less complex code and thus better design. The number of a method's parameters is used to analyze how much information is passed to the method when it is called, a reasonably small value being better for good design. Unfortunately even these metrics are probably affected by the growth of the system. Up to a certain threshold the design is as good as one with a smaller value. Therefore the proportion of methods having a very poor value for a metric, thus indicating a poor structure, is probably a better metric.

### 3.1.4 Knowledge transfer

Improved knowledge transfer is a favorable property for any software development project. We can analyze both the breadth and depth of a developer's understanding of the developed system. The breadth of understanding characterizes how many modules of the system a developer knows. The depth of understanding characterizes how well a developer knows a specific module. We can analyze the understanding also from the perspective of an individual module. For example, if a particular module is understood well by only one developer, there is a high risk of having a critical person, who cannot be replaced easily.

**Research question 5**: Does pair programming affect knowledge transfer within the team?

Williams and Kessler (2002) propose that pair programmers, especially if the pairs are rotated, know more about the overall system, but neither they nor other researchers have analyzed this claim more carefully or reported any data about this aspect.

When pair programming is used, tasks (use cases in this experiment) are allotted to only half the number of worker units compared to solo programming projects. Therefore in a pair programming team each developer participates in twice as many tasks as in a solo programming team, if the same tasks are completed in both teams. This distributes a developer's involvement to the development of different modules more broadly in pair programming teams. However, this also means that each developer spends less effort working with each module.

The depth of a developer's understanding of a module is probably related to the amount of her involvement in the development of that module. The acquired depth of understanding may also be affected by the type of involvement, i.e., solo programming or pair programming. The effect of pair programming may be positive when a person learns new things from the partner or when a partner allows solving a problem that the developer alone could have spent huge amounts of time without learning anything. These positive situations may realize especially when a developer works with a more skillful partner. The effect of pair programming on the acquired depth of understanding may be negative when a person does not have to think deeply enough about the solution and existing code, and thus the new understanding may be quite superficial. Finding a solution to a difficult problem alone can sometimes be very educational.

**Hypothesis 5.1**: Higher involvement in the development of a module leads to higher understanding of that module.

**Hypothesis 5.2**: In the PP teams more developers understand each module well than in the SP teams.

**Hypothesis 5.3**: In the PP teams each developer understands more modules well than in the SP teams.

**Hypothesis 5.4**: In the PP teams developers achieve deeper understanding of a module than in the SP teams with the same amount of involvement with that module.

After the project the developers answer to a web questionnaire, where they evaluate the degree of their involvement in and understanding of the developed modules.

### 3.1.5 Enjoyment of work

Higher enjoyment of work is likely to increase work motivation and thus productivity and quality. It also decreases the possibility of a person leaving her job, which would almost always cause a negative effect for the performance of a project.

**Research question 6**: Does pair programming affect the enjoyment of work?

Williams and Kessler (2002) report that almost all who try pair programming like it. I have personally heard developers commenting that the higher confidence on the solutions when building critical systems with a pair reduces the stress they experience due to work.

**Hypothesis 6.1**: In the PP teams developers enjoy their work more than in the SP teams.

The overall enjoyment of work can only be measured by asking it from the developers. The developers who use pair programming can evaluate which they like more, pair programming or solo programming, and which they consider better for the overall success of this kind of a project.

### 3.1.6 Effort estimation

More accurate effort estimates help planning and controlling a project. Effort estimates can be done by different people, e.g., by a project manager, by the development team collectively, e.g., as in the XP planning game (Beck 2000) or by the developer or pair who is going to implement a task, e.g. as in the XP iteration planning (Beck 2000). Some estimates are typically done in the beginning of a project or an iteration and they may be refined later when the implementation becomes closer or just before the implementation starts.

**Research question 7**: Does pair programming affect effort estimation accuracy?

Nawrocki and Wojciechowski (2001) reported that the standard deviation of development times and program sizes were smaller for programs developed by pairs. This may be, e.g., due to a pair's lower probability of facing problems too hard to solve or being able to solve them faster. Smaller deviations in realizations between different pairs should help making the estimate by anyone, pair or team. Höst and Wohlin (1998) reported that effort estimates for a programming task combined from several developers were better than those from individual developers. I assume this effect can be seen already when making estimates by a pair vs. individual.

**Hypothesis 7.1**: The PP teams estimate use case effort more accurately than the SP teams in the beginning of an iteration.

**Hypothesis 7.2**: Pairs estimate use case effort more accurately than solo programmers just before its implementation.

From a project's perspective the estimates made already in the beginning of an iteration are much more important, because the iteration's resourcing/scoping decisions are based on these. From the same perspective, the sum of the errors in all of the estimates matters most, because if positive and negative estimation errors compensate each other, the overall estimate for the project was perfect and the project achieved exactly what was planned. Sometimes the estimates for individual use cases may go more wrong than the estimate for a set of use cases because in the latter case it may be easier to estimate some logical whole than possibly overlapping use cases. However, the true accuracy of estimates needs to be analyzed on a use case basis. Because different teams implement different amounts of use cases, comparison of the accuracy of the estimates can be done only for those use cases that are implemented by all the teams.

All teams report the estimates done per use case in the beginning of an iteration and updated estimates done just before starting the implementation of a use case.

## 3.2 Experiment

The context for the experiment is discussed below along with the instrumentation required for the data collection.

### 3.2.1 Project

The experiment was performed at Helsinki University of Technology (HUT) in the spring of 2004 during a project course teaching the Java 2 Platform Enterprise Edition (J2EE) technology. The course contained first a two-week training period containing about 15 hours of lectures about the subject given by professional J2EE developers. The theory taught in the lectures was applied in practice in two personal home assignments taking a few hours each. Both the lectures and the home assignments were mandatory for all participants. After the training period the participants started a nine-week development project in four-person teams.

Passing the course required participation to the lectures, following certain work practices as defined by the course organizers, and answering a couple of inquiries. The evaluation of the course was announced to be on pass/fail scale.

The topic of the project was developing a distributed, multi-player casino system using the J2EE technologies. The project included developing, testing and delivering the software, which was described in a requirements specification (see Appendix A) written by the course organizers. The specification was comprehensive discussing the main domain concepts, potential user groups, and some non-functional requirements. Each of the 30 use cases was described in a separate about one-page long use case description (see Appendix B). The desired user interface was specified by providing static HTML-pages of all parts of the user interface (see Appendix C).

The students were given a pre-made 10-page long technical specification and implementation of the core architecture including examples of suitable J2EE design patterns and a build/deployment script. The core architecture contained about 1000 lines of code[6] plus almost 1500 lines of comments. The development environment was provided pre-installed in the laboratory's computer class, but clear instructions were provided on how to set up the environment to any other computer.

The tools used were J2EE 1.4 SDK as the development platform, XDoclet 1.2 for generating the J2EE interface descriptors etc., JBOSS 3.2.1 with Tomcat 4.1.24 and Hypersonic SQL database as the application server, Eclipse 3.0 as the IDE, Ant 1.6.0 as the build tool, and CVS 1.11 as the version control tool.

### 3.2.2 Experimental design

Five four-person teams did a similar project simultaneously but independently of other teams. All the teams had to work mostly as a co-located team and use the same development process, work practices, tools, and specifications.

The experiment had a one-factor randomized design (Juristo 2001 p. 85-86), where the only factor was the type of programmer collaboration. The studied alternatives were pair programming (PP) and solo programming (SP). Either of the alternatives was randomly assigned to each participating group. The PP teams had to use pair programming for all development work and the SP teams were not allowed to use pair programming for more than occasional collaboration. The PP teams had to participate in a one hour lecture discussing what pair programming is before the experiment.

### 3.2.3 Experimental subjects

The experiment was marketed as a practical J2EE course for the computer science students at HUT through the university's news groups and Software Business and Engineering (SoberIT) laboratory's web pages. The course was arranged solely in order to get participants to the experiment. The pair programming experiment and the requirement of using pair programming by half of the participants were mentioned in the course brochure. The only mandatory prerequisite for participation was the basic knowledge of Java programming language. The course was not mandatory for anyone. Students majoring in Software Engineering were allowed to include the course to their major subject studies, but for others it was a totally optional course.

Twenty-four students signed up for the course. Four of them gave up after the training period, because they were not able to allocate the required effort for the project. Thus the total number of participants assigned to the teams was 20. They were all at least 4th year students at HUT, and the time during which they had been actively doing programming tasks varied between 1.5 and 10 years (average 4.7 years) of which 1-6 years (average 2.2 years) was using Java. The average grade from their previous programming courses was

[6] Only non-blank lines of code within method bodies included.

3.9 on scale 1-5 (5=best), and they all personally considered being similar or somewhat better than average programmers when compared to their fellow computer science students at HUT. As most of the computer science students at HUT work at software companies during their studies the participants can be claimed to be similar people to those developing software in industry.

Before the experiment the participants were ranked by their programming skills. A comparison value for ranking was calculated for each participant based on the time she spent on the two personal J2EE programming assignments, her previous programming experience, her average grade from programming courses, and her personal opinion on her capability as a programmer compared to her fellow computer science students at HUT. After this five names at a time were taken from the top of the list and randomly assigned into five different teams. Thus the result was five teams with one person from each skill quartile.

Finally three teams were randomly selected to act as the PP teams and two as the SP teams. The number of the PP teams was higher because the total number of the teams was uneven and we were more interested in getting observations from the PP teams than from the SP teams.

### 3.2.4 Development process

The participants had to follow certain work practices. The practices were described in a mostly pre-written project plan document (see Appendix D) given to the teams and on a 1-hour lecture. The project plan also listed the goals of the project in the following priority order:

1) follow the defined work practices
2) report the required data in a disciplined fashion
3) minimize the amount of bugs
4) implement as many use cases as possible
5) do not waste effort on unnecessary things

The project effort was fixed to 400 hours, i.e., 100 hours per person. Each person had to spend at least 75% of her hours in team sessions lasting 4-8 hours. Co-location of the team was required for all team sessions. It is the natural setting of developing software and otherwise the pair programmer teams would have automatically had lots of co-location and other teams probably not because typically students have problems arranging times for team work sessions.

The projects were divided into three phases as shown in Figure 1. The first phase was project planning and studying lasting one week and requiring about 10 hours per person. It was followed by two implementation iterations each lasting four weeks and requiring about 45 hours per person. The teams were allowed to divide the effort as they wanted during the iterations, but they had to plan the weekly effort usage and the times of the team sessions in the beginning of the project. In the end of both iterations, the teams had to deliver certain documents and all code to the course organizers. The code was delivered by delivering the whole version control system repository.

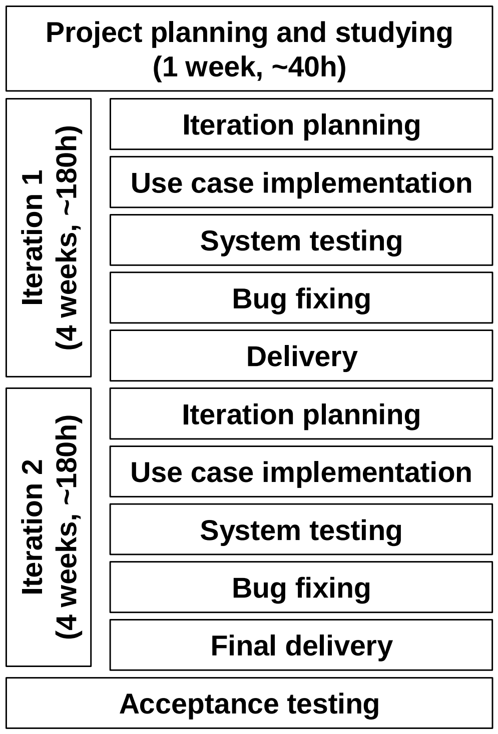


**Figure 1** The development process.

The mandatory practices included co-located team sessions, iteration planning, steps for iteration execution (implementing use cases, system testing, fixing bugs, updating technical specification), collective ownership, version control, coding standard, continuous refactoring, unit testing, system testing, time reporting, defect reporting, source code size reporting and documenting. For half of the teams pair programming was a mandatory practice and other teams were not allowed to use it for more than occasional collaboration. All the teams had to implement the use cases in the same order. Using any existing code other than that provided by the course organizers was forbidden. Communicating with the other teams was not allowed. The detailed guidelines for these work practices are included in the project plan in Appendix D.

### 3.2.5 Instrumentation

The effort data was collected using a web-based system called Trapoli developed at HUT. Each participant reported her work hours per each task immediately after a work session. Typical tasks, e.g., implementing a certain use case, were preconfigured to the system. For each reporting entry a work type such as programming, pair programming or testing was required. The same system was used to collect the task effort estimates in the beginning of the iterations and the updated estimates just before starting the implementation of a task.

