Leopold-Franzens
University of Innsbruck

Institute of
Computer Science

Research Group
Quality Engineering

Master Thesis

Adoption of Decision Deferring Techniques in Plan-driven Software Projects

A Controlled Experiment

Author
Bakk.techn. Michael Schier

Supervisor
Dr.Barbara Weber

April 21, 2008

"Planning is everything. Plans are nothing."

Field Marshal Helmuth Graf von Moltke

Abstract

Today, a trend towards agile software development approaches can be observed caused by shortened product lifecycles and the striving to optimize time to market. As a consequence, traditional plan-driven approaches are more and more forced to take a back seat in development processes producing products for rapidly changing markets.

This thesis discusses the idea of deferring design decisions in plan-driven software development approaches and advocates the adoption of techniques supporting such concepts. At this, an experiment is conducted taking a journey as metaphor for a software development project. The results of it essentially corroborate the thesis that software projects can benefit from the use of design decision deferring techniques. In addition to that, it fortifies the assumption that such techniques reduce the amount of project plan adjustments in case of conflicts caused by unforeseen events.

Acknowledgement

I would like to thank all people who have helped and inspired me in developing the Alaska simulator and while writing this master thesis during the last seven months.

First of all, I want to thank my supervisor **Dr. Barbara Weber** for her guidance, her perpetual energy and enthusiasm. She and **Werner Wild** whom I also want to thank provided Stefan and me with tons of inspirations and feature requests, and gave us helpful feedback with regard to functionality and usability of Alaska. Furthermore, Barbara spent much time on shaping this thesis and her constructive reviews were crucial for finishing it on time.

Next, I want to thank **Bakk. techn. Stefan Zugal** for being a nice and studious development partner throughout the whole implementation. Without being a well-rehearsed team, conducting inspiring discussions and the existence of mutual comprehension, we would not have been able to finish implementation that fast and at such a high level of quality.

I am also grateful to **Bakk. techn. Christian Haisjackl** who acted as software tester and provided us with helpful bug reports and feedback about Alaska's usability, and to **Mag. (FH) Sandra Naschberger** for proofreading this thesis.

Finally, I want to thank **my parents** for supporting me mentally as well as financially. Without their help, I would not have had the chance to study Computer Science — *thank you for giving me the freedom to live the life I wish to live!*

Declaration of Authorship

I, Bakk. techn. Schier Michael, 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 (Schier Michael)

Contents

Chapter 1.

Introduction

1.1. Background

At the early beginnings of the computer era when a software development project's content was limited to small, well arranged requirements, hard tasks could be only found in the field of programming. It was more important to create correct and efficient programs than worrying about coordination matters. By the time when the number of requirements rose, companies were forced to employ more and more developers in order to master a system's completion within a given project time bound. This trend let other problems come to the fore. Suddenly, project leaders were confronted with challenges like breaking down complex tasks into smaller, meaningful steps without losing too much flexibility and at the same time keeping the system maintainable as well as fulfilling certain security and quality standards.

> *"To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming had become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them. It has created the problem of using its products."*
>
> *Edsger W. Dijkstra [Dij72]*

These changes combined with the appearance of better and faster computers at lower prices are the driving factors which stood behind the famous software

crisis in the mid 1960's. The first time in history, the costs of developing software exceeded those of producing hardware. Due to missing methodologies and techniques to handle and coordinate big software projects, many of them ended up in chaotic situations and failed accordingly. Developers began to realize this lack of concepts which led to a huge problem-solving discussion and finally to the definition of the term *Software Engineering* at the NATO-conference in Garmisch-Partenkirchen in 1968 [NR86]. During the following years, besides improvements in the field of implementation, people started thinking about how to get a grip on coordination of software development in the large with a special focus on the interplay of different project phases and reasonable delegation of working tasks to single project members. Today, software development process models can be roughly divided into two categories: **plan-driven process models** and **agile process models**.

1.1.1. Plan-driven process models

In plan-driven process models, similar activities are combined to phases which are executed sequentially. Results of these phases are documented which motivates the nomenclature of a *document- or specification-driven process model*. The primary target is to follow the plan, stick to a sequential execution order and avoid setbacks to previous phases where possible. Prominent representatives of this approach are the Waterfall Model [Roy70] and the V-Model [Hes08]. The discovery of the fact that pure sequential models tend to be more and more unrealistic to be carried out, motivated people with great project management experience like Barry Boehm to define plan-driven process models supporting iterations. As a prominent example, the Spiral model [Boe88] runs through different project phases several times and produces within each iteration a prototype which is refined this way up to an operational prototype. Further process models favoring the idea of introducing cycles in the development process are the Personal Software Process [Hum94], the Team Software Process [Hum99] as well as the Rational Unified Process [JBR99]. The central idea behind all these approaches is to specify the development plan and the software's design upfront

before starting the actual implementation. To conclude the observation of plan-driven process models, it is important to notice that a strong focus lies on a complete plan whereas in agile approaches, the project plan concretizes during the whole project's lifetime and is frequently subject of change.

1.1.2. Agile process models

Looking at agile software development methodologies, the interaction between customer and contractor gains more importance. Due to their iterative development procedure, customers can review the system in an earlier state and hence can provide the developers with feedback more quickly. Because of *shortened development cycles*, costs for requirements change and error correction can be cut down. A common misconception when dealing with agile approaches is that developers who are not familiar with the ideas behind them or who do not understand the difference between deferring and skipping design decisions might put it on one level with a chaotic approach where no planning takes place [Pry02]. Examples of this paradigm are eXtreme Programming [Bec00], Scrum [Sch04], the Dynamic Systems Development Method [CV98] and Lean Software Development [PP06]. Compared to their counterparts, agile methods try to tighten the software development process and to increase flexibility.

Another difference is that agile methods are profit-oriented in the sense of producing *as much business value as fast as possible*. The idea behind is that the 80/20 principle which states that 20% of the business generates 80% of the profits also applies to software products. In an agile project, the members try to identify these 20% of functionality and aim to implement them first. The resulting effect is that project constraints like quality, cost and time are obeyed but the project's scope is kept flexible which has positive implications on, for instance, scenarios with tighter project deadlines [Bec00]. In such a case, the impact of dropping remaining low-value feature requests on the system's overall business value is minimized in contrast to similar scenarios occurring in plan-driven approaches. As opposed to that, plan-driven approaches fix their scope but keep quality variable. The plan is created in such a way that the ordering of tasks

allows an efficient implementation but their business values are not taken into consideration. This seems to be a reasonable procedure because no early releases are intended from which the customer could benefit. Besides that, the estimation of a feature's business value would have to be done before implementation can start which delivers a more imprecise estimation as in agile approaches where later iterations allow a re-estimation of the expected business value. According to Jim Highsmith, the typical agile developer essentially says:

> *"We will give you a plan based on what we know today; we will adapt the plan to meet your most critical objective; we will adapt the project and our plans as we both move forward and learn new information; we expect you to understand what you are asking for - that flexibility to adapt to changing business conditions and absolute conformance to original plans are incompatible objects."*
>
> *Jim Highsmith [Coh06]*

1.1.3. Historical evolution of software process models

Figure 1.1 provides an overview of the emergence of software project methodologies in history. As mentioned earlier, the Waterfall Model and the V-Model were the first approaches and are typical representatives of plan-driven methods. In the mid-eighties, people became aware of the weakenings of such approaches, which initiated the development of new, less rigid and more agile methodologies like the Spiral Model [Boe88], which introduces iterations to the so far nearly linear development process. During the 1990s, more and more people picked up the agile idea, and popular lightweight approaches like Scrum, eXtreme Programming and Crystal [Coc04] increasingly were applied to software projects. At the same time, the ideas behind former plan-driven methods were improved and new approaches appeared, like the V-Model 97 and the W-Model.

Today, the majority of software projects still make use of plan-driven approaches [dG06], but there is a cognizable trend towards agile methods. The shortening of software development projects' life-cycles caused by an increase of market dynamics due to the globalization and other economic phenomena degrade the conditions for inflexible plan-driven approaches and force more and

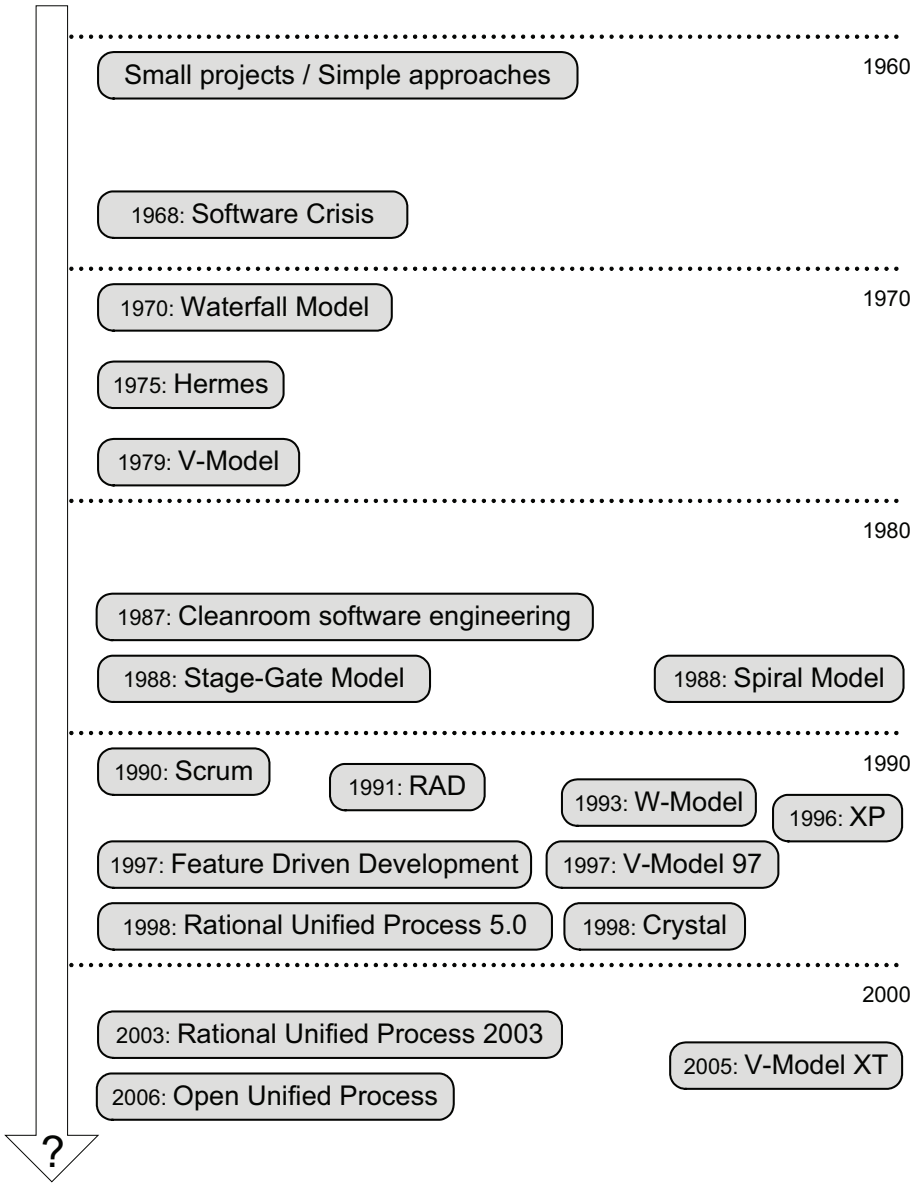


Figure 1.1.: History of software project methodologies

more companies to search for alternatives. This evolution is the main driving force behind the present boom of agile methodologies which can be observed today [Bar07].

1.2. Research Objectives

The goal of this thesis is to investigate how the plan-driven approach – commonly regarded as a rigid methodology – performs in environments characterized by unstable system requirements and dynamic project parameters. In such an approach, fixed procedures are used to regulate changes and hierarchical organizational structures are means of establishing order. As a consequence, an increase of control typically leads to an increase of order and occurring difficulties are primarily solved by reductionist task breakdown and allocation. The underlying assumption is that *risk* is adequately predictable to be managed by complex, detailed upfront planning.

Besides risk as an important indicator value for software projects, the system's *business value* for the customer as a measure of success and the *quantity of project plan adaptations* as a measure for dynamics inside the software project are of great concern in this work. Business value is in this context not only meant to be a measure for the amount of financial benefit from a software system, but also includes positive effects on a company's workflow, on the employee's motivation etc. [DK02]

We focus in this thesis on the plan-driven approach. Due to various software development techniques, it is in this approach to some degree possible to defer design decisions and to furnish planning with a certain amount of flexibility. Surely, this cannot be compared to possibilities available when applying agile methods but the important question to investigate in this context is whether a sophisticated design can compensate financial losses caused by forced project plan adaptations and whether resulting positive effects can be observed (see figure 1.2). These considerations immediately lead us to this thesis' central research topic:

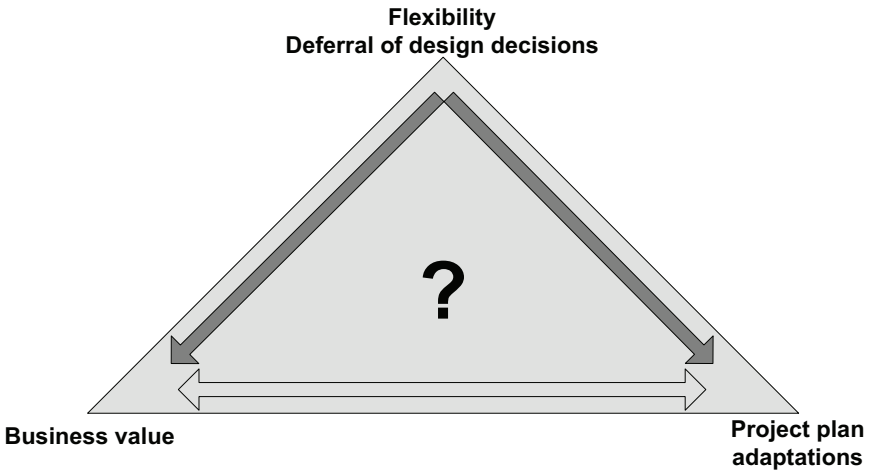


Figure 1.2.: Relationship between three important software project indicator values

We investigate the impact of the adoption of methodologies allowing the deferral of design decisions on two project indicator values: **business value** and **project plan adaptation frequency**.

