Agile versus Plan-Driven Approaches to Planning-A Controlled Experiment

MASTER THESIS

Stefan Zugal Bakk.techn.

LEOPOLD-FRANZENS
UNIVERSITY OF INNSBRUCK

INSTITUTE OF
COMPUTER SCIENCE

RESEARCH GROUP
QUALITY ENGINEERING

Supervisor:
Dr. Barbara Weber

September 28, 2008
“It’s better to be roughly right than precisely wrong.”

John Maynard Keynes
Abstract

Scarce resources, fierce competition as well as ever changing business demands require software development approaches which are efficient, flexible and keep time to market low. While anecdotal evidence suggests that agile approaches satisfy the necessary requirements and outperform plan-driven approaches, there is still a lack of scientific results.

This thesis compares the adoption of plan-driven versus agile approaches for planning. Since studying real projects would be too cost and time intensive, a journey is deployed as a metaphor for a software development project. In an experiment, students simulate the undertaking of a voyage with the usage of a travel simulator. The gathered data suggests that best results can be achieved by adopting agile methodology, but there is no statistical significance that agile approaches perform better than plan-driven ones on average. Manual analysis showed that the weak performance of agile journeys was not caused by conceptual problems, but by an insufficient implementation of agile practices as well as missing tool support.
Acknowledgment

I would like to thank all people who supported and inspired me in conducting my master thesis. I am thankful for all the constructive criticism, feedback and encouragement, which helped me to do my best.

First of all, I want to thank Dr. Barbara Weber for supervising the thesis, her endless motivation and support. She and Werner Wild provided well thought out feedback and inspiration for developing the Alaska travel simulator and helped us to shape a common vision. I want to thank Barbara once more for the support and feedback she gave me when writing this thesis.

Next I want to thank DI Michael Schier for carrying out the development work with me. Without his encouraging work it would not have been possible to implement the travel simulator in time. I want to be thankful for all the constructive discussions and feedback which helped to develop the simulator in superior quality.

Further I want to thank all people who gave useful insights in shortcomings in the simulator as well my thesis. Their feedback helped to improve the work steadily. Special thank goes to Christian Haisjackl Bakk.techn. who thoroughly tested the Alaska travel simulator and helped to eliminate bugs and fine tune the application. I also want to thank my sister Petra Zugal for proof reading this thesis.

Last but not least I want to thank my parents for continuously supporting me throughout studying computer science and cheering me up when I felt down.
Declaration of Authorship

I, Stefan Zugal Bakk.techn., declare that this thesis and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a research degree at this University.
- No part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Date

Signature (Stefan Zugal)
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>5</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>7</td>
</tr>
<tr>
<td>Declaration of Authorship</td>
<td>9</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>15</td>
</tr>
<tr>
<td>1.1. Background</td>
<td>15</td>
</tr>
<tr>
<td>1.1.1. Plan-Driven Process Models</td>
<td>16</td>
</tr>
<tr>
<td>1.1.2. Agile Process Models</td>
<td>17</td>
</tr>
<tr>
<td>1.2. Research Objectives</td>
<td>18</td>
</tr>
<tr>
<td>1.3. Research Method</td>
<td>19</td>
</tr>
<tr>
<td>1.4. Related Work</td>
<td>21</td>
</tr>
<tr>
<td>1.5. Overview</td>
<td>27</td>
</tr>
<tr>
<td>2. Concepts</td>
<td>29</td>
</tr>
<tr>
<td>2.1. Planning Behavior in Plan-Driven versus Agile Software Development</td>
<td>29</td>
</tr>
<tr>
<td>2.2. User Stories</td>
<td>38</td>
</tr>
<tr>
<td>2.2.1. States</td>
<td>40</td>
</tr>
<tr>
<td>2.2.2. Location</td>
<td>41</td>
</tr>
<tr>
<td>2.2.3. Weather</td>
<td>42</td>
</tr>
<tr>
<td>2.2.4. Cost</td>
<td>42</td>
</tr>
<tr>
<td>2.2.5. Duration</td>
<td>42</td>
</tr>
<tr>
<td>2.2.6. Business Value</td>
<td>43</td>
</tr>
<tr>
<td>2.2.7. Reliability</td>
<td>44</td>
</tr>
</tbody>
</table>
### Contents

2.2.8. Availability ................................................. 45  
2.3. Project Specific Limiting Factors and Basic Conditions .......... 46  
2.4. Project Dynamics and Unforeseen Events .......................... 47  
2.5. Agile Principles and Techniques .................................. 49  
2.5.1. Prioritization .............................................. 49  
2.5.2. Eliminate Waste ............................................. 52  
2.5.3. Defer Commitment ........................................... 54  
2.6. Interplay of Concepts ........................................... 56  

3. Architecture and Development Approach ............................... 59  
3.1. Development Approach ........................................... 59  
3.1.1. Values ...................................................... 60  
3.1.2. Principles ................................................... 61  
3.1.3. Practices .................................................... 62  
3.2. Alaska Toolset .................................................. 63  
3.3. Alaska Travel Simulator .......................................... 65  
3.3.1. Alaska Core ................................................ 66  
3.3.2. Alaska User Interface ....................................... 69  
3.3.3. Alaska Help ............................................... 72  
3.4. Third Party Frameworks ......................................... 72  
3.4.1. Eclipse Rich Client Platform ................................. 73  
3.4.2. Graphical Editing Framework ................................ 74  
3.4.3. FitNesse .................................................... 75  
3.4.4. JUnit ....................................................... 76  
3.4.5. XStream .................................................... 77  

4. Experiment ...................................................................... 79  
4.1. Basic Terminology .................................................. 79  
4.1.1. Objects ....................................................... 80  
4.1.2. Subjects ..................................................... 80  
4.1.3. Independent Variables ....................................... 81  
4.1.4. Response Variables ........................................... 81
4.1.5. Experimental Designs .................. 82
4.1.6. Hypotheses .......................... 83
4.2. Experiment Design ........................ 83
4.3. Experiment Execution ..................... 86
4.4. Data Analysis Procedure ................... 87
  4.4.1. Data Validation ....................... 88
  4.4.2. Data Analysis ......................... 88
4.5. Experiment Results ........................ 91
  4.5.1. Interpretation of Results ............... 93
4.6. Risk Analysis and Mitigation ............... 96
  4.6.1. Internal Validity ...................... 97
  4.6.2. External Validity ..................... 99
4.7. Discussion .............................. 100

5. Summary 103

A. Data of the Experiment 107

List of Figures 109

List of Tables 111

Bibliography 113
Chapter 1.

Introduction

1.1. Background

Software engineering is still a quite young discipline. Starting in the 1940s, programming was the main task to be accomplished. Even though programs were small in comparison to nowadays applications, scarce hardware resources hindered software development. A reasonable part of the development work consisted of devising code, which could get by on the available hardware. Therefore a common conception was that more powerful hardware would solve most of the programmer's problems [Dij72]. While the performance of hardware indeed increased dramatically, software development found itself in a crisis in the mid 1960s. So far, possible implementations were limited by the available hardware resources, thus also limiting the application’s complexity. The increasing computing power eased programmers of the burden of scarce resources, but introduced apparently even bigger problems in terms of complexity.

“The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he had to dream about them and, even worse, he had to transform such dreams into reality! Is it a wonder that we found ourselves in a software crisis?”

Edsger W. Dijkstra [Dij72]
Chapter 1. Introduction

In order to tackle these problems, the first official conference about software engineering has been held in 1968 in Garmisch-Partenkirchen, well known as the NATO Software Engineering Conference [NR86], where the term software engineering has been coined. Profound development methodologies were required urgently as the following citation of Ronald Graham, member of Bell Labs and participant of the conference, suggests: “We build systems like the Wright brothers built airplanes – build the whole thing, push it off the cliff, let it crash, and start over again” [NR86].

First evolving software development processes showed a strong bias of mathematics and classical engineering disciplines. Even though they already included aspects which go beyond implementation issues, other important issues of software and its development have not been understood properly yet. For instance, in construction engineering it is not possible to build houses iteratively, while in software development it certainly is. Furthermore, social and business factors were not taken into account. In order to tackle these problems too, agile approaches rose in the 1990s. Today, software development processes can roughly be divided into plan-driven as well as agile process models.

1.1.1. Plan-Driven Process Models

The first software development approaches evolved belong to plan-driven models. As the name suggests, the focus is put on the plan, which defines the execution of the project. Considerable effort is invested in the elaboration of a plan in the very beginning of the project, the up-front planning represents a key point of these process models. Project phases are executed sequentially, communication between phases is usually one-way and carried out by documents. For instance, the requirements of software are documented in the very beginning and later on used for the architecture and implementation. Stepping a phase back is only allowed in exceptional situations, e.g. to resolve contradictions. The Waterfall Model [Roy70] and V-Model [Hes08] are classical plan-driven process models.
Experience showed that the strict execution seems to be unrealistic in real-world projects [Alb98]. In order to integrate the redoing of project phases, plan-driven models with iterative execution have been devised. The development process runs through all phases multiple times, each cycle represents an iteration. As a result after each run a prototype of the application is available. The Spiral Model [Boe88], Personal Software Process [Hum94], Team Software Process [Hum99] as well as the Rational Unified Process [JBR99] implement the idea of iterative development. Even though these models provide more flexibility in the execution, the planning activity remains in the beginning of the project.

1.1.2. Agile Process Models

Agile process models provide a further increase of flexibility. Instead of trying to cancel out all uncertainties by elaborating sophisticated up-front plans, uncertainty is accepted and plays an important role in the process models. According to [Boe81] uncertainty decreases as the project progresses. Thus only parts of the development plan which deal with the near future are elaborated in detail, aspects lying far ahead are only sketched roughly. In addition planning is extended from the very beginning of the project to its whole runtime, it becomes a continuous activity. The plan is deliberately subjected to change and is adapted throughout the project if necessary. By integrating new gained knowledge into the plan, uncertainties are diminished, risks can be reduced [Coh06]. Project phases in agile process models are usually interwoven and run in parallel, the execution is carried out iteratively. In contrast to plan-driven process models after each iteration working software is deployed.

Agile software development processes try to minimize the overhead of excessive documentation and up-front specification. Instead of anticipating the future and user’s demands, communication and quick feedback is fostered, the involvement of the customer in the project plays an important role. In addition also social factors are taken into account, team work and human values are considered. The goal is to have everyone contribute the best he has to offer to the team’s success [Bec00]. Agile processes not only provide flexibility but also the process itself is
flexible, i.e. the development process is adapted as more knowledge about the project is gained [Coh06]. Process models describe best practices and techniques, but it is up to the development team to apply or abandon them [Bec00]. For instance best practices which involve very close human interaction cannot be applied if the developers do not feel comfortable.

“Stay aware, adapt, change!”

Kent Beck [Bec00]

Finally it should be mentioned that agile software development focuses on business priorities. The goal is to leverage each team member’s skills and generate the highest business value for the customer. First agile process models evolved in the 1990s, nowadays among others eXtreme Programming [Bec00], Scrum [Sch04], Dynamic Systems Development Model [CV98] and Lean Software Development [PP06] are known.

1.2. Research Objectives

This thesis investigates the differences between plan-driven and agile software development processes with respect to planning. As pointed out in 1.1, the plan-driven approach is commonly regarded as a rigid methodology, while agile software development is known to be more flexible [Coh06]. In order to compare the success of projects, the generated business value is considered. The performance of either approach will be evaluated in two different environments, one of them exhibiting a high degree of uncertainty, the other rather stable conditions. By providing different degrees of uncertainty, it shall be investigated whether the flexibility provided by agile approaches results in an effect on the generated business value.

We investigate the impact of adoption of plan-driven versus agile methodology with respect to the success of the project, measured in terms of the generated business value.
Experience reports suggest that the adoption of agile approaches yield positive effects for software development projects [MM06], [MK07], [MP08]. So far only very few studies with proper scientific scrutiny have been published in conference proceedings and journals [MP08]. Thus this thesis generates and analyzes data about the planning behavior of the plan-driven versus agile methodology to back up experience reports with profound scientific results.

1.3. Research Method

Our research method is based on literature, which provides guidelines for setting up and conducting experiments about software development [FP97], [Bro90], [KPP+02]. The research objective is to compare the performance of plan-driven versus agile approaches with respect to the success of the project. Due to limitations in budget and time, it is not feasible for us to carry out software development projects twice, once for each approach. Thus we utilize a vacation trip as a metaphor for a software development project. As discussed in detail in Chapter 2, we consider the metaphor suitable due to many parallels found in either situation. Since we consider only the planning behavior, we can abstract from details, which would differ fundamentally.

In parallel to this thesis, the master thesis of Michael Schier [Sch08] has been conducted. Schier investigates the impact of deferring design decisions with respect to the business value and project plan adaption frequency. The applied research method differs only in the evaluation of data generated by the experiment (cf. Chapter 4). The experiment setup itself supports the research objectives of both theses. Figure 1.1 illustrates the applied research method, a short illustration of the method’s phases follows subsequently.

1. Implementation As first step, a simulator is implemented, which enables its user to plan and execute a journey supporting the concepts described in Chapter 2. Since plan-driven and agile methodology should be compared, the simulator (subsequently called Alaska travel simulator) provides user interfaces for either approach.
Chapter 1. Introduction

1.3. Research Method

Literature about software experiments ([FP97], [Bro90], [KPP+02]) provides several design guidelines for setting up an experiment. We make use of these and conduct experiments of plan-driven approaches by utilizing a vacation trip as metaphor for a software development project. The reason for doing so lies in the multitude of parallels between both terms: In both cases, tasks with different characteristics await their execution, design decisions have to be made and agility is in demand when it comes to unforeseen events (described in detail in chapter 2). Based on this idea, our experiment evolves as follows (depicted in figure 1.3):

1. Implementation
   Creation of a travel simulator system

2. Experiment
   Execution of simulated journeys by a set of test persons.

3. Data Analysis
   Separation into two clusters, evaluation of experiment output and statistical analysis.

4. Conclusion
   Discussion of experiment results

![Figure 1.3: Progression of the journey paradigm study](image)

Figure 1.1.: Visualization of the applied research method [Sch08]

2. **Experiment** In order to generate data needed for the comparison of the development approaches, an experiment is conducted (cf. Chapter 4). The afore implemented Alaska travel simulator provides the necessary tool support for undertaking the virtual journeys. The experiment is carried out at the University of Innsbruck, students participate as test persons.

3. **Data Analysis** In the next step the data collected in the experiment is validated and analyzed. According to the adopted approach, the data set is grouped into plan-driven as well as agile journeys. Profound statistical methods are deployed for inferring conclusions. An extensive explanation of the validation and analysis can be found in Section 4.4.

4. **Conclusion** Finally conclusions based on voyages have to be transferred back into the context of software development (cf. Subsection 4.6.2). Findings can now be discussed and further research questions identified.
1.4. Related Work

An empirical study is really just a test that compares what we believe to what we observe [EPV00]. While empirical studies are widely spread and used in other engineering disciplines, in software engineering they are still underused. As a result, software engineering researches tend to do not validate their research ideas by conducting adequate experiments [ZW97]. A problem with respect to experiment design is the difference between individual software projects, which makes comparison difficult [Bas96]. This thesis can be seen as an empirical study investigating the differences between the plan-driven and agile software development approach. The applied research method has been outlined in Section 1.3, the experiment’s setup, conduction and data analysis is illustrated in Chapter 4.

The research in agile methodology is currently lagging behind the state-of-practice of agile software development. While the amount of anecdotal evidence has grown steadily over the past years, the amount of empirical research evidence is still growing too slowly. Individual agile practices have been studied to a certain extent but results have been quite diverse [MP08]. As pointed out in Section 1.2, several experience reports in the field of agile methodology have been published so far, the most interesting with respect to this thesis will be summarized subsequently. Additionally relevant experiments and books are taken into account.