The use of the CVS version control system was mandatory and source code was collected for analysis by collecting the whole CVS repositories in the end of both implementation iterations. Check-in timestamps in the repositories can be used to cross-check the reliability of the time reporting entries.

Each team had to report found defects by writing them down into a simple table. Defects related to unfinished use cases were not reported. Each report had to contain a short description of the defect, related use case and source code class, finding activity (development/system testing), who found the defect and when, and the current status (open, fixed).

Two inquiries were made using web forms (see Appendix E). The first collected the background information of the participants before the projects, and the second asked some opinions about the project and the developed system after the project.

One of the participants was a graduate student, who had a special assignment of making notes on how his team followed the process and reporting guidelines. His notes can be used to evaluate the reliability of the reported data.

## 3.3 Realities affecting the experimental design

The goals for the experimental design were to have a close to industrial like context (e.g., team size, the skills of the developers, size and complexity of the developed software, and the type of the development process and practices) and be able to collect the data reliably. Arranging full control of the participants was of course impossible and therefore special care was needed for ensuring that the experimental design supported getting reliable data and enforcing compliance to the defined work practices.

### 3.3.1 Team size

It is not easy to get a large number of students to a voluntary course requiring both a large amount of effort and working tightly in a large team during times suitable for all team members. In theory the team size could have been four or six or even eight, but because the expected number of students was low and the software size unnatural for a larger team, four was selected. Four was also easier for the students because they had to arrange several co-located team sessions. However, a larger team would have been a more realistic choice when compared to industry and would have also provided a more interesting setting for studying knowledge transfer within the team.

### 3.3.2 Project topic

There are several requirements for the technologies and the domain for this kind of an experiment. They have to be interesting from the perspective of the students in order to get a high number of participants. They should be ageless in order to be able to replicate the experiment later. Platform independency and free development tools also improve the possibilities for replication. The domain has to be familiar to the participants or simple enough to be described unambiguously in order to be able to move to the development work soon. The technologies have to be similar to those used in the industry in order to be able to generalize the results. Objective testing of the developed software must be possible and preferably efficient in order to evaluate the quality. Building the casino system using the J2EE technologies matches most of these requirements quite well.

A software project contains either developing further some existing software or starting the project from scratch. When considering an experiment, a larger size of existing software would bring more realism to the project, especially when the experiment covers only quite a limited amount of effort. Existing software would also enforce or guide students in a certain direction what comes to the software architecture, which is good for the experiment, because big differences in the architecture may be major factors affecting project effort and software quality later in a project. However, getting familiar with existing software requires lots of effort from the developers, which would make the effort available for the development work smaller. It may also be more complicated to design an assignment on top of existing software.

I chose to provide a small, well-documented implementation of core architecture in order to force certain aspects of the architecture and to alleviate the steep learning curve of the J2EE technology. I was fortunate enough to have two professional J2EE developers to prepare the core architecture and give training of the architecture and J2EE to the participants.

### 3.3.3 Facilities

In an optimal, fully controlled setting all teams would have their own working room, and a researcher would always be present observing the work. In this experiment there were one large room with 12 computers and one small room with two computers available. The rooms were open for the students 24 hours a day 7 days a week and the students were encouraged to avoid simultaneous work even in the larger room. Only the participants in this experiment used the rooms during the experiment. All the work stations were similar. They were powerful enough for fluent development work and had 19" monitors. Large tables and rolling chairs provided a proper environment for pair programming. The students were allowed to work at home with non-development tasks such as system testing for at most 25% of the project effort.

### 3.3.4 Development process

In order to be able to evaluate the effects of the pair programming practice the software development process and all the other development practices had to be as similar as possible in all teams. Collecting reliable data also presumes something from the process. Making sure that the participants understand and follow the defined process sets several requirements for the process as described in Table 4.

Where the process conformance can be clearly observed, it can be ensured quite well by the threat of failing the course if not following the process. Elsewhere it is necessary to trust the morale of the participants after emphasizing them the importance of following the process.

**Table 4** The requirements for the development process.

| Requirement | Reasoning/consequences |
|---|---|
| The amount of process discipline required from the teams should be rather low so that the process is realistic and easy to follow. | • Voluntary process conformance is crucial due to the lack of full control during the project. If the process does not feel like a natural way of working the conformance suffers.<br>• Collecting the data needed for the experiment unavoidably causes some bureaucracy, but all unnecessary bureaucracy should be minimized. |
| The process should be incremental with fixed dates for the iterations' ends. | • Deadlines force the teams to divide work more evenly during the project and to converge the work into a working product.<br>• Common deadlines simplify course arrangements. |
| Two of the three project control variables, effort, scope and quality must be fixed. | • Fixing simplifies data analysis. For example, comparing the productivity is very hard if all variables have different values between the teams. |
| Effort and quality must be fixed, and thus scope is variable. | • Effort is fixed because then the students know the required effort beforehand.<br>• Quality is fixed to high because it is a realistic and quite unambiguously communicable choice. |
| The use cases must be implemented in the specified order, one use case at a time per individual/pair. | • The same implementation order simplifies comparing the achieved scope and thus the productivity between teams. |

## 3.4 Summary

The experiment presented here tried to answer several research questions related to the effects of pair programming. The hypotheses related to the research questions were derived based on literature and my personal experiences. There are many challenges in measuring the effects proposed in the hypotheses. These challenges and the resulting measures were discussed after each hypothesis. The experiment was performed at Helsinki University of Technology with five four-person teams as subjects. The subjects were older students having quite a lot of programming experience. Each team used 400 hours for developing a software system based on the same requirements specification given to all teams. Three of the teams were using pair programming for all development work. All teams followed a similar development process. A time reporting system, a version control system, defect sheet and web inquiries were used to collect the data required by the research. Even though aiming for as realistic setting as possible the course context set certain limitations to this kind of an experiment.

# 4 Results and discussion

This chapter presents the results of the experiment. The results are also discussed and compared to previous research. Threats to the validity of the results are described in the next chapter.

All five teams finished the projects according to the schedule and spending the required 400 hours of effort. However, the analysis contains only two pair programming teams (PP1 and PP2) and two solo programming teams (SP1 and SP2). The third pair programming team abandoned the use of pair programming in the middle of the project, and was therefore removed from the analysis. The removed team was the least successful team based on the amount of use cases they were able to finish.

## 4.1 Productivity

**H 1.1:** The PP teams have higher project productivity than the SP teams.

**H 2.1:** The PP teams have lower use case productivity than the SP teams.

The number of implemented use cases by each team can be seen in Table 5. The PP2 team had started four more use cases, but three were not finished and one was rejected in the acceptance testing of the delivered system. All the other teams had started one more use case, and all the implemented use cases were accepted.

Both SP teams finished more use cases than the PP teams. The differences in the amounts of implemented use cases are enlarged by the smaller implementation effort required for the latter use cases (see Figure 3). Therefore we estimated the size of each implemented system in addition to the number of use cases by summing up the sizes of all implemented use cases. The size of each use case was estimated by the median of the effort different teams spent on implementing the use case. The last row of Table 5 shows the sum of the sizes of implemented use cases. Based on this data the project productivity of the PP teams was 29% lower ((190-266)/266) corresponding to 41% higher effort. This finding refutes hypothesis 1.1.

**Table 5** Use cases implemented by the different teams (μ=mean).

| | **PP1** | **PP2** | **SP1** | **SP2** | **$\mu_{PP}$** | **$\mu_{SP}$** | **PP vs. SP** |
|---|---|---|---|---|---|---|---|
| Number of implemented use cases | 20 | 10 | 25 | 27 | 15 | 26 | -42% |
| System size (i.e. sum of use case sizes) | 226 | 154 | 258 | 273 | 190 | 266 | -29% |

The PP1, SP1 and SP2 teams spent about the same amount of hours for use case implementation work (250h, 250h, 266h) but PP2 only 229h. The rest of the 400h effort was spent for non-implementation tasks (see Figure 2). PP2 spent considerably more effort than other teams in studying tools and technologies. However, if PP2 had been given the same amount of effort to spend on use cases they would have implemented only 5-6 more use cases, still far less than the others did. Both of the PP teams reported more effort than the SP teams on bug fixing and re-testing activities in the end of iterations, but the reason for this is mainly that the SP teams reported many hours for this task instead of a specific use case implementation task already before system testing. The SP teams reported hours for bug fixing and re-testing tasks only after system testing.

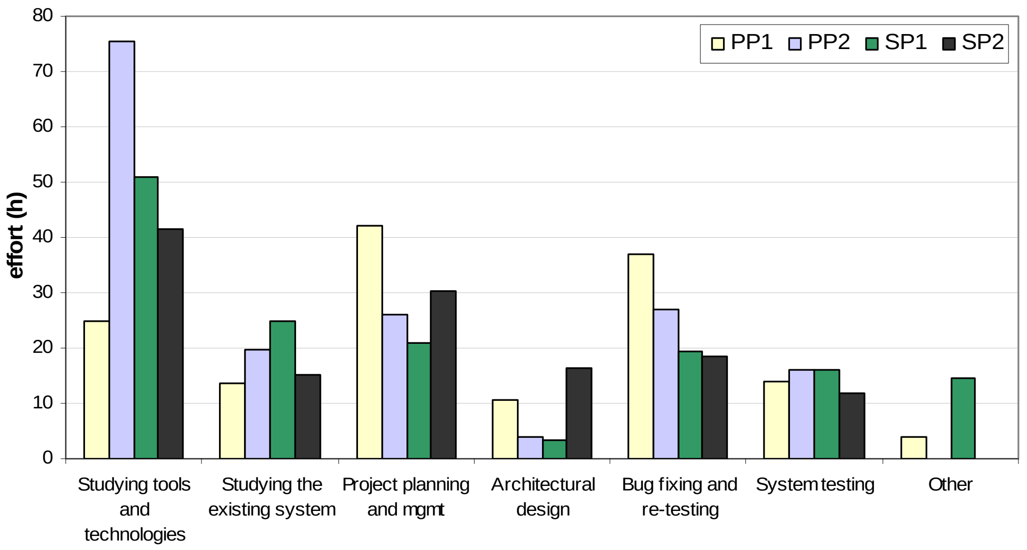


**Figure 2** Effort spent on other than use case implementation tasks.

The implementation effort of each finished use case is shown in Figure 3. There is a gap in the line of the SP1 team, because they did not finish use case 22. Both of the PP teams performed really poorly in implementing the first three (PP2) or four (PP1) use cases. Thereafter the differences in effort between the PP and SP teams are minimal. Actually, PP1 spent for their last use cases (17-20) less effort than either of the SP teams.

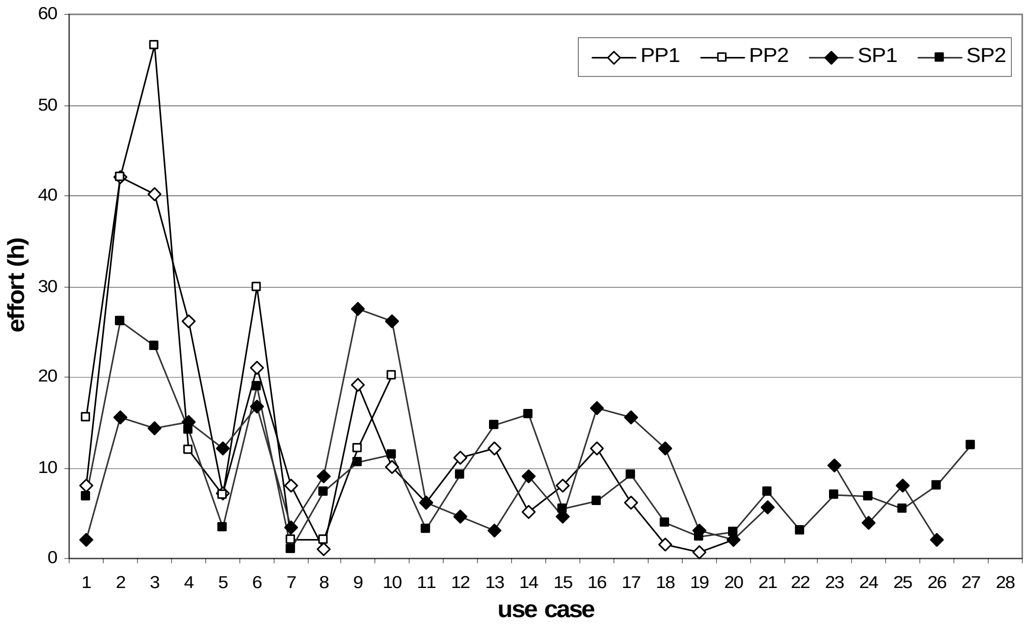


**Figure 3** The effort used per use case.

Table 6 shows the implementation efforts for different subsets of the use cases. For use cases 1-10, which were implemented by all four teams, the PP teams used on the average 44% more effort than the SP teams. If we leave out use cases 1-4, where the PP teams performed very poorly, the PP teams used 5% less effort than the SP teams.