In connection to that, we examine whether there are cognizable positive effects going hand in hand with the use of such techniques in terms of business value increase. Additionally, we analyze whether a decrease of forced project plan adaptations (caused by unexpected events) can be observed.

In this thesis, the modern agile approach is not taken into consideration. Differences between the agile and the plan-driven approach are discussed in further detail in the thesis of Stefan Zugal [Zug08], who was extensively involved in the development of a simulator software also used in this work. Zugal examines both approaches and studies planning behavior, requirements change robustness and effectiveness, whereas this thesis only focuses on the adoption of decision deferring techniques in plan-driven software projects, i. e. how successful traditional approaches are in comparison to such approaches enhanced with previously mentioned concepts.

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):

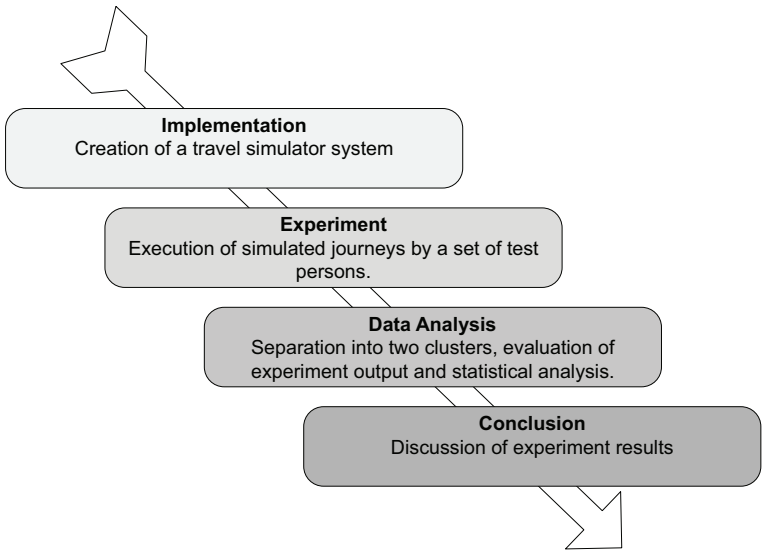


Figure 1.3.: Progression of the journey paradigm study

1. **Implementation**: A travel simulator is implemented which tries to offer the user the most important aspects of a real journey. It is designed in such a way that the user understands the program handling intuitively and can fully concentrate on the main task: The stategic planning of a journey (see chapter 3).

2. **Experiment**: During the experiment, test persons develop travel plans for two different scenarios and simulate their execution in order to evaluate their project management skills. At this, every planning step is logged to be available for later data analysis (see section 4.3).

3. **Data Analysis**: The collected results are split up into two clusters – one cluster containing pure plan-driven journeys and the other journeys whose test persons made use of decision deferring techniques (explained in more detail in section 2.6). Subsequently, statistical analysis is conducted which should optimally highlight relations between project indicator values, and the results are prepared graphically (see section 4.4).

4. **Conclusion**: Based on the analyzed data, we draw conclusions about the benefits of adopted decision deferring techniques and further examine whether there are differences in the magnitudes of project plan adaptations when comparing both approaches (see section 4.7).

1.4. Related Work

Many publications can be found in the field of planning software development projects. Probably the most important representatives coming from the agile universe are Kent Beck's books of "The XP series" [Bec00], which start the discussion by underlining the interaction of four important project variables: cost, time, quality and scope (see figure 1.4). As mentioned before, Beck points out that both programmers and business people often are not aware of the potential which lies in the effective management of software projects' scope. He pleads for a slimming of scope by reducing waste in the sense of unnecessary or valueless functionality. The following citation underlines the positive effects of such practices:

> *"You implement the customer's most important requirements first, so if further functionality has to be dropped it is less important than the functionality that is already running in the system."*
>
> *Kent Beck [Bec00]*

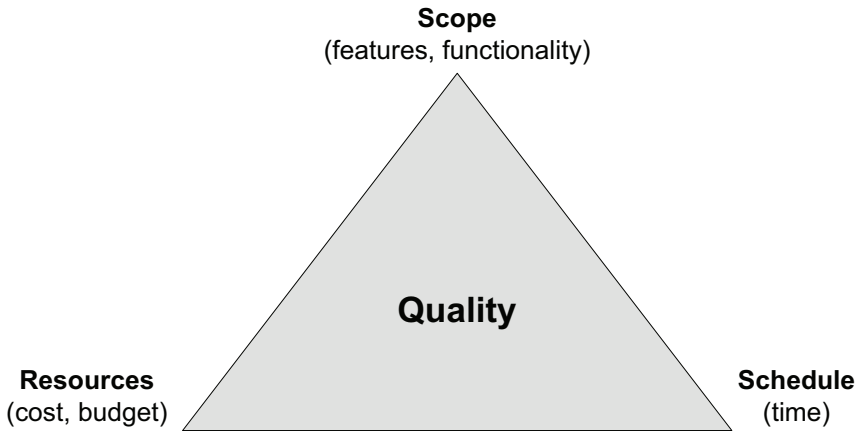


Figure 1.4.: Software development from the perspective of a system of control variables

Furthermore, Beck illustrates how the cost of change behaves differently in software projects using plan-driven and agile approaches respectively (figure 1.5). In this connection, he highlights the positive implications with regard to a flattened, non-exponential cost curve which accompany the techniques of iterative planning and choice deferral. In order to give project members a concrete idea

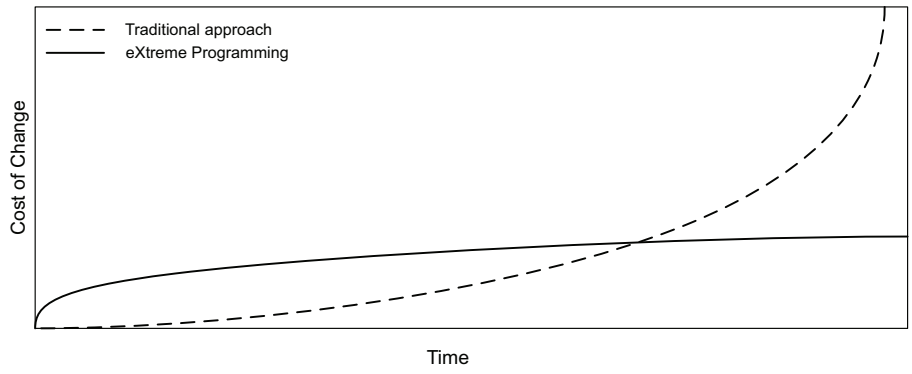


Figure 1.5.: Comparison of traditional and XP cost curve

of how to plan in an agile fashion, he defines the term *Planning Game* at which the goal is to maximize the value of software produced by the team. Based on this value, further indicators like cost of development and its risk are deduced.

The provided strategy is to invest as little as possible to put the most valuable functionality into production as soon as possible, but only in conjunction with the programming and design strategies designed to reduce risk. According to Beck, this can be done among other things by sorting of functionality by value and risk in descending order. The developers then inform the customer about the expected velocity in terms of ideal engineering time per calendar month which assists him to choose the scope of functionality to be implemented within the next iteration or release.

The second book in the XP series, "Planning eXtreme Programming" [BF00] again by Kent Beck and Martin Fowler underlines the importance of planning and deferred commitment (discussed in more detail in [MSWW03]) in XP projects and the impact which working together with a customer and delivering features incrementally have on agile planning behavior. The emphasis lies on the provision of hints for ordering features, how planning and status meetings shall be organized, how visual graphs can be used to monitor project progress and how bug tracking and fixing should be handled. A fact of special interest in connection with this thesis is that the authors also list planning a trip and car driving as metaphors for software development and point out parallels.

Another interesting book dealing with agile methodologies is "Implementing Lean Software Development" by Mary and Tom Poppendieck [PP06]. They see development as a process of transforming ideas into products and identify two schools of thought offering approaches to such transformations. The former is the deterministic school of thought which starts by creating a complete product definition and then creates a realization of that definition whereas the latter is the empirical school of thought beginning with a high-level product concept and subsequently establishing well-defined feedback loops that adjust activities so as to create an optimal interpretation of the concept. Moreover, they believe that development processes dealing with changing environments should be empirical processes because such processes are the best way to adapt change. History showed that especially software development processes feature a high potential of unsteadiness and change which leads the Poppendiecks to the conclusion that such processes should be understood best as empirical processes.

Similar to Beck, they endorse the idea of eliminating waste and try to implement it by encouraging engineers to develop a deep understanding of what customers value and a closely associated deep understanding of what the technology can deliver. Based on that, all steps done in processes should have a focus on value-creating activities and should try to improve capability to deliver value in so far as this is possible. With regard to the amount of produced code, this means that features in a system have to be aggressively limited to only those that are absolutely necessary to add value. Another positive side effect is that the code base is kept small, simple and clean resulting in a decreased level of complexity during later iterations which avoids that costs rise exponentially as depicted in figure 1.6.

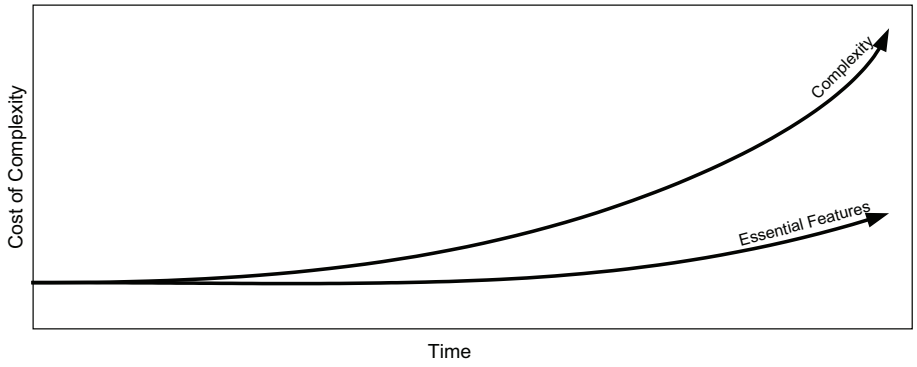


Figure 1.6.: The cost of complexity

Another important concept which is discussed in the book is the *Deferred Commitment*, which tells developers to abandon the idea of a full specification and to delay decisions as far as possible. This does not imply not to plan any tasks in advance at all, but pleads for making decisions reversible where possible. It also does not demand for complete flexibility but instead asks for the maintenance of options at points where change is likely to occur.

Mike Cohn's book "Agile Estimating and Planning" [Coh06] focuses on agile approaches in general and starts with the clarification of the difference between them and chaotic approaches. He explains the purpose of planning and what mistakes can be made in planning software projects. Mike Cohn outlines techniques

for estimating needed time and effort, how to perform a meaningful prioritisation based on four factors - value, cost, new knowledge and risk - and fortifies his argumentation with examples as well as popular theories coming from the product development sector like the Kano Model [Kan84]. The core chapter deals with scheduling and handling release planning - how velocity influences iteration planning, which length iterations should have, why buffers are essential in the presence of uncertainty and what things should be considered when planning for multiple-team projects. The book closes with a case study, points out once more the difference between conventional and agile planning techniques and provides guidelines for efficient agile estimating and planning.

Tom deMarco points out in his book "Waltzing With Bears: Managing Risk on Software Projects" [dL03] how managers often underestimate requirements and doom projects with their too optimistic thinking and unrealistic expectations. In this context, he explains the impact of risk on scheduling software projects and underlines the importance of buffers and flexibility justified by the difficulty of risk management (*"Risk management is project management for adults"*).

Another book [deM86] by the same author focuses on the importance of the adoption of good estimation techniques to calculate a software project's effort as exact as possible. Based on this, deMarco further explains how to do cost and time planning to countervail deadline exceeding and support precocious detection of errors.

Yet another interesting book is "Balancing Agility and Discipline - A Guide for the Perplexed" by Barry W. Boehm and Richard Turner [BT03]. The authors examine the aspects of agile and plan-driven methods and provide an approach to balancing by examples and case studies. They are convinced that best development strategies combine attributes of both ways of thinking and provide with their work a practical guidance for software developers. Furthermore, Boehm and Turner suggest not to adapt existing processes to a project but instead to build a new process from existing approaches, tailored for the target project.

> *"Agility and discipline: These apparently opposite attributes are, in fact, complementary values in software development. Plan-driven developers must also be agile; nimble developers must also be disciplined. The key*

> *to success is finding the right balance between the two, which will vary from project to project according to the circumstances and risks involved. Developers, pulled toward opposite ends by impassioned arguments, ultimately must learn how to give each value its due in their particular situations."*
>
> *Barry W. Boehm, Richard Turner [BT03]*

In "Managing the Software Process" [Hum89], Watts Humphrey points out the importance of a sophisticated project plan and discusses important topics like size's measurement, how estimations can be implemented, how to identify productivity factors and how task scheduling can be conducted. Furthermore, he compares different planning models and suggests how to track projects best.