In “Evaluation of Test-Driven Development” [WG07] an industrial case study about test-driven development (TDD) [Bec02] has been conducted. The authors evaluate the adoption of TDD at EPCOS [Epc], one of the largest manufacturer of passive components for electronic devices. In a rather small project (about three man years) a typical three tier application with average complexity has been developed, TDD has been adopted for the development of the database, web services as well as the client. The researchers investigate claims made by the TDD community, including the applicability of TDD, perception for stakeholders and code quality. These issues will be shortly illustrated subsequently.
**TDD can be applied anywhere in any project** As already mentioned, TDD was deployed for the development of the database, client as well as web services. While TDD fitted very well for the development of web services and business logic, problems occurred when testing security relevant issues. Since authentication mechanisms in web services depend on IP addresses, testing would involve a setup with multiple machines and IP addresses, thus involving additional effort for creating the test environment. For the database and user interface development similar problems regarding the test environment showed up. The authors conclude that the adoption of TDD fails if it either is technically extremely difficult / impossible to test code using unit tests or the time required for the test setup is not in relation with manual verification.

**Code quality will be higher** For the assessment of the code quality, the McCabe cyclomatic complexity [McC76] has been used. Results showed moderate complexity, comparable with other applications developed using TDD. The researchers highlight that the measurements can only be seen as an indication, more samples would be required to speak of evidence.

**The TDD process is more enjoyable for the stakeholders** Besides technical aspects also the stakeholder’s satisfaction has been evaluated. Programmers appreciated the fact that they were eased of the burden of writing lengthy documentation before writing code. Interestingly, for new team members the lack of written documentation was not an obstacle. Tests provided adequate documentation about the application’s functionality, for architectural aspects small documents sufficed. Interviews with customers revealed their satisfaction with TDD, since they are more involved in the development process.

“The Costs and Benefits of Pair Programming” [CW01] is another interesting paper describing the applicability of agile methodology by Alistair Cockburn and Laurie Williams. In comparison to the previous paper they do not conduct an experiment themselves, but summarize and discuss the results of existing studies. The paper explores eight paths of software engineering and organizational effectiveness and shows their connection to pair programming. In order to provide a
profound scientific reasoning, conclusions are made based on experimental data as well as interviews. Next, three interesting aspects will be summarized briefly.

**Economics** According to the authors, the affordability of pair programming plays a central role for its deployment. Skeptical managers assume that pair programming will double the code development expenses. Indeed a controlled experiment showed that pair programming increases development costs, but merely by 15%. Figure 1.2 shows the relative development time of individual– versus pair programming for three subsequently implemented applications. For the implementation of the first application, teams were not well practiced yet, development team rose by 50%. Subsequent programs show the expected 15% rise. Even though an increase of code development costs has to be accepted, the authors point out that the increase in code quality will compensate additional expenses. They provide calculations which show that even by assuming very conservative values, huge savings are possible when taking into account the correction of defects. Cockburn and Williams conclude that pair programming can thus be justified even on purely economic basis which clearly contradicts the skeptical managers’ opinion.

**Design Quality** Besides reducing the defect rate of the developed software, pair programming has also a positive influence on the design quality. An experiment showed that pairs produce programs with same functionality but consistently less lines of code in comparison to individually developed software (cf.
Chapter 1. Introduction

Figure 1.2). The authors believe that this fact may be an indication that the pairs had better designs.

**Continuous Reviews** Another interesting aspect of pair programming pointed out by the paper are the reviews which are automatically and continuously performed. Two programmers work on the same machine, but only one of them can use the keyboard. The partner who is currently not typing, constantly reviews the written code, allowing feedback cycles of a few seconds. By getting feedback quickly, defects can be avoided very early in the development process. Since the costs of removing defects exponentially increase as the project progresses, finding defects early offers potential for drastic savings.

As the paper summarized afore, [SSSH01] deals with the evaluation of pair programming in experiments. The authors state that anecdotal evidence suggests that eXtreme Programming [Bec00] practices are beneficial, scientific studies are still lacking. In order to foster experiments validating the benefits of pair programming, they devise a framework, including the definition, planning, operation, and interpretation of experiments. The work focuses on evaluating software quality as well as costs and uses object-oriented design metrics to assess the developed software. The paper does not include any data collected so far, but announces that experiments will be conducted based on the devised framework.

In “XP in a Safety-Critical Environment” [PP02] Mary Poppendieck briefly discusses an experience report of the adoption of XP [Bec00] in a pharmaceutical project lead by Ron Morsicato. The development of safety critical systems involves according to Morsicato the following issues:

- Understanding all situations in which hazardous conditions may occur
- Future changes to the system have to take this prior knowledge into account

The traditional “freeze up-front” approach addresses the first issue by freezing the design up-front, but has severe problems with dealing with the second requirement. Ad hoc approaches on the other hand allow to include changes in the system, but do not ensure that existing control mechanisms still work. Morsicato conceives XP as a third approach, which satisfactorily addresses both
1.4. Related Work

needs. By deploying iterative development it is more likely to find all failure modes, since knowledge about possible failures is created as the development progresses. Also the integration of control for new hazardous conditions is possible. Automated tests help to preserve existing functionality, thus also fulfilling the second requirement.

“Just because you’re doing XP doesn’t mean you abandon good software engineering practices. It means that you don’t have to pretend that you know everything there is to know about the system in the beginning.”

Ron Morsicato [PP02]

The article further highlights that refactoring as well as automated testing play an essential role for the development of reliable systems. According to Morsicato, refactoring is especially important in embedded software because one never knows how the device will work at the beginning of the software project. Especially weapon systems are often built with bleeding edge technology, so there is no way to get a complete picture of all the failure modes. Furthermore refactoring helps to simplify design and deal with inevitable design flaws. Morsicato believes that refactoring in combination with effective use of planning and on-going testing is the best way to develop safe software.

Another related article is “Multiple Perspectives on Executable Acceptance Test-Driven Development” [MM07] by Grigori Melnik and Frank Maurer. By utilizing a case study they investigate whether executable acceptance testing (EATT) is an effective tool for communicating, clarifying and validating business requirements on a software project. 13 to 18 people were working on the monitored project, which lasted 10 months. The customer role was represented by a domain expert, who was full-time available for the project. The FIT [MC05] testing framework has been deployed for the automated execution of acceptance tests. According to the authors, testers noted that FIT is simple and easy to learn.

In order to specify the acceptance tests, the customer often paired up with developers and testers, which fostered feedback, communication and team spirit. An commonly heard criticism of EATT is that customers are not able to write
acceptance tests themselves, i.e. it is too complicated. The case study showed that 40% of all tests were written by the customer, 30% by developers and the remaining 30% by testers. The customer testified that specifying acceptance tests was not particularly hard, the hardest thing was to be disciplined enough to do so. The authors admit that the study may not be representative, since the customer came from an information system background. Another interesting aspect of the study is that the initially specified test cases have been enhanced by further scenarios in 30 – 50% while implementing the user stories. This effect can be explained by the continuous learning about the domain and the system. The authors conclude that EATT has positive effects for customers, developers as well as testers. As pointed out before when discussing “Evaluation of Test-Driven Development” [WG07], customers appreciate the involvement in the development process. Developers valued that tests could be made more readable and intuitive in comparison to JUnit tests, but were less enthusiastic about FIT from time to time when specifying tests seemed to be more work than doing the same in JUnit. Testers pointed out several improvements from their point of view: the testing process became more focused, reoccurring bugs could be caught by specifying adequate tests and the code became cleaner.

Work discussed so far backed up findings by case studies or experiments. The book “Agile Estimating and Planning” [Coh06] by Mike Cohn does not provide such, but is substantiated by more than 20 years of experience. The book is of special interest with respect to this thesis, because it provides explanations why and how agile planning works. Starting with pointing out the purpose of planning and possible failings, Cohn illustrates how agile planning allows to develop plans which are more accurate and flexible. Central points are the cone of uncertainty as well as learning about the project and product. The former point describes the uncertainty with respect to the progression of the project. While estimations usually vary from 0.6 to 1.6 times the actual value in the beginning of the project, uncertainties decrease as the project advances. Thus the later estimations are made, the preciser they are. The learning about project and product is one of the explanations for the decreasing uncertainty. The more knowledge the team has, the preciser estimations can be made.
Cohn therefore suggests continuous planning as well as plan adoptions throughout the project if necessary. Plans which involve remote future should be only coarse grained and will be elaborated as more knowledge is gained. Thus estimations become more precisely, uncertainties and risks are decreased. Cohn stresses that agile planning shifts the emphasis from the plan to the planning.

The book “eXtreme Programming Explained” [Bec00] by Kent Beck introduces an agile software development process model. Basic ideas are to leverage people’s qualities, provide a sustainable social environment and most important “embrace change”. Using the metaphor of driving a car, Kent introduces the central motto: “Stay aware, adapt, change”. Thus the idea is not to ignore uncertainties, but to accept them and actively deal with them in the software development process. In the following values, principles, practices and expected benefits or drawbacks of eXtreme Programming (XP) are illustrated. Further nowadays widely accepted agile techniques and concepts like user stories, pair- and test-first programming are introduced.

1.5. Overview

The rest of the thesis is structured as follows:

Chapter 2 introduces relevant concepts of software development process models and the journey metaphor. Connections, similarities as well as limitations will be illustrated.

Chapter 3 deals with the software architecture of the Alaska travel simulator. Furthermore interesting aspects with respect to the adopted development approach are highlighted.

Chapter 4 explains the setup of the experiment conducted in course of this thesis. It introduces basic terminology, lists factors endangering the experiment’s validity, explains data analysis procedures and discusses the results.

Chapter 5 concludes the thesis with a summary.
Chapter 2.

Concepts

This chapter describes the concepts of a software development project and highlights connections to a travel scenario. Section 2.1 explains the planning behavior in plan-driven as well as in agile software development and compares the approaches. User stories are discussed in Section 2.2, subsequently Section 2.3 deals with project limiting factors. Section 2.4 deals with unforeseen events and dynamics of software development projects. Next, agile principles are illustrated in Section 2.5. The chapter is concluded by a summary and the interplay of concepts in Section 2.6.

2.1. Planning Behavior in Plan-Driven versus Agile Software Development

Software Project

This work focuses on the comparison of agile and plan-driven software development. Concepts relevant for the plan-driven approach are explained in detail in [Sch08], therefore only the basic notions will be introduced in order to see the main differences to the agile approach. Concepts related to agile software development will be shown in detail.

The agile development paradigm describes two main roles: customers and developers [Bec00]. The customer ensures that the software will meet the needs
Chapter 2. Concepts

of the user [Coh04]. He provides the requirements, answers questions about functionality and uses the software. Developers design, implement and test the software. Besides the major roles, many others exist, e.g. interaction designer, project- and product managers. Roles are not fixed and rigid, the goal for people is not to fill abstract roles, but for each team member to contribute all he can to the team [Bec00]. Furthermore extensive communication and collaboration is required, all people involved should think and work as a team [Bec00].

The progress of a project can be divided into different phases [Roy70]:

- **Requirement Analysis and Specification** The requirements of the software to be developed are collected, i.e. the customer is interrogated to provide information about the product intended to be build.

- **System Design and Specification** Having gathered the requirements, the system design is developed.

- **System Implementation** According to the system design and specification the software is implemented.

- **Integration and System Testing** To ensure a working system, possible modules are integrated into the final system, the functionality of the software is verified.

- **Delivery, Deployment and Maintenance** The software is delivered and deployed, the system is maintained.

No matter whether the development approach belongs to the plan-driven paradigm, including Waterfall Model [Roy70] and V-Model [Hes08] or to the agile ones like Extreme Programming [Bec00] or Scrum [Sch04], each project consists of the phases listed above. The main difference between the two paradigms with respect to planning consists in the way how the phases are organized and plans developed. Phases in plan-driven development are executed sequentially, stepping back to a previous phase is only allowed in exceptional situations. For instance, if the system design contains contradictions, which cannot be implemented, one has to take on step backward and redesign the system. The design of the software is
specified up-front and fixed. Agile approaches follow the iterative development model. The process is divided into multiple iterations, where each iteration runs through all phases. Furthermore the phases are interwoven and run in parallel. For instance in concurrent engineering [PP06] the specification emerges from the development process. After each iteration working software is delivered [Coh06]. Apparently not all functionality can be included, but since the product is working, it can be used already, thus generating business value. Furthermore it serves the customer to develop knowledge about their demands: The usefulness of existing functionality can be determined and new requirements can be discovered. Software is delivered more often; resulting feedback of users is gained quicker [PP06]. Therefore the software is constantly adapted according to the needs of the customer, resulting in a solution which fits the requirements of the customer.

It should be noted that some improved plan-driven process models also support iterations [Boe88], [JBR99]. Nevertheless the planning is done up-front, which differentiates them from agile approaches. Figure 2.1 shows the phases of Waterfall Model [Roy70] on the left as well as the overlapping phases of agile approaches on the right [TN86]. The two letter codes in the diagram are abbreviations of the phases introduced afore.

Instead of creating a fine grained requirements specification and design, only the most important and elementary features are taken into account [Coh04].

---

**Figure 2.1.: Comparison of Waterfall [Roy70] versus Iterative Model [TN86]**
Chapter 2. Concepts

Consequently after the first iteration, a first version of the system can be presented. When discussing the results of the first iteration with the customer, new requirements may show up [Coh06]. Since the system design is not too detailed, it is easier to introduce new requirements into the system. Additional knowledge about the project as well as the development process can be considered in the next iteration [Coh06]. Therefore not only the product evolves iteratively, but also the development process can be adapted according to the customer’s needs [DCH03]. In eXtreme Programming this concept is carried to the extremes, Kent Beck says: “Do the simplest thing that could possibly work”, meaning that only the uttermost important functionality should be implemented in each iteration. If necessary the feature can be extended in the next iteration. On the other hand, if it turns out that the functionality is actually not needed, not too many resources have been wasted.

“In practice, the detailed design of software always occurs during coding, even if a detailed design document was written ahead of time.”

Mary and Tom Poppendieck [PP06]

Competition as well as scarce resources require software projects to be efficient and on time while producing high-quality software. Ever-changing business demands depend on a development approach, which allows changing or adding requirements in a very flexible way [PP06]. Plan-driven approaches come up with a very detailed up-front plan, trying to include all possible uncertainties. When starting a plan-driven project, a lot of effort is spent on creating a plan, preparing exact requirements specifications and on a detailed design of the software. The documents created in this phase are successively used to implement, test and finally release and maintain the software.

While it is hard, but still manageable to design software from a fixed set of requirements, it is impossible to include all possible changes up-front. Therefore plan-driven approaches work well as long as the requirements do not change. Apparently requirements on software can change rapidly, especially when new business opportunities arise [PP06]. Therefore the software development process
must be able to comprise the changes in order to reduce time-to-market and support the company’s flexibility.

Agile approaches provide the required flexibility to deal with changes. As already mentioned, the development is divided into multiple iterations, enabling the customer to come up with new requirements after each iteration. Instead of focusing on the production of documents (e.g. plans), the planning activity plays a central role [Coh06]. In contrast to plan-driven development, planning becomes a continuous activity. Not only the product and process evolve, but also the plan. It may be sufficient to have a rather coarse plan first of all. When the project evolves, knowledge about the product (e.g. new requirements) and the project (e.g. Is a certain technology useful for our product?) are gained [Coh06]. By integrating these new facts the plan becomes more accurate and detailed, thus reducing risk and uncertainties.

Uncertainty represents another important aspect of planning. While it is widely known and accepted that each plan has to deal with uncertainties arising during the project, it is not clear how to treat them. Plan-driven approaches cope with this problem by putting a lot of effort in developing sophisticated plans in the beginning of the project, which try to incorporate unforeseen events. In addition to the missing flexibility arising, also another problem has to be taken into account: the accuracy of estimations. When a project is in the very beginning, estimations tend to be rather imprecise. [Boe81] describes the accuracy of estimations with respect to the phases of Waterfall Model. Based on these findings, Steve McConnell coined the term “cone of uncertainty” [McC96], which is depicted in Figure 2.2. It shows that typical estimations made in the beginning of the project vary from 0.6 to 1.6 times the final value. As development progresses, estimations become more and more exact.