**Table 6** The efforts spent for different subsets of the implemented use cases.

| **Use cases** | $\mu_{PP}$ | $\mu_{SP}$ | **PP vs. SP** |
|---|---|---|---|
| 1-10 | 191h | 133h | +44% |
| 1-4 | 121h | 59h | +107% |
| 5-10 | 70h | 74h | -5% |

The reason for the lower project productivity of the PP teams seems to be the considerably larger effort they spent on the three or four first use cases. This decreased the whole project's productivity. Later in the projects the PP teams finished use cases with about the same or even lower effort as the SP teams. It must be noted that because the PP teams spent much time with the first use cases, they may have learned something that helped them also with the later use cases. The SP teams also spent slightly more time for bug fixing, which actually meant additional effort related to the use cases, but is not included in the efforts in Table 6. We can argue that pair programming is more productive for a project, if the project is long enough to compensate the learning time inefficiency or if there is no learning time because people already know each other and have used pair programming before.

The PP teams had lower productivity in the three or four first use cases, but thereafter pair programming took same or smaller effort than solo programming refuting hypothesis 2.1. There was almost no difference in the amount of effort spent for non-implementation tasks. Therefore the advantages of pair programming for the project's productivity seem to be based on the decreased use case implementation effort after the learning time.

Previous research has also noticed the learning time with pair programming, but there is no data about the reason for it, i.e., is it, e.g., due to learning to use pair programming for the first time or learning to work with a new partner. In Williams' (2000) experiment the pairs did much worse in the first assignment (60% effort increase) than in the later assignments (15% effort increase) with the same pair. In the experiment by Ciolkowski and Schlemmer (2002) the additional effort for pair programming was 9% in both studied iterations indicating no learning effect. However, in that experiment the developers had worked as a team for several weeks before the observed iterations, inspecting and modifying the requirements and design documents. It seems that this kind of teamwork removes the inefficient learning time for pair programming. In my experiment at least four use cases were implemented before everyone had pair programmed with everyone once, which might explain the poor performance of the PP teams with the first three or four use cases.

**H 2.2:** Higher use case complexity favors PP teams as measured on use case productivity.

The diamonds in Figure 4 show the complexity of each use case as evaluated by the solo programmers (1=very easy, 5=very difficult). The squares show the difference in use case implementation effort between the SP and PP teams. If there were a correlation between the complexity and efficiency of pair programming the curves should follow each other. However, there is no correlation (r=-0.02) between the two variables, refuting H 2.2.

The lack of correlation contradicts with the recommendation by Williams and Kessler (2002) and the opinion of pair programmers with whom I have talked. It may be that the feeling of the usefulness of pair programming comes from the assumed higher resulting quality, and thus developers' impressions are not solely based upon effort differences.

The researchers of the social facilitation theory dealing with the impact of social presence on individual performance have found that social facilitation effects impair performance in case of complex tasks (Aiello and Douthitt, 2001). These studies have focused on studying persons who are not familiar with each other, which was also the case in the early phase of our experiment, where the pairs performed poorly.

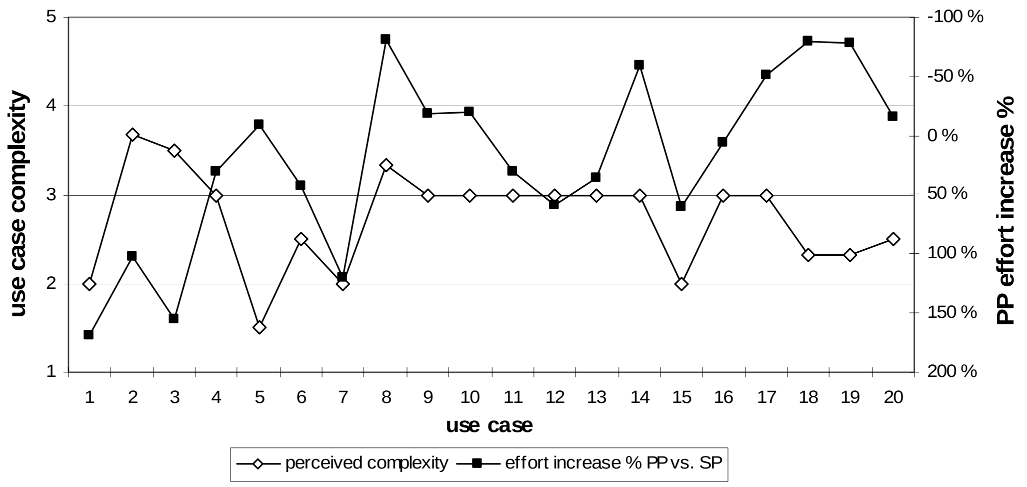


**Figure 4** Perceived complexity of a use case vs. effort increase of pair programming.

There may be a problem in analyzing the data this way, because the productivity of a team probably increases during the project due to learning of the domain etc. The increase may happen faster in the PP teams if the claim of better knowledge transfer in the PP teams is true. In addition, learning to do pair programming may be a factor increasing the productivity of the PP teams during the project as was seen in Figure 3.

## 4.2 Defects

**H 3.1**: After coding and unit testing the PP teams have fewer defects than the SP teams.

**H 3.2**: After system testing and bug fixing the PP teams have fewer defects than the SP teams.

Defects found by a team itself during system testing at the end of an iteration are called pre-delivery defects. Defects found during development are also counted as pre-delivery defects if they are related to a use case, which the responsible developer/pair considered to be ready. Defects found after delivery by an external tester are called post-delivery defects. The amounts of pre- and post-delivery defects normalized by the number of implemented use cases are listed in Table 7.

**Table 7** The amount of defects per implemented use case.

| | PP1 | PP2 | SP1 | SP2 | $\mu_{PP}$ | $\mu_{SP}$ | PP vs. SP |
|---|---|---|---|---|---|---|---|
| Pre-delivery defects | 0.95 (19/20) | 0.75 (9/12) | 1.60 (40/25) | 0.78 (21/27) | 0.850 | 1.189 | -29% |
| Post-delivery defects | 0.30 (6/20) | 0.33 (4/12) | 0.12 (3/25) | 0.04 (1/27) | 0.317 | 0.079 | +303% |
| Sum | 1.25 (25/20) | 1.08 (13/12) | 1.72 (43/25) | 0.81 (22/27) | 1.167 | 1.267 | -8% |

The SP teams found more pre-delivery defects than the PP teams, but there may have been differences in the system testing even though the total effort for it between the teams was similar. The last row of the table lists the sum of pre- and post-delivery defects and is the best estimate of the total number of defects in the code at the point of time when the responsible developer(s) considered it ready. According to this data, pair programmers made 8% less defects to the code during the development supporting hypothesis 3.1.

Interestingly, after delivery the PP teams had considerably more defects than the SP teams, refuting hypothesis 3.2. The SP teams found and fixed a larger proportion of defects before the delivery than the PP teams. The percentual difference is huge, because the absolute amounts of post-delivery defects were so small. The small number is partially explained by the focus of the post-delivery system testing, performed by the author of this thesis, on the basic functionality and common error situations instead of more exotic error situations, which the teams themselves had tested. The PP teams may have had a less careful attitude towards finding defects during system testing, because they may have relied too much on the peer-review during pair programming.

## 4.3 Design quality

**H 4.1:** The PP teams create better software design than the SP teams.

The source code of the final systems was analyzed using Metrics 1.3.5[7] plug-in for the Eclipse IDE. All the systems contained lots of code comments (4200-5600 lines) from which a large portion was used for generating J2EE bean classes automatically using the XDoclet tool. Neither the comments nor the automatically generated code was included in the analysis. Table 8 shows the analyzed metrics for all final systems and the core architecture on top of which all systems were built. The LOC metric contains only non-blank and non-commented lines inside method bodies. It must be noted that each system contains a different amount of use cases, and thus different amounts of code.

All the method level metrics, when analyzing the averages of all methods within a system, show slightly better values for the PP teams. However, it may be that the values correlate with the LOC of the system, which was higher for the SP teams. Therefore it is not possible to say whether the smaller values are due to smaller LOC or due to use of pair programming.

Analyzing the percentage of bad methods should be less dependent on the size of the software. The proportion of very long methods is smaller for the PP teams. The proportion of very complex methods is clearly smallest for PP2, but for other teams there are no differences. The proportion of methods with a long parameter list is clearly best for SP2 and worst for PP1.

Based on this analysis it is not possible to draw any conclusions on the effects of pair programming on the design quality. The differences and superiority between the PP and SP teams depend on the metric used, and the metrics may be affected by the size of the analyzed software, which was larger in both SP teams. Thus, we cannot say anything conclusive regarding hypothesis 4.1.

[7] Available at http://metrics.sourceforge.net/

**Table 8** Source code metrics.

| Metric | CORE | PP1 | PP2 | SP1 | SP2 |
|---|---|---|---|---|---|
| **System level metrics** | | | | | |
| Number of use cases | 0 | 20 | 10 | 25 | 27 |
| LOC | 1022 | 4180 | 2635 | 4956 | 5550 |
| Number of methods | 202 | 565 | 525 | 643 | 704 |
| **Method level metrics** | | | | | |
| LOC (avg.) | 4.82 | 7.16 | 4.96 | 7.61 | 7.67 |
| McCabe cyclomatic complexity (avg.) | 1.69 | 2.25 | 1.71 | 2.35 | 2.36 |
| Number of parameters (avg.) | 0.72 | 0.72 | 0.64 | 0.73 | 0.80 |
| **Percentage of bad methods** | | | | | |
| LOC > 50 | 0.00% | 2.12% | 0.76% | 2.33% | 2.27% |
| McCabe cyclomatic complexity > 10 | 0.50% | 2.30% | 0.95% | 2.49% | 2.27% |
| Number of parameters > 5 | 0.99% | 1.77% | 0.76% | 0.78% | 0.57% |

## 4.4 Knowledge transfer

After the project all developers were asked to evaluate for each Java package their:

- involvement, i.e., how much did you participate in its development
- understanding, i.e., how well do you understand its internal structure

The scale was: none (1), little (2), some (3), quite a lot (4), very much (5). The individual answers are listed in Appendix F. All teams had the same ten packages originating from the core architecture given to them.

**H 5.1:** Higher involvement in the development of a module leads to higher understanding of that module.

There was a high correlation (r>0.5) between the involvement and understanding for eight of the ten packages, when taking into account all levels of involvement and understanding. This finding supports hypothesis 5.1.

Figure 5 shows the average number of persons per package in the PP vs. SP teams, who were involved “quite a lot (4)” or “very much (5)” in the development of the package. The PP teams had a higher value for six packages, the same value for two packages and a lower value for only two packages. In the PP teams, on average 1.3 of the four persons were involved at least “quite a lot” in the development of each package, compared to 1.1 in the SP teams.

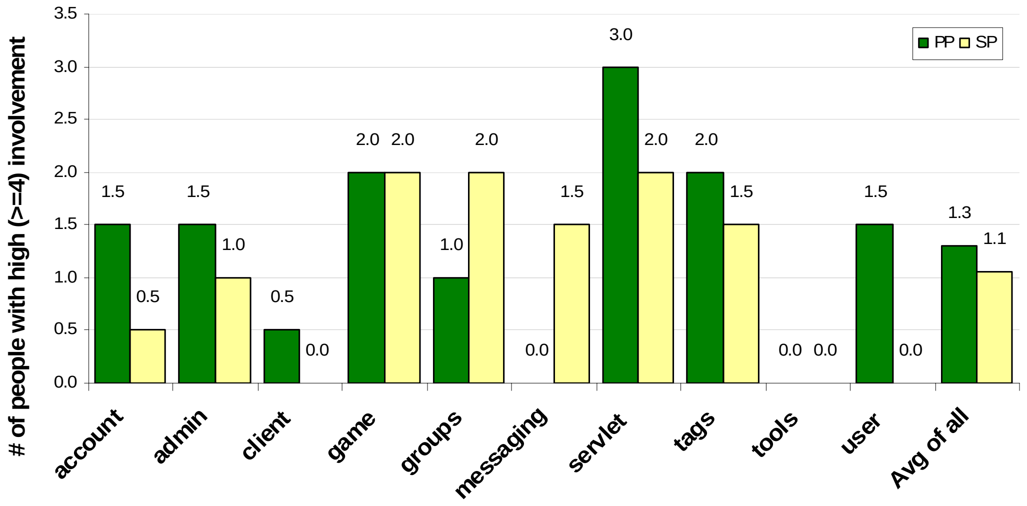


**Figure 5** The distribution of the involvement to the development of each module.

**H 5.2:** In the PP teams more developers understand each module well than in the SP teams.

Figure 6 shows the average number of persons per package in the PP vs. SP teams, who understood "quite a lot (4)" or "very much (5)" of the package. The PP teams reported higher values for seven, the same for two and lower for one package than the SP teams. In the PP teams, on average 1.8 of the four persons understood at least "quite a lot" about each package. In the SP teams the value was 1.4, supporting hypothesis 5.2.