The theory and practice of current measurement techniques and their problems are described in Capers Jones' book "Applied Software Measurement" [Jon91]. The procedure of collecting "function-point metrics", the metric that will probably replace lines of code as the standard measure of program size, is discussed as well. An interesting fact is that Jones integrates a huge amount of quality and productivity data and analyzes it in several chapters to substantiate his argumentation.

In their book "Applied Software Project Management" [SG05],Andrew Stellman and Jennifer Greene write about the concepts behind the creation-process of a project plan. They favour the plan-driven approach and argue that it is crucial to have knowledge about the project's scope up-front. In addition to that, they point out the importance of maintaining a risk plan to manage a project's plan with respect to the definition of buffer times and exception handling. Stellman and Greene see the project manager as central coordiating individual and also discuss issues in connection with a possible lack of leadership but surprisingly do not mention the present trend of delegating responsibility to team members.

> *"When the project begins, the project manager has a unique role to play. The start of the project is the time when the scope of the project is defined; only the project manager is equipped to make sure that it's defined properly. Everyone else has to play a role later on..."*
>
> *Andrew Stellman, Jennifer Greene [SG05]*

Similar to Stellman and Greene, Pankaj Jalote summarizes the most important concepts of plan-driven project management in his book "Software Project Management in Practice" [Jal02]. He discusses several effort estimation approaches and divides the scheduling process in an overall part defining the rough contents and a detailed part further refining the project plan. Besides other topics like quality planning, he puts a focus on risk management and explains assessment techniques for identification and prioritisation of potential risks. Based on this information, Jalote pleads for controlling techniques to monitor and track identified risks. Another major topic in his book is Requirements Change Management. At this, Jalote enumerates several mechanisms that are crucial when facing sudden requirements changes, which are: the estimation of the effort needed for the change requests, the reestimation of the delivery schedule, the performance of a cumulative cost impact analysis, etc.

1.5. Overview

The remainder of this thesis is structured as follows:

Chapter 2 describes the fundamental concepts of a software development project and the journey metaphor used in the experiment. At this, it highlights parallels between them and tries to explain the legitimacy of the chosen metaphor.

Chapter 3 explains important concepts of the travel simulator's architecture, lists used technology and frameworks and points out the composition of plug-ins Alaska consists of.

In chapter 4, the controlled experiment conducted in connection with this thesis research question is described. First, the explanation of basic terminology with regard to software experiments is given. Next, the experiment's design is demonstated, before describing its actual execution. The core of chapter 4 is the data analysis phase as well as the discussion of obtained results. It is concluded by identifying risks which threaten the experiment's validity and lists procedures to mitigate them.

Chapter 2.

Concepts

This chapter explains the core concepts of a software development project and points out connections to potential counterparts in a travel scenario. A detailed and compact summary of parallels can be found in table 2.1 on page 50. In section 2.1, we explain the terms *software development project* and *project plan* followed by existing *roles* in such a project (section 2.2). Next, *use cases* are discussed in section 2.3 and subsequently, *basic conditions* and *limiting factors* in section 2.4 as well as *project dynamics* in connection with *unforeseen events* are explained (section 2.5). Finally, we list *techniques and concepts for the deferral of design decisions* in section 2.6 and conclude the chapter with a comparison of concepts in a software development project and a journey (section 2.7).

2.1. The Software Development Project and the Project Plan

Software project

A software development project is always shaped by the customer's demands and visions with respect to the system's functionality. What the project's devolution will look like and how the interaction with the customer will emerge strongly depends on the chosen software development method. Popular examples are the Waterfall Model [Roy70], the Spiral Model [Boe88], the Unified Process [JBR99], the V-Model [Hes08] as well as eXtreme Programming [Bec00] and Scrum [Sch04]. As a typical representative of a plan-driven, linear, non-iterative software devel-

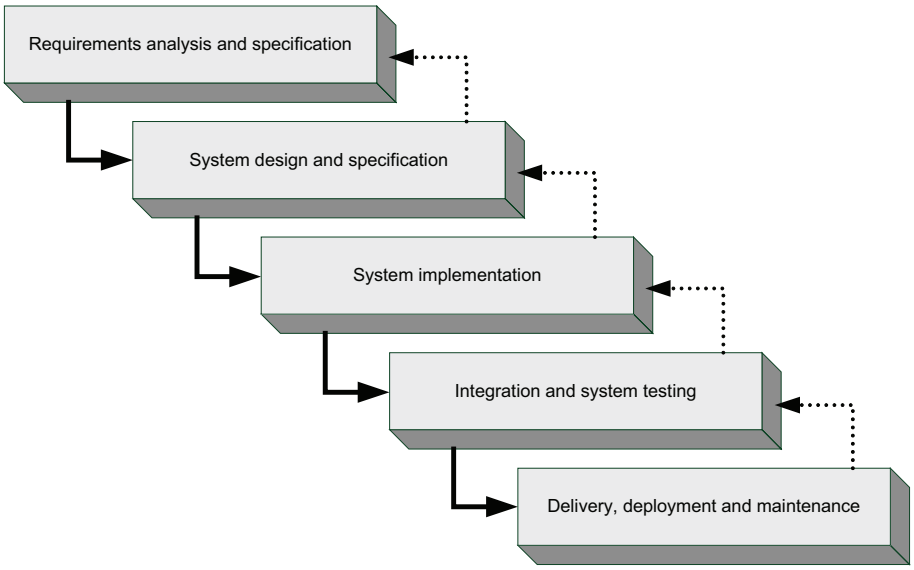


Figure 2.1.: Consecutive phases in the waterfall model

opment model, we use the **Waterfall approach** (figure 2.1). A specific feature of this approach is the fact that a project can be separated into several phases, which have their outputs strictly defined. At this, the output of one phase is the input of its succeeding phase. This characteristic makes the model's name obvious.

Beginning with the **requirements analysis and specification phase**, the customer creates a document which describes as exact as possible the functionality which the system has to provide. Based on this information, the contractor composes a second document, the requirements specification, which describes the system's functionality from the contractor's point of view. In this connection, the required functionality is divided into *use cases* (see section 2.3 on page 38) defining the interactions between an external actor and the system under consideration to accomplish a goal. In both documents, the system is seen as a black box and consequently, only questions dealing with what is needed are of concern.

How features are finally implemented is constituted in more detail during the **system design and specification phase**. The resulting system specification is

a refinement of the requirements specification and contains detailed information about their demanded behavior, besides single methods and their signatures. Hereby, the behavioral description is often formulated using natural language and has to be as precise as possible to avoid the emergence of questions during the succeeding **implementation phase**.

This period of a software development project appears to be the most cost intensive one. As a result, it is critical for a project's success to have the preceding phases successfully completed. Now making a step backwards forced by inconsistencies found in the specification may for example render several other already implemented features unnecessary which inevitably results in a tremendous financial loss.

In order to countervail arising implementation errors and possible misinterpretations of the specification document, all specified methods of the software project as well as test cases defined in the requirements specification are run against the system. It has to be emphasized that these tasks executed during the **integration and system testing phase** do not lead to a complete proof of the correctness of the program but only test a certain set of key functionality and try to get a high amount of code coverage.

After satisfyingly completing this phase, the system is delivered to the customer and installed on his hardware during the **delivery, deployment and maintenance phase**. After successfully integrating the system into the IT landscape of the customer, training courses for client-side employees are often performed to get them into touch with the novel software. Important challenge is the one of providing support in case of errors. Often, not all faults can be detected during the verification phase and may occur at run time. The removal of such software flaws can take diverse forms. In the best case, the customer can get rid of the error by himself by modifying documented parameter values. If it comes to the pinch, the system cannot be repaired on site and has to be revised by a return to antecedent project phases which again results in high costs for the development company and/or the customer depending on their contract.

The careful reader might have become aware of the fact that such a strict linear software development model holds a high potential of risk. The idea of perfectly

completing one phase before going to the next one turned out to be an illusion [McC04]. An example of a scenario where such an approach causes severe problems is the customer's uncertainty about his conception of the required system. Too often, it happens that it needs a prototype to help the customer sketching what is demanded. Without that help, mismatches between the customer's and the developer's view on the system will be detected during later phases which again results in higher costs. Another imaginable situation is that while specifying the system, the developers are not aware of its complexity and this problem is not discovered until the implementation phase. Again, an expensive step backwards is needed. David Parnas describes the problems of such inflexible processes in his article [PC86] quite well:

> *"Many of the (system's) details only become known to us as we progress in the (system's) implementation. Some of the things that we learn invalidate our design and we must backtrack."*
>
> *David Parnas*

These and other troubles have led to the development of a whole set of modified waterfall models where most of them are covered in McConnells book [McC96].

Journey metaphor

As metaphor for a software development project, a journey is used. The main purpose of a voyage typically is the maximization of fun and travel experience. In connection with that, also some constraints and restrictions have to be taken into consideration like being on time for the return flight or staying within the travel budget. In a similar way, targets and constraints can be set up in a software project like implementation of the core functionality, reaching a certain level of usability, keeping expenses below a given bound or meeting the project's deadline. Additionally, other parallels can be pointed out like the money available for a software project and the size of the traveler's budget as well as the implementation of use cases and the execution of touristy actions.

A journey can as well be divided into several phases where the **pre-planning phase** is the first one of those. During this phase, the traveler tries to identify several activities of special interest and tries to create a plan involving such tasks

as well as other necessities like accommodations and car drives. A reasonable travel plan can only be created when several action specifics as well as global parameters (e. g. time, budget, constraints) are taken into account. After its successful compilation, bookings can be made before the pre-planning phase ends and the actual journey starts.

During the **traveling phase**, unforeseen events (e. g. road closure, civil war, discovery of new attractions) might occur, which may reduce the journey's expected outcome or even make the travel plan's execution impossible. Analogously to this, in a software development project such happenings can be a sudden customer requirements change, the detection of an underestimation made in some previous phase or the loss of key developers during the implementation phase. As previously described, such events demand countermeasures which may force the developers to step back to an earlier phase and for example adapt the system's specification. In the case of the journey, the invalid travel plan forces the traveler to interrupt the voyage and step back to the planning phase to modify it. This may comprise postponing actions, skipping non-available actions or inserting new actions. Once the schedule fits the traveler's needs, he can proceed again to the traveling phase and continue the journey to visit sights, go from one place to another or perform diverse activities.

Finally, the traveler finishes his journey and returns home – this is when the **postproduction phase** begins. In the context of a journey, pending bills are paid, holiday photographs are printed and a balance is drawn. Based on that material, the journey's success can be estimated.

All these parallels motivate the establishment of a three-phase mapping between journey stages and software project phases as depicted in figure 2.2. In the first phase which we call **planning phase**, the traveler performs all planning tasks needed before and during its journey. In the succeeding **execution phase**, actions, which are not asociated with planning tasks, are being executed. A modification of the travel plan can only be done by stepping back to the planning phase whereas the finalization of the journey leads to the final **postproduction phase**. Analogous to this, requirements are analyzed and specified as well as the system's design is being worked out in a software project during the **planning**

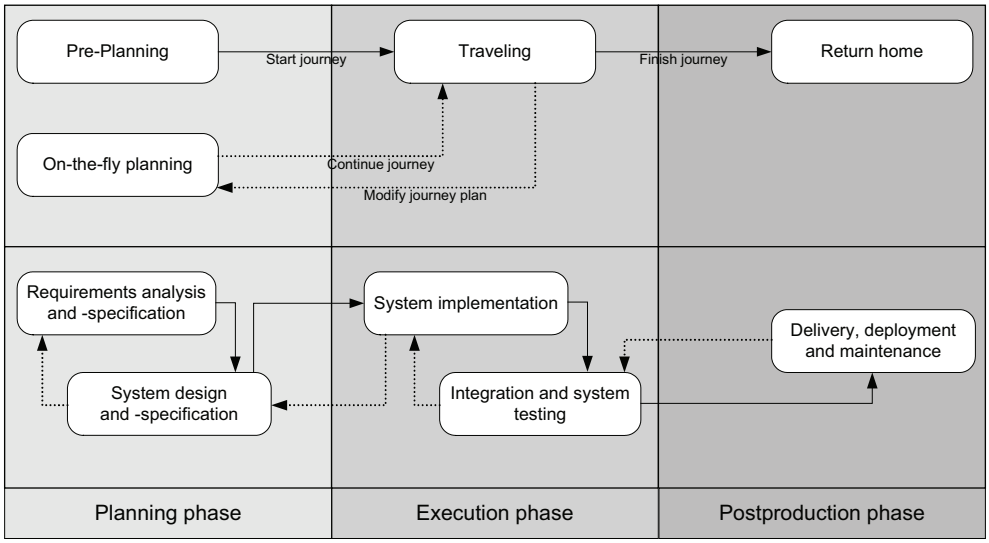


Figure 2.2.: Comparison of phases in waterfall model and journey metaphor

phase. The following **execution phase** comprises the implementation of the software and its test. At last, the system is delivered, installed and maintained during the **postproduction phase**.

2.2. The Software Developer and the Customer

Software project

The customer works out the requirements specification and defines the content of the project together with the lead developer and/or the project leader. The actual implementation work is done by the software development team whereas the detection of problems in the specification and the occurrence of requirements changes have to be handled by both parties.