Agile approaches resolve the issue of imprecision by continuously adapting the plan throughout the project. As described earlier, gained knowledge is incorporated into the plan [Coh06]. According to the cone of uncertainty, estimations become more accurate the later they are made. Therefore also the plan itself gets preciser as estimations are adapted. These observations show that it is not nec-
Chapter 2. Concepts

Figure 2.2.: Cone of Uncertainty

necessary, in most cases it will be even a waste of resources, to have a very detailed up-front plan, since the plan is very likely to be imprecise.

“Plans are nothing; planning is everything.”

Dwight D. Eisenhower [Eis]

It is important to stress that agile approaches also develop a plan up-front, but a not too detailed one; it is refined during the project. A common misconception consists of overlooking that agile approaches also put a lot of effort in developing plans. In contrast to plan-driven methods, planning plays an even bigger role, since it is expanded from the up-front phase to the whole project life-cycle.

Journey Metaphor

Journeys yield many aspects which can also be found in software development projects. In a voyage the goal is to maximize fun and travel experiences. Among many other factors, the financial situation, marital status and employment im-
pose certain restrictions. For instance, a typical employee in Austria does not have more than 5 weeks of holiday at his disposal, also his budget is restricted. Even though time and money are scarce, attractions are to be visited, trips to be made and a little bit of culture should not be missing. While traveling, unforeseen events may occur: roadblocks thwart the original traveling route or the traveler finds out about a great tourist attraction from a holiday acquaintance.

A quite similar situation can be found in software projects. Here the goal consists of maximizing the yield for the company. While implementing the project several constraints have to be obeyed: First of all, the budget must not be overrun and deadlines have to be kept. Furthermore the product has to meet certain requirements like usability, user friendliness, stability and many others. Finally the software should provide all the functionality the customer demanded. All of this should be accomplished, even if developers get sick or resign and new requirements appear.

The parallel, we are especially interested in, is the planning activity in journeys as well as in software projects. In both situations good planning is essential for success. Also the amount of planning required in both situations can be considered rather high. For instance, when traveling, an overall program should be assembled, hotels and flights must be booked and information about tourist attractions obtained. In software developments projects considerable effort must be put in planning to control complexity. Due to many parallels we consider journeys to be a suitable metaphor for software development projects.

Also for journeys the planning behavior can be plan-driven as well as agile. In the first case, the traveler predefines all attractions to be visited up-front and books them accordingly, thus fixing the schedule. An agile traveler would also identify interesting places up-front, but book only where necessary, e.g. the flight and accommodation for the first night. For the case the traveler wants to change the route or stay some days longer at some nice place, the plan-driven approach would involve canceling the bookings. The agile traveler has only booked where necessary, thus saving cancelation fees. In the plan-driven approach, planning is limited to the beginning of the voyage. The agile approach requires constant
planning in order to perform bookings. In exchange, the traveler is free to adapt his plan while traveling.

Like software development, journeys can be divided into different phases: up-front planning-, traveling- and finally the post production phase. The phases are described with respect to the agile planning approach. The plan-driven counterpart is illustrated in [Sch08].

**Up-front Planning Phase** In the very beginning of a journey, the traveler gathers information about possible travel destinations and figures out the most appealing tourist attractions. Taking into account the constraints, an initial plan is assembled, which comprises the identified activities as well as the traveling route and accommodations. Since the plan is developed in an iterative way, the first version may be not too detailed and cover only the most important parts like the flight and housing. Before the actual journey starts, bookings are made, if necessary. In terms of a software project, the team specifies the requirements and creates the initial project plan.

**Traveling Phase** When leaving home and starting the journey, the traveling phase is entered. From this point in time unforeseen events require the traveler to update the plan. For instance, road blocks can make the original route impassable. Besides suddenly appearing events, also more predictable factors must be accredited. The weather can be seen as such an influence, since it is neither totally deterministic nor completely random. Due to weather forecasts, the approximate weather for the next couple of days is known, but the exact weather cannot be predicted. Therefore the traveler must keep the actual weather forecasts in the back of his mind to update the plan accordingly and maximize his travel experience. After reorganization the plan must still satisfy the journey’s constraints. In this phase the continuous planning becomes apparent: while the journey progresses, the traveler updates the plan to comprise the afore mentioned factors.
2.1. Planning Behavior in Plan-Driven versus Agile Software Development

Software projects have to cope with similar problems. Requirements which pop up or change during development, or developers getting sick are typical examples of unforeseen events. Furthermore also projects have to deal with factors which can only be approximated. For instance, the time it takes to implement a feature can only be estimated. Whenever the time needed deviates from the estimation, the project leader must decide how to deal with additional / missing time. Like in journeys, the reorganization must not violate the project’s constraints.

As pointed out in [dL03], a deliberate dealing with risks like unforeseen events and uncertainties is essential for success. While ignoring risks very likely leads to failure, trying to excluding all risks is neither possible nor promising. Voyages as well as software development projects always imply certain risks, thus effective risk management is required in both situations in order to ensure success.

Post-production Phase   After finishing the voyage the post-production phase starts. In terms of a journey, the traveler may create a photo album, tell his friends about the experiences and keep in touch with holiday acquaintances. For software projects, the developed software must be deployed and maintained, for internal used software the staff trained.

To compare the phases in software development projects with the ones in the journey metaphor, 3 phases have been devised. Each one represents a different kind of activity. In the planning phase all planning activities are performed, during the execution phase the plans are implemented. Finally in the post-production phase the results from the execution phase are maintained.

Each phase in the journey metaphor is mapped to one or more of these 3 phases. The up-front planning is mapped to the planning phase, since only planning is done. The traveling phase is mapped to the planning – as well as to the execution phase, because the traveler is traveling, but is also constantly updating the plan according to unforeseen events and uncertainties. Finally the post-production phase is mapped to the post-production phase. Figure 2.3 shows the phases as well as the mappings.
2.2. User Stories

In agile software development projects, user stories [Coh04] are used to define the requirements and the basic unit of planning. In contrast to use cases [ABCP02], user stories are much less detailed and written by the customer or development team. A user story “describes functionality that will be valuable to either a user or purchaser of a system or software” [Coh04] and is composed of the following three parts:

1. Short written description of the story
2. Conversation about the story
3. Tests that convey and document details
2.2. User Stories

The description is rather short and provides just enough information to estimate the priority and effort it takes to implement it. The most important parts are the conversation and test, they provide the details of the story [Coh04]. When user stories are written, only part 1 is elaborated, parts 2 and 3 are postponed until the story is about to be implemented - the work is deferred until it is actually required. If it turns out during development that the user story is not useful anymore, not much time has been wasted.

Each user story is prioritized according its value, cost, risk and knowledge [Coh06] (more about prioritization in Section 2.5.1). The team evaluates how long it will take to implement each story, denoted in story points. Initially story points have no connection to the working hours it takes to build a feature. Nevertheless the relations between user stories must be correct, i.e. a story with 6 story points takes 3 times longer to implement than a story with 2 points [Coh06].

The prioritized user stories are then divided into iterations. The development team specifies how many story points can be achieved per iteration and assigns the stories accordingly. No iteration is allowed to contain more stories than can be expected to be finished, higher prioritized user stories must be finished first. The amount of story points which are completed in one iteration is called the velocity [Coh06]. The estimation of the velocity for the first iteration is likely to be rather imprecise, but is refined for subsequent iterations. Again the iterative nature enables the development team to adapt and improve the development process as the project progresses. If the velocity is known, story points can be related with real working hours.

Wrapping up, each user story runs through the states depicted in Figure 2.4. State NEW represents an identified, but not yet recorded user story. By writing the story, it switches to state WRITTEN. Remember that only a short description of the story is written down, just enough information to estimate the size of the story. Stories are prioritized, reaching state PRIORITIZED and assigned to an iteration, entering state SCHEDULED. Finally the story is implemented or canceled if not needed anymore, reaching states IMPLEMENTED & TESTED respectively CANCELED.
If a story could not be implemented in the current iteration, its new priority is assessed and scheduled in upcoming iterations.

**Journey Metaphor**

User stories are represented in journeys as activities, accommodations and routes, further on labeled by the umbrella term actions. For the plan-driven approach, actions are mapped to use cases [Sch08]. Each action is available in a location (the term is explained in 2.2.2) and can only be performed if the traveler is currently at the location. During the day the traveler attends a certain amount of activities, at the end of each day exactly one accommodation is required. Routes denote a special type of action, since they allow the traveler to get from one location to another. In context of software development, routes therefore depict dependencies between user stories.

2.2.1. States

The life cycle of actions is determined by the state chart diagram depicted in Figure 2.5. Each action starts in state NEW, where the traveler has considered the action to be interesting, but has not scheduled it yet. When the traveler actually decides to schedule the action (i.e. add it to the plan), the state switches to PLANNED. As long as the action resides in state PLANNED, the traveler is free to
2.2. User Stories

![Action states diagram](image)

Figure 2.5.: Action states, adapted from [Sch08]

reschedule it any time. If the availability (cf. section 2.2.8) is low, the action may be booked - moving to state BOOKED. At any point in time the booking can be canceled. If all preconditions have been satisfied (correct location, enough money, constraints not violated, ...), the action can be executed now - entering state STARTED. For the case the action can be executed successfully, state DONE is reached, FAILED otherwise. By canceling the action, CANCELED becomes the current state.

The life cycle of user stories and actions (cf. Figures 2.4 and 2.5) inhibit certain similarities. In both situations, the value of the action / user story (subsequently called item) is determined (cf. state NEW). Thereafter the item is scheduled, cf. states PLANNED / SCHEDULED. Loops between PRIORITIZED and SCHEDULED respectively PLANNED visualize adaptations which are required due to unforeseen events. Finally items may be executed or canceled if not needed anymore.

2.2.2. Location

Each journey consists of a set of locations the traveler can visit. Each location offers a set of actions. There exists a many-to-one relationship between actions
and location, i.e. each action is offered by exactly one location. Locations are connected by routes, which allow the traveler to get from one location to another. Locations represent prerequisites relations between actions, as the traveler has to move to another location before executing an action.

2.2.3. Weather

The weather plays a central role to simulate uncertainties. Good weather influences the travel experience in a positive way, while bad weather diminishes the gain. The weather changes from day to day, but not during a day. Each location has its own weather characteristics, described by stability and tendency. The stability $s_w \in [0;1]$ is a measurement for the fluctuation of the weather. Low stability results in labile weather, high stability in stable weather. The overall trend of the weather is specified by the tendency $t_w \in [-1;1]$. A negative tendency results on average in a weather which influences the travel experience negatively, analog for positive values. Figure 2.6 shows a sample of the weather simulation for $t_w = 0.2$ and $s_w = 0.5$.

2.2.4. Cost

To perform an action, the traveler must pay a certain amount of credits. There may be actions, which do not cost anything, but there are no actions the traveler can make money out of them. If the traveler desires to execute an action, the price must be paid immediately.

2.2.5. Duration

The duration describes how much time the traveler will spent for an action. In most cases, the time is not known beforehand, only a range with minimum and maximum duration is available. No action can last longer than a single day, therefore the maximum duration must not be longer than the duration of the day. The actual duration is computed using a symmetric triangular distribution
2.2. User Stories

Figure 2.6.: Exemplary devolution of weather for $t_w = 0.2$ and $s_w = 0.5$

function, bounded by the minimum and maximum duration. Accommodations play a special role with respect to duration: they always last all night long.

### 2.2.6. Business Value

The business value describes the gained travel experience. Since the goal of the journey consists of collecting as much experience as possible, the business value must be optimized. Each action inhibits a certain amount of business value. Even though the maximum achievable business value $BV_{\text{max}}$ is known upfront, the actual gained value can only be estimated due to the following uncertainties.

First, actions can be susceptible to weather $\text{rand}_{\text{weather}}$ (cf. section 2.2.3). The **weather influence** $\text{weather}_{\text{inf}} \in [0;1]$ describes how strong the business value depends on the weather. For actions with a weather influence of 0, the obtained value is independent of the weather. A value of 1 represents actions which strongly depend on the weather, i.e. the influence of the weather on the action is 100%. In other words, the weather may increase the business value to its
maximum or decrease to the minimum. Therefore it is of great importance for the traveler to take care of the weather, since it can influence the gains tremendously.

Second, each action has an inherent distribution \( \text{rand}_{\text{inherent}} \), which describes the distribution of the business value without considering the weather. Put differently, this factor describes how the traveler will conceive the action. For a typical action, the normal distribution represents the inherent distribution: Most likely the gained business value will be around the expected value, with moderate deviations.

Formula 2.1 defines the computation of the business value, taking into account the previously introduced factors. Note that the gained value can neither become negative nor greater than the maximum business value.

\[
BV = \left( \text{rand}_{\text{inherent}} + \left( \text{rand}_{\text{weather}} - \frac{1}{2} \right) \cdot \text{weather}_{\text{influence}} \right) \cdot BV_{\text{max}} \quad (2.1)
\]

### 2.2.7. Reliability

To assist the traveler by estimating the risk of an action, the reliability has been developed. It describes how strong the business value of an action varies in values of \([0;1]\). A low value represents actions, whose business value fluctuates strongly. High values mean that the gained business value will be very likely the expected one. The reliability takes into account the weather tendency \( t_w \) and stability \( s_w \), but not the actual weather, therefore the reliability estimation is also available in the up-front planning phase. During the traveling phase the traveler has to consider the current weather to adapt the reliability accordingly. Since the weather is known exactly, much preciser estimations can be made. In combination with the expected business value \( BV_{\text{exp}} \) the traveler has a good measurement to decide whether it is worth considering an action.

Formulas 2.2, 2.3 and 2.4 define the computation for the expected weather, reliability and expected business value. The expected weather \( w_{\text{exp}} \) is computed
from the weather tendency $t_w$ and stability $s_w$ as well as the expected value $\mu_{w}$ from the distribution function describing the fluctuation of the weather. The reliability $r_{BV}$ depends also on the variance of the inherent distribution $\sigma_{\text{inherent}}$ and the weather influence $w_{\text{influence}}$. For the computation of the expected business value $BV_{\text{exp}}$ the expected value from the inherent distribution $\mu_{\text{inherent}}$ and the maximum business value $BV_{\text{max}}$ are required.

\[
w_{\text{exp}} = \frac{t_w + 1}{2} \cdot s_w + \mu_{w} \cdot (1 - s_w) \tag{2.2}
\]
\[
r_{BV} = [1 - 2 \cdot \sigma_{\text{inherent}} - (1 - s_w) \cdot (1 - |t_w|) \cdot |w_{\text{influence}}|]_0^1 \tag{2.3}
\]
\[
BV_{\text{exp}} = \left( \mu_{\text{inherent}} + \left( w_{\text{exp}} - \frac{1}{2} \right) \cdot w_{\text{influence}} \right)_0^1 \cdot BV_{\text{max}} \tag{2.4}
\]

### 2.2.8. Availability

Not all actions are permanently available: Hotels may be booked up, attractions closed due to maintenance work. The availability $a \in [0; 1]$ describes how certain an action will be available. When executing the action, there will be a chance of $100 \cdot a\%$ that the action is executable. By booking an action the availability is ensured to be 1, i.e. the action is available for sure. Four aspects have to be taken into account when booking actions: First of all the booking deadline must be obeyed; It is only possible to book an action a certain amount of days before executing it. Secondly, the action must be paid immediately. Thirdly only a percentage of the paid money will be refunded when the action is canceled, according to the action’s cancelation fees. Finally booked actions cannot be rescheduled.
2.3. Project Specific Limiting Factors and Basic Conditions

**Software Project**

When developing software, multiple restrictions have to be taken into account, time and money are only the most obvious ones. According to the company, several guidelines / regulations may exist, which restrict the development process. For instance in closed-source development it must be ensured that the code does not get disclosed, no matter whether intentional or accidentally. Other companies may require extensive documentation produced throughout the development. To ensure satisfiable code quality, testing phases may be prescribed.