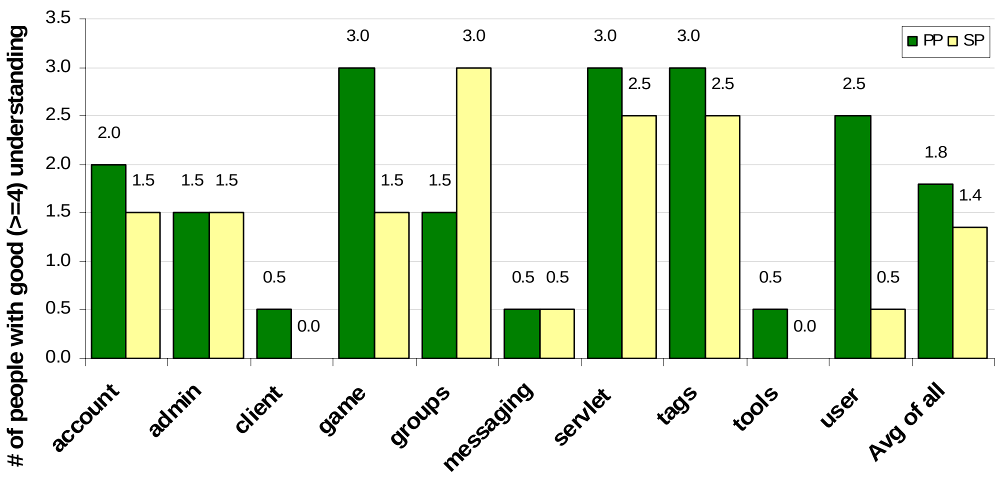


**Figure 6** The distribution of the understanding of each package.

**H 5.3:** In the PP teams each developer understands more modules well than in the SP teams.

Table 9 shows the number of packages that the PP vs. SP developers on the average understood with at least a certain depth of understanding. For example, on the average the developers in the PP teams understood at least "little" about 8.9 packages. The differences between the PP and SP teams are quite small and depend on the depth of understanding. When analyzing good depth of understanding (≥4), the PP developers knew 32% (4.5 vs. 3.4) more packages compared to the SP developers. This supports hypothesis 5.3, but one must note that for the other thresholds of the depth of understanding the situation is different.

**Table 9** The number of the packages understood with at least a specific depth.

| Depth of understanding | Number of packages (max=10) | |
|---|---|---|
| | **Average of PP developers (N=8)** | **Average of SP developers (N=8)** |
| ≥ little (2) | 8.9 | 8.5 |
| ≥ some (3) | 6.9 | 7.1 |
| ≥ quite a lot (4) | 4.5 | 3.4 |
| = very much (5) | 0.8 | 1.0 |

**H 5.4:** In the PP teams developers achieve deeper understanding of a module than in the SP teams with the same amount of involvement with that module.

Table 10 shows the averages of each developer's all package evaluations. In the PP teams a slightly higher (2.8 vs. 2.6) involvement in the development of different packages lead to a slightly higher (3.1 vs. 3.0) depth of understanding than in the SP teams. The difference was larger in the involvement than in the understanding, which refutes hypothesis 5.4. It seems that the developers in both kinds of teams achieve about the same depth of understanding after a similar amount of involvement.

**Table 10** The averages of each developer's all package evaluations.

| PP teams | | | | SP teams | | |
|---|---|---|---|---|---|---|
| **Dev.** | **Involvement** | **Understanding** | | **Dev.** | **Involvement** | **Understanding** |
| P1 | 3.0 | 3.4 | | S1 | 2.6 | 2.7 |
| P2 | 3.0 | 3.0 | | S2 | 2.6 | 2.9 |
| P3 | 2.6 | 3.1 | | S3 | 2.2 | 2.0 |
| P4 | 3.2 | 3.6 | | S4 | 2.4 | 3.4 |
| P5 | 2.4 | 2.3 | | S5 | 2.8 | 2.7 |
| P6 | 2.7 | 3.2 | | S6 | 2.6 | 3.3 |
| P7 | 2.5 | 2.0 | | S7 | 3.0 | 3.6 |
| P8 | 3.0 | 4.2 | | S8 | 2.9 | 3.4 |
| Avg. | 2.8 | 3.1 | | Avg. | 2.6 | 3.0 |

Analyzing the answers of all individual developers increased the sample size from four teams to 16 developers and allowed us to make a statistical analysis of the significance of the differences between pair programmers and solo programmers. According to the Mann-Whitney U-test (Siegel 1956) none of the differences related to knowledge transfer were statistically significant, which was quite natural due to the small sample size.

## 4.5 Enjoyment of work

**H 6.1:** In the PP teams developers enjoy their work more than in the SP teams.

After the project, the developers were asked in a web questionnaire about their feelings about the way their team did the development work in general (Table 11). The scale was from "1-It was terrible" to "5-I liked it a lot". The most satisfied developers were in the SP2 team. However, seven developers in the PP teams liked the way they worked (answered 4 or 5), but in the SP teams only five had this opinion. Thus the data gives some support for H 6.1

**Table 11** The enjoyment of the way the team did the development work.

| Team | Developers' enjoyment |
|---|---|
| PP1 | {5, 4, 4, 4} |
| PP2 | {5, 4, 4, 2} |
| SP1 | {4, 3, 2, 2} |
| SP2 | {5, 5, 5, 5} |

Two other questions were also asked (Table 12). Three of the eight developers in the PP teams preferred pair programming and four solo programming. However, only two considered pair programming the more successful choice for this kind of a project.

**Table 12** The feelings of the developers (N=8) in the PP teams about pair programming.

| Question | PP | SP | Neutral |
|---|---|---|---|
| Which do you *like* more, pair programming or solo programming? | 3 | 4 | 1 |
| Which do you consider better for the overall success of this kind of a project? | 2 | 5 | 1 |

## 4.6 Effort estimation

The initial effort estimates for the use cases were made in the beginning of an iteration by the team collectively. The estimates were updated just before the implementation of the use case by the responsible individual or pair.

**H 7.1:** The PP teams estimate use case efforts more accurately than the SP teams in the beginning of an iteration.

Table 13 shows the sum of the estimation errors for each team. Both PP teams and the SP1 team did quite well in the estimation in the beginning of the iterations. The SP2 team spent considerably less effort than they thought. The bad estimates by the SP2 team give some support for hypothesis 7.1.

**Table 13** The sum of the estimation errors for all use cases each team implemented.

| Team | Sum of the errors in the initial estimates |
|---|---|
| PP1 | -61h |
| PP2 | -39h |
| SP1 | 59h |
| SP2 | -205h |

**H 7.2:** Pairs estimate use case efforts more accurately than solo programmers just before its implementation.

Table 14 shows the number of good updated estimates each team made. A good estimate means here that the estimate was at most 50% higher or lower than the realized effort. The analysis contains only use cases 1-10 that were finished by all the teams. The estimates made by solo programmers compared to those made by pairs are clearly better (SP avg. 75% vs. PP avg. 45%). The differences are almost similar even if the first four use cases, where pair programming was most inefficient were removed. The finding contradicts with hypothesis 7.2. It may be that people are quite experienced in estimating their own work, but new pair programmers do not know how they should consider the fact that two people participate in the development work.

**Table 14** The amount of good updated estimates (error max. 50%, use cases 1-10).

| Team | Good updated estimates |
|---|---|
| PP1 | 60% (6 of 10) |
| PP2 | 30% (3 of 10) |
| SP1 | 60% (6 of 10) |
| SP2 | 90% (9 of 10) |

## 4.7 Summary

The SP teams had clearly better project productivity. However, the difference was largely due to the PP teams spending considerably more effort for the first three or four use cases. This phenomenon was probably caused by the learning time involved before pair programming becomes efficient. Later in the projects the PP teams spent same or even slightly smaller amount of effort for implementing use cases than the SP teams. The claim that pair programming would be most useful with complex tasks was not supported by this experiment at least from the perspective of productivity.

The code written by pair programmers contained fewer defects per use case when the teams started system testing. However, the SP teams were much more successful in finding and fixing the defects and in the end of the project they delivered systems with lower number of defects per use case. The code from the PP teams had slightly better quality measured by the method size and method complexity metrics. However, the explanation behind this difference may be the potential correlation of the software size and these metrics. The SP teams developed larger systems, and therefore these metrics may show worse values for them.

In the SP teams developers had high involvement with more packages than developers in the PP teams. Probably related to this, in the PP teams each package was understood well by more developers, and each developer understood more packages well in the PP teams.

When asking for opinions from the developers which they like more solo programming or pair programming, both alternatives got about the same number of supporters. However, most people liked working in projects using pair programming.

Both PP teams and the other SP team made good initial effort estimates, but the other SP team made a large estimation error. However, solo programmers were more often successful than pair programmers with their updated estimates made just before the implementation of a use case.

# 5 Evaluation of the experiment

This chapter analyzes different threats to the validity of the results. Then the strengths of the experimental design are discussed and finally improvements to the experimental design are proposed.

## 5.1 Threats to internal validity

The average skill level of the teams was balanced, but there may have still been skill differences affecting the performance of the teams. The skills of a team's most skillful person may be very important in a project including learning quite a challenging new technology. A team may save lots of effort when they have someone who can solve the hardest problems quickly. The two most experienced developers having ten years of programming experience were in the two most productive teams SP1 and SP2, and the third experienced one (eight years) in the third productive team PP1. In the last two teams the most experienced developer had seven or six years of programming experience.

The participants were not under full control of the researcher during the project. This may have affected how disciplined they were in following the development practices and reporting requirements. It has also been proposed that pair programmers have more discipline in following the process, which may have caused bias between solo and pair programming teams. The change log data in the version control system has not been analyzed for cross-checking the reliability of the time reporting data.

The realized effort was collected per use case, but allocating the hours for a certain use case was not necessarily uniform between the teams when use cases related to each other were implemented. For example, different teams may have reported architectural work related to several use cases in a different way. Also some of the hours that were reported to bug fixing and re-testing tasks before system testing should have been allocated to the related use cases. These problems did not affect the project productivity analysis, but may have affected the use case productivity analysis.

The differences in the project productivity were quantified by measuring the sizes of the use cases based on the average effort spend on implementing them. This is not an accurate measure, and for the latter use cases it may have given too small values, because the size was estimated based on only the efforts spend by the most productive teams.

The analysis of the correlation between the use case complexity and the usefulness of pair programming may have been be inaccurate, because the productivity of the PP teams may have changed during the project in a different way than that of the SP teams. The productivity may increase faster in the PP teams if the claim of better knowledge transfer is true. In addition, the learning time related to pair programming may be a factor increasing the productivity of the PP teams during the project, but also making them less productive in the beginning of the project.

No guidelines were given to the teams on how to make effort estimates except that by whom and when they should be made. Some teams probably did not spend much thought for making the estimates. The tool for collecting the estimates was also clumsy, and some teams had to be advised during the project to enter the estimates in the system in a correct way.

The acceptance testing was done by the author of this thesis, who knew the number of bugs the teams had found themselves and whether a tested system was done by an SP or PP team. This may have biased the testing activity.

Evaluating design quality with code metrics may have been unreliable, especially because the compared designs had different amount of functionality and code in them. There is no common understanding about the best code metrics indicating good design, and code metrics may be awkwardly affected by the total size of the software. The core architecture given to the basis of the projects also certainly affected the code metrics of the final systems.

The inquiry about the involvement to and understanding of the packages had many deficiencies. It was made after the project and through the web. It is not possible to say how many respondents checked the source code when answering, or how much time they spent answering the inquiry. Respondents may have had a different understanding of the scale used, e.g., they may have compared themselves to other members of the team. The biggest problem is that the answers were based on opinions, not on some objective test of understanding or on time reporting data of involvement.

## 5.2 Threats to external validity

The number of the teams was so low and the variations in the response variables within the SP and PP teams so high that there was no sense doing analysis of the statistically significance of the team level results.

The requirement for using pair programming for all coding tasks was not the most natural and probably also not the most beneficial choice. The optimal amount of using pair programming may be anywhere between using it for all development work by everyone and using it for nothing.

This study did not observe how the students really did pair programming, e.g., how actively they switched roles and communicated during the pair programming sessions. The students were told on a lecture how to do pair programming, but it may not have been enough for becoming an efficient pair programmer. For example, Dick and Zarnett (2002) report about a case where switching roles did not work in spite of frequent intervention by the team coach.

The proposed benefit of pair programming as a means of decreasing the amount of external interruptions during the development work may not become as apparent in a student project as in a typical office environment. In this experiment each team was having a kind of a meeting whenever they were doing team development sessions and there were not any co-workers around them. Therefore the number of interruptions was by default minimized.