Journey metaphor

In the journey metaphor, as depicted in figure 2.3, a traveler preferring to plan his trip on his own is the projected model for both - the client and the project manager. He knows his personal preferences and perceptions of a nice journey

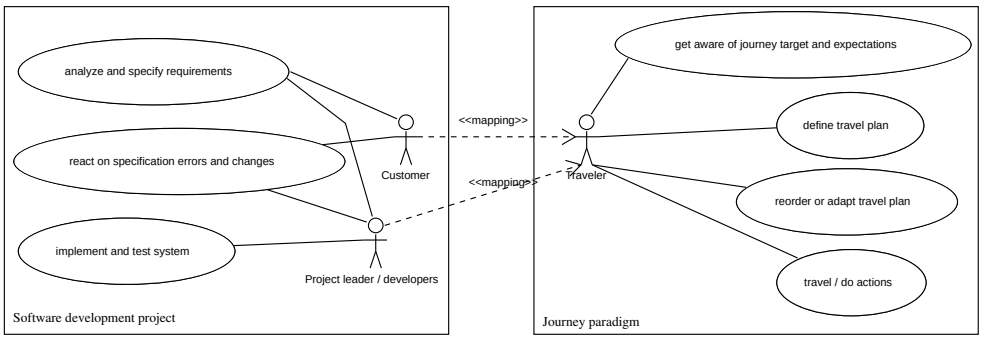


Figure 2.3.: Two on one role mappings between software development project and journey metaphor

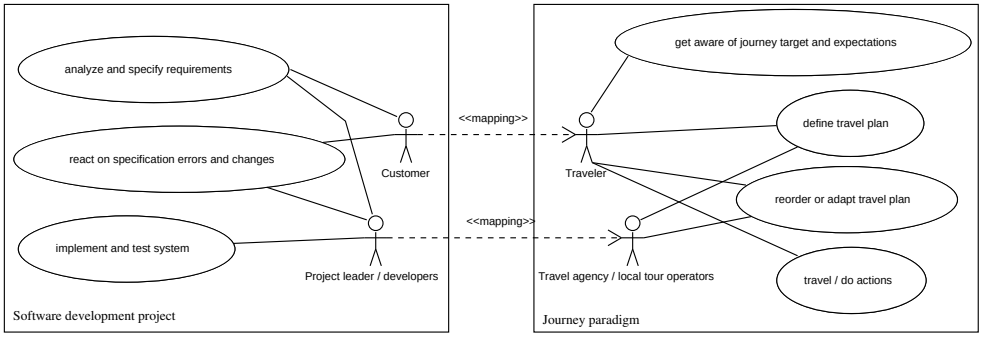


Figure 2.4.: Two on two role mappings between software development project and journey metaphor

perfectly well and can start planning based on this knowledge. This allows us to exclude potential mismatches between customer and developer which happen frequently in real software projects. Also pending reactions on changed parameters and unforeseen events can be solely decided by the traveler whereas in the software project scenario an agreement between both parties is crucial. Thus, the traveler can be seen as the union of both roles from the software development world. In contrast to that stands the second possibility of role mapping visualized in figure 2.4 where the travel agency and/or local tour operators as second party come into play. They assist the traveler in the definition of his travel plan and can help him react on unforeseen events during the voyage.

2.3. Use Cases

Software project

A use case [ABCP02] in a software development project describes the system's behavior as response to an actor's interaction with it. Actors can be human users as well as other software systems. A use case is often formulated as sequence of single steps between the system and the actor from the point of view of the latter. According to Bittner and Spence, *"Use cases, stated simply, allow description of sequences of events that, taken together, lead to a system doing something useful"* [BS02]. Use cases are characterized by a set of information, among which the name, the description, participating actors, the status, triggers and preconditions as well as post conditions are the most important ones. Besides that, other indicator values can be assigned to a use case which often cannot be determined or estimated in the design phase and become known not until its implementation being finished. Examples are a use case's duration, its complexity and its risk with regard to imprecise specification, to insufficient domain knowledge and to inexperience within a technology in need.

Figure 2.5 is a simplified state diagram of a use case's life cycle. A feature described by a use case is requested by the customer before it is analyzed by the project team. When its implementation is expected to be too costly or impossible, the feature request is declined. Otherwise, it is accepted and inserted into the project plan. In a next step, the feature is implemented by the software developers and gets tested in a succeeding phase. When conceptual errors are detected during implementation, the feature's specification has to be reconsidered. After finishing the test phase, the feature gets either deployed or detected errors have to be removed first. After the feature's delivery, still errors may occur which take the feature back to the analysis phase depending on the existence of a support contract and the defects' seriousness.

Journey metaphor

When again considering the journey metaphor, use cases are being modeled as **actions** which are separated into **accommodations**, **activities** and **routes**.

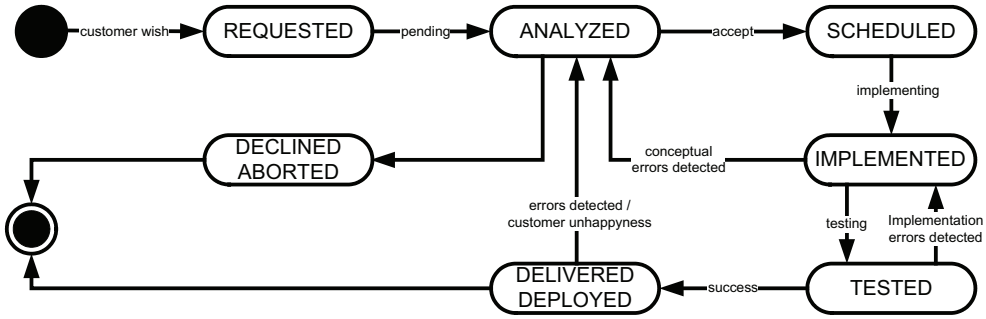


Figure 2.5.: Possible states of a software use case

Routes are indispensable when wanting to move from one location to another which is a reflection of core features in the development progress needed for example to satisfy the preconditions of succeeding use cases. Accommodations as well as activities are bound to specific locations and can only be performed when the traveler currently stays there. They are complementary in terms of execution time: Accommodations can only be visited at the end of a day to spend the night there whereas activities are intended to be carried out during a day.

Figure 2.6 illustrates the mapping of use cases onto a journey's actions. A bank customer wants a cash machine to provide several services like the withdrawal and deposit of money as well as freezing a credit card in case of theft. Such services are described in the requirements specification which the software developer has to implement. In the journey metaphor, the role of the developer is taken by the traveler who also has to accomplish several tasks. The observation can be concluded by stating that there exists both a mapping of roles and a mapping of tasks (use cases onto actions).

Actions can be characterized similar to use cases by several metrics. Considering our travel simulation, we tried to identify fundamental properties of travel actions and included support for the following characteristics into it.

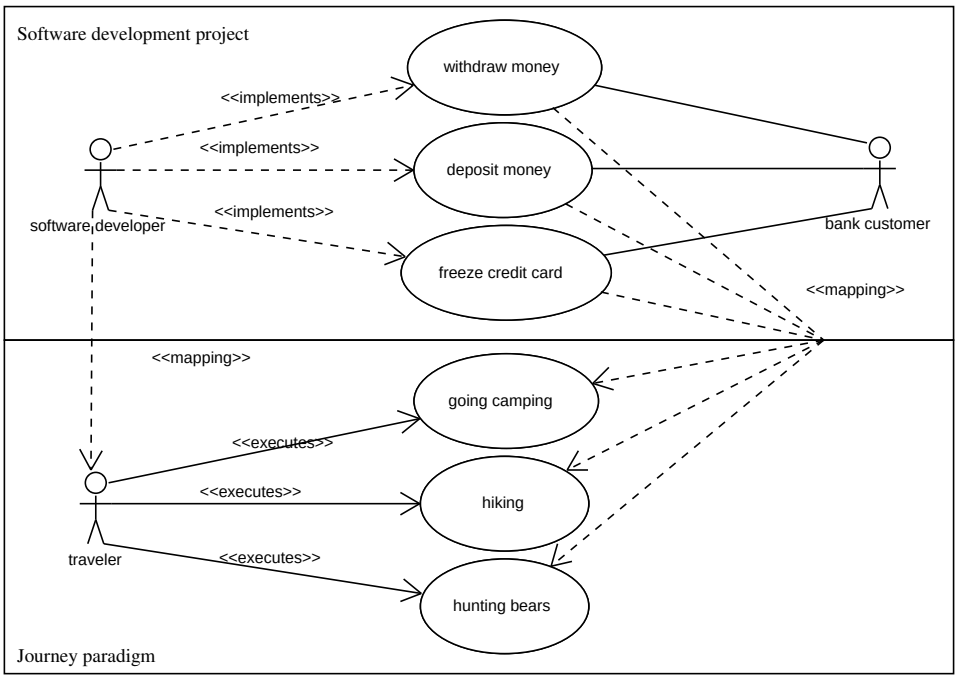


Figure 2.6.: Mapping of use cases onto actions in a journey

2.3.1. Location

An action is always bound to one location and can only be executed when the traveler currently stays there. Routes are a special case of actions and connect two locations. They can be taken in both directions i. e. they can be executed in both locations. Figure 2.7 illustrates an exemplary composition of actions and locations.

Cost

Each action is associated with a price which the traveler has to pay to be able to execute it. Furthermore, an action can be booked in advance to guarantee its availability at execution time. The booking process is also reversible supporting cancelation of bookings for which cancelation fees are charged. The magnitude of the cancelation fees is time dependend.

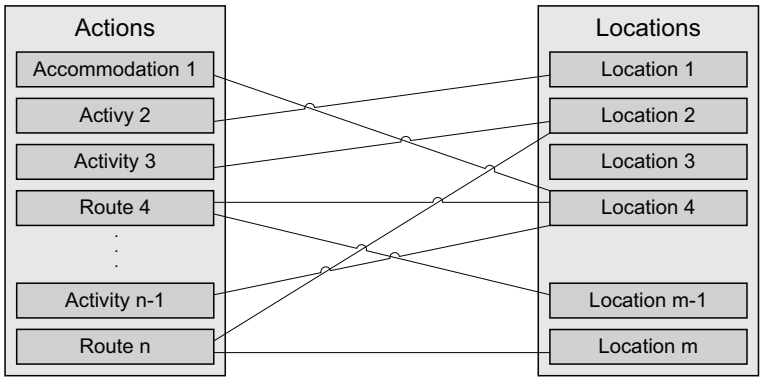


Figure 2.7.: Showcase of a a set of actions associated with locations

2.3.2. Duration

An action has either an exact duration or a duration range. This means that when the traveler wants to know an action's duration in advance, he may only get information about a time interval in which the actual duration will lie. This value is calculated randomly from a symmetric Triangular distribution. The main reason here for not taking a Normal distributed random value is that it is unbounded whereas the Triangular distribution is bounded to an intervall $[a; b]$. Figure 2.8 shows the symmetric Triangular distribution from which random values are drawn to determine the duration of an action with given duration range $[60; 120]$.

2.3.3. Business Value

An action's business value depends on two parameters: Its local certainty and the global weather influence. For each action, its maximum business value BV_{max} is known. The calculation of the actual business value BV is done as follows:

$$BV = \left(rand_{loc} + \left(w_n - \frac{1}{2} \right) \cdot w_{inf} \right) \cdot BV_{max} \tag{2.1}$$

At this, $rand_{loc}$ is a random value drawn from a bounded distribution (Beta distribution, Triangular distribution, Uniform distribution, etc.) specifically de-

fined for the action. It is bounded to the range [0;1]. $w_n \in [0;1]$ is the local weather value at day n and $w_{inf} \in [0;1]$ the weather influence factor regulating the impact of the local weather on the action's business value. A location's current weather depends on the local weather characteristics given by weather tendency $t_w \in [-1;1]$ and stability $s_w \in [0;1]$. At this, s_w defines the variance of the weather's fluctuation whereas t_w indicates whether these fluctuations are positively or negatively oriented i. e. have a positive or negative impact on the business value. The weather for a certain day is calculated based on this information and the previous day's weather. Fluctuations of the weather are simulated by random values drawn from the weather fluctuation distribution ψ. Formuli 2.2 to 2.4 explain how the weather in the simulation is calculated and figure 2.9 illustrates an exemplary devolution of the weather at a certain location. In this graph, a higher stability would reduce the size of fluctuations and a higher tendency would shift weather values up.

$$f_\Delta(x|a,b,c) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \le x \le c \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c \le x \le b \\ 0 & \text{for any other case} \end{cases} \in [0;1] \tag{2.2}$$

$$\psi(x) = \begin{cases} f_\Delta\left(x \left| \frac{s_w}{2}, \frac{1}{2}, \frac{1+(1+t_w)(1-s_w)}{2} \right.\right) & \text{for } -1 \le t_w \le 0 \\ f_\Delta\left(x \left| \frac{1-(1-t_w)(1-s_w)}{2}, \frac{1}{2}, \frac{2-s_w}{2} \right.\right) & \text{for } 0 \le t_w \le 1 \\ 0 & \text{for any other case} \end{cases} \tag{2.3}$$

$$w_n = \left[\begin{cases} \frac{t_w+1}{2} \cdot s_w + \mu_\psi \cdot (1-s_w) & \text{if } n = 0 \\ w_{n-1} + 2 \cdot rand_\psi - 1 & \text{if } n > 0 \\ \text{undefined} & \text{else} \end{cases} \right]_0^1 \tag{2.4}$$

The business value is a measure for the travelers contentedness after the action's execution. This means that the more fun and satisfaction he has, the higher this value is. The sum of all gained business values is the score of the whole journey and reflects its overall success.