Furthermore there exist restrictions, which are specific for each project. User stories can have prerequisites or depend on each other. For instance, the implementation of the user interface can only be accomplished, if the business logic has already been finished. Other user stories are mutual exclusive, which can be the case if multiple variants have been proposed. Some user stories are mandatory, while other are optional and should only be implemented, if enough resources are left.

**Journey Metaphor**

Similar restrictions can be found in the journey metaphor: the time is limited by the duration of the day as well as the journey duration, also the budget is constricted. Accommodations represent activities, which have to be performed regularly, like testing or integration builds. Actions can be restricted by several constraints: prerequisite between two actions demands the execution of the first action as a precondition for the execution of the second one. The coexistence constraint specifies that either both or none of two actions may be executed, while mutual exclusion dictates that at maximum one action out of two can be executed. Such constraints are present in each journey’s planning: “We must take federal road B36 to get to the silver mine” (prerequisite) or “Either we go
mountain climbing or -biking, but cannot carry the equipment for both” (mutual exclusion).

Afore mentioned constraints can be divided into two categories: execution- and termination constraints. The first category represents constraints which restrict the team by implementing features / the traveler by executing actions (e.g. mutual exclusion). The latter class describes constraints constricting the completion of the project, iteration / journey. The coexistence constraint belongs to this category.

### 2.4. Project Dynamics and Unforeseen Events

#### Software Project

Unforeseen events can mitigate the success or even fail software development projects. Especially in highly competitive business fields, time-to-market as well as flexibility require a software development process which can quickly adapt to new requirements and incorporate changes. If the development process is too rigid, the company cannot take advantage of suddenly appearing business opportunities, therefore losing customers to competitors [PP06]. Not only new requirements demand a flexible approach, but also diminished resources like a shortened budget or sick developers.

Whenever the uncertainty in a project is high, agile approaches benefit from the iterative planning behavior (cf. Section 2.1). Since new requirements are collected after each iteration, new demands can be incorporated quickly, which in turn leads to sooner available features. On the other hand, when it turns out that user stories are not necessary anymore, they can be dropped [Coh06]. If the budget gets shortened or less developer are available, user stories with lowest expected business value can be omitted. The software can still be developed – according to the new availability of resources.

Set-based design [PP06] is another method to deal with uncertainty. For critical irreversible decisions multiple options are built and the best one is selected.
when the deadline for making the decision is reached. Therefore the decision which option is used can be deferred (cf. Subsection 2.5.3). By elaborating multiple options, knowledge about the product is created [PP06]. In [PP06] the usage of set-based design is illustrated by the following example: Consider a completely unmovable deadline. In order to meet the deadline, three teams A, B and C are arranged. Team A develops a very simple solution, team B the preferred solution, while Team C elaborates the ideal solution. The development of team A will certainly meet the deadline, while team B and C may not finish in time. Thus in any case a solution can be presented and the deadline will be met. In the ideal case, team C manages to finish the development and the ideal solution can be delivered.

Journey Metaphor

Unforeseen events are also very likely to happen in journeys. Consider a traveler exploring a country with his car. While traveling, road closures or traffic jams prevent the traveler from arriving in time or at all. In order to continue the voyage, he is forced to find out an alternative route and activities. Already booked actions have to be canceled and cancelation fees have to be paid. Therefore time and money invested in booking are wasted. In the context of software development, this reflects the resources wasted in specifying and designing all aspects of the software up-front. Even though the software would work perfectly, it will not be used, since the requirements have changed.

Consider again the traveler, who has now found out that there is a great tourist attraction available at the location he is currently staying. If he has already worked out and fixed the actions for the next few days, it will not be possible to visit the new attraction without paying high cancelation costs. In the sense of software development, a new requirement has shown up, which cannot be handled. In both situations a too rigid plan diminishes the success of the project. The rigidity of the plan is mainly determined by the commitments made - in the case of a voyage the bookings.
2.5. Agile Principles and Techniques

Set-based design can also be found the journey metaphor. It is not possible to actually perform two actions at the same time, but it is feasible to plan them in parallel. By scheduling actions in parallel, options are created. To illustrate the usage of set-based design consider a tourist attraction, which is susceptible to weather. It promises a high business value if the weather is good, but a very low one if the weather is bad. The traveler schedules an action, which is weather independent in parallel to the afore mentioned one. In the case of good weather, the first action will be chosen, for bad weather the second one. By adopting set-based design the traveler can cancel out the risk of unstable weather forecasts.

2.5. Agile Principles and Techniques

The following three subsections describe agile principles and techniques in software development as well as its connections to the journey metaphor. Subsection 2.5.1 deals with the concept of prioritization. Subsequently in 2.5.2 the elimination of waste is discussed. Finally 2.5.3 describes how to defer commitment.

2.5.1. Prioritization

Software Development

Virtually all software development projects are limited in time and money, therefore it may not possible to implement all intended user stories. Prioritization strives for implementing the features which will lead to the highest gain [PP06]. The idea of prioritization is motivated by the “80/20 principle” [Koc04] which states that 80 % of the value is delivered from 20 % of the features. Prioritization ensures that these 20 % are implemented. User stories are prioritized according to expected value, cost, knowledge and risk [Coh06].
Chapter 2. Concepts

**Value** The first factor involved in prioritization is the financial value of the user story. It defines how much money the organization will make or save by having the feature implemented.

**Cost** Another important aspect are the costs involved. The cost of a feature depends on the effort required for implementation, e.g. development time, testing and integration. The cost of a user story represents an essential prioritization factor, since monetary resources are usually low.

**Knowledge** Knowledge about the project and product is essential for success. Acquiring knowledge about the project enables the team to work efficiently, while knowledge about the product fosters the development of the intended software. By acquiring knowledge, uncertainty is reduced [Coh06]. Therefore user stories which help to gain knowledge improve the development process and product.

**Risk** According to [Coh06], “risk is anything that has not yet happened and that would jeopardize or limit the success of the project”. Typical risks of software development projects are schedule (“Are we on time?”), cost (budget overrun) and functionality (“Can we make that feature work?”) risk. The combination of risk and value is a useful factor for assessing the priority of a user story [Coh06]. The classification in high / low risk and value is illustrated in Figure 2.7. High-risk high-value user stories should be implemented first, since they deliver the most value and eliminate significant risk. Subsequently high-value low-risks should be treated, because they also offer high value, but do not help to diminish risk. As last, low-risk low-value feature are addressed - even though they yield little value, there is not much risk involved. High-risk low-value user stories should be avoided if possible - there is no reason for taking a high risk to gain only little value [Coh06].
2.5. Agile Principles and Techniques

![Classification in high / low risk and value](Coh06)

**Journey Metaphor**

Prioritization is also a central topic with respect to planning journeys. As described in Section 2.3, resources are restricted in voyages. Most vacation areas provide more attractions than the traveler can visit. Therefore the voyager must decide which activities seem more attractive. Such prioritization is performed in everyday’s journey: “Should I go parachuting or hiking?”. Personal preferences and interests are represented by the business value (cf. Section 2.2.6) in the journey metaphor.

Similar to prioritization in software development, value, cost and risk are taken into account. Knowledge does not play a role for prioritization, since there is no concept of gaining knowledge by executing actions in the journey metaphor yet.

**Value**  The value of the action, measured in business value.

**Cost**  Like in software development, money as well as time are the basis for the assessment of the cost. In journeys also the prerequisites have to be taken into account. For instance the drive to and from the action’s destination have to be included.
Risk In the journey metaphor, the risk of an action is determined by the weather, weather influence, inherent distribution and cancelation fees (cf. Sections 2.2.3, 2.2.6, 2.2.8). The traveler is assisted in assessing the risk by the reliability (cf. Section 2.2.7). Similar to software development, it makes sense to categorize actions according to risk and value (cf. Figure 2.7). Nevertheless the schedule strategy has to be adapted slightly. High-risk low-value actions should still be avoided and low-risk low-value actions scheduled last. There is a little difference for high-risk high-value and low-risk high-value actions. Since the risk defined in the journey metaphor cannot be eliminated by executing the action, there is no gain in scheduling high-risk high-value before low-risk high-value. Instead the execution should depend on the current knowledge of the risk factors (weather, weather influence and inherent distribution). For the inherent distribution and weather influence it is not possible to acquire additional knowledge. Conversely, the weather of the current day is known. Therefore the traveler should decide for high-risk high-value actions spontaneously and only if the weather is appropriate. On the other hand, low-risk high-value actions can be scheduled upfront.

2.5.2. Eliminate Waste

Software Development

The elimination of waste is one of the basic principles of lean software development [PP06]. Originally the idea of eliminating waste comes from the Toyota Production System, which is according to Taiichi Ohno “the absolute elimination of waste” [Ohn88]. The first step of eliminating waste consists of the recognition of value. Then, anything which does not add value or prevents the customer from immediately receiving it, can be removed [PP06].

In manufacturing inventory is waste, it has to be handled, moved, stored and retrieved [PP06]. In software development the inventory is the partially done work and has the same negative properties as inventory in manufacturing: it gets lost, grows obsolete, hides quality problems and ties up money [PP06].
The biggest source of waste in software development are extra features [PP06]. Again the “20/80 principle” [Koc04] comes into play: only about 20% of the features and functions in typical software are used regularly. Therefore a huge amount of development work has been wasted for features which are only rarely used, but add complexity to the software and thus increase costs of further development [PP06].

**Journey Metaphor**

Also in journeys the elimination of waste can help the traveler to yield more from invested resources. As pointed out before, the first step consists of determining what real value is - in the context of the journey metaphor the business value. Therefore anything which does not add business value should be avoided. Since business value is gained from executing actions only, the focus should be put on ensuring that the desired actions can be executed. All acting, which does not support the execution of actions and therefore the acquisition of business value can be considered waste. Money and time represent the basic resources of a journey. Availability and constraints also restrict the execution of actions and are therefore a source of waste. In order to eliminate most waste, the traveler has to focus on these factors.

**Money**  
Similar to software development, actions which do not deliver a high business value (with respect to their cost) can be considered waste. By prioritizing actions (cf. Section 2.5.1), actions yielding the most value are preferred, waste is diminished. Additional monetary waste arises from unnecessary or too early booking of actions, since due to cancelation fees only a part of the paid money is refunded.

**Time**  
Like wasted money, wasted time stems from actions whose ratio of business value to time is low. For routes this ratio is typically rather low, therefore a traveler should avoid too much needless driving. Another source of wasted time are plans which do not exploit the full length of the day. When abstracting from
the fact that travelers need rest, it would be the best to utilize every minute of the day for visiting attractions. Since the duration of actions is not exactly known in advance, the traveler must use the maximum duration of the action to estimate its duration in the plan (cf. Section 2.2.5). For the case the action did not take as long as expected, the remaining time can be used for additional actions.

Availability  Actions are only available to a certain percentage (cf. Section 2.2.8). Thus the traveler has to take care of the availability in order to ensure that the action is available, i.e. book the action if necessary. Booking ensures that actions are available for sure, but also creates monetary waste if the booking must be canceled and cancelation fees apply.

Consider the case the traveler plans to visit a museum, but forgets about the booking. When he arrives at the museum, no more visitors are admitted. Therefore he either waits until other visitors leave the museum or comes off empty-handed. In the first case this constitutes a waste of time, in the second also a waste of money.

Constraints  As described in Section 2.3, constraints restrict the execution of actions. In order to execute the most valuable actions, the traveler must take care of constraints which restrain their execution. For instance some action can only be executed by travelers who also perform some preliminary activity, e.g. for the advanced climbing course the basic course must be taken first. If the traveler forgets about the basic course, he is not admitted for the advanced course and therefore isn’t permitted for the more valuable action.

2.5.3. Defer Commitment

Software Development

The idea of deferring commitment consists of scheduling irreversible decisions for the last responsible moment [PP06]. If possible, decisions should be made
2.5. Agile Principles and Techniques

reversible, so they can be made and then changed easily [PP06]. In software
development, design decisions are a good example of commitments. If the de-
sign is fixed and cannot be changed easily, it may be difficult to introduce new
features which do not fit the design. By deferring the design decision to the last
responsible moment (or making it reversible), the software remains flexible.

A strategy for deferring commitment consists of introducing options in critical
regions [PP06]. The software design for instance is not required to be fully
flexible, but should leave up options at critical points. When setting up options
it is important to assess whether the option is actually required, since each option
causes additional cost. A software design can build options for persistency by
coding against a standardized database interface. The cost of this option would
be sacrificing vendor specific database calls. On the other hand, the decision
which database is utilized, is made reversible.

Journey Metaphor

Deferring commitment can also be found in the journey metaphor. According to
the life cycle of actions (cf. Section 2.2.1), the traveler is free to reschedule an
action as long it is not booked. A booked action can only be rescheduled if the
booking is canceled first. If high cancelation fees have to be paid, rescheduling
a booked action can be seen as a decision, which is not reversible without se-
vere financial loss. Therefore booking actions which imply high cancelation fees
resembles committing to an action.

As a flexible software design allows deferring commitment, options in a journey
enable the traveler to postpone the decision between actions. An option in a
travel plan is nothing but two or more actions scheduled in parallel. Typically
options are used to back up against weather independent actions. Consider
the case the traveler takes into account an action which is highly dependent
on the weather. Therefore the gained business value strongly correlates with the
weather, which is not known in advance. To deal with this uncertainty, a weather
independent action is scheduled in parallel. For the case the weather turns out to
be bad, the weather independent action can be executed. On the other hand for
good weather the execution of the weather depended may yield better results.

As in software development the usage of options may involve costs. If the
actions for building up the option have to be booked, at least one of them must
be canceled. Therefore the traveler must keep in mind that options are not for
free and only useful as long as the cost for building the options does not outweigh
the gains.

### 2.6. Interplay of Concepts

The concepts described so far are listed in Table 2.1, the left column lists the
concept in context of software development, the right one in terms of the journey
metaphor.

<table>
<thead>
<tr>
<th>Software development project</th>
<th>Journey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project plan</td>
<td>Travel plan</td>
</tr>
<tr>
<td>Software developer and customer</td>
<td>Traveler</td>
</tr>
<tr>
<td>Basic constraints, limiting factors</td>
<td>Location, budget, time, other constraints</td>
</tr>
<tr>
<td>Sudden change of requirements</td>
<td>Unforeseen events, weather, inherent distribution</td>
</tr>
<tr>
<td>Project time frame</td>
<td>Duration of vacation</td>
</tr>
<tr>
<td>Project budget</td>
<td>Financial situation of traveler</td>
</tr>
<tr>
<td>User stories (key features, non-critical ones)</td>
<td>Actions(routes, accommodations, activities)</td>
</tr>
<tr>
<td>Business value of implemented use case</td>
<td>Positive experience of done activity</td>
</tr>
<tr>
<td>Imprecise specification, unrealistic estimates</td>
<td>Uncertainty of an action’s cost and duration</td>
</tr>
<tr>
<td>Estimation of use case risk</td>
<td>Estimation of a non-booked action’s availability</td>
</tr>
<tr>
<td>Cost of canceling a feature’s implementation</td>
<td>Cancelation fee of a booking</td>
</tr>
<tr>
<td>Options for deferring commitment</td>
<td>Parallel scheduled actions</td>
</tr>
</tbody>
</table>

Table 2.1.: Comparison of elements in a software project and a journey [Sch08]

Table 2.2 summarizes the main differences between the plan-driven and agile
planning approach.