Data about whether the team members were familiar with each other before the project, or what kind of personalities were in each team was not collected. Differences in these variables may have affected the success of the teams. Potential participants knew before entering the course that 50% of the participants must use pair programming. This may have kept away people who are strongly against pair programming.

The J2EE technology was new to the participants. Therefore the context was learning a new technology instead of routine development with familiar technologies. However, learning new technologies is not uncommon even in industry projects.

One of the PP teams abandoned the use of pair programming during the project, because they found it inefficient when the work involved lots of studying. They believed that their productivity would increase without pairing. This team was removed from the data analysis, because the data was not comparable. Their productivity was the worst of all teams. If they had continued using pair programming and the data were included in the analysis, the average productivity of the PP teams may have decreased.

## 5.3 Strengths of the experimental design

The experiment was done in project context with a moderately large effort and co-located teams. The requirement of co-location is important, because it alone may increase the knowledge transfer within a team considerably. Co-location also enforces simultaneous work between the developers making the development more realistic.

The participants quite well represented industrial professionals. Their programming experience was on average 4.7 years and many had worked as professional software developers during their studies. Also due to the content of the course, the participants were students who were likely to be interested and skilled in programming work.

The project topic and technologies enticed quite many students to voluntarily take the laborious course. The training and core architecture allowed the students the start real work quite soon, and there were almost no problems related to the requirements specification, core architecture or any other materials.

One of the participants was a graduate student, who made a special exercise by observing how well his team followed the process and how reliable the reported data was. His observations were valuable for analyzing the threats to the validity of the results and for conceiving improvements to the experiment.

Compared to what other researchers have previously been able to study, this experiment was worth carrying out. Ciolkowski and Schlemmer (2002) are the only researchers, whose experimental setting has been orderly planned and reported. However, they analyzed only the programming phases of the project, not including, e.g., design activities. They also failed in studying any defect metrics. Williams (2000) reported her results of the team experiment only very shortly and Baheti et al. (2002) did not use either a randomized study design or similar projects for the different teams. All the other previous studies have concentrated on observing pairs as isolated entities.

## 5.4 Improvements to the experimental design

The long list of threats to the validity of the results indicates that the experimental design has space for improvements. The recommendations presented below are valuable for anyone who is thinks of replicating this experiment or designing a similar experiment. Some of the improvements can be easily implemented if considered already when planning an experiment. Others require additional resources compared to the experiment described in this thesis.

There should be more control on the process conformance. The researcher could participate to a couple of development sessions with each team in order to ensure they start using the mandatory practices. The researcher should make sure during the project that the data is reported on time and immediately ask for missing data. Later in the project the pair programming sessions should be observed somehow, at least randomly.

There should be a couple of exercises before the real project done by the teams in order to decrease the effects of getting familiar with other team members and learning new tools and technologies, which may decrease the productivity in the beginning of the project in a different way in different teams. In accordance with these exercises pair programming could be taught in practice by an on-site expert. By extending these exercises a little bit more the whole development process could be taught in practice by doing a mini iteration before the project.

Fixing the scope of the project instead of effort would simplify the analysis of design quality and defect counts. The students would also be even more motivated, when they had the possibility of finishing the project in less than the nominal effort required for getting the study credits. The trade-off is that the maximum amount of effort that may be required from a team is not known in advance, and also software quality may suffer, if the students hurry too much in finishing the work.

The analysis of the design quality could be improved by reviewing certain parts of the code by some external experienced developers. Even thought their findings would be subjective, the results might be more reliable than analysis made by code metrics.

An alternative experimental design would be to have all teams make two projects, one using pair programming and another using solo programming. All teams should use pair programming and solo programming in the same order, because the experience of using pair programming may affect how the team works afterwards more than if the team uses solo programming for the first project. Two projects would allow blocking the effect of different teams. This design would require either doubled effort from the participants or smaller projects. Also another assignment should be prepared.

Rotating the pairs only after finishing a task makes organizing the work difficult in a four-person team. When a pair finishes a task, the other pair is typically still working and the two have to find some tasks they can do alone until the other pair is ready. Rotating the pairs after each session would simplify the rotation, but it might be inefficient because one of each pair would not be familiar with the incomplete tasks. However, I have interviewed an industrial four-person project team where the rotation of the pairs after a constant time was considered a useful practice.

The inquiry about the understanding of the packages should be made in a controlled environment, and preferably so that the participants have the source code available. The complexity of the use cases should be asked at least initially immediately after finishing a use case.

Some guidelines for making the effort estimates should be taught and the estimates for each use case should be collected in a simpler way than into a clumsy time reporting system.

If there are top performers considerably better than the average participants they should not be assigned to the experimental teams unless all teams can be provided with a top performer with similar skills. Otherwise students of this kind should be collected to a team of their own and thus let them still participate to the course, but not the experiment.

The project was arranged as a part time course requiring about 10 hours/week. If a full time project of 2-3 weeks could be arranged, it would imitate better a real software project.

## 5.5 Summary

There are many threats to the validity of the results. The threats to the internal validity are related to, e.g., different top performers in the teams, limited control of the process conformance, difficulties in reporting work hours for different tasks in a similar way, measuring the scope of the project, potentially varying productivity during the projects, lack of guidelines for making effort estimates, potentially biased acceptance testing, use of code metrics for evaluating design quality and too loosely controlled inquiry of the understanding of the packages.

Threats to the external validity are related to the small number of teams, unnatural level of pair programming, lack of data on details within pair programming sessions, superficial training of pair programming, lack of data about people's familiarity with each other before the experiment, context as a student project, new implementation technology, and one of the teams abandoning the use of pair programming in the middle of the experiment.

Despite of the long list of threats to the validity of the results, this was one of the first experiments studying pair programming in a team context. The participants were also quite experienced programmers, even though they were students. The materials for the project were of high quality and there were no practical problems during the experiment. The results of this experiment should be interpreted carefully, but lots of ideas for improving the experimental design have been identified.

# 6 Conclusions

## 6.1 Summary and conclusions

This work studied the effects of pair programming on development effort, software quality, knowledge transfer, enjoyment of work, and effort estimation accuracy. The results certainly shed some more light on the topic, even though this experiment, like all the previous ones, contained several deficiencies such as the small sample size. Hopefully this work invites others to execute even better pair programming experiments.

The PP teams had 29% lower project productivity than the SP teams. However, the reason was the considerably larger effort they spent for the first three or four use cases. The inefficiency was probably caused by the learning time involved in getting familiar with new people and with the pair programming practice. Later in the projects the PP teams spent 5% less effort than the SP teams for implementing the use cases. If the inefficient learning time is not taken into account, the productivity of the PP teams seems to be equal to that of the SP teams. In a typical organization the learning time can usually be neglected because most people already know each other and at least after the first pair programming project are familiar with it. Even if there still were some learning time involved, the cost of a day or two per developer for the learning is insignificant. The claim that pair programming is most useful with complex tasks was not supported by this experiment at least from the perspective of the required effort.

The code written by pair programmers contained 8% less defects per use case when the responsible developers considered the code ready. However, the SP teams were much more successful in finding and fixing the defects and in the end of the project they delivered systems with lower number of defects per use case. This indicates that pair programmers write code with fewer defects, but the benefits may be lost unless careful system testing is performed.

The PP teams had slightly better design quality measured by the method size and complexity metrics. However, the explanation may be the potential correlation between software size and these metrics. The PP teams developed systems with less functionality, and therefore these metrics may show better values for them.

In the PP teams developers generally had high involvement with more packages than developers in the SP teams. Probably related to this, in the PP teams there were generally more developers (1.8 vs. 1.4) with good understanding of each package, and each developer understood more packages (4.5 vs. 3.4) well. These differences indicate better knowledge transfer within the PP teams.

Even though half of the developers in the PP teams enjoyed solo programming more than pair programming and half vice versa, most of them liked working in the PP teams. Thus developers' feelings about pair programming should not hinder its deployment.

Both PP teams and the other SP team made good initial effort estimates, but the other SP team made a large estimation error. However, solo programmers were more often successful than pair programmers with their updated estimates made just before the implementation of a use case.

It seems that the use of pair programming leads to fewer defects in code after coding and better knowledge transfer within the development team without requiring additional effort if the learning time can be avoided. These benefits are likely to decrease the further development costs of the system and increase an organization's productivity due to improved competence of the developers.

## 6.2 Future work

I will package all the necessary materials and publish them on the web in order to provide more help for those interested in replicating this experiment. I believe that with only minor modifications the experiment package can be used for studying some other development practice such as test driven development. I am also planning improving the experimental design and then replicating the experiment with a larger number of students at Helsinki University of Technology or in co-operation with some other educational institution.

My studies about pair programming will extend to case studies at companies using pair programming. In companies it is very challenging to arrange even quasi-experiments, but also case studies can give valuable information on, e.g., how pair programming should be done in practice.

# REFERENCES

Aiello, J., Douthitt, E., 2001. Social Facilitation From Triplett to Electronic Performance Monitoring, *Group Dynamics: Theory, Research and Practice*, **5**(3), pp. 163-180.

Arisholm, E., 2002. Design of a controlled experiment on pair programming, *ISERN 2002 Annual Meeting*. 30.9.-1.10.2002. Available: http://fc-md.umd.edu/projects/Agile/ISERN/Arisholm.ppt. Referenced 15.4.2005.

Baheti, P., Gehringer, E. and Stotts, D., 2002. Exploring the Efficacy of Distributed Pair Programming, *Extreme Programming and Agile Methods - XP/Agile Universe 2002*, August 4-7 2002, pp. 208-220.

Beck, K. Extreme Programming Explained. 2000. Addison-Wesley.

Boehm, B. and Turner, R., 2003. Balancing Agility and Discipline. Addison-Wesley.

Brooks, F., 1995. The Mythical Man Month: Anniversary Edition. Addison-Wesley.

Bryant, S., 2004. Double trouble: Mixing qualitative and quantitative methods in the study of eXtreme Programmers, *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing*, September 26-29 2004, pp. 55-61.

Cao, L. and Xu, P., 2005. Activity Patterns of Pair Programming, *Proceedings of the 38th Hawaii International Conference on System Sciences*, January 3-6 2005, pp. 88.1-10.

Canfora, G., Cimitile, A. and Visaggio, C., 2004. Working in pairs as a means for design knowledge building: an empirical study, *Proceedings of the 12th IEEE Workshop on Program Comprehension,* June 24-26 2004, pp. 62-68.

Ciolkowski, M. and Schlemmer, M., 2002. Experiences with a Case Study on Pair Programming, *Workshop on Empirical Studies in Software Engineering*, December 9-11 2002.

Cockburn, A. and Williams, L., 2000. The Costs and Benefits of Pair Programming, *Extreme programming examined*, Addison-Wesley, pp. 223-243.

Constantine, L., 1995. Constantine on Peopleware. New Jersey: Prentice Hall P T R.

Coplien, J.O. and Harrison, N.B., 2004. Organizational Patterns of Agile Software Development. Upper Saddle River: Pearson Prentice Hall.

Cusumano, M., MacCormack, A., Kemerer, C.F. and Crandall, B., 2003. Software Development Worldwide: The State of the Practice. *IEEE Software*, **20**(6), pp. 28-34.

DeMarco, T. and Lister, T., 1999. Peopleware, 2$^{nd}$ edition. New York: Dorset House Publishing.

Dick, A.J. and Zarnett, B., 2002. Paired Programming & Personality Traits, *Proceedings of Extreme Programming and Agile Processes in Software Engineering*, May 26-29 2002, pp. 82-85.

Domino, M., Collins, R., Hevner, A. and Cohen, C., 2003. Conflict in Collaborative Software Development, *Proceedings of the SIGMIS Conference '03*, April 10-12 2003, pp. 44-51.

Fenton, N.E. and Pfleeger, S.L., 1996. Software Metrics: A Rigorous & Practical Approach, Second Edition. International Thomson Computer Press.

Gallis, H., Arisholm, E. and Dybå, T., 2003. An Initial Framework for Research on Pair Programming. *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, 30.9.-1.10.2003, pp. 132-142.

Hanks, B., McDowell, C., Draper, D. and Krnjajic, M., 2004. Program quality with pair programming in CS1, *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education,* June 28-30 2004, pp. 176-180.

Heiberg, S., Puus, U., Salumaa, P. and Seeba, A., 2003. Pair-Programming Effect on Developers Productivity, *Proceedings of Extreme Programming and Agile Processes in Software Engineering*, May 25-29 2003, pp. 215-224.

Höst, M. and Wohlin, C., 1998. An Experimental Study of Individual Subjective Effort Estimations and Combinations of the Estimates, *Proceedings of the 20th international conference on Software engineering*, April 19-25 1998, pp. 332-339.

Jensen, R., 2003. A Pair Programming Experience. *CrossTalk*, **16**(3), pp. 22-24.