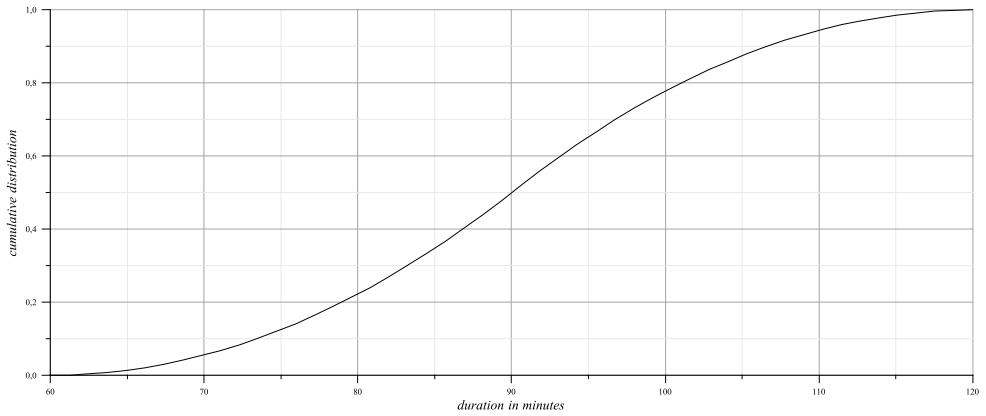


Figure 2.8.: Cumulative distribution function of the symmetric Triangular distribution

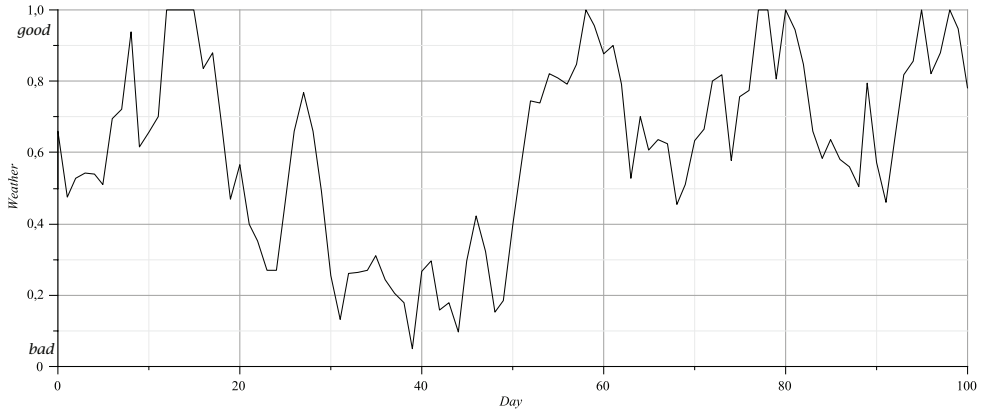


Figure 2.9.: Devolution of a location's weather with parameters $t_w = 0.2$ and $s_w = 0.6$

2.3.4. Reliability

The reliability $r_{BV} \in [0; 1]$ of an action provides information about how sure the estimated expected business value $BV_{exp} \in [0; BV_{max}]$ will be reached. It is directly connected to the overall distribution's standard deviation and is calculated independently from BV_{exp} as follows:

$$r_{BV} = [1 - 2 \cdot \sigma_{loc} - (1 - s_w) \cdot (1 - |t_w|) \cdot |w_{inf}|]_0^1 \tag{2.5}$$

$$BV_{exp} = \left(\left[\mu_{loc} + \left(w_{exp} - \frac{1}{2} \right) \cdot w_{inf} \right]_0^1 \right) \cdot BV_{max} \tag{2.6}$$

$$w_{exp} = \frac{t_w + 1}{2} \cdot s_w + \mu_\psi \cdot (1 - s_w) \tag{2.7}$$

Again, t_w and s_w define the local weather's characteristics. μ_{loc} and σ_{loc} are the action distribution's mean value and standard deviation, w_{inf} the influence of the weather on the action, w_{exp} the expected weather and μ_ψ the mean value of the weather fluctuation distribution.

2.3.5. Availability

Another important value is the action's availability $a \in [0; 1]$. It is especially important in case of unbooked actions. If an action has no booking for its day of execution, there is only a chance of $100 \cdot a\%$ that it is executable.

2.3.6. States

For actions, as depicted in figure 2.10, seven different states are defined. In the initial state `NEW`, an action is taken into account for execution but is not yet scheduled. The scheduling of it results in the state `PLANNED` from which it can reach the state `BOOKED` by booking the action before the booking deadline. Another possible successor state is `CANCELED` where the traveler drops the action from his plan without performing it. A third alternative is moving to state `STARTED` from which the two final states can be reached: `DONE` in case of success and `FAILED` otherwise. In all states except the initial state, the starting time is

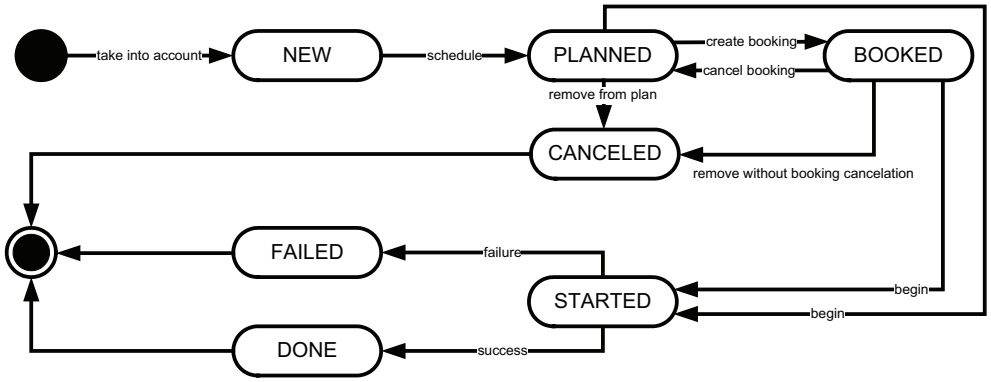


Figure 2.10.: Possible states of a travel action

known and can be used for reservations and weather forecasts. The execution of an action begins with entering the `STARTED` state and finishes in one of the succeeding final states. In both cases, important values like gained business value, duration and expenses are available from that point on.

2.4. Project Specific Limiting Factors and Basic Conditions

Software project

In a software development project, elementary resources like time, man power and money are limited. In addition to that, further rules and diverse constraints have to be considered. It might for example happen that two use cases describe two different user interaction scenarios which provide equivalent functionality. At some point in time in the project devolution, a decision has to be made about taking exactly one of these two possibilities and dropping the other one. Another potential constraint would be the implementation of one use case as precondition for a succeeding one.

Journey metaphor

Such restrictions can also be found in the journey metaphor at many places: Every day lasts for a fixed amount of minutes and has to have an accommodation at its end. Actions are bound to locations and can only be executed when the traveler currently stays there, owns enough money to pay for it and still has enough time available before going to the hotel. Other constraints can establish relations between actions like mutually excluding each other, demanding for coexistence of two actions or taking the execution of an action as prerequisite for the following one. It might also happen that the visit of certain sights is mandatory or a maximum number of executions of a single action must not be exceeded.

Basically, we distinguish between execution constraints and termination constraints. Execution constraints have to be satisfied to enable the developer to implement the feature / to allow the traveler to execute the action targeted by the constraint. Termination constraints on the other hand must be fulfilled in order to complete the iteration or release / finish the journey.

2.5. Project Dynamics and Unforeseen Events

Software project

Especially in domains with little a priori knowledge about the range of required functionality and in areas where customer demands behave unstable, a huge amount of dynamics with regard to the requirements specification can be expected. These sudden changes on global project parameters like budget, timeframe and quality and the occurrence of unforeseen events force the project development team to frequently modify and adapt the development plan. Such events might be upcoming feature requests, changes on pending or already implemented features like functionality adaption and modification of their risk estimations and compensation operations of failed feature implementations caused by underestimated risk or lately detected conceptual mismatches. It is obvious that particularly in such domains, deferred planning behavior or even agile planning approaches are beneficial. The more changes are expected and the more

thorough the planning has been done, the more time and money the adaptation process will take. With many decisions as long delayed as possible, the amount of needed modifications on the project plan can be significantly reduced [PP06].

Journey metaphor

It is easy to imagine that in a journey similar scenarios might happen. Considering a voyage of a personality who prefers to know perfectly well what he can expect to get, the travel plan will consequently be worked out in detail. As pointed out before in conjunction with a software development project, unforeseen events might happen. Imagine furthermore that a road closure happens prohibiting the traveler to reach the next destination in his plan. He is now forced to modify it in a way that he can continue his journey without losing too much time and money. In such a scenario, this might emerge as a hard task if for example succeeding actions have already been booked and cannot be canceled without causing high costs. Another feasible event is that after arriving at a certain location, the traveler gains knowledge about a special attraction which turns out to be very appealing to him. Unfortunately, his meticulous travel plan does not allow him to stay at this location any longer and refuses him to benefit from this opportunity. Otherwise, it would be quite devastating for his budget to force a visit of this attraction because of the resulting cancelations of several other planned tasks. Again, the observation may be made that the more tasks are planned and/or booked in advance, the less flexible the traveler can react to unforeseen changes and the more these changes influence the journey budget negatively.

2.6. Techniques and Concepts for Deferring Design Decisions

In order to countervail a high number of changes to the travel plan and to compensate unforeseen events, the traveler is given the opportunity to plan whole packages with predefined time and location bounds instead of single actions. These packages have the advantage that the decision about their content can be

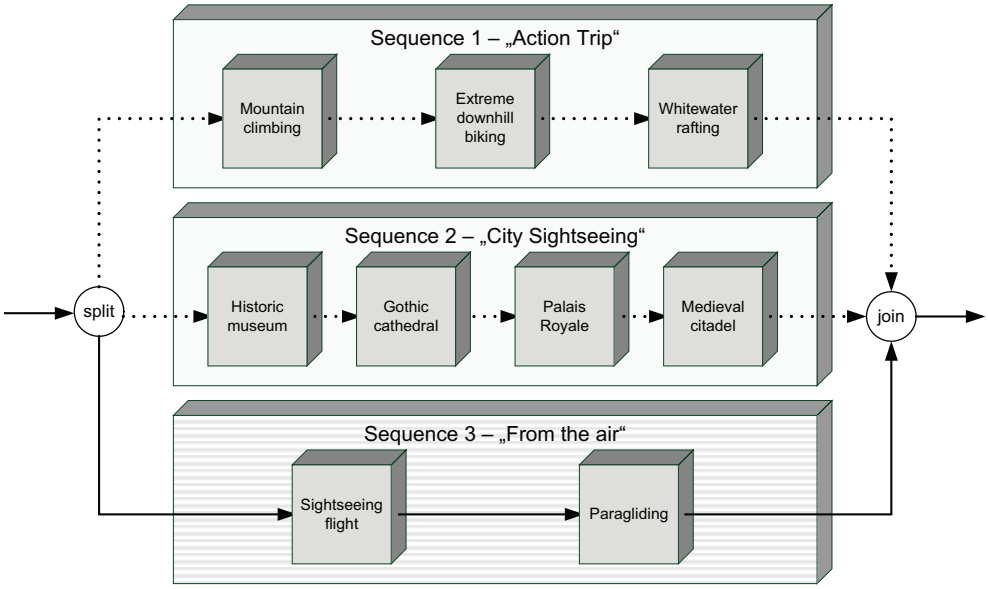


Figure 2.11.: Tour packages using late binding technique

deferred. A typical example would be a whole-day package at a certain location where the traveler can decide at the day of the visit whether he chooses the indoor or the outdoor program at which the content of both programs is fixed.

This type of package can be compared to some degree with the concept of **late binding** (figure 2.11) in object oriented systems. During design time, the developer subclasses existing classes and overrides some methods to adapt behavior. When working with variables having the type of the superclass, the compiler is not able to correctly decide at compile time which implementation to take when invoking an overridden method. This decision is deferred to runtime where the environment executing the application has more information about the content of the variable and can therefore choose the implementation accordingly [CT06], [Str00].

The other kind of package provided in the journey metaphor behaves in a similar way by also deferring the design decision but it handles the content definition differently. With this package, the traveler can configure the content freely by choosing actions from a fixed set. Obviously, this version gives the traveler more

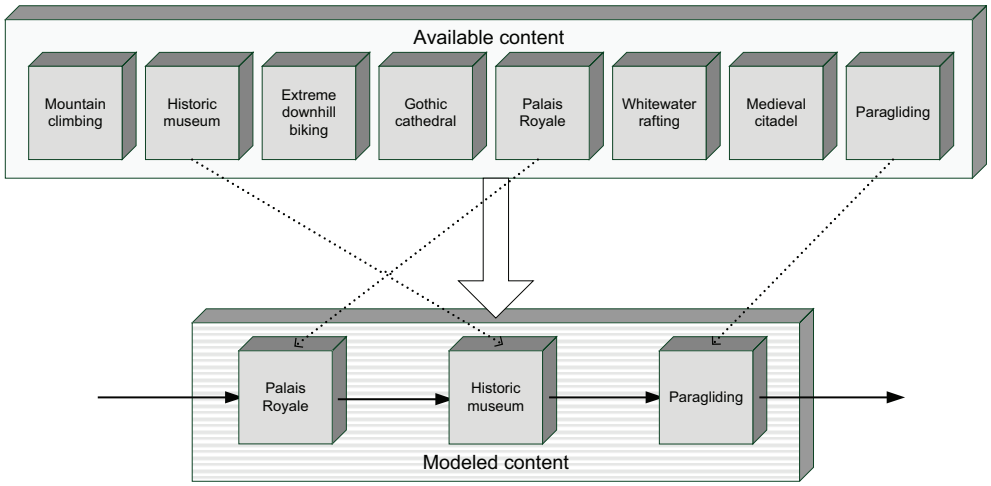


Figure 2.12.: Tour packages using late modeling technique

opportunities but on the other hand also more responsibility - he now has to check by himself whether the planned tasks are feasible with regard to parameters like cost, time and availability. A quite similar design concept can be also found in the context of workflow management systems where this technique is called **late modeling** (figure 2.12) [HHJHS97].