Figure 2.8 shows a business class diagram of all relevant objects of a jour-
ney. A Journey represents the central element of the model. For each Journey,
Constraints can be defined. The location attribute lists all Locations the trav-
erer can visit. Each Location contains several Actions, refined as Activity,
Accommodation and Route. In the right lower corner unforeseen events are
### 2.6. Interplay of Concepts

<table>
<thead>
<tr>
<th>Description</th>
<th>Plan-Driven</th>
<th>Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time spend on planning</td>
<td>High</td>
<td>High during the entire project</td>
</tr>
<tr>
<td>Time of planning</td>
<td>Beginning of project</td>
<td>Guideline, help for decision making</td>
</tr>
<tr>
<td>Perception of plan</td>
<td>Schema for execution</td>
<td>Making decisions at the last responsible moment</td>
</tr>
<tr>
<td>Strategies to deal with uncertainty</td>
<td>Careful upfront planning</td>
<td>reducing cost of change by planning for change</td>
</tr>
</tbody>
</table>

Table 2.2.: Comparison of plan-driven and agile approach [WW08]

modeled as Event, may occurring only in certain Locations and either introduce new Actions (NewActionEvent) or change the parameter of existing ones (ChangeActionEvent). Concepts involving Box and associated classes are explained in detail in [Sch08].
Figure 2.8.: Business class diagram of a journey [Sch08]
Chapter 3.

Architecture and Development

Approach

Aspects concerning the architecture of the Alaska travel simulator as well as the applied development approach are discussed in this chapter. This section only highlights the most interesting decisions and experiences and should not be seen as an extensive documentation. Section 3.1 starts by pointing out issues about the applied development approach. Sections 3.2 and 3.3 discuss the architecture of the Alaska travel simulator. Section 3.4 finishes the chapter by a short description of third party frameworks used.

3.1. Development Approach

This section highlights some interesting aspects of the development approach applied for the implementation of the Alaska travel simulator. We mainly adopted methods and practices of eXtreme Programming (XP) [Bec00]. Subsequently values, principles and practices, which were of special interest are illustrated. The focus is put on the application of the afore mentioned concepts, therefore they will not be explained in detail.
Chapter 3. Architecture and Development Approach

3.1.1. Values

Values are the roots of the things we like and don’t like in a situation [Bec00]. They form the basic building block of XP. Without adopting values, constitutive principles cannot work, practices will fail. We will now shortly illustrate how we fostered values described in [Bec00] as well as the resulting synergies.

Feedback and Communication

Especially in the implementation of the user interface helpful feedback could be gathered and used to improve usability and user-friendliness. In order to receive as much feedback as possible, we have set up a weekly meeting in which the latest advances but also problems could be discussed. In the initial phase of the project, feedback was achieved through acceptance– as well as unit tests. The development has been carried out iteratively, working software was delivered in each of the weekly meetings, thus permitting quick feedback. Since it was of uttermost importance to develop a user interface which is easy to understand and use, persons which were not involved in the development process evaluated the simulator. A value closely connected to feedback is communication. Since the development team has been rather small (2 people plus supervisor), direct and effective feedback as well as quick responses were possible.

Simplicity

Simplicity constitutes an especially important factor in our context. First of all the system must be simple and easy to use in order to ensure the internal validity of the experiment (cf. Subsection 4.6.1). Secondly the Alaska travel simulator will be further extended and maintained. In order to ease the task for successive development the architecture and program code have to be clear and concise. For this reason, code and architecture have been constantly refactored [FBBO99]. Additionally, the application’s architecture has been created incrementally [Bec00]. By adopting this technique it was possible to keep the
3.1. Development Approach

design simple and avoid unnecessary generalities. Furthermore we always tried to implement “the simplest thing that could possibly work” [Bec00] and realized only features we were sure that they were actually required, thus minimizing unnecessary code and eliminating waste [PP06].

Courage and Respect

As mentioned before also persons who were not involved in the implementation provided useful feedback for the improvement of the simulator. Dealing with this kind of feedback requires having respect for those people. Problems disclosed by outstanding persons tend to be considered as “obvious” and “trivial”, even if their resolution is essential for a well working application. Thus before classifying feedback as useless, we took a step back and reconsidered issues. For giving feedback, a considerable amount of courage is required. When discussing we tried to be as critical and objective as possible, even if the results of the discussion imposed severe effort in terms of refactoring or disposing existing code.

3.1.2. Principles

Principles bridge the gap between values and practices, they represent domain-specific guidelines for life [Bec00]. Extreme Programming describes several principles, two of them were of especial interest for our development: quality and improvement.

Quality

The quality of software in terms of usability and robustness plays a central role in the Alaska travel simulator, since the results of the experiment (cf. Chapter 4) rely on its accuracy. Errors in the program could lead to inaccurate measurements and therefore to distorted data. If the program crashes during the experiment, precious data is lost. To countervail errors, pair– and test-first programming (cf. 3.1.3) have been deployed. The creation of automated tests using JUnit [Bec04]
Chapter 3. Architecture and Development Approach

as well as FitNesse [MC05] ensures the correct operation of the application, additionally the risk of introducing errors by refactoring [FBBO99] is reduced.

**Improvement**

“Do the best you can today, striving for awareness and understanding necessary to do better tomorrow” [Bec00] describes the idea of steady improvement. This principle could be observed best in the evolution of travel simulator’s user interface. There has been no graphical user interface while developing the business logic except FitNesse tests. Early versions of the simulator included only the basic features, which have been assessed in our weekly meetings. The feedback helped devising new features, which were actually needed and fitted the user interface best. Therefore continuous improvement has been a key principle for the implementation.

3.1.3. Practices

Practices are techniques which are applied by XP teams day-to-day [Bec00]. This section shortly describes the most important practices we adopted. More specifically, the usage of *pair programming* and *test-first programming* will be illustrated.

**Pair Programming**

The idea of pair programming consists of letting two programmers work on the same machine. Through the close collaboration, very short feedback loops are possible. The resulting code is generally less likely to have errors and is more compact [CW01], [Nos98]. As pointed out in [Bec00] it is up to the team to decide whether pair-programming is useful for the current development work. Pairing for the implementation of sophisticated algorithms is certainly a good idea, while routine tasks perhaps do not require pair programming. We made use of pair programming for the development of the business logic, but decided to
implement the user interface by ourselves. In retrospect the pairing was essential for developing business logic which was flexible enough to support the simulation of agile as well as plan-driven journeys.

**Test-First Programming**

As pointed out in Subsection 3.1.2, quality played a central role in the development. In order to foster quality, we adopted test-first programming [Bec02]. When using the test-first technique, automated tests are written before coding. JUnit [Bec04] and FitNesse [MC05] (cf. Subsection 3.4.3) were chosen as testing frameworks. JUnit focuses, as the name suggests, on the support of unit testing [Bec02]. Tests are written in Java [Ull07] and therefore only readable by developers. In FitNesse, tests are expressed as tables as part of a HTML page and can be edited by a Wiki [Klo06]. When talking about testing scenarios, FitNesse tests are much more convenient than JUnit tests, since they have a graphical representation. For further clarification comments and images can be included. Creating and editing tests is also possible for domain specialists. The usage of JUnit in combination with FitNesse allowed us to develop comprehensive tests: Single units of code have been tested with JUnit. FitNesse tests were used for testing functionality which involved multiple units.

“If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.”

_Edsgen W. Dijkstra [Dij72]_

**3.2. Alaska Toolset**

The Alaska travel simulator is the major application which has been developed for the conduction of this thesis’ experiment. Meanwhile several other components are available in order to support the creation, maintenance and evaluation of journeys. Figure 3.1 shows their interplay. The life cycle begins with the *Alaska*
travel configurator, which provides a convenient graphical user interface for the creation and maintenance of journey configurations. Journey configurations are deployed as zipped files, containing the configuration as well as additional resources (e.g. images). The Alaska travel simulator imports these configurations at startup and enables the player to undertake the journeys using either the agile or plan-driven user interface. All relevant data is logged through a data access interface to an arbitrary data store. At the moment, MySQL [Kof07] and XML file storage are available. Further applications also access the log data via the provided data access interface. For instance, the Alaska report generator [Tri08] creates reports including basic statistical metrics, e.g. mean business value and cost. For more sophisticated data analysis the Alaska data evaluator is currently under development.

The implementation of the travel simulator has been part of this thesis. The travel configurator and data evaluator have been funded by the e-learning [Ele08] initiative of the University of Innsbruck. Finally the report engine has been developed at the University of Bolzano as part of a Bachelor thesis.
3.3. Alaska Travel Simulator

The Alaska travel simulator has been implemented as an Eclipse Rich Client Platform application (cf. Subsection 3.4.1) and therefore consists of a composition of plug-ins. Architectural considerations will be illustrated subsequently. Figure 3.2 shows the plug-ins involved in the application, third part plug-ins are colored light gray, own plug-ins slightly darker. The plug-ins Graphical Editing Framework and XStream are shortly described in Subsections 3.4.2 and 3.4.5. Alaska Core, Alaska UI and Alaska Help have been developed on our own and will be explained in detail now. Please note that subsequent class diagrams are intended to sketch the architecture’s concepts only and therefore do not represent a detailed documentation of the system.

The travel simulator has been implemented by Michael Schier and myself. Business logic which is used in the plan-driven as well in the agile approach has been developed together. Also visual components which are shared by both approaches were created corporately. Based on the common components, Michael wrote the user interface for the plan-driven approach, my task was to implement the agile counterpart.
Chapter 3. Architecture and Development Approach

3.3.1. Alaska Core

In the plug-in *Alaska Core*, the business logic of the travel simulator is contained, i.e. it implements the concepts discussed in Chapter 2. Conceptually classes can be divided into static as well as dynamic behavior. The first category deals with objects, which do not change during the simulation of the journey. Information like possible events which may occur or locations belong to this category. On the other hand the core plug-in also has to deal with dynamic aspects of the journey like the weather or the journey plan. In addition to the purely business logic related implementation also the logging mechanism is included in this component.

**Services**

The exchangeability of services [Erl05] has been interesting for the implementation of the business logic. It allowed to decouple certain functionality during the development and foster modularization. For instance, the computation of the weather’s devolution in a journey has been implemented as a service. In the first place, a random function was sufficient. Later on a more sophisticated weather behavior was needed; the random weather service was simply replaced by a more sophisticated one.

In more technical terms, a service is specified by a Java interface. Figure 3.3 shows the interplay of an exemplary service, the registry and a client. The interface `IWeatherService` defines the behavior of the service, `RandomWeatherService` and `SophisticatedWeatherService` are compliant implementations. Services register themselves at the `ServiceRegistry` and use the class of the service interface as unique key for registration. Consequently at most one implementation of a service is available at the same time. Clients use the service registry to retrieve a specific service, using the service `interface` as unique identifier. Calls to services should always be made through the service interface in order to guarantee exchangeability.
3.3. Alaska Travel Simulator

Logging and Persistence

Besides the business logic the logging mechanism is an essential part of the core plug-in. It provides support for persisting journeys by logging each planning relevant step, making it possible to reload journeys step-by-step (cf. Subsection 3.3.2). The data can be currently logged either to a XML file or to a database. In order to be able to use the logging data for further analysis in other tools, generated XML files conform to the MXML format [MXM]. For demonstration purposes the XML file serialization is ideal, since no additional setup is required. The database variant provides the necessary support for the experiment’s setup (cf. Chapter 4).
Figure 3.4 illustrates the architecture of the afore mentioned concepts. The access to log data is defined by IPromReader and IPromWriter, which allow to read / write in-memory representations of the MXML format. The specific implementations implement the unmarshaling / marshaling of data, currently available for XML and databases. Application code which makes use of this persistence support always deals with classes representing the MXML format. The responsibility of AbstractLogger and subclasses is therefore to convert a planning relevant step of a journey into an element of the MXML format. Implementations of IJourneyRestorer perform the conversion in the opposite direction. By separating the concerns of logging and persisting we avoid duplication and ease the support of new storage methods. The construct of IPromWriter and IPromReader as well as implementations constitute a usage of the Strategy Pattern [GHJ94].

Action Factories

Figure 3.5 shows the parallel hierarchies of configuration proxies and Action objects. Proxies represent tourist attractions available in a journey, while Action objects constitute attractions, which the traveler has scheduled already. To keep the code generic, in most cases the specific subtype of proxy is not known. If an Action object should be instantiated for a proxy, the developer does not know whether he must create an Accommodation, Activity or Route. Therefore the responsibility of creation is moved into the ActionConfigurationProxy class, subtypes implement the behavior appropriately. The proxy class acts as a factory for Action objects, thus implementing the Factory Pattern [GHJ94].

Configuration Proxies

Among other factors, the business value of an action can change due to events. In order to isolate these changes from other games, travel specific objects do not directly access the journey configuration, but through a proxy object. Any modifications caused by events are stored in these intermediary objects. Since
3.3. Alaska Travel Simulator

Figure 3.5.: Proxies act as factories for the creation of Actions

Figure 3.6.: Proxy caches a configuration

proxy objects are travel-specific, changes are not propagated to other travels. This structure can be seen as a modified version of the Proxy Pattern \cite{GHJ94} with the same intention but no common supertype of proxy and “proxied” class. The common supertype is intentionally missing in order to prevent developers from confusing configurations with their corresponding proxy. Figure 3.6 depicts the interaction of configuration, proxy and client.

3.3.2. Alaska User Interface

All concepts related to the user interface of the travel simulator are implemented as part of the UI plug-in. It makes use of the core plug-in’s business logic and provides the visual support for undertaking travels. The experiment conducted in course of this master thesis compares the agile with the plan-driven approach. Therefore the user interface provides tools supporting agile as well as plan-driven methodology. In order to reach a consistent user interface, common components have been developed which are used in either approach. Besides the classical
Chapter 3. Architecture and Development Approach

Figure 3.7.: User interface model for separation of business logic and ui

object oriented concepts of reuse by inheritance and composition [Joh99], the usage of Eclipse views [CR08] proved to be a convenient technology for reusing visual components. For developing a rich user interface GEF (cf. Subsection 3.4.2) has been utilized. Subsequent architectural issues of the user interface are discussed.

User Interface Model

As shown in the plug-in composition of the Alaska travel simulator (cf. Figure 3.2), the user interface plug-in builds on the core plug-in. In order to separate the business logic components in the core plug-in from the user interface, a user interface model has been inserted in between. Figure 3.7 shows the user interface model on the left and the core components on the right. Classes of the user interface model reference core classes and provide additional user interface specific functionality. If a core class has multiple visual representation, multiple user interface model classes exist, cf. AgileAction, PlanDrivenAction versus the single core class Action. Concepts regarding the user interface only do not have corresponding core classes, cf. PlanningArea in the user interface model, but no associated class in the business logic. The structure can be seen as an application of the Decorator Pattern [GHJ94], where user interface model classes enhance the functionality of core classes.
Restoration of Journeys

The restoration of journeys plays an important role with respect to the evaluation of the experiment (cf. Chapter 4). Loading the final state of a journey does not suffice, since the planning behavior must be observable. Put differently, it must be possible to load a journey step-by-step to analyze the journey appropriately. In order to provide the necessary fine-grained restoration support, each planning relevant step is represented as a command. When restoring the journey, commands are executed sequentially, allowing the user to run through the journey step-by-step. The command encapsulates all necessary information to carry out the step. If a step-by-step analysis is not required, all commands are executed once, therefore reaching the end of the journey.

The journey restoration mechanism is an application of the Command Pattern [GHJ94]. Figure 3.8 illustrates the architecture’s structure. An AbstractLogRestoreCommand provides basic functionality required for all restoration commands. AbstractAgileLogRestoreCommand includes support for the restoration of agile journeys, AbstractPlanDrivenLogRestoreCommand for plan-driven ones. Subclasses represent concrete commands which are responsible for the restoration. For instance, MoveAgileActionCommand moves an action within the agile user interface. Commands which are applicable to either approach directly subclass AbstractLogRestoreCommand. The JourneyRestorer executes the commands, as pointed out either one by one or all at once.