Juristo, N. and Moreno, A.M., 2001. Basics of Software Engineering Experimentation. Kluwer Academic Publishers.

Keefer, G., 2003. Extreme Programming Considered Harmful for Reliable Software Development 2.0. Available: http://www.avoca-vsm.com/Dateien-Download/ExtremeProgramming.pdf. Referenced 14.2.2005.

Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K. and Rosenberg, J., 2002. Preliminary guidelines for empirical research in software engineering, *IEEE Transactions on Software Engineering*, **28**(8), pp. 721-734.

Lui, K. and Chan, K., 2003. When Does a Pair Outperform Two Individuals, *Proceedings of Extreme Programming and Agile Processes in Software Engineering*, May 25-29 2003, pp. 225-233.

McCabe, T., 1976. A Software Complexity Measure, *IEEE Transactions on Software Engineering*, **2**(4), pp. 308-320.

McDowell, C., Bullock, H., Fernald, J. and Werner, L., 2002. The Effects of Pair-Programming on Performance in an Introductory Programming Course, *ACM SIGCSE Bulletin*, **34**(1), pp. 38-42.

McDowell, C., Hanks, B. and Werner, L., 2003a. Experimenting with Pair Programming in the Classroom, *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, 30.6.-2.7.2003, pp. 60-64.

McDowell, C., Werner, L., Bullock, H. and Fernald, J., 2003b. The Impact of Pair Programming on Student Performance, Perception, and Persistence, *Proceedings of the 25th International Conference on Software Engineering*, 3-10 May 2003, pp. 602-607.

Müller, M.M., 2004a. Are Reviews an Alternative to Pair Programming?, *Empirical Software Engineering*, **9**(4), pp. 335-351.

Müller, M.M., 2004b. Should we use programmer pairs or single developers for the next project?, *Technical Report 2004-8, Faculty of Informatics, Universität Karlsruhe*.

Müller, M.M. and Padberg, F., 2004. An Empirical Study about the Feelgood Factor in Pair Programming. *Proceedings of Software Metrics, 10th International Symposium on,* 11-17 September 2004, pp. 151-158.

Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C. and Balik, S., 2003. Improving the CS1 Experience with Pair Programming, *ACM SIGCSE Bulletin,* **35**(1), pp. 359-362.

Nawrocki, J. and Wojciechowski, A., 2001. Experimental Evaluation of Pair Programming. *Proceedings of the 12th European Software Control and Metrics Conference*, April 2-4 2001, pp. 269-276.

Nosek, J., 1998. The Case for Collaborative Programming. *Communications of the ACM*, **41**(3), pp. 105-108.

Padberg, F. and Müller, M.M., 2004. Modeling the Impact of a Learning Phase on the Business Value of a Pair Programming Project, *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, 30.11.-3.12.2004, pp. 142-149.

Padberg, F. and Müller, M.M., 2003. Analyzing the Cost and Benefit of Pair Programming, *Proceedings of the Software Metrics Symposium*, 3-5 September 2003, pp. 166-177.

Parrish, A., Smith, R., Hale, D. and Hale, J., 2004. A Field Study of Developer Pairs: Productivity Impacts and Implications, *IEEE Software*, **21**(5), pp. 76-79.

Rostaher, M. and Hericko, M., 2002. Tracking Test First Pair Programming – An Experiment, *Extreme Programming and Agile Methods - XP/Agile Universe 2002*, August 4-7 2002, pp. 174-184.

Siegel, S., 1956. Nonparametric statistics for the behavioral sciences. McGraw-Hill Kogakusha.

Stephens, M. and Rosenberg, D., 2003. Extreme Programming Refactored: The Case Against XP. apress.

Van Deursen, A., 2001. Program Comprehension Risks and Opportunities in Extreme Programming, *Proceedings of the Eighth Working Conference on Reverse Engineering*, October 2-5 2001, pp. 176-185.

Williams, L., 2000. *The Collaborative Software Process PhD Dissertation*, University of Utah.

Williams, L. and Erdogmus, H., 2002. On the Economic Feasibility of Pair Programming, *In International Workshop on Economics-Driven Software Engineering.*

Williams, L. and Kessler, R., 2002. Pair Programming Illuminated. Addison-Wesley.

Williams, L. and Kessler, R., 2000. All I Really Need to Know about Pair Programming I Learned In Kindergarten. *Communications of the ACM*, **43**(5), pp. 108-114.

Williams, L., Kessler, R., Cunningham, W. and Jeffries, R., 2000. Strengthening the Case for Pair-Programming. *IEEE Software*, **17**(4), pp. 19-25.

Williams, L., McDowell, C., Nagappan, N., Fernald, J. and Werner, L. Building Pair Programming Knowledge through a Family of Experiments, *Proceedings of the 2003 International Symposium on Empirical Software Engineering.* 30.9.-1.10.2003, pp. 143-152.

Williams, L., Shukla, A. and Antón, A., 2004. An Initial Exploration of the Relationship Between Pair Programming and Brooks' Law, *Proceedings of the Agile Development Conference*, June 22-26 2004, pp. 11-20.

Wilson, J., Hoskin, N. and Nosek, J., 1993. The Benefits of Collaboration for Student Programmers. *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education*, pp. 160-164.

# Appendix A. Requirements specification

This appendix contains the requirements specification given to the student teams.

## 1. Purpose of the document

The system to be built is a distributed, web-based, multiplayer card game gambling platform including implementations of certain card games.

The intended audience of this document is described in table 1.

**Table 1.** Possible readers of this document.

| Group of the readers | Reasons for reading |
|---|---|
| System developers | To understand what functions and properties the system must contain |
| Testers | To test the system against the requirements |
| Project team | To follow-up the status of the project against the requirements |

## 2. Business goals

The customer wants to start an on-line casino for the students at HUT in order to steal their money before they spend them on pizza and beer.

## 3. Main domain concepts

Table 2 describes the main domain concepts and figure 1 shows the relationships between them.

**Table 2.** Main domain concepts.

| Concept | Description |
|---|---|
| Casino | Casino is the place where everything in the system happens. |
| Player | Players are the main actors in the system. They play games, form groups, communicate with each other etc. in the casino.<br>A player can also act in the role of *a group boss* or *a table boss* for certain groups and tables. The boss is able to perform some additional operations. |
| Group | A group consists of players that often play games together. Each group has a boss, who can invite, accept and remove members to and from the group. |
| Account | Each player has an account, to which he can transfer money using his credit card number. The account is updated when the player spends or wins money in the casino. |
| Transaction | Whenever a player spends money (bets in a game, or performs an operation) or receives money (wins in a game round) a transaction is stored to his account. |
| Game | There are several different games in the casino such as Black Jack, video poker, stud poker etc. Each game has different rules which describe how the game is played: e.g. dealing and changing cards, betting, hand comparisons, min. and max. number of players etc. A game can be a *single player game* than is played alone or a *multiplayer game* that needs at least two |

| | |
|---|---|
| | players. |
| Table | Multiplayer games are played in tables. Only one type of game can be played in a table. Table is either *public* i.e. visible to all players or *private*. The order of *seats* in the table, i.e. who is sitting next to each other, is significant as stated in game rules. A player in a table may be either active or passive (not participating to a game round). One of the players in a table is the table boss. The players in a table may or may not belong to the same group. |
| Round | When a game is played there are consecutive game rounds. During one round the cards are dealt, (changed), bets placed and winner determined. |

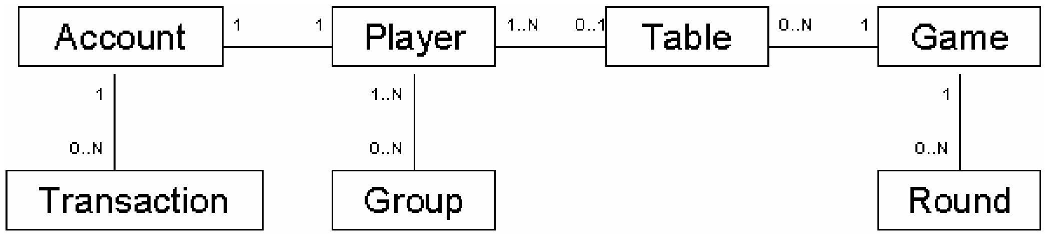


**Figure 1:** Relationships between main domain concepts.

## 4. System overview

Casino-HUT is a distributed, web-based, multiplayer card game gambling platform. It allows users to play card games against each other or against computer players with or without money. Users can form permanent groups to simplify starting a game session. Users can buy game tokens by giving their credit card number and cash their wins to a bank account, but the system does not contain an automatic integration to the credit card/banking systems.

The system is hosted by the Casino of Otaniemi. The administrator of the system can view reports of players, played games and cash flows.

## 5. User groups

Table 3 describes the intended users of the system.

**Table 3.** Users of the system.

| User group | Description | Number of users |
|---|---|---|
| Administrator | System administrators can change "every-thing" in the system. | a couple of persons |
| Player | Players play games in the casino and may also perform some administrative tasks. Table and group bosses are also ordinary players. | dozens |

## 6. Functional requirements

**Table 4.** Use cases.

| # | Use case | Notes |
|---|---|---|
| - | 0.1 register | Implemented in the core. |
| - | 0.2 login | Implemented in the core. |
| 1 | 0.3 logout | |
| 2 | 1.1 start new game | |
| 3 | 1.2 play poker machine | |
| 4 | 1.3 transfer money | |
| 5 | 2.1 modify personal information | |
| 6 | 2.2 play roulette | |
| 7 | 2.3 view account | |
| 8 | 3.1 send message | Sending to a single player only. |
| 9 | 4.1 browse account history | |
| 10 | 4.2 view casinos account summary | |
| 11 | 4.3 view casinos account details | |
| 12 | 4.4 view players account report | |
| 13 | 4.5 view top players | |
| 14 | 5.1 create team | |
| 15 | 5.2 explore teams | |
| 16 | 5.3 send message | Sending to a team. |
| 17 | 5.4 invite to team | |
| 18 | 5.5 join team | |
| 19 | 5.6 leave team | |
| - | ~~5.7 modify team~~ | Removed before the project. |
| 20 | 5.8 request to join team | |
| 21 | 6.1 accept request | Accept joining both to a table and team. |
| 22 | 6.2 send message | Sending to a table. |
| 23 | 6.3 invite to table | |
| 24 | 6.4 join table | |
| 25 | 6.5 leave table | |
| 26 | 6.6 request to join table | |
| 27 | 7.1 remove registered player | |
| 28 | 8.1 play Indian poker | |
| 29 | 9.1 play draw poker | |
| 30 | 10.1 play stud poker | |
| 31 | 11.1 play blackjack | |

## 7. Properties (quality requirements, non-functional requirements)

The performance of the system must be adequate (<2 second delays on player actions) with at least 50 simultaneous players on 2Ghz/512Mb Win2000 server, and 1GHz/256Mb Win2000 client.

The reliability of the system is very important as the server will not be continuously monitored.

Special care should be put to guarantee that the number of remaining bugs is as low as possible, because the system will be maintained by non-professionals, who will not be very good in fixing the system.

The usability of the system is not an issue. Do the simplest possible user interface from the implementation point of view.

## 8. Constraints

### 8.1 Technologies

The system must be implemented using Enterprise Java Beans (EJB) and other J2EE technologies. The following tools must be used:

- J2SDK 1.4.1
- JBoss 3.2.2

### 8.2 User interface

The user interface consists of three parts.

| Web browser | Text based game window(s) | Text based system info window |
|---|---|---|
| • creating, exploring and modifying groups/tables<br>• viewing reports<br>• sending invitations<br>• sending requests for joining a group/table<br>• administrator's operations | • playing games | • receiving on-line invitations<br>• messages from/to on-line players |

**Web browser**

The web browser interface should be minimalist, no fancy usability tricks are required and Javascript must not be used. JSP technology should be used to generate the pages.

**Text based game window**

The user interface for the games must be text based. The output must follow the one shown below.

Example: Video poker

```
Select bet (1-5, 0=quit):3
Your cards 5H, 8S, 5C, KD, QD
Select cards to hold (1,2,3,4,5): 1 3
Your cards: 5H, 5S, 5C, 2D, 3D
You win 15!
Select to double (y/n): y
Select high or low (h/l): h
Card 4h. You lose!

Select bet (1-5, 0=quit):3
...
```

Example: 5 card stud poker

```
Your cards 5H, 8S
Mikko's cards XX, QD

Mikko raises 5, pot=5
Check, raise, drop (c, r [1-10], d):r 10
Mikko checks 10, pot=30

Your cards 5H, 8S, KS
Mikko's cards XX, QD,9S

Pass, raise (p, r [1-10]):r 10
Mikko raises 5, pot=50
Check, raise (c, r [1-10]):c

Your cards 5H, 8S, KS, 5S
Mikko's cards XX, QD,9S, 6S

...
```

## 9. References

| Name of the document | URL |
|---|---|
| General poker rules | http://www.casino-info.com/gambling_tips/poker.html |
| Draw poker rules | http://www.pagat.com/jerrycooley/drawpoker.html |
| Stud poker rules | http://www.pagat.com/jerrycooley/studpoker.html |
| Video poker example | http://www.pelaamo.ray.fi/pelit/jokeripokeri.html |
| Poker hand ranking | http://www.pagat.com/vying/pokerrank.html |

# Appendix B. An example of a use case description

This appendix contains one of the use case descriptions given to the student teams.