In the development of software systems, similar concepts of deferring design decisions can be found. A quite popular technique is used by the eclipse community [FB07] which is called "API first" [dR05]. The idea behind it is to first make an agreement on a common interface which suits the requirements as good as possible. The actual implementation of the underlying functionality is typically deferred to later phases. The advantages of such a proceeding are that decisions about implementation details can be shifted chronologically which increases flexibility and that other parties can already make use of the provided interface. This fact also motivates the demand on the API to be as stable as possible and asks for special procedures when interface changes are indispensable.

Another parallel where design decisions are being deferred is called "Information Hiding" [Par71]. Roughly speaking, it deals with the encapsulation of information in modules, classes etc. to avoid that functionality using these units

gets in touch with implementation details and internal data. This technique not only simplifies the use of modules but also makes it easier for the module maintainer to modify internals at a later point in the project without having to worry that the module's users have to adapt their code accordingly.

2.7. Interplay of Concepts

Table 2.1 sums up all discussed concepts of software development projects and lists their counterparts in the journey metaphor. How these elements are connected in our perception of a journey is depicted in the class diagram of figure 2.13. The central element is the `Journey`, which has a name, a traveler, a state and holds global parameters like the booking duration as well as the time available for traveling. Further attributes are the start `Location`, the current `Location` and the current time. Besides that, the `Journey` knows about `Constraints` which have to be obeyed and `Events` which might occur. Finally, it contains all `Locations` available in the simulation.

Software development project	**Journey**
Project plan	Travel plan
Software developer and customer	Traveler
Basic constraints, limiting factors	Location, budget, time, other constraints
Sudden change of requirements	Occurrence of unforeseen events, weather
Project time frame	Duration of vacation
Project budget	Financial situation of traveler
Use cases(key features, non-critical ones)	Actions(routes, accommodations, activities)
Requirements and Design phase	Planning the journey at home
Start of the implementation phase	Start of the journey
Practices for deferring design decisions	Tour packages
Business value of implemented use case	Positive experience of done activity
Imprecise specification, unrealistic estimates	Uncertainty of an action's cost and duration
Estimation of use case risk	Estimation of a non-booked action's availability
Cost of canceling a feature's implementation	Cancellation fee of a booking

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

A `Location` is characterized by a name, its coordinates and parameters defining the local weather conditions. Each `Location` has some `PlanItems`. A `PlanItem` is a generic type for packages (`Boxes`) and `Actions`. `PlanItems` can be members of the `JourneyPlan` and have similar to the `Journey` a state as well as a

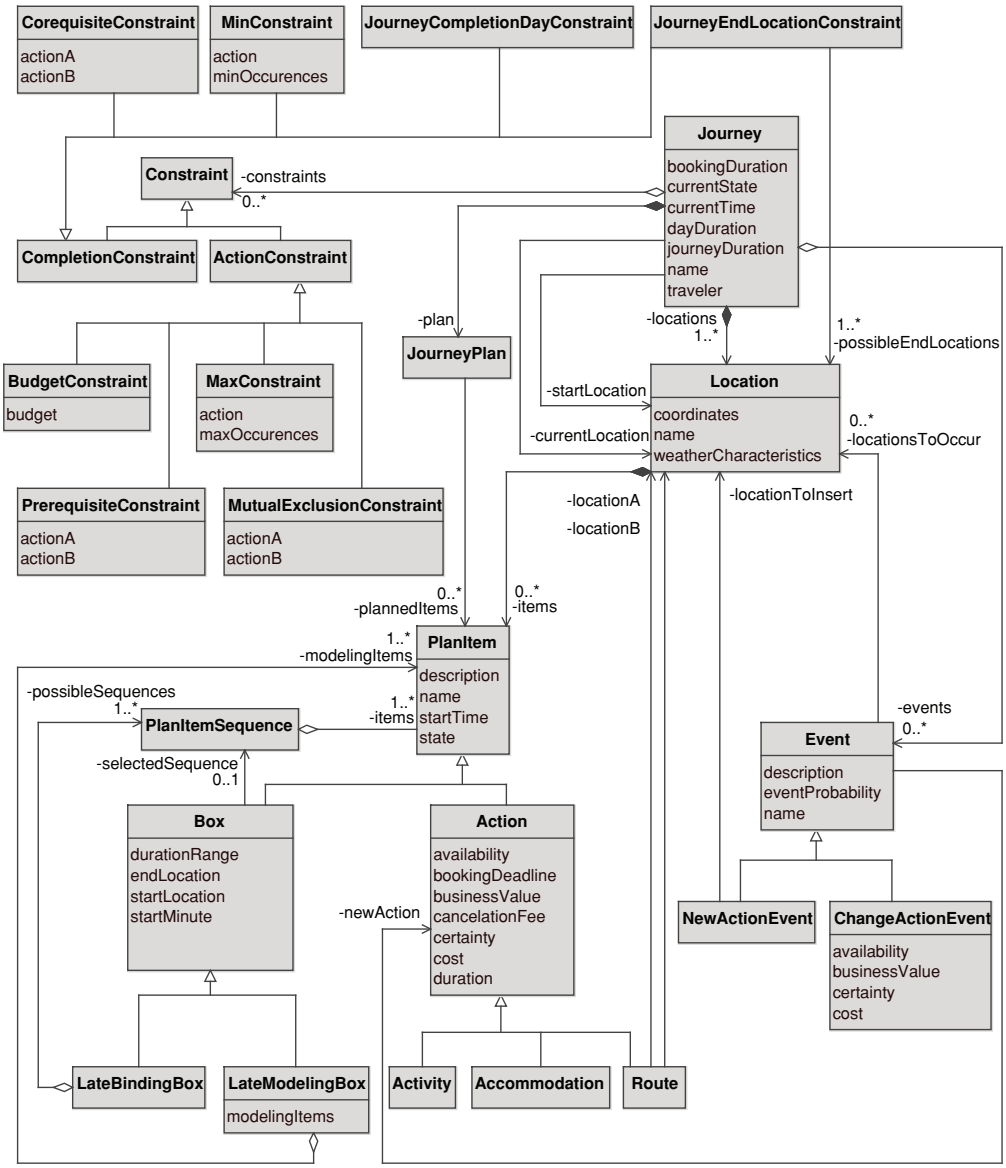


Figure 2.13.: Business class diagram of a journey

name, a description and a start time. `Boxes` last for a certain duration and have predefined start and end `Locations`. Furthermore, their starting time on a day is also fixed. `Boxes` are divided into `LateBindingBoxes` and `LateModelingBoxes`. The former contain several `PlanItemSequences` among which the traveler can choose from and the later contain a set of `PlanItems` which the traveler can use to flexibly build a specific `PlanItemSequence`. `Actions` are characterized by a business value, the certainty about the business value, the `Action`'s cost, its duration, its availability, the number of days before which a booking can be made as well as the cancelation fees. They again can be categorized into `Activities`, `Accommodations` and `Routes`.

Chapter 3.

Architecture of Alaska

Alaska is a journey simulator which is used in the experiment described in chapter 4. The development was done using Sun's Java 1.6 [Ull07] and making use of the Eclipse RCP framework [DFK+04]. As a consequence, the entire Alaska application builds on top of Eclipse plug-ins, as described in section 3.1. The composition of plug-ins reflects the system's layered architecture, which is discussed in great detail in section 3.2.

3.1. Plug-in composition

One of Eclipse's key concepts is its plug-in architecture in which basic units are modeled as plug-ins. This technique eases the addition of new features – they can be integrated by extending existing plug-ins or creating entirely new plug-ins. The resulting relativly low technology risk is one of the main reasons why Alaska was developed following this strategy. As depicted in figure 3.1, the toplevel plug-in is `alaska.ui`, which depends on `alaska.core`, `alaska.help` as well as the Graphical Editing Framework [MDG+04]. Below, XStream[1] as a persistency plug-in and the actual Eclipse RCP components are of relevance. All plug-ins prefixed with `alaska.` have been developed especially for the Alaska simulator.

[1] http://xstream.codehaus.org

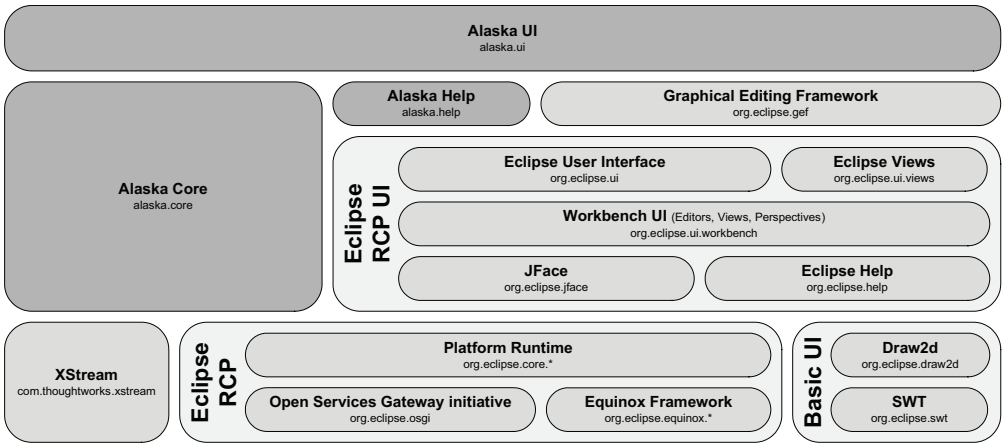


Figure 3.1.: Alaska's plug-in composition

3.1.1. Eclipse Rich Client Platform

The Eclipse Rich Client Platform is used to abstract the application from the underlying operating system. In order to stay operable on other operating systems, only the two lowermost plug-ins in figure 3.1 have to be replaced by their appropriate counterparts compatible with the target platform. Besides abstraction, the RCP is among many other tasks responsible for running the platform, handling plug-in management and providing an extension registry.

3.1.2. Basic User Interface

SWT, the Standard Widget Toolkit [NW04], offers platform-independent functionality for building graphical user interfaces having a native style. This is done by making use of platform-specific operating system features and having customized implementations for different platforms but providing the developer with a common interface. Draw2d is built on top of SWT and is a lightweight toolkit of graphical components called figures. In this context, "lightweight" means that figures are just Java objects with no corresponding resource in the operating system. This connection to graphical elements of SWT is established by `EventDispatchers` and `UpdateManagers`, as illustrated in figure 3.2. Be-

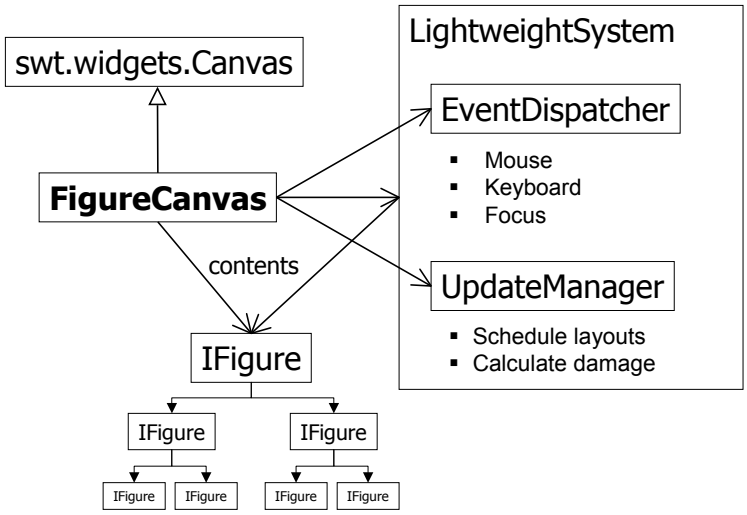


Figure 3.2.: Draw2d's lightweight figures

sides that, Draw2d comes along with a coordinate system allowing the translation between absolute and relative coordinates and provides connections between graphical elements which have definable decorations and different routing styles.

3.1.3. Eclipse Rich Client Platform User Interface

In order to provide RCP applications with a consistent, professionally looking graphical surface, several plug-ins have been developed among which most are based on SWT. JFace [Dau07] extends SWT's functionality by extending it with the model-view-controller pattern and introducing several helper classes supporting the developer in the creation of clean, well-structured code. On top of JFace lies the actual Eclipse Workbench user interface which comprises many concepts like Views, Editors, Perspectives, Dialog, Wizards etc. familiar from the Eclipse IDE. Additionally, a framework for managing help content is included.

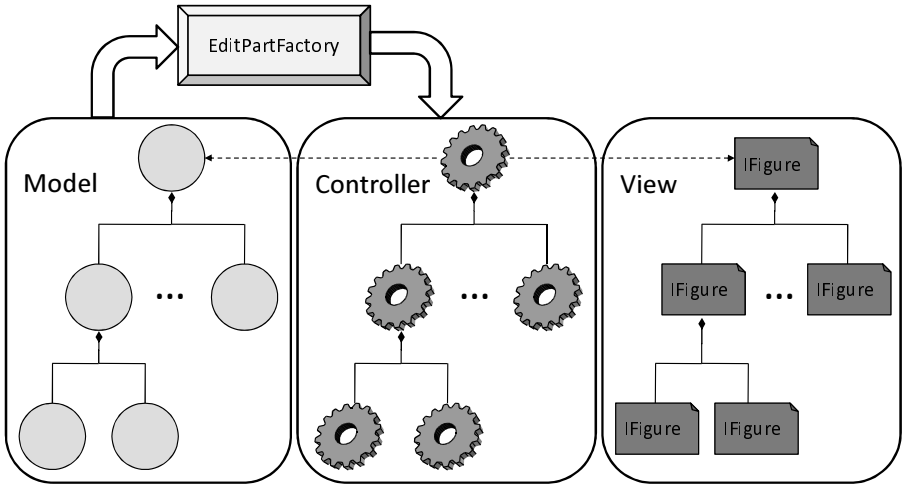


Figure 3.3.: Visualization mechanism producing a graphical representation of a model

3.1.4. Graphical Editing Framework

GEF, the Graphical Editing Framework, is built on top of Draw2d and is primarily used for visualizing models and interacting with them. Visualization is handled by so called `EditPartFactories` which create `EditParts` based on inserted models, as depicted in figure 3.3. Such an `EditPart` plays the role of a controller in the MVC-pattern, creates a graphical representation of the model and bridges both parts. The `EditPart` is also responsible for creating sub elements by querying the model for children and producing based on them new `EditParts`, again utilizing the `EditPartFactory`.