Another interesting aspect of the journey restoration is the update mechanism of the user interface. As illustrated before, we devised a user interface model in order to separate the business logic from user interface components. User interface model classes act as Observables, the user interface itself as Observer, cf. Observer Pattern [GHJ94]. In other words, the user interface registers Listeners at the model classes. Whenever changes in the model occur, listeners are invoked and the user interface is updated accordingly. In the context of the journey restoration this behavior simplifies the implementation of the commands.
Changes are only made to the user interface model and propagated to the user interface accordingly.

3.3.3. Alaska Help

The help plug-in enhances the travel simulator with documentation and tutorials. It relies on Eclipse’s help system [CR08], which provides support for well-structured and easy accessible help sites. The help system basically consists of a web server providing the help pages as HTML. The structure is specified in XML files, for which Eclipse’s IDE provides a convenient editor. Due to the web server based architecture it is possible to either integrate the help functionality in RCP applications or run the server standalone and let the client access the help pages via the network.

3.4. Third Party Frameworks

Third party frameworks involved in the implementation of the Alaska travel simulator are shortly illustrated here. Besides introducing used technologies, it should be motivated why they have been used. Therefore the focus is put on key features and special benefits or drawbacks as well as experiences made. Sub-
section 3.4.1 shortly introduces Eclipse’s Rich Client Platform, 3.4.2 highlights the most important facts of the Graphical Editing Framework. Subsections 3.4.3 and 3.4.4 deal with testing frameworks FitNesse and JUnit. In Subsection 3.4.5 the XStream persistence library is explained.

3.4.1. Eclipse Rich Client Platform

The Eclipse Rich Client Platform (RCP) is a generic platform for running applications [ML06]. It enables developers to build software on top of the Eclipse infrastructure. The reuse of existing functionality comes along with several advantages. First of all, reoccurring routine tasks can be delegated to the platform. Further on, these components are well tested and maintained by the Eclipse development team, which ensures the functionality and compatibility of subsequent versions. The flexible architecture of the Eclipse platform allows developers to integrate any third part library, the Eclipse Foundation itself provides numerous components.

Another major advantage of using RCP stems from it’s flexible architecture, which is based on the concept of plug-ins. The platform runtime constitutes the platform’s basis. When the application starts, it searches for available plug-ins and installs them accordingly. Thus any component, which is contributed to a RCP, must be in form of a plug-in – any RCP application can be seen as a compilation of plug-ins. By forcing the usage of plug-ins, the reuse of existing software as well as modularization is fostered.

Figure 3.9 shows the described architecture. The platform runtime on the very bottom is responsible for the correct handling of plug-ins. The standard widget toolkit (SWT) provides basic user interface components, whereas JFace supplies higher level support [CR08]. Finally the UI plug-ins implement top level concepts for ui structuring, e.g. perspectives, editors and view, cf. [CR08]. Custom plug-ins contribute additional functionality. In this context, also the Java Development Tooling (JDT, the Java development environment of Eclipse) can be seen as a RCP application.
The learning curve of RCP is rather steep in the beginning, since many concepts have to be understood. Nevertheless the initial effort pays off immediately if a more sophisticated user interface is required. As already mentioned before, the platform comes also in handy when implementing routine tasks. Packaging and deploying applications does not constitute problems either, the Eclipse IDE [DFK+04] comes along with appropriate tool support. The only drawback to mention is the amount of unnecessary code which is contained in auxiliary plug-ins. In the case of the Alaska travel simulator, less than 5% of the application’s size stem from our application code.

3.4.2. Graphical Editing Framework

The Graphical Editing Framework (GEF) provides support for the development of rich graphical representation of models [MDG+04]. It is freely available from the Eclipse Foundation under terms of the Eclipse Public License. GEF is build upon the graphical drawing framework Draw2D, which provides a lightweight graphical system based on heavyweight SWT controls. Lightweight in this context means that components from the Draw2D framework are not dependent on the underlying operating system. More precisely, Draw2D handles mouse and painting events occurring from a SWT control and delegates them to figure objects. These are in turn responsible for painting themselves accordingly.
Therefore figures are simple Java objects, which can be created and destroyed independent of the underlying graphics system, which makes the painting more efficient and flexible.

Another interesting architectural aspect of GEF is the support of the Model-View-Controller pattern [GHJ94]; the application can be clearly separated into model, view and controller. Afore mentioned figures represent the view classes. **EditParts** create and manage figures and update the model accordingly, they represent the controller classes. An **EditPartFactory** is responsible for creating controllers for given models.

### 3.4.3. FitNesse

According to its developers, FitNesse is a software development collaboration tool. It enables customers, testers and programmers to learn what their software should do, and to automatically compare that to what it actually does [MC05]. The core of FitNesse is the Fit [MC05] acceptance testing framework. Fit basically offers the possibility to read HTML pages and invoke the system under test as specified in the pages. According to the test outcome, corresponding HTML pages are created which report success, failures and errors. FitNesse enhances Fit with a Wiki and web server in order to provide a convenient environment for test execution, creation and maintenance.

Figure 3.10 shows the interplay of the HTML pages with FitNesse and the system under test. Tests are formulated in a Wiki and represented as tables. If the tests should be executed, an ordinary HTTP request is sent to the FitNesse web server, specifying the tests to be run (1). FitNesse uses the Fit framework to parse the corresponding page and determines the tests to be called as well as the parameters. For the communication between FitNesse and the system under test, fixtures are required. In the case of Java applications, a fixture is an ordinary class which subclasses existing fixtures of the framework. It’s purpose is to convert the request from FitNesse (2) to calls of the system under test (3). FitNesse again uses the Fit framework to collect and interpret the results (4).
and to generate appropriate HTML pages which are sent back to the requester (5). In the example table of Figure 3.10 the business value of actions Climbing, Hiking and Biking is compared. The first two values match the expected ones, the second is 50 lower than expected - table cells are colored appropriately.

The usage of FitNesse has proved to be especially beneficial during the development of the business logic. The graphical representation of the tests helped to discuss relevant features at the appropriate level of abstraction and made the current functionality easier to explain. In comparison to JUnit (cf. Subsection 3.4.4) the initial effort for the creation of a test is high, since also the fixtures have to be implemented. This drawback is outweighed quickly if the number of tests using the same fixtures increases.

### 3.4.4. JUnit

JUnit [Bec04] is a framework for testing Java applications, focusing on unit tests. It is widely used in software development and freely available. Several tools have been developed for a convenient integration of the framework: including automated build server scripts for running tests, support for IDEs and many more. In comparison to FitNesse (cf. Subsection 3.4.3), tests are represented by Java classes. The framework examines test classes and invokes methods which
match the criteria for test methods via Java’s reflection mechanism. We decided for the usage of JUnit since it is easy to use, well integrated in the Eclipse IDE and freely available. Furthermore it’s functionality supplemented with the features of FitNesse very well.

3.4.5. XStream

XStream [XSt] is a framework for XML persistency support. We decided for XStream since it is flexible, small (about 400 kb) and open source. Positive experiences with the software backed up our decision. The framework is freely available under terms of the Berkeley Software Distribution License [Grü06]. XStream makes use of Java’s reflection mechanism and comes along with a default persisting strategy, therefore no cumbersome XML format specification is required. Nevertheless it provides enough flexibility to deal with more advanced problems like the Eclipse class loader hierarchy [CR08].
Chapter 4.

Experiment

This chapter deals with the experiment conducted in the course of the master thesis. It starts with the explanation of the basic terminology in Section 4.1 and presents the experiment’s design in Section 4.2. In Section 4.3 the execution of the experiment is illustrated, in Section 4.4 the validation and analysis of gathered data is performed. Section 4.5 presents the experiment’s results, Section 4.6 deals with risks endangering the validity of the experiment and countermeasures. Finally Section 4.7 concludes this chapter with a discussion of the results.

4.1. Basic Terminology

Before the actual experiment will be explained, basic terminology is introduced. Subsections 4.1.1 to 4.1.6 explain all relevant concepts in detail, Figure 4.1 shows their interrelations. Subjects act on objects, relevant data is measured as response variables. The experimenter can influence subjects by adapting independent variables and gauge any difference in the subjects’ behavior. If the collected data allows the inference of statistically significant results, the stated hypothesis $H_0$ can be rejected.
Chapter 4. Experiment

4.1. Objects

The physical entities on which the experiment is run are called objects [JM01]. Objects can be material as well as immaterial, e.g. a task, problem or product. In experiments conducted in software development, the development process or some methodology may be the object.

4.1.2. Subjects

The entity who applies the methods or techniques to the objects is called the subject [JM01]. Subjects may be persons, physical or non-physical things and serve as source for experimental data. When conducting the experiment, subjects are divided into groups. The experiment’s setup defines the environment for each group, allowing the experimenter to reason about the influence of the environment’s parameters on the subjects. In the context of experiments in software development, programmers could be subjects, which apply a certain development process (object).

Figure 4.1.: Basic concepts of an experimental setup [Sch08]
### 4.1. Basic Terminology

#### 4.1.3. Independent Variables

Independent variables, also referred as factors, are those that are manipulated by the experimenter [HL05]. It is assumed that all other impacts on the results can be kept stable. Therefore by running the experiment multiple times with different values (called factor levels) for the independent variable the experimenter can gain information about the influence of the independent variables. For a controlled experiment the set of independent variables is kept small. The more independent variables exist, the more complex their interrelations become. If only a single independent variable occurs, we speak of a single factor experiment [KL99].

To illustrate the concept of independent variables, consider an experiment which deals with the usefulness of automated tests. The subjects would be the developers who write the automated tests, the object is the usage of automated tests, the independent variable the amount of automated tests. The experimenter could then vary the amount of automated tests and check whether there is any correlation with the quality of the developed software.

#### 4.1.4. Response Variables

The outcome of an experiment is referred as the response or dependent variable [JM01]. According to the experiment’s type, multiple response variables may be present. Response variables depend on the independent variable and therefore measure the effect of applying the independent variable on the factor levels. The analysis of the results is based on the correlations between dependent and independent variables.

In the thought experiment involving automated tests illustrated in the preceding section (4.1.3), the quality of the developed software represents a response variable, the amount of automated tests an independent variable. By using this setup, the experimenter can point out connections between the usage of automated tests and the quality of the resulting software.
4.1.5. Experimental Designs

The design of the experiment must be devised carefully in order to ensure the validity of the experiment. According to [TD07], the major research designs can be classified into three types: randomized, quasi-experiment and non-experiment. For randomized experiments a random assignments into groups can be achieved. If no randomized assignment is possible, but multiple measurements or groups are used, a quasi-experiment is conducted. If neither randomization nor multiple measurements or groups are used, we speak of a non-experiment. Figure 4.2 depicts the classification.

The internal validity of an experiment is the approximate truth about inferences regarding casual relationships [TD07]. It is an indicator whether the effects on the response variables are caused by the independent variables or alternative causes. Randomized experiments provide the highest internal validity, therefore the experimenter should strive for randomized experiments in order to ensure the validity of the analysis.
4.1.6. Hypotheses

A hypothesis is a specific statement of prediction. It describes the results the experimenter expects [TD07]. In the context of an experiment, hypotheses always appear in pairs. The null hypothesis represents the contrary of the experimenter’s prediction, while the complementary alternative hypothesis resembles the prediction. This pair of hypotheses is always mutual exclusive, i.e. only one of them can hold. For the null hypothesis we use the abbreviation $H_0$, for the alternative $H_1$.

Based on the type of prediction, a hypothesis can be either one-tailed or two-tailed [TD07].

One-tailed One-tailed hypotheses specify a direction, i.e. either an increase or decrease is expected. An experiment which investigates in the effect of a fertilizer could formulate the one-tailed hypothesis that using fertilizer A yields a larger crop than using fertilizer B.

Two-tailed Hypotheses which do not specify a direction are called two-tailed. In other words, the hypothesis states that results will differ significantly, but does not say anything about the direction. For instance a drug treatment for depression has been tested on animals, but not on humans. The experiment’s two-tailed hypothesis may be that the medicament has a significant impact, but it is not sure whether the intended or side-effects are stronger.

4.2. Experiment Design

The design of the experiment conducted for this master thesis is explained in this section. Table 4.1 shows a short summary of the most relevant design decisions, Figure 4.3 illustrates the interplay.

As subjects Bachelor students of Computer Science at the University of Innsbruck were chosen. The students were asked to plan journeys using the Alaska
Chapter 4. Experiment

<table>
<thead>
<tr>
<th>Design terminology</th>
<th>Corresponding element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subjects</td>
<td>Bachelor students of Computer Science</td>
</tr>
<tr>
<td>Objects</td>
<td>Two game configurations of Alaska</td>
</tr>
<tr>
<td>Independent variable</td>
<td>Planning approach: agile or plan-driven</td>
</tr>
<tr>
<td>Response variable</td>
<td>Overall business value of journey</td>
</tr>
<tr>
<td>Experiment design type</td>
<td>Randomized single factor experiment with repeated measurement</td>
</tr>
<tr>
<td>Hypothesis</td>
<td>The adoption of the planning approach does not increase the reached business value</td>
</tr>
</tbody>
</table>

Table 4.1.: Experiment’s design

travel simulator, one in an agile manner, the other one following the plan-driven approach. In order to prevent learning effects, two journeys have been provided: Alaska and California. These journeys resemble the experiment’s objects. The journey to Alaska exhibits a higher degree of uncertainty and unforeseen events in comparison to the Californian travel. Furthermore Alaska’s actions involve a higher risk on average. Therefore planning the Californian voyage requires less effort for dealing with uncertainties and can be considered easier. The difference in the journey scenarios forces a separate analysis of the Alaska’s and California’s data.

As independent variable the deployed planning approach is used. The gained business value represents the dependent variable.

Figure 4.4 shows the basic setup of the experiment. The subjects are randomly divided into two groups of equal size, subsequently called group A and group B. For the first run the Californian travel scenario is selected. As pointed out before, less uncertainties and risk are involved in this scenario. Students start with this scenario to make the beginning a little bit easier. Group A starts with the plan-driven approach, group B with the agile one. After completing the journey, groups swap the approach and plan the journey to Alaska. By using this setup each subject makes use of both planning approaches and takes part in both travel scenarios.
4.2. Experiment Design

Chapter 4. Experiment

Design terminology

<table>
<thead>
<tr>
<th>Corresponding element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
</tr>
<tr>
<td>Bachelor Computer Science program students</td>
</tr>
<tr>
<td>Object</td>
</tr>
<tr>
<td>Two game configurations of Alaska</td>
</tr>
<tr>
<td>Independent variables</td>
</tr>
<tr>
<td>quasi independent:</td>
</tr>
<tr>
<td>choice of using design deferring techniques</td>
</tr>
<tr>
<td>Response variables</td>
</tr>
<tr>
<td>overall business value of journey</td>
</tr>
<tr>
<td>absolute change frequency of game plan</td>
</tr>
<tr>
<td>Experiment design type</td>
</tr>
<tr>
<td>Quasi-experimental design, unbalanced single factor experiment with repeated measurement</td>
</tr>
<tr>
<td>Hypotheses</td>
</tr>
<tr>
<td>Design decision deferring techniques do not have an impact on business value and number of project plan adjustments</td>
</tr>
</tbody>
</table>

Table 4.1: Elements of this experiment’s design

Game configurations
Bachelor students
Recording of data
Approach: agile/plan-driven
Business value
Hypothesis
H0
XOR
confirm
disprove
Plan journey
in
Alaska
California

As a consequence, the experiment can be seen as randomized single factor experiment with repeated measurement investigating the effects of the adoption of the planning approach on the business value. The interest in the relation between these parameters leads to the formulation of the following one-tailed null hypothesis:

- $H_0^{BV}$: The adoption of the planning approach does not increase the reached business value.