## Play poker machine

### Short description

The actor plays poker machine.

### Actors

Player.

### Preconditions

The actor is logged on and has a valid session.

### Postconditions

The actor's winnings or losses are updated to his account.

### Trigger

The actor chooses to play poker machine.

### Normal flow

The actor may play several rounds. System displays the actor's account balance and current bet. Bet is initially 1 euro, but between rounds actor may choose a bet of 1, 2, 3, 4 or 5 euros.

The actor starts a round. System deals actor 5 cards from one freshly shuffled 52-card deck. The actor selects any number of cards to keep. System deals new cards to replace discarded cards. The actor wins or loses according to following table:

| **Combination** | **Win (including bet!)** |
|---|---|
| Straight flush (e.g. 3-4-5-6-7 of spades) | 20x |
| Four of a kind (e.g. 4 jacks) | 15x |
| Full house (e.g. 3 tens and 2 eights) | 8x |
| Flush (e.g. all hearts) | 4x |
| Straight (e.g. 4-5-6-7-8 regardless of suit) | 3x |
| Three of a kind (e.g. 3 fives) | 2x |
| Two pairs (e.g. 2 fives and to sevens) | 2x |

### Alternative flows, errors

If the actor doesn't have enough money on his account, system displays an error message.

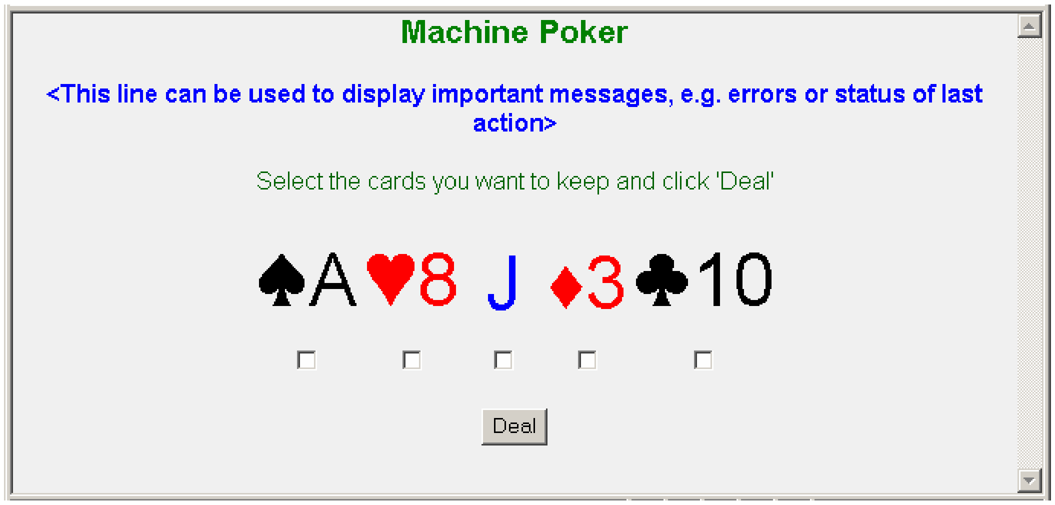
If there is a database or communication error, a descriptive error message is shown.

### Features not to be implemented

-

# Appendix C. An example of an HTML layout

This appendix contains one of the HTML layouts given to the students teams. The layouts were provided as static HTML pages, which the teams had to implement as dynamically generated pages.



Machine Poker

<This line can be used to display important messages, e.g. errors or status of last action>

Select the cards you want to keep and click 'Deal'

♠A ♥8 J ♦3 ♣10

# Appendix D. Project plan

This appendix contains the partially written project plan given to the student teams. It contains the guidelines for the development process.

# Project Plan

*This is a partially completed template of a project plan. Remove/replace texts in italics when completing the plan.*

## 1. Introduction

### 1.1 Purpose and scope of project

The purpose of the project is to build an on-line casino software for the Casino of Otaniemi. The project includes developing, testing and delivering the software, which is described in the requirements specification written by the customer. Project effort is limited to N hours during which a maximum amount of value for the customer should be produced.

### 1.2 Terminology and definitions

*None, yet.*

## 2. Stakeholders and staffing

### 2.1 Project group

Present the actual project group; the members and their contact information (Name, Telephone, E-mail)

### 2.2 Other stakeholders

Customer:N.N.

## 3. Goals and end criteria

### 3.1 Goals of the customer

| Goal | Verification criteria |
|---|---|
| 1. Team follows the defined work practices | Analyzing data from CVS and time reporting system and produced documentation. |
| 2. Team reports their work in a disciplined fashion | Analyzing data from CVS and time reporting system and produced documentation. |
| 3. Number of bugs in the delivered system is as low as reasonably possible | Number of bugs found by an objective tester after the project. |
| 4. Amount of implemented features is as great as possible | Number of features implemented. |
| 5. No effort is wasted on unnecessary things not forced in the process or requirements specification | Analyzing data from the time reporting system. |

### 3.2 Project abort criteria

Project is cancelled if any member of the group is not able to put the required amount of effort in the project.

### 3.3 Project end criteria

The project ends on the day defined in the schedule in section 6.

## 4. Resources and budget

### 4.1 Personnel

*The first version of the project plan should contain estimates of the hours each individual group member will spend on the project on various weeks. The amount should be roughly the same between all members. If personal absences are known beforehand, they should be noted here as restrictions and be considered when dividing effort.*

*Plan the time and location for all group sessions in advance. Fill each member's hours for each week (group session hours + personal hours).*

| Wk | Session 1 | Session 2 | Session 3 | Session 4 | ***Member 1*** | ***Member 2*** | ***Member 3*** | ***Member 4*** | Total |
|---|---|---|---|---|---|---|---|---|---|
| **1** | lecture | lecture | | | 9 (0+9) | 9(0+9) | 9 (0+9) | 9 (0+9) | 36 |
| **2** | lecture | | | | 11 (0+11) | 11 (0+11) | 11 (0+11) | 11 (0+11) | 44 |
| **3** | *Mo 8-13 T-bld* | | | | 10 (5+5) | 10 (5+5) | 10 (5+5) | 10 (5+5) | 40 |
| **4** | *Mo 9-12 A218* | *We 9-12 A218* | *Th 15-18 A218* | | *12 (9+3)* | *12 (9+3)* | *12 (9+3)* | *12 (9+3)* | *48* |
| **5** | *Mo 8-11 X123* | *We 8-12 X123* | *Th 8-11 X123* | | *12 (10+2)* | *12 (10+2)* | *12 (10+2)* | *12 (10+2)* | *48* |
| **6** | | | | | | | | | |
| **7** | | | | | | | | | |
| **8** | | | | | | | | | |
| **9** | | | | | | | | | |
| **10** | | | | | | | | | |
| **11** | | | | | | | | | |
| **T** | | | | | **120** | **120** | **120** | **120** | **480** |

### 4.2 Materials

*Key hardware resources and premises where the work is done should be listed here. If the availability of these resources is not self-evident the restrictions must be documented.*

## 5. Work practices and tools

*This section lists all practices, methods, and tools used in the project. Each chosen topic and its application to the project must be listed and shortly discussed here.*

### 5.1 Practices

*Describe here all planned work practices at least briefly. The following practices are mandatory for all teams.*

**Work organization an planning**

**Co-located team and group sessions.** The team must do all development work together in group sessions lasting 4-8 hours. Each person may spend max. 20 hours for personal work during the project (weeks 3-11), e.g. for studying technologies and tools, but no production code may be developed during the personal time.

**Iteration planning**

Iteration planning must be done in the beginning of iterations 1 and 2.

In the beginning of Iteration 1 the team takes one requirement at a time in the priority order and estimates collectively the effort required to implement (=design, code, unit test and comment code) it. This is repeated until the team is sure it has estimated more requirements than they can implement in the iteration. The estimates are stored in Trapoli (Modify Tasks: Est. Effort field) for the corresponding tasks (each requirement has a corresponding task). In the beginning of Iteration 2 those requirements that were not already implemented are re-estimated, and enough new ones estimated.

In addition, the team plans tasks, which are not related to implementing individual requirements and estimates their effort. Such tasks are, e.g., meetings, architectural code, system testing, bug fixing after system testing and documenting. New tasks and their effort estimates are stored into the Trapoli system.

After the iteration planning, the plan must be copy-pasted from Trapoli to section 6 of the project plan.

**Iteration execution**

1. Make the minimum architecture. In the beginning of an iteration, do the minimum core architecture required for those requirements that you believe you are able to implement during that iteration. Don't spend too much time on this, but try to do and report most architectural work later as a part of implementing individual requirements.

2. Pick a requirement. A person/pair takes a requirement from the top of the requirements list. Preferably only one person/pair is responsible for implementing a certain requirement, but when unavoidable (e.g. in the beginning of the project) you may split one requirement to several tasks that are concurrently made by several persons/pairs.

3. Implement the requirement. First, spend a few minutes re-estimating how much it will take from you/the pair to implement the requirement, and store the new estimate as the first effort left value for the task in Trapoli, i.e. add a row (hours done=0.1, hours left=[new estimate]). Then design, code, unit test and comment the code of the requirement.

A person/pair is responsible for the work, but of course may ask help from other team members. If the specs are ambiguous or the team considers it is not able to implement a requirement, contact t76633-customer#soberit.hut.fi and describe your problem.

4. Repeat steps 2-3 until all hours reserved for implementing the requirements during the iteration have been used. Complete a partially finished requirement, even if the hours run out during its implementation.

5. Execute system testing and fix bugs.

6. Update the technical specification.

7. If you are still below the allocated hours for the iteration, e.g. because no time was needed for bug fixing, implement one more requirement and test it.

**Development**

**Collective ownership.** Anyone in the team is allowed to change any code.

**Version control.** Source code must be managed using CVS version control system through Eclipse. Changes must be committed to the repository at least whenever a requirement has been both coded and successfully unit tested. CVS server provided by the course must be used, or otherwise a zipped project repository must be delivered in the same way as project documentation in the end of iterations.

**Pair programming.** (PP TEAMS) All production code and unit tests must be written by two persons sitting in front of one computer. The pairs must be actively switched, but preferably after completing a requirement. Of course both pairs do not finish their work at the same time. In these cases you can either do some (non-programming work) alone for a while or switch pairs in the middle of implementing a requirement. The maximum amount of time for programming without changing the partner is 12 hours.

**Pair programming.** (NON-PP TEAMS) You are not allowed to implement a whole requirement with another person sitting with you on the same workstation. You are allowed to communicate frequently with other members, and even implement some parts of the requirement together, if you find it useful. If you work together with another person, you must always track the following thing: how many minutes, who participated, what requirement.

**Coding standard.** Sun's Coding Conventions for the Java Programming Language must be followed everywhere in the code.

**Unit testing.** Unit tests must be written to all reasonable places in the source code. Unit tests must be written using EJBUnit or JUnit.

**System testing.** In the end of each implementation iteration, the produces system must be tested. Each team member must spend 3 hours for running ad-hoc tests for all implemented requirements based on the requirements specification.

**Reporting**

**Time reporting.** Time reporting must be done using the Trapoli system. By default it contains a task for each requirement in the requirement specification. Teams must add other required tasks to the system. The effort estimates done in the beginning of an iteration must also be reported to the system (planned hours field). Realized effort must be reported after each working session.

The hours spent before the actual project on lectures and doing personal exercises must not be reported to Trapoli.

**Bug reporting.** Bug reporting must be done to a text-file/Excel sheet (see template). All bugs found in the system are reported, except those that are related to the code of a requirement currently under development.

**Source code size reporting.** Lines of code (LOC) must be calculated in the end of iterations. A program called CCCC is used to calculate the LOC.

*Documentation.* ***The following documents must be produced:***

- A short project plan (this document refined)
- Reasonable code comments for all relevant methods
- A short technical specification (system overview, architecture, design principles, high-level module descriptions, …). The level of detail should be defined based on what the team considers useful for their own purposes during the development.
- Final report (see template, contains e.g. summaries of effort used, bug counts, LOC, evaluation of the developed system, course feedback)

Documents must be delivered in HTML format to the course via the delivery system **by 23:59 on Sunday** in the end of project planning and implementation iterations.

**Forbidden things**

You are not allowed to use existing code related to the business domain of the exercise except the code given to you by the course organizers. However, you are allowed to use small code snippets e.g. patterns related to the architectural solutions of the system from J2EE textbooks or other public sources.