Generally speaking, `EditParts` perform graphical editing in the sense of that they delegate work to sub `EditParts`, display feedback during complex interactions with the user and manipulate the model in the way denoted in figure 3.4. Interactions having the form of SWT Events triggered by user input like mouse clicks or keyboard strokes are handled by corresponding tools which then forward the event as `Request` to an `EditPart` responsible for the graphical element which received the user input. The `EditPart` then asks installed `EditPolicies` having one or more roles like `CONNECTION_ROLE` or `LAYOUT_ROLE` to handle the

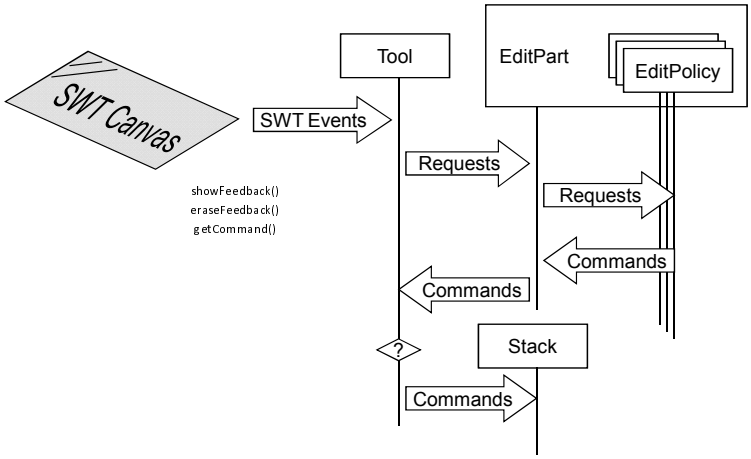


Figure 3.4.: Delegation of `Requests` to `EditPolicies` for `Command` creation

`Request` which may result in the creation of `Commands` responsible for updating the model. In this context, roles are used for correct `Request` routing and enable interchangeability of `EditPolicies`. The graphical user interface itself consists of a hierarchy of `Figures` and is usually kept up to date by utilizing listeners hooked on the model. Furthermore, useful workbench functionality comes along with GEF which comprises an undo/redo mechanism, key-bindings as well as contributions having the form of actions, menus and toolbars.

3.1.5. XStream

Its creators describe XStream[2] as simple library to serialize objects to XML and back again. A convenient property is the one that most objects can be serialized without the need for specifying what the XML-representation should look like. This fact in combination with the promise that the standard mapping behavior is kept stable throughout further releases of the persistency framework make XStream a powerful tool. It reduces the needed amount of refactoring drastically when developers are confronted with model changes. Finally, XStream is probably best known for the idea of producing clean XML by dropping informa-

[2] `http://xstream.codehaus.org`

tion which can be deduced by using the reflection mechanism and for having an efficient full object graph support avoiding duplicates of objects.

3.1.6. Alaska Help

This plug-in is based on the Eclipse Help plug-in and contains a detailed documentation of the Alaska journey simulator in the form of HTML pages and screen-casts. By extending the native Eclipse help system, features like a keyword search and a table of contents can be used out of the box along with the standard Eclipse look and feel.

3.1.7. Alaska Core

The Alaska Core plug-in contains the actual program logic and implements all concepts discussed in chapter 2. Besides providing the business model for the journey simulator, it comprises logging functionality crucial for persisting a travel's progress as well as decisions made by the traveler. Furthermore, several services are included, which are responsible for simulating the journey environment. This comprises making decisions about actions' availability, calculating their durations as well as performing local weather forecasts.

3.1.8. Alaska User Interface

The Alaska User Interface plug-in can be roughly categorized into two parts which have different responsibilities. The task of the first one of these is obviously to provide the application user with an attractive and intuitive graphical interface. This exercise was accomplished by extensively making use of previously discussed plug-ins like the Eclipse UI Framework and the Graphical Editing Framework. Beneath a powerful calendar editor, six different views exist with each of them supporting the user in his decisions by providing him with information structured in a compact way. The second part of the plug-in is used to refine the core functionality in order to support plan-driven-specific as well as agile-specific behavior.

3.2. Three-layered Architecture

Similar to many other well-structured desktop applications, Alaska can be divided into three different layers: The persistency layer, the presentation layer and the layer containing the business logic. It is important to avoid an overlap of these layers to keep the system well-structured and maintainable and to ease replacement of single modules. Figure 3.5 summarizes the most important com-

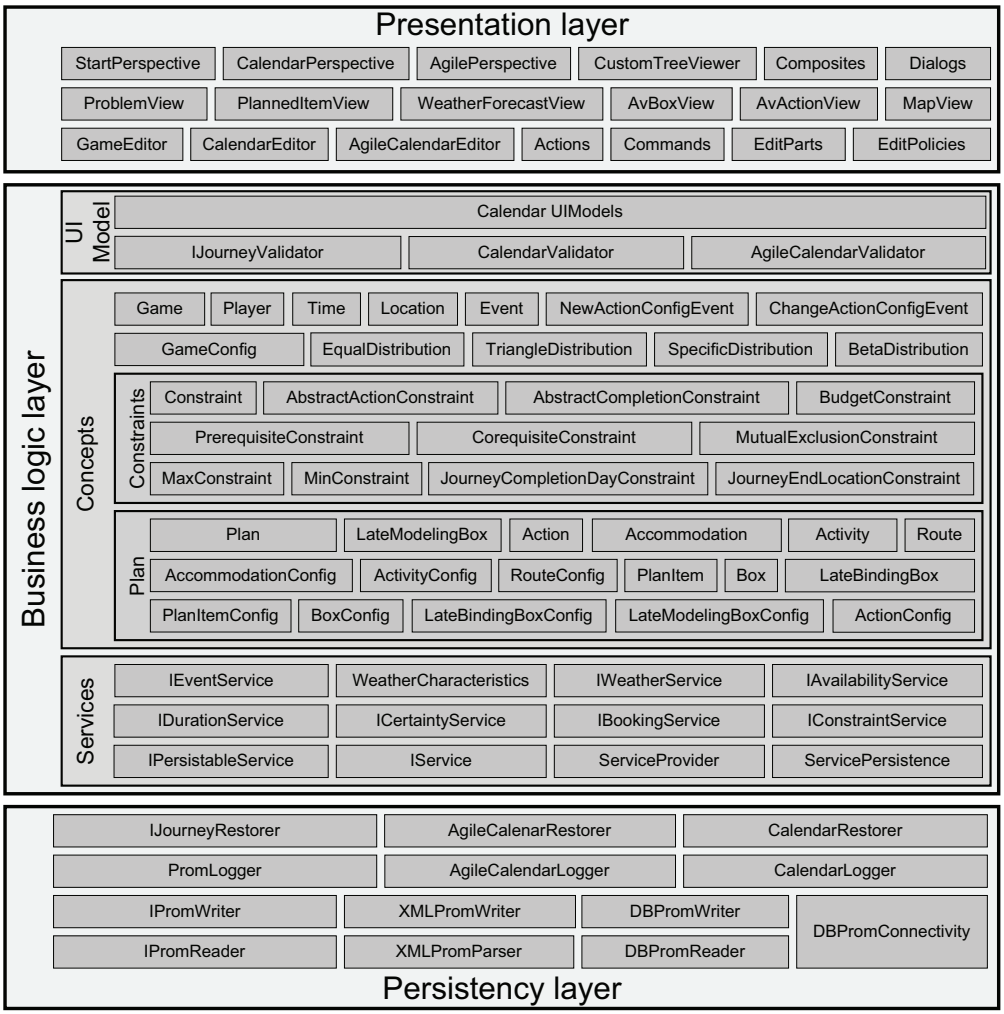


Figure 3.5.: Three-layered architecture of Alaska

ponents of the Alaska application which are discussed in the remainder of this chapter.

3.2.1. Presentation Layer

The presentation layer contains all graphical components responsible for enabling a human being to intuitively use the Alaska system. The composition of single elements is defined in the perspectives at which the `StartPerspective` is empty and is used after the program's startup to act as a transitional solution before one of the two main perspectives becomes active depending on the traveler's choice of planning methodology. Both, the `CalendarPerspective` and the `AgileCalendarPerspective` have very much in common in order to avoid that the users have to learn the program operation twice (see figure 3.6).

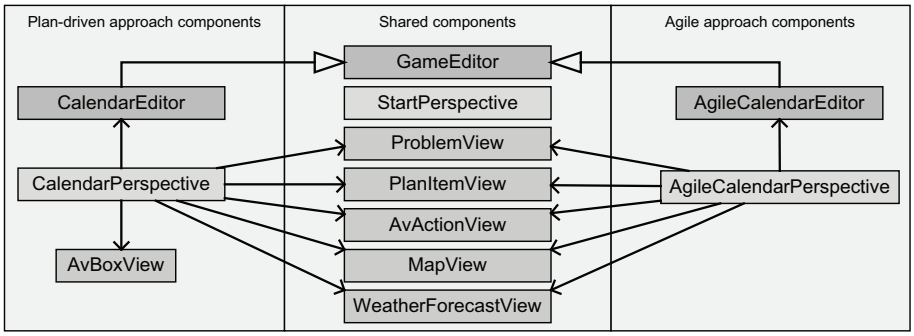


Figure 3.6.: Sharing components to increase usability and to flatten the learning curve

The `CalendarPerspective` contains a `CalendarEditor` which is a subclass of `GameEditor` and allows the traveler to schedule actions in a calendar-like fashion. Besides a smart Drag and Drop concept, context menus provide the user with all needed functionality. In contrast to the `AgileCalendarEditor`, the `CalendarEditor` is capable of handling travel packages which have the form of `LateBindingBoxes` and `LateModelingBoxes` whereas the `AgileCalendarEditor` allows the scheduling of parallel `Actions`.

Common in both perspectives are further the `ProblemView` informing about inconsistencies originating from an invalid travel plan, the `WeatherForecastView`

presenting a graphical forecast of a chosen location's weather, the `PlannedItemView` providing the traveler with more detailed information and control mechanisms of scheduled elements, the `MapView` offering a scrollable and zoomable map interface supporting the player geographically in planning decisions and the `AvActionView` listing available `Actions` and in combination with their characteristics. Similar to the `AvActionView`, but only available in the `CalendarPerspective`, is the `AvBoxView` containing available tour-packages and serving as drag source for the `Editor`. Further graphical elements of the presentation layer are customized `Composites`, diverse `Dialogs` for e. g. realizing the modeling of `Boxes`' content and informing about occurring `Events` and a customized `TreeViewer`.

The interaction between the presentation layer and the business logic layer reflects architectural core concepts of GEF and makes extensive use of `EditParts` and `Commands` created by corresponding `EditPolicies`. Figure 3.7 clarifies the interplay of these classes by using a move operation performed on an `Action` in the `CalendarEditor` as example. First, the `Action` is dragged which is recognized by the operating system and forwarded by SWT as `Event` to some `EventDispatcher`. This class then detects that the `Event` is a `SelectionEvent`, which leads to another forwarding to GEF's `SelectionTool`. The `SelectionTool` delegates the work to its `DragEditPartsTracker`, which tracks the position and status of dragged `EditParts` and creates `Requests`. Such a `Request` is used like a query to obtain a `Command` from the target `EditPart`. Usually, `Command` creation is delegated to installed `EditPolicies` which enables the developer to move code which modifies business objects away from `EditParts`.

In a second step, the user drops the `Action` which is again noticed by an `EventDispatcher`. In a similar way as before, the `MouseEvent` is forwarded to the `DragEditPartsTracker` which now executes the `Command` obtained in the previous step. An important fact is that the abstraction from the user interface data – in this case the conversion from mouse coordinates of the `CalendarEditor` to a `Time` object in the game – is performed by `EditPolicies`.