To gather data required for analysis, a logging mechanism has been implemented in the Alaska travel simulator. All relevant planning activities are immediately logged in a central database. Users are uniquely identified by their host name, access to the database is restricted appropriately. The starting of journeys is restricted by password protection. This procedure allows a very accurate and low risk data assessment, since data is collected automatically without any user interaction and no tampering of data is possible.

In mathematical terms, $n$ subjects $S_i$ with $i + \leq n$ were divided into two groups of size $\frac{n}{2}$ each. Each subject $S_i$ plans two journeys, using the agile as well as
plan-driven approach. Two travel scenarios $T_A$ (Alaska) and $T_C$ (California) are provided to prevent learning effects. By conducting the runs illustrated in 4.5, we acquire $2n$ results $R^{T_A, Agile}_j$, $R^{T_A, Plan}_j$, $R^{T_C, Agile}_j$ and $R^{T_C, Plan}_j$ with $j \in \{1, \ldots, n\}$.

### 4.3. Experiment Execution

The experiment has been conducted in April 2008 at the University of Innsbruck and was part of the courses Software Development and Project Management as well as Advanced Topics in Software Engineering. Due to organizational reasons, it was split up in 4 sessions; a total of 56 Bachelor and Master Students participated. To avoid a too high variance of the subject’s experience in planning, the data of the latter group had to be discarded.

The experiment started with a presentation of the theoretical foundations. At first the objectives of the experiment were explained to the students. Subsequently the basic ideas of agile as well as plan-driven planning were introduced. Also the connections to the journey metaphor and the concepts (e.g. uncertainties, risk) were highlighted. Finally the experiment’s procedure was outlined.
Thereafter students were randomly divided into 2 groups (A, B) of equal size. Group A started by adopting the plan-driven approach, group B made use of the agile process. As illustrated in Figure 4.4, the experiment was divided into 2 runs. In the familiarization phase of each run the students had to watch a screencast which explains the usage of the Alaska travel simulator, additionally help pages were provided. In order to further familiarize with the usage of the travel simulator, students performed a testing journey, which was not taken into account for the data analysis. All in all the familiarization phase took 20 minutes. The remaining 40 minutes were used for planning the journey which was relevant for the experiment’s results. After a short break, groups were asked to swap approaches and perform the second run. In the first run, the Californian travel scenario had to be managed, in the second the journey to Alaska. The structure of each run is depicted in Figure 4.5.

4.4. Data Analysis Procedure

The analysis of data is divided into two phases: the first phase ensures the validity of the used data, cf. Subsection 4.4.1. In the second phase (cf. Subsection 4.4.2) the collected data is analyzed according to well-established statistical methodologies.
Chapter 4. Experiment

4.4.1. Data Validation

As pointed out in Section 4.2, a logging mechanism is used to collect data. The correctness of the logging system is assured by automated tests [MC05], [Bec04]. In order to ensure data consistency, journeys stemming from the testing phase as well as duplicates from the same host have to be removed.

The analysis of data plausibility is based on the student’s seriousness and the respecting of the experiment’s setup. The first condition is violated if the reached business value of a journey is a lower outlier (smaller than median minus three times the inter quartile range). One student had to be discarded because of this rule. The data of another two students was rejected, since they did not follow the experiment’s setup. In comparison to [Sch08], the state at the end of the journey is not taken into account for assessing data plausibility. If students are not able to finish the journey appropriately, this may be an indication for a problem with the planning approach used and cannot be discarded.

For further analysis of the data plausibility we make use of box-whisker-plot diagrams, which visualize the distribution of the data, particularly outliers. The boxes are bounded by the 25% – and 75% quantiles, the line inside box represents the median. Above and below of boxes whiskers are drawn, all results outside of the whiskers are considered to be outliers. The length of a whisker is at most 1.5 times the interquartile range (abbreviated IQR, 75% quantile - 25% quantile) [Tuk77]. Figure 4.6 shows the boxplot for the results of the conducted experiment. No outliers appear, therefore the data can be considered to be valid from this point of view.

4.4.2. Data Analysis

This section deals with the analysis of gathered data and interpretation of results. Subsection 4.4.2 provides a descriptive analysis, in 4.4.2 the testing of the hypothesis stated in 4.2 is performed.
Descriptive Analysis

Table 4.2 shows descriptive metrics of the collected data. Since some journeys had to be discarded during data validation, results of 47 subjects remain for analysis. The lowest business value results from a journey in Alaska, planned in the agile user interface. Also for the Californian travel scenario lowest values stem from a student who used the agile user interface. The same pattern can be found for the 25% quantile, mean and 75% quantile. Interestingly, the maximum business value is always reached by travels planned using the agile user interface. The standard deviation of journeys planned in the agile user interface seems to be higher than for those planned in the plan-driven counterpart.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Approach</th>
<th>N</th>
<th>Min</th>
<th>25% Q.</th>
<th>Mean</th>
<th>75% Q.</th>
<th>Max</th>
<th>Stdv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alaska</td>
<td>Agile</td>
<td>24</td>
<td>0</td>
<td>2418</td>
<td>3235</td>
<td>4165</td>
<td>6080</td>
<td>1576</td>
</tr>
<tr>
<td>Alaska</td>
<td>Plan-driven</td>
<td>23</td>
<td>2244</td>
<td>2849</td>
<td>3563</td>
<td>4265</td>
<td>4963</td>
<td>1145</td>
</tr>
<tr>
<td>California</td>
<td>Agile</td>
<td>23</td>
<td>2086</td>
<td>3356</td>
<td>4247</td>
<td>5063</td>
<td>5983</td>
<td>778</td>
</tr>
<tr>
<td>California</td>
<td>Plan-driven</td>
<td>24</td>
<td>3482</td>
<td>4113</td>
<td>4617</td>
<td>5152</td>
<td>5820</td>
<td>657</td>
</tr>
</tbody>
</table>

Table 4.2.: Descriptive Statistics
Chapter 4. Experiment

The results suggest that by the use of the agile user interface, the best as well as worst results can be achieved. Most agile journeys seem to be substandard in comparison to plan-driven ones, since the mean value is lower in both travel scenarios. An extensive discussion of possible causes can be found in Section 4.5 and 4.7.

Hypothesis Testing

In order to test the hypothesis stated in Section 4.2, the business values gained in agile as well as plan-driven journeys is compared. As pointed out before, the travel scenarios Alaska ($T_A$) and California ($T_C$) differ significantly in the expected business value and have to be treated separately. Therefore the business value of journeys $R^{T_A, Agile}$ is compared with $R^{T_A, Plan}$, voyages $R^{T_C, Agile}$ with $R^{T_C, Plan}$ (cf. Section 4.2).

For a statistical significant comparison we use the t-test. This test assesses whether the means of two groups are statistically different from each other [TD07]. The idea behind the t-test is to compare the difference between to group means with the variability of the corresponding groups. Only if the ratio of means difference to variability is high enough, the means can be considered statistically significant different. The computation of the t-test will not be explained in detail, interested readers may want to look it up in [TD07]. For the analysis of this thesis’ data SPSS [DS07] is utilized.

The t-test requires the underlying samples to emanate from a normal distribution. To verify that data fulfills this requirement, the Kolmogorow-Smirnow is applied [Sta08]. This test is used to check whether two given probability distributions differ. In our case we check if the distribution of the data differs from the normal distribution. The test must be applied to all four groups $R^{T_A, Agile}$, $R^{T_A, Plan}$, $R^{T_C, Agile}$ and $R^{T_C, Plan}$. Several other tests exist to determine the probability distribution of samples, the Kolmogorow-Smirnow has been chosen since it is known to be robust and works also for small samples [TD07].
Another prerequisite for the t-test is the test for equality of variances [TD07]. Even though the t-test works for equal as well as varying variances, different methodologies have to be applied. SPSS automatically performs the Levene test for equality of variances [Sta08] and applies the appropriate methodology. Therefore the Levene test is not further explained. An extensive documentation can be found in [Sta08].

Putting all together, the data analysis is performed as follows.

1. Testing for normal distribution of
   a) $R_{Ta,Agile}$
   b) $R_{Ta,Plan}$
   c) $R_{Ta,Agile}$
   d) $R_{Ta,Plan}$

2. After the Kolmogorow-Smirnow passes for all groups, the t-test is applied, SPSS automatically checks for equality of variances and applies the appropriate methodology. The following combinations are tested
   a) $R_{Ta,Agile} - R_{Ta,Plan}$
   b) $R_{Ta,Agile} - R_{Ta,Plan}$

4.5. Experiment Results

According to Table 4.2, 94 travels have been analyzed, 47 thereof planned with use of the agile user interface, 47 in the plan-driven counterpart. As described in the previous section, at first the Kolmogorow-Smirnow test was applied to each group. SPSS was used for applying the tests, results are listed in Table 4.3. If the resulting significance is greater than 0.05, the sample emanates from a normal distribution [DS07], all groups fulfill this condition and the t-test can be applied. The data on which the descriptive statistics is based on can be found in Appendix A.
Chapter 4. Experiment

Following the procedure outlined before, the t-test can now be deployed to find out whether the reached business value differs significantly. Again SPSS was used to carry out the test, results are listed in Table 4.4. The second column lists the significance of the Levene test, both values are lower than \(0.05\), therefore the variances are not equal. SPSS thus applies the t-test for non-equal variances. For the rejection of the null hypothesis, a significance level of smaller than \(0.05\) is required (cf. third column). None of the two scenarios deliver a lower significance, therefore the null hypothesis cannot be rejected. It is important to see that this does not mean that the null hypothesis is accepted, it only suggests that there is not sufficient evidence against it [Fis66]. The underlying data for the tests can be found in Appendix A.

Table 4.3.: Results of the Kolmogorow-Smirnow test

<table>
<thead>
<tr>
<th>Group</th>
<th>Significance</th>
<th>Normal Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R_{TA,Agile})</td>
<td>0.409</td>
<td>yes</td>
</tr>
<tr>
<td>(R_{TA,Plan})</td>
<td>0.894</td>
<td>yes</td>
</tr>
<tr>
<td>(R_{TC,Agile})</td>
<td>0.629</td>
<td>yes</td>
</tr>
<tr>
<td>(R_{TC,Plan})</td>
<td>0.971</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 4.4.: Results of the Kolmogorow-Smirnow test

Table 4.4 shows the lower and upper bounds of the 95% confidence interval. The difference of the samples’ means is between the lower and upper bound with a certainty of 95% [Zar98]. Since the lower bound is negative, but the upper positive, none of the approaches yields a statistical significant higher business value. The following section provides explanations for this result.

Table 4.4.: Results of the t-test

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Sign. L.</th>
<th>Sign. T.</th>
<th>95% Conf. Interval</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alaska</td>
<td>0.02</td>
<td>0.369</td>
<td>-1060</td>
<td>404</td>
</tr>
<tr>
<td>California</td>
<td>0.002</td>
<td>0.185</td>
<td>-916</td>
<td>175</td>
</tr>
</tbody>
</table>

92
4.5. Experiment Results

4.5.1. Interpretation of Results

Literature about agile approaches highlights the superiority to plan-driven methodology, especially in highly dynamic environments [PP06], [Coh06], [Lar03], [Lef07], [Coh04]. As pointed out, we think that a journey is a suitable metaphor for software development projects (cf. Chapter 2). Hence the adoption of agile approaches should be beneficial for planning a travel. Also the metrics presented in 4.4.2 suggest that the highest business value can be reached by using agile methodology. In order to discover the reasons for the failing of agile voyages, logged journeys have been analyzed manually. As result, the following problems have been identified: inadequate planning, complexity and lack of agile principles.

**Inadequate Planning**

The Alaska travel simulator provides two user interfaces for planning a journey: the plan-driven and agile user interface. Former restricts the flexibility of planning and prescribes the adoption of the plan-driven approach. For the agile user interface such behavior is not possible, since the agile approach cannot be dictated by tool support. Put differently, the user interface for the plan-driven approach supports plan-driven methodology only, while the agile user interface supports plan-driven, agile, chaotic planning as well as any intermediary approach. Figure 4.7 visualizes the connection between user interface and supported planning approaches.

As pointed out, the Alaska travel simulator enables the user to plan the travel in an agile manner, but there is no way to enforce it. Agile methodologies need
Chapter 4. Experiment

a high degree of flexibility, therefore the user interface for planning agile travels constricts the user’s freedom only where necessary. For instance it is possible to assemble a journey in the up-front planning phase which cannot be executed, since accommodations are missing. This behavior is legal, since the user may want to choose the accommodations during the travel-phase.

For the case the user does not plan the voyage sufficiently, he uses the agile user interface, but actually follows the chaotic approach [WW08]. Manual analysis of the journeys showed that 10 out of 47 journeys were chaotic, i.e. involved very little or no planning at all. 15 people invested considerable time in up-front planning, but did not adapt their plans during the traveling phase, therefore adopting the plan-driven approach. Altogether journeys of 25 out of 47 students were accounted as agile journeys, which were actually chaotic or plan-driven. Table 4.5 shows the mean business value of the agile, chaotic and plan-driven approach.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Approach</th>
<th>Mean BV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alaska</td>
<td>Agile</td>
<td>4279</td>
</tr>
<tr>
<td>Alaska</td>
<td>Chaotic</td>
<td>2383</td>
</tr>
<tr>
<td>Alaska</td>
<td>Plan-driven</td>
<td>2596</td>
</tr>
<tr>
<td>California</td>
<td>Agile</td>
<td>4417</td>
</tr>
<tr>
<td>California</td>
<td>Chaotic</td>
<td>3614</td>
</tr>
<tr>
<td>California</td>
<td>Plan-driven</td>
<td>4228</td>
</tr>
</tbody>
</table>

Table 4.5.: Mean values of approaches

Complexity

Even though the journey metaphor represents a rather simple model and abstracts away from several details, the scenarios are still rather complex. Both travel scenarios include 43 actions, most of them restricted by constraints. The voyage to Alaska provides up to 5 unforeseen events, the Californian up to 4. Each event requires the traveler to restructure the plan. Money is restricted too
and booking deadlines and –fees demand careful booking. Finally the weather introduces additional uncertainties, especially in the Alaskan scenario.

The traveler must take all these factors into account when assembling a valid journey plan. The Alaska travel simulator provides a validation mechanism in order to support the user. It takes the current plan as input and comes up with a list of problems. When using the plan-driven approach, the traveler is not allowed to start the traveling-phase as long as there are problems found by the validation. Since the traveler cannot modify the plan during the journey (only exception: if events occur), it wouldn’t be sensible to start a journey with preprogrammed problems. For the agile approach this procedure is not possible, since the traveler is allowed and even encouraged to change his plan while traveling. Thus the voyage can be started independent of the validation. The validation of agile journeys is much more complicated, since also parallel and missing actions must be considered. For some cases it cannot be decided whether an action is missing intentionally or has been forgotten. Therefore the validation procedure reports phantom problems too, which makes it harder for the traveler to identify the actual issues.

Putting it all together, the validation mechanism helps to reduce the complexity of planning by pointing out possible errors. In the plan-driven version the traveler is forced to remove all problems. The validation for the agile approach also supports the traveler, but is much more complex and the resolution of problems is not enforced. Additionally phantom errors make the interpretation of results harder. Thus the chance for getting stuck in the middle of nowhere is much higher for agile journeys due to the higher complexity and responsibility of the traveler. 8 agile travels (out of 47) show troubles which would have been prevented by the validation mechanism of the plan-driven approach. For instance, not enough money was available to return home and missed booking deadlines caused serious business value losings.
Lack of Agile Principles

Another factor which cannot be prescribed by tool support is the usage of agile principles (cf. Section 2.5). The Alaska travel simulator provides support for the adoption of these principles, but cannot enforce them. In order to prioritize actions (cf. Section 2.5.1), a sorting mechanism has been implemented, which enables the user to sort actions by time, money as well as business value (expected, expected per time, expected per credit). Deferring commitment (cf. Section 2.5.3) is fostered by reminders for booking deadlines and the possibility of scheduling actions in parallel. In order to eliminate waste (cf. Section 2.5.2), the traveler can make use of the expected costs, money as well as expected timeframes for the action’s execution.