Non-pair programming teams are encouraged not to search for and read material about pair programming during the project.

## 5.2 Tools

*List tools used in the project.*

- Eclipse 2.1
- Ant
- CVS 1.11
- J2EE SDK 1.4
- JBoss 3.2
- MS Frontpage for documentation?

## 6. Phasing

*This section lists the phases of the project, their deliverables and any critical dates during the phases. The project must follow the schedule presented below. N=number of people in the group.*

### 6.1 Tentative schedule

| Week | Event | Effort |
|---|---|---|
| **1-2** | **Training** | |
| **1-2** | Training sessions<br>• Tu 17.2. 14-15: course overview, 15-18: J2EE lecture (Servlets, JSPs), HelloWorld-exercise<br>• Th 19.2. 14-18: development technologies: J2EE lecture (EJBs) and more advanced exercise<br>• Tu 24.2. 14-16: J2EE design patterns, 16-18: core architecture and use cases<br>• Th 26.2. 14-15: development process, 15-17: tools, 17-18.30: pair programming (only for pp groups)<br>Personal studying<br>• exercises (6h) | N*20h |
| **3** | **Project planning iteration** | |
| **3** | Project planning and studying | N*5h |
| **3** | High level architectural design | N*5h |
| **x.y.** | DL: Project plan, architectural design | |
| **4-7** | **Iteration 1** | |
| **4** | Iteration planning | N*1h |
| **4-6** | Design & implementation | N*30h |
| **7** | System testing, bug fixing | N*8h |
| **x.y.** | DL: Project plan, architectural design | |
| **8-11** | **Iteration 2** | |
| **8** | Iteration planning | N*1h |
| **8-10** | Design & implementation | N*30h |
| **11** | System testing, bug fixing | N*15h |
| **x.y.** | DL: Project plan, architectural design | |

### 6.2 Project planning

Goals:

- Plan when, how much and where the team works
- Read the requirements specification
- Study the new technologies, practices, and tools
- Study the provided EJB example
- Complete the project plan based on the pre-filled template (this document)

- Do quick high-level architectural design and prototyping based on the requirements specification. Design is refined later when implementing the individual requirements.

Deliverables:

- project plan
- a draft of the technical specification

**6.3 Iteration 1 & 2**

Goals:

- Implement (design, code, test) as many requirements as possible
- Execute system testing and fix bugs found

Deliverables:

- technical specification
- software (in CVS)

# Appendix E. Inquiry forms

This appendix contains the inquiry forms filled by the participants before and after the project.

## Inquiry 1: Background information

Please, answer the following questions. The data is used to form groups with equal programming experience.

**Name**

**Student ID**

**How much programming experience do you have?**
Sum up the periods of calendar time you have been actively programming (related to work, studies or hobbies).
I have programmed about years, of which about years using Java.

**How much pair programming experience do you have?**
select

**Evaluate your skills as a programmer against other computer science students at HUT?**
select

**What is your average grade from programming intensive courses at HUT?**
about (e.g. 4.2)

## Inquiry 2: Final Inquiry

Please answer the following questions carefully. Answering the open questions (text areas) is not mandatory, but I would appreciate a lot all comments you have.

### Identification

**Student ID**

---

### Development work in general

**Estimate how much work you did co-located with:**

**0 other members** %
**1 other members** %
**2 other members** %
**3 other members** %

**Did you like the way your team did the development work in general?**

Select

**Why/why not?**

What do you think were the major factors that affected (positively or negatively) the productivity or quality of work in you team?
E.g. were you team sessions successful (why/why not), were there some external factors that affected the project?

---

### Pair programming

**Did you enjoy doing pair programming?**

Select

**Why/why not?**

**Which do you *like* more, pair programming or programming alone?**

Select

**Which do you consider better for the overall success of this kind of a project: pair programming or programming alone?**

Select

Any comments on pair programming vs. programming alone?

---

## Use cases

**Evaluate the intellectual difficulty of the implementation work of the use cases?**

For example

- if a use cases required a very clever solution, but then the total effort was small, select "very difficult"
- if a use case required lots of easy work, select "easy"

| | |
|---|---|
| I did not code it | 0.3_logout |
| I did not code it | 1.1_start_new_game |
| I did not code it | 1.2_play_pokermachine |
| I did not code it | 1.3_transfer_money |
| I did not code it | 2.1_modify_personal_information |
| I did not code it | 2.2_play_roulette |
| I did not code it | 2.3_view_account |
| I did not code it | 3.1_send_message (to a single player) |
| I did not code it | 4.1_browse_account_history |
| I did not code it | 4.2_view_casinos_account_summary |
| I did not code it | 4.3_view_casinos_account_details |
| I did not code it | 4.4_view_players_account_report |
| I did not code it | 4.5_view_top_players |
| I did not code it | 5.1_create_group |
| I did not code it | 5.2_explore_groups |
| I did not code it | 5.3_send_message (to a group) |
| I did not code it | 5.4_invite_to_group |
| I did not code it | 5.5_join_group |
| I did not code it | 5.6_leave_group |

| | |
|---|---|
| I did not code it | 5.7_modify_group |
| I did not code it | 5.8_request_to_join_group |
| I did not code it | 6.1_accept_request (table and group) |
| I did not code it | 6.2_send_message (to a table) |
| I did not code it | 6.3_invite_to_table |
| I did not code it | 6.4_join_table |
| I did not code it | 6.5_leave_table |
| I did not code it | 6.6_request_to_join_table |
| I did not code it | 7.1_remove_registered_player |
| I did not code it | 8.1_play_indian_poker |
| I did not code it | 9.1_play_draw_poker |

## Modules

Evaluate for each module (defined by subdirectories in the source code tree):

- your involvement, i.e. how much did you participate in its development
- understanding, i.e. how well do you understand its internal structure
- quality of code, i.e. quality of internal structure and module design

| Module | Involvement | Understanding | Quality of Code |
|---|---|---|---|
| account | Select | Select | Select |
| admin | Select | Select | Select |
| client | Select | Select | Select |
| game | Select | Select | Select |
| groups | Select | Select | Select |
| messaging | Select | Select | Select |
| servlet | Select | Select | Select |
| tag | Select | Select | Select |
| tools | Select | Select | Select |
| user | Select | Select | Select |

# Appendix F. Detailed data

**Table 1** Effort (hours) spent for each finished use case.

| # | Use case name | PP1 | PP2 | SP1 | SP2 |
|---|---|---|---|---|---|
| 1 | UC_0.3_logout | 8.1 | 15.6 | 2.0 | 6.8 |
| 2 | UC_1.1_start_new_game | 42.1 | 42.0 | 15.5 | 26.1 |
| 3 | UC_1.2_play_pokermachine | 40.1 | 56.6 | 14.3 | 23.5 |
| 4 | UC_1.3_transfer_money | 26.1 | 12.0 | 15.0 | 14.2 |
| 5 | UC_2.1_modify_personal_information | 7.1 | 7.0 | 12.1 | 3.4 |
| 6 | UC_2.2_play_roulette | 21.1 | 30.0 | 16.8 | 19.0 |
| 7 | UC_2.3_view_account | 8.1 | 2.0 | 3.5 | 1.1 |
| 8 | UC_3.1_send_message | 1.1 | 2.0 | 9.1 | 7.3 |
| 9 | UC_4.1_browse_account_history | 19.1 | 12.2 | 27.6 | 10.6 |
| 10 | UC_4.2_view_casinos_account_summary | 10.1 | 20.2 | 26.2 | 11.4 |
| 11 | UC_4.3_view_casinos_account_details | 6.1 |  | 6.1 | 3.3 |
| 12 | UC_4.4_view_players_account_report | 11.1 |  | 4.7 | 9.3 |
| 13 | UC_4.5_view_top_players | 12.1 |  | 3.1 | 14.7 |
| 14 | UC_5.1_create_group | 5.1 |  | 9.0 | 15.9 |
| 15 | UC_5.2_explore_groups | 8.1 |  | 4.6 | 5.5 |
| 16 | UC_5.3_send_message | 12.1 |  | 16.5 | 6.3 |
| 17 | UC_5.4_invite_to_group | 6.1 |  | 15.6 | 9.2 |
| 18 | UC_5.5_join_group | 1.6 |  | 12.1 | 3.9 |
| 19 | UC_5.6_leave_group | 0.6 |  | 3.1 | 2.4 |
| 20 | UC_5.8_request_to_join_group | 2.1 |  | 2.1 | 2.9 |
| 21 | UC_6.1_accept_request |  |  | 5.6 | 7.4 |
| 22 | UC_6.2_send_message |  |  |  | 3.0 |
| 23 | UC_6.3_invite_to_table |  |  | 10.3 | 7.0 |
| 24 | UC_6.4_join_table |  |  | 3.9 | 6.8 |
| 25 | UC_6.5_leave_table |  |  | 8.0 | 5.4 |
| 26 | UC_6.6_request_to_join_table |  |  | 2.1 | 8.1 |
| 27 | UC_7.1_remove_registered_player |  |  |  | 12.4 |
| 28 | UC_8.1_play_indian_poker |  |  |  |  |

**Table 2** The evaluations of the involvement in and understanding of the Java packages.

| Developer | Packages | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Involvement | | | | | | | | | | Understanding | | | | | | | | | |
| | account | admin | client | game | groups | messaging | servlet | tags | tools | user | account | admin | client | game | groups | messaging | servlet | tags | tools | user |
| PP1 | 4 | 4 | 1 | 3 | 3 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 1 | 4 | 4 | 2 | 4 | 4 | 3 | 4 |
| PP2 | 3 | 2 | 2 | 4 | 3 | 2 | 4 | 4 | 3 | 3 | 3 | 2 | 2 | 4 | 3 | 2 | 4 | 4 | 3 | 3 |
| PP3 | 4 | 1 | 1 | 3 | 2 | 3 | 4 | 3 | 1 | 4 | 4 | 2 | 1 | 4 | 3 | 3 | 4 | 4 | 2 | 4 |
| PP4 | 2 | 4 | 4 | 4 | 4 | 2 | 3 | 4 | 1 | 4 | 2 | 4 | 4 | 4 | 4 | 3 | 4 | 5 | 1 | 5 |
| PP5 | 3 | 1 | 1 | 4 | 2 | 2 | 4 | 3 | 1 | 3 | 4 | 1 | 1 | 3 | 2 | 2 | 3 | 3 | 1 | 3 |
| PP6 | 3 | 3 | 1 | 4 | 2 | 1 | 4 | 4 | 2 | 3 | 3 | 3 | 1 | 4 | 2 | 3 | 5 | 4 | 3 | 4 |
| PP7 | 3 | 3 | 1 | 3 | 1 | 2 | 4 | 3 | 2 | 3 | 2 | 2 | 1 | 2 | 1 | 2 | 3 | 3 | 2 | 2 |
| PP8 | 4 | 4 | 1 | 3 | 5 | 1 | 4 | 4 | 1 | 3 | 5 | 5 | 3 | 4 | 5 | 4 | 4 | 4 | 4 | 4 |
| SP1 | 2 | 3 | 1 | 3 | 2 | 4 | 3 | 3 | 2 | 3 | 2 | 3 | 1 | 3 | 3 | 3 | 4 | 3 | 2 | 3 |
| SP2 | 2 | 1 | 1 | 3 | 4 | 1 | 3 | 5 | 3 | 3 | 3 | 2 | 1 | 3 | 4 | 2 | 3 | 5 | 3 | 3 |
| SP3 | 5 | 5 | 1 | 1 | 1 | 1 | 5 | 1 | 1 | 1 | 4 | 4 | 1 | 1 | 1 | 1 | 5 | 1 | 1 | 1 |
| SP4 | 1 | 1 | 1 | 4 | 5 | 4 | 3 | 3 | 1 | 1 | 3 | 4 | 2 | 4 | 5 | 3 | 3 | 4 | 3 | 3 |
| SP5 | 2 | 4 | 2 | 3 | 3 | 4 | 3 | 3 | 2 | 2 | 3 | 4 | 2 | 3 | 4 | 2 | 2 | 3 | 2 | 2 |
| SP6 | 3 | 2 | 1 | 4 | 3 | 2 | 4 | 4 | 2 | 1 | 4 | 3 | 1 | 3 | 4 | 4 | 5 | 5 | 3 | 1 |
| SP7 | 3 | 1 | 2 | 4 | 5 | 2 | 4 | 3 | 3 | 3 | 4 | 1 | 3 | 4 | 5 | 3 | 4 | 4 | 3 | 5 |
| SP8 | 2 | 2 | 1 | 4 | 4 | 2 | 4 | 4 | 3 | 3 | 3 | 3 | 2 | 4 | 4 | 3 | 4 | 5 | 3 | 3 |

**Scale for the answers:**

1 - None

2 - Little

3 - Some

4 - Quite a lot

5 - Very much