Even though the travel simulator provides these features, not all students took advantage of them. The manual analysis showed that journeys with low business value lacked the usage of agile principles. A common mistake which appeared was missing adaptions of the plan. If the execution of an action does not take as long as expected, the remaining time can be spent for other actions. Unforeseen events may introduce actions with high business value. Below-average travels lacked adaptions to these situations and exhibited serious waste of time and money. Other travels which did not take advantage of prioritization involved the execution of many actions but could not reach a high business value. Lack of deferring commitment caused extra expenses, in turn the traveler could not afford additional actions.

Further conclusions and discussion can be found in Section 4.7.

4.6. Risk Analysis and Mitigation

This section briefly explains the concept of validity, threats which endanger it as well as countermeasures. When designing the experiment, we took care of threats mitigating internal as well as external validity.
As pointed out in Section 4.1.5 the internal validity of an experiment is the approximate truth about inferences regarding casual relationships [TD07]. It is an indicator whether the effects on the response variables are caused by the independent variables or alternative causes. According to [TD07], internal validity deals with the question: “Assuming that there is a relationship in this study, is the relationship a causal one?” To diminish risks endangering the internal validity, the experimental setup must ensure that the independent variables are the only one which are varied. All other influences must be kept stable.

External validity deals with the generalization of results [TD07]. In most experiments it is impossible to assess the whole population, e.g. all software developers around the world. Therefore for conducting the experiment, a sample is drawn from the population. Conclusions are therefore always made based on the data gathered from the sample and have to be generalized back. If the samples are not drawn properly, the generalization may not be valid. In order to ensure external validity, samples should be big enough and selected randomly, replication of experiments also helps to mitigate risks [TD07].

### 4.6.1. Internal Validity

As explained, internal validity deals with question whether effects on the response variables are caused by changes of the independent variables or have other causes. In the context of our experiment, the task is to ensure that the reached business value depends on the adopted planning approach to a high degree, other factors should be insignificant.

The student’s experience in planning represents a parameter which can have a strong influence on the gained business value. In order to balance the level of experience, only Bachelor students were taken into account for data analysis. The data of Master students participating has been discarded. Furthermore it was possible to acquire a considerable amount of students (a total of 56), after data validation and plausibility check 47 were left. By having a sufficiently large sample the threat of unequal experience can be relativized.
Another threat to consider is the motivation of students, since carelessly planned journeys will certainly yield a lower business value on average. Due to organizational reasons, the experiment has been split up in four sessions, in which no more than 20 students attended. Therefore it was possible to monitor the student’s activities and verify their work enthusiasm. Only one student did not work properly, his journey has been marked and removed during the data validation phase.

The respecting of the experiment’s setup represents a further threat to the internal validity. If students use the same approach or journey twice, learning effects cannot be prevented. Due to the small group size, it was possible to immediately detect people which did not follow the instructions. Two students escaped our vigilant eyes, their misbehavior could be identified during the data validation.

An additional factor influencing the business value is the usability of the travel simulator. If the user interfaces for plan-driven and agile simulation vary in terms of user-friendliness and understandability, differences in resulting business value may stem from the user interfaces, but not from the adoption of the approach. In order to countervail this issue, user interfaces differ only where necessary to support the corresponding approach. Furthermore the usability of the simulator has been evaluated by persons, who were not involved in the development. Thus we can ensure that the usage of the travel simulator is easy and can be learned within a short time.

Finally inaccuracies in the measurement of the business value have to be considered as a threat of internal validity. As illustrated in Section 4.2, a logging mechanism automatically collects all relevant data and therefore eliminates the risk of inaccurate measurements.
4.6. Risk Analysis and Mitigation

4.6.2. External Validity

The external validity deals with the generalization of results (cf. Section 4.6). In our experiment we have to consider whether the results in context of the journey metaphor are also applicable to software engineering.

First of all it has to be ensured that journeys represent a suitable metaphor for software development projects. As illustrated in Chapter 2, we think this can be assured due to many parallels found in either domain. Summing up, in both situations planning represents an essential activity. Resources are scarce and plans are restricted by multiple constraints. The goal is to deliver the highest value for the customer, which is complicated by uncertain environments and unforeseen events.

The choice of subjects could also be seen as a threat to external validity, since students are not experts in the field of planning. Even though students cannot be considered to be experts, the management and planning of software projects is taught at the University of Innsbruck in course of the Bachelor study. Thus students participating know the principles of planning. Furthermore existing work shows that experiments with students as subjects are transferable and can give valuable insights into a problem domain [Run03], [Hou99].

The experiment focuses on the planning behavior of the plan-driven versus agile approach only. Even though we consider a journey as a suitable metaphor for a software development project, social aspects cannot be taken into account. Agile approaches consider human resources as the most precious asset. Social factors play an important role, the focus is put on leveraging people’s qualities [Bec00]. Communication, feedback, courage and respect are basic values of the agile methodology XP [Bec00], many practices deal with teamwork and collaboration. Our journey metaphor does not take these parameters into account yet, thus considerable potential of agile approaches lies idle.
4.7. Discussion

The results presented in Section 4.5 suggest that there is no significant difference of the adopted planning approach with respect to the business value reached. Since literature about agile approaches praises its superiority to plan-driven methodology \cite{PP06}, \cite{Coh06}, \cite{Lar03}, \cite{Lef07}, \cite{Coh04} data has been analyzed manually in a second step. The analysis showed that there exist several problems with the identification of agile processes and the usage of tools. Reasons and consequences from this findings are discussed now.

While it is possible to prescribe the plan-driven approach by restricting the tool support, this procedure is not feasible for agile methods. As pointed out in Subsection 4.5.1, several journeys were planned using the agile user interface, but actually adopted chaotic or plan-driven methodology. This raises the interesting question: What does it mean to be agile? Terms found in domain of agile approaches, like test-driven-development \cite{Bec02}, extreme programming \cite{Bec00} or Scrum \cite{Sch04} became buzzwords and companies or even university institutions pretend to be “agile”. As seen in the experiment, used tools are not necessarily an indicator for the adopted approach. Subsection 4.5.1 highlights that only 22 out of 47 journeys planned in the agile user interface actually utilized agile methodology, while another 25 voyages can be considered to be plan-driven or chaotic. In order to assess the degree of agility, the whole process as well as its devolution has to taken into account. As suggested in \cite{Coh06}, the process must not be necessarily fixed throughout the project, improvements are intended and help to improve efficiency.

The results highlight that the agile approach yields a higher variance of the achieved business value than plan-driven methodology. The higher variance is a logical conclusion of the increased flexibility and responsibility in the agile approach. The achieved business value depends more on the personal’s strengths and weaknesses. Therefore before blindly adopting agile approaches, it should be verified that agile concepts are properly understood. It is not necessary that all employees are gifted, but at least for each team there should be one person who
owns the abilities to leverage agile principles and coaches his co-workers appropriately [Coh06]. To sum up, agile approaches promise a higher gain, but also require gifted and well-trained people. If these preconditions are not satisfied, failing is very likely.

Another issue raised by the experiment’s results is the availability of tool support for agile approaches. If the degree of freedom is high, decision making requires deliberating about multiple variants. Recommendation systems [SWvDvdA08] can support people in the process of decision making. Especially for students, who performed bad in the agile approach, it would be interesting to see whether the usage of recommendation systems helps to improve results. In order to investigate in this question, further master theses are currently conducted at the University of Innsbruck, which deal with the usage of afore mentioned techniques.
Chapter 5.

Summary

In this thesis we investigate the impact of the adoption of plan-driven versus agile methodology. The generated business value of the project is taken as measurement for its success. In order to see the effect of the adopted planning approach with respect to the level of uncertainty of the environment, two different project settings were evaluated.

Literature suggests that the comparison of software development projects is difficult due to many differences between individual projects [Bas96]. Furthermore developing the same software two times in order to compare the adopted approaches is nearly unfeasible due to restricted monetary resources as well as time. In addition it is not possible to prevent learning effects if the same team is used for development. In a nutshell, restricted resources made it impossible to study the adoption of the planning approach in real software development projects.

In order to be able to compare planning approaches without having to carry out cost and time intensive projects, we deploy a journey as a metaphor for a software development project. As pointed out in Chapter 2, we believe that due to many parallels found in either situation, a voyage represents a legitimate metaphor with respect to planning. Undertaking a journey would apparently not result in cost savings in comparison to software development, thus we provide an application, which allows the simulation of a voyage. The implemented software supports all essential concepts of our journey metaphor.
Chapter 5. Summary

By utilizing the journey metaphor in combination with the travel simulator, studying plan-driven versus agile methodology became feasible within the scope of this master thesis. The friendly support of Prof. Dr. Ruth Breu and Dr. Barbara Weber made it possible to conduct an experiment at the University of Innsbruck in which students used the simulator for undertaking journeys. As pointed out in Chapter 4, we provided two different journey scenarios with different levels of uncertainty. Students simulated the travel in both scenarios, switching the approach for the second simulation. Thus we were able to compare plan-driven versus agile planning behavior, prevent learning effects and gather sufficient data to perform a quantitative analysis.

We expected that travels planned using the agile user interface would yield a significantly higher business value than those planned using the more rigid plan-driven interface. Nevertheless quantitative analysis showed that quite the contrary was the case: even though the highest business value always stemmed from journeys planned in the agile user interface, on average travels planned in the plan-driven user interface yielded higher business values (statistically not significant).

To find out why the results do not fit our hypothesis, we performed a qualitative (manual) analysis of the journeys. The analysis showed that over 50% of the journeys planned in the agile user interface actually adopt a chaotic or plan-driven approach. Some of the students tried to adopt agile methodology, but failed to implement the most critical agile practices. In addition, the agile user interface could not provide enough support to decrease complexity which led to overstrained students.

Summing up, we could not find significant differences in the adoption of the planning approach due to problems in the experiment’s setup. Nonetheless we were able to identify further research topics and issues:

- **User Support** Apparently users have been overstrained by the flexibility of the agile user interface. Further experiments which will be conducted
should identify whether it is possible to decrease complexity by deploying decision support engines.

- **Empirical Software Engineering** As recently pointed out on a conference [MP08], there is still a lack of empirical studies validating agile methodology. Problems still exist in the conceptualization and conduction of experiments [EPV00]. This thesis showed that it is easy to force people to use rigid methodologies, but hard to assess quantitatively whether a person used an agile approach.

- **Applicability of Agile Methodology** As the results suggest, some students were overstrained by flexibility and tended to deploy chaotic or plan-driven approaches. If not at least one experienced team member is capable of understanding and adequately deploying agile methodologies as well as coaching co-workers, adoption will likely result in a failure.

- **What Does It Mean to Be Agile?** Agile software development processes are very flexible and allow adopters to select appropriate practices and techniques according to a specific project. As pointed out in [Bec00], there is no strict definition of being agile. The experiment showed that used tools are by far no indicator of the adopted approach.
## Appendix A.

### Data of the Experiment

<table>
<thead>
<tr>
<th>Business Value</th>
<th>Alaska</th>
<th>California</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agile</td>
<td></td>
<td>Agile</td>
</tr>
<tr>
<td>6080.23</td>
<td>3547.60</td>
<td>5983.04</td>
</tr>
<tr>
<td>5304.36</td>
<td>3417.46</td>
<td>5873.46</td>
</tr>
<tr>
<td>5025.49</td>
<td>3411.27</td>
<td>5842.68</td>
</tr>
<tr>
<td>4547.46</td>
<td>3293.65</td>
<td>5552.67</td>
</tr>
<tr>
<td>4440.70</td>
<td>2974.04</td>
<td>5272.47</td>
</tr>
<tr>
<td>4211.33</td>
<td>2542.10</td>
<td>5063.41</td>
</tr>
<tr>
<td>4028.66</td>
<td>2376.74</td>
<td>5006.23</td>
</tr>
<tr>
<td>3990.40</td>
<td>1376.47</td>
<td>4982.98</td>
</tr>
<tr>
<td>3973.58</td>
<td>1375.23</td>
<td>4979.54</td>
</tr>
<tr>
<td>3703.45</td>
<td>810.15</td>
<td>4748.89</td>
</tr>
<tr>
<td>3674.33</td>
<td>0.0</td>
<td>4694.24</td>
</tr>
<tr>
<td>3553.05</td>
<td>0.0</td>
<td>4194.20</td>
</tr>
</tbody>
</table>


List of Figures

1.1. Visualization of the applied research method [Sch08] . . . . . . . 20
1.2. Comparison of pair versus individual programming . . . . . . . 23

2.1. Comparison of Waterfall [Roy70] versus Iterative Model [TN86] . 31
2.2. Cone of Uncertainty . . . . . . . . . . . . . . . . . . . . . . . . 34
2.3. Phases in Journey Metaphor and Software Project . . . . . . . . 38
2.4. Life cycle states of a user story . . . . . . . . . . . . . . . . . . . 40
2.5. Action states, adapted from [Sch08] . . . . . . . . . . . . . . . . 41
2.6. Exemplary devolution of weather for $t_w = 0.2$ and $s_w = 0.5$ . . 43
2.7. Classification in high / low risk and value [Coh06] . . . . . . . . 51
2.8. Business class diagram of a journey [Sch08] . . . . . . . . . . . . 58

3.1. Alaska Toolset . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
3.2. Plug-in composition of the travel simulator [Sch08] . . . . . . . . 65
3.3. Services as easily exchangeable parts of functionality . . . . . . 67
3.4. Support of XML as well as databases for persistence . . . . . . 67
3.5. Proxies act as factories for the creation of Actions . . . . . . . . 69
3.6. Proxy caches a configuration . . . . . . . . . . . . . . . . . . . . 69
3.7. User interface model for separation of business logic and ui . . . 70
3.8. Commands for the restoration of journeys . . . . . . . . . . . . . 72
3.9. Eclipse Rich Client Platform’s architecture . . . . . . . . . . . . 74
3.10. Interaction of FitNesse with tests and system under test . . . . . 76

4.1. Basic concepts of an experimental setup [Sch08] . . . . . . . . . . 80
4.2. Classification of experimental designs [Sch08] . . . . . . . . . . . 82
List of Figures

4.3. Experiment’s setup, adapted from [Sch08] .......................... 85
4.4. Experiment’s phases [Sch08] ........................................... 86
4.5. The structure of a single run [Sch08] ................................. 87
4.6. Boxplot visualizing the distribution of the results ................. 89
4.7. Planning approach support of user interfaces ..................... 93
List of Tables

2.1. Comparison of elements in a software project and a journey [Sch08] 56
2.2. Comparison of plan-driven and agile approach [WW08] . . . . . . 57
4.1. Experiment’s design . . . . . . . . . . . . . . . . . . . . . . . . . 84
4.2. Descriptive Statistics . . . . . . . . . . . . . . . . . . . . . . . . 89
4.3. Results of the Kolmogorow-Smirnow test . . . . . . . . . . . . . 92
4.4. Results of the t-test . . . . . . . . . . . . . . . . . . . . . . . . . 92
4.5. Mean values of approaches . . . . . . . . . . . . . . . . . . . . . 94
Bibliography


Bibliography


Bibliography


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Publisher</th>
<th>Year</th>
</tr>
</thead>
</table>
Bibliography

[NR86] Peter Naur and Brian Randell, editors. Software Engineering, Report of a conference sponsored by the NATO Science Committee, Garmisch-Partenkirchen, October 1986. NATO Science Committee.


[SWvDvdA08] Helen Schonenberg, Barbara Weber, Boudewijn van Dongen, and Wil van der Aalst. Supporting Flexible Processes Through Log-