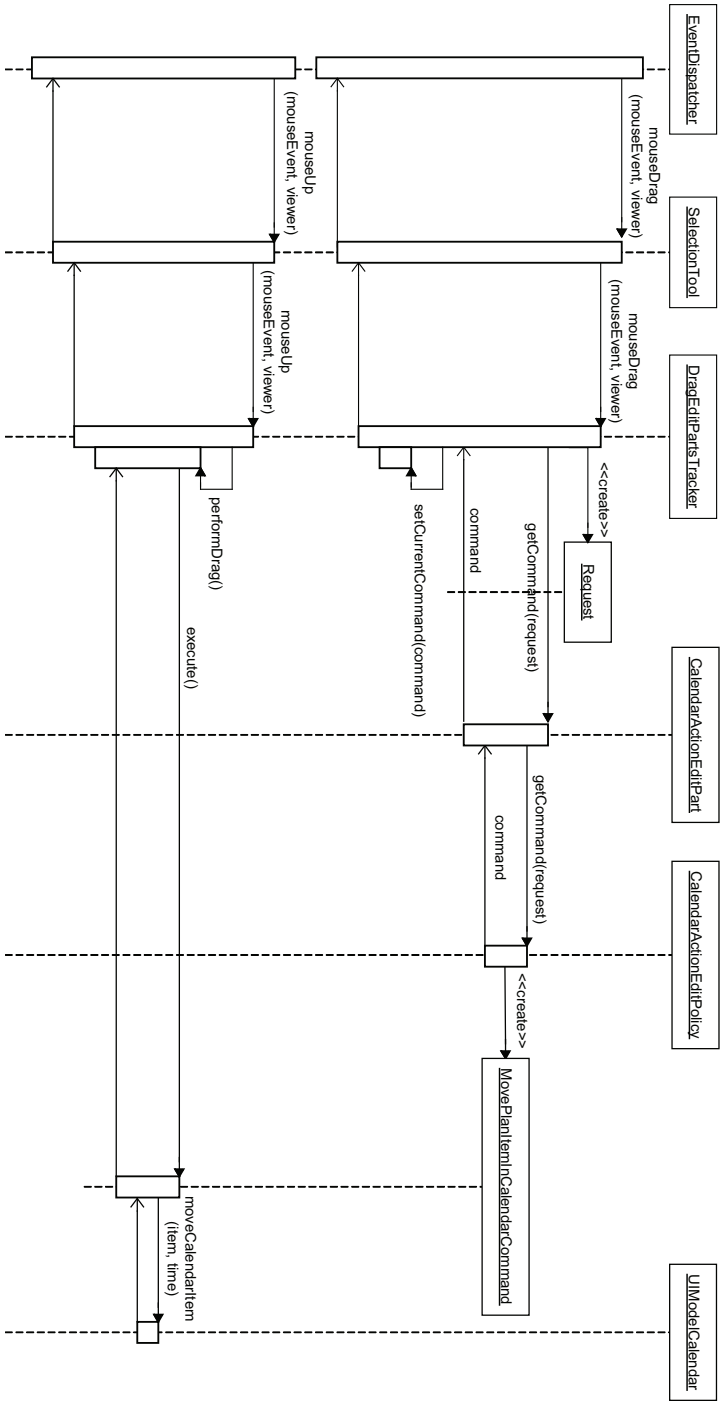


Figure 3.7.: Sequence diagram of an `Action` move operation

3.2.2. Business Logic Layer

The business logic layer contains all core functionality needed to simulate a journey. This layer acts as connection between the persistency layer used for persisting core data as well as tracking core operations and the presentation layer responsible for visualizing data and providing the user with an interface. The business logic layer can again be divided into three sub categories.

Concepts

The central part of the business logic layer is the one containing all conceptual elements discussed in chapter 2 (see figure 2.13 on page 51). A `Game` holds all information relevant for planning a journey and knows its `Player`, the current `Time` as well as the current `Location` and has a reference to the journey `Plan`. The `Plan` contains a list of `PlanItems`, which have a start `Time` and can be either `Actions` or `Boxes`. `Actions` are separated into `Accommodations`, `Activities` and `Routes`, and `Boxes` can be divided into `LateBindingBoxes` and `LateModelingBoxes`.

The difference between these classes and the corresponding configurations is that for example an `Action` represents a traveler's operation between two points of time and can be seen as instance of its configuration, whereas a configuration holds information valid for all instances like their name or description. Figure 3.8 illustrates the dependencies between configurations and instances and the elements in between called proxies. A configuration stores all static, immutable information and is used within several different games, whereas proxies are modifiable objects encapsulating configurations and storing changes caused by occurring events. When instances have to work with data from the configuration, they ask instead the corresponding proxy object which provides the actual data specific for the current game. The creation progress goes from left to right – configurations create proxies when a new game is created and proxies create corresponding `Actions` when needed during the game. Technically seen, this design reflects the use of two design patterns: The "Factory Method Pattern"

and a slightly modified version of the "Proxy Pattern" without a common super class [GHJ94].

When considering `Action`'s class hierarchy, it can be observed that `Action` provides many methods for manipulating itself or subclasses. An `Action` is always aware of its `State` and even remembers past state transitions. Core methods work based on this information and with the help of diverse services in order to check whether state transitions are possible and to trigger such transitions. The key idea behind this is that subclasses can extend and adapt these methods to guarantee correct behavior. An example might be that upon execution of an `Accommodation` it has to be checked whether the current day is not the last day of the journey, whereas when planning a `Route`, there just has to be enough time left on the current day.

User Interface Model

The user interface model, as the name already suggests, is a model encapsulating the actual business model but containing also information relevant for visualization. A simplification of this relation is depicted in figure 3.9 and points out the parallelism of the class hierarchy – `UIModelCalendarBox` and `UIModelCalendarAction` are user interface models of `Box` and `Action` respectively. A common superclass `UIModelCalendarItem` provides advantages by enabling polymorphism similar to `PlanItem` in the core model. Finally, `UIModelCalendar` is the correspondent to `Plan` and coordinates `UIModelCalendarItems`.

Besides that, the user interface model is enriched with a notification mechanism which informs listening components like elements of the graphical user interface about model changes. Figure 3.10 illustrates the dependencies and shows that the flow of information about a change is going up the model hierarchy and is distributed by a top-level element in the presentation layer to corresponding lower-level elements. More precisely speaking, model changes appear when methods are called by controller classes on a `UIModelCalendarAction` or on a `UIModelCalendarBox`. Depending on the change, a `PropertyChangeEvent` is generated which holds all information about the occurred change and this event is then forwarded to the top-level model element, which is `UIModelCalendar` in

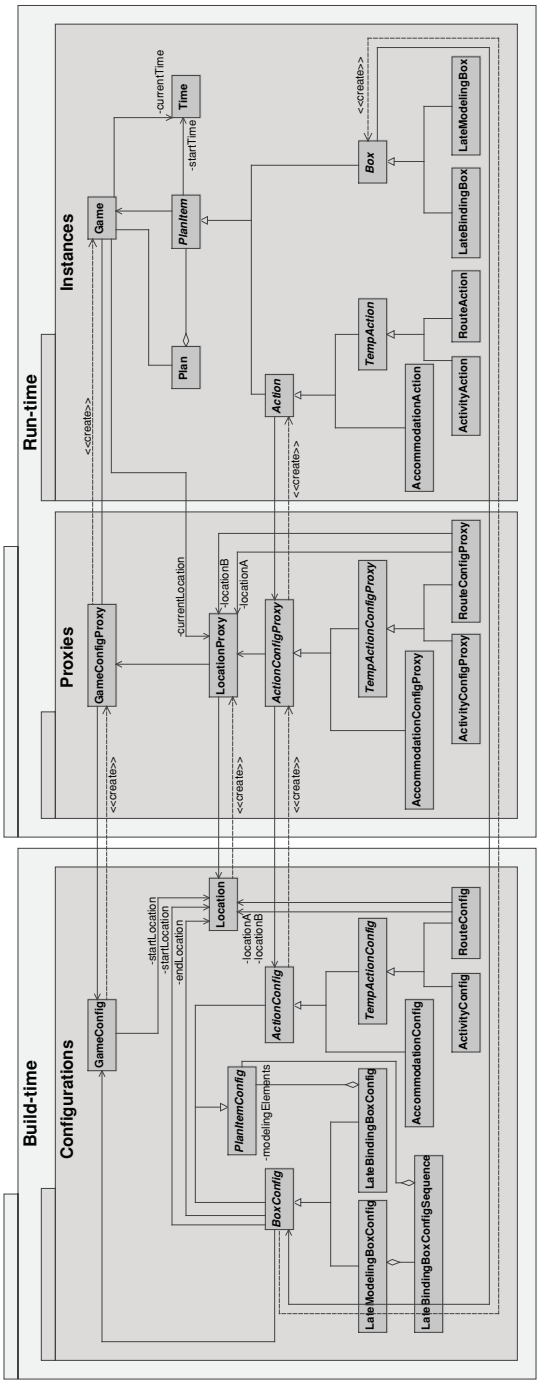


Figure 3.8.: Proxies acting as intermediates between configurations and instances

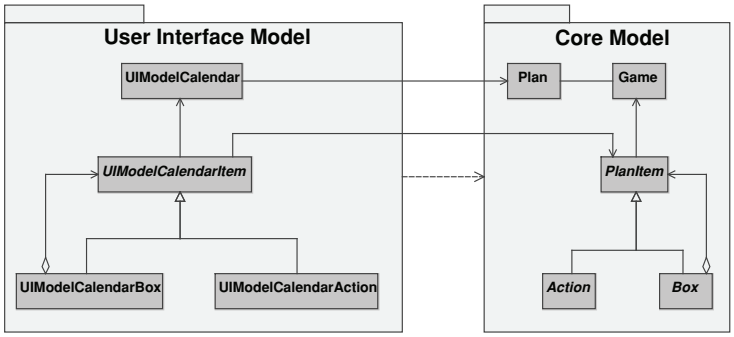


Figure 3.9.: User interface model as an extension of the core model

the plan-driven perspective. `UIModelCalendar` is a subclass of `AbstractJourney` which again is a subclass of `UIModel`. The latter implements a `PropertyChangeSupport` which allows an arbitrary number of listeners to register themselves and to get informed about changes. In our scenario, the `CalendarEditor` is informed which the actual step across the border to the presentation layer. The `CalendarEditor`, which is a subclass of `GameEditor`, has two kinds of duties: First, it has to find the `EditPart` belonging to the modified model with the help of a mapping and to trigger an update to refresh graphical elements. The second task is to inform global components like actions, views and the status line.

Furthermore, the user interface model extends the set of functionality provided by the core model to some degree and supports methodology-specific concepts like boxing in plan-driven journeys and parallel execution paths in agile voyages. Additionally, it contains validation mechanisms which check the integrity of planned journeys (see figure 3.11). Since agile and plan-driven journeys have a different conception of validity, two specific validators are needed at which the following discussion focuses on the `CalendarValidator` responsible for the plan-driven `UIModelCalendar`.

The first thing the `CalendarValidator` checks is the availability of `Actions`. In this context, `Actions` lying in the past are not taken into consideration whereas `Actions` of the current day of which the availability is exactly known have to be available. In addition to that, it is checked whether future `Actions`' expected

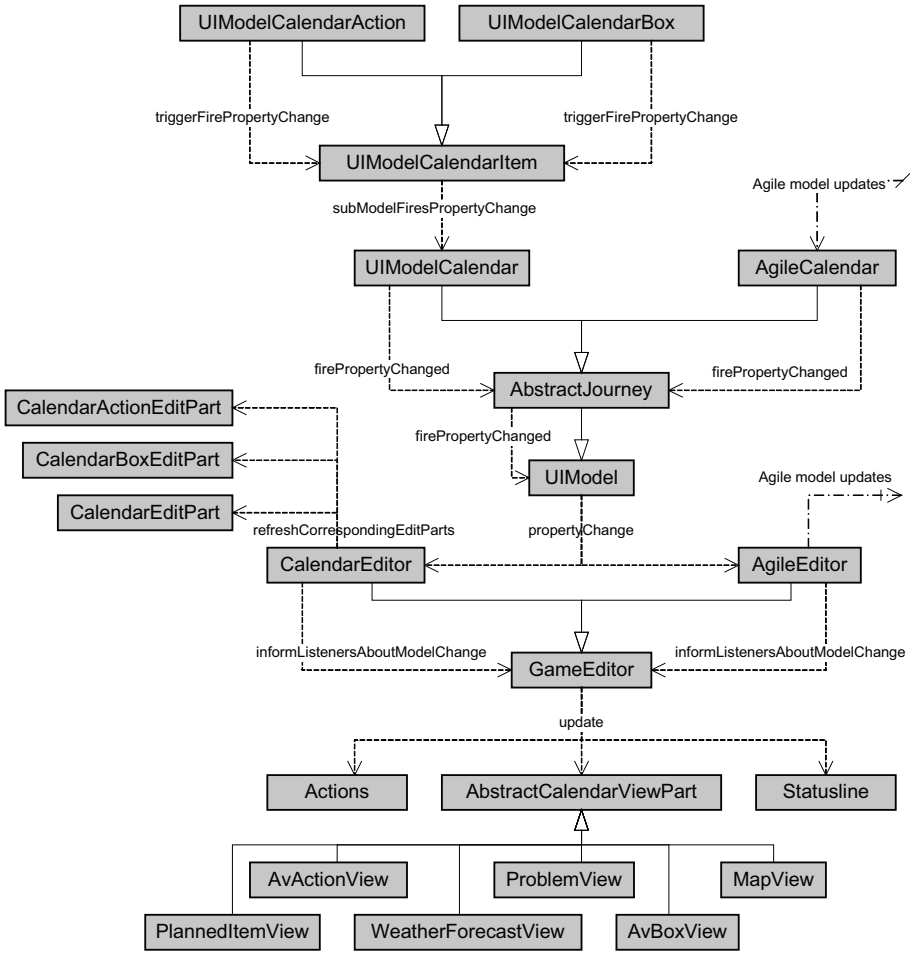


Figure 3.10.: Change-notification mechanism in the user interface model

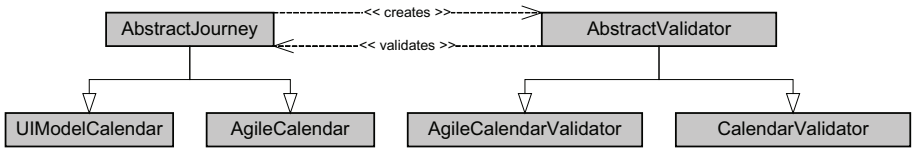


Figure 3.11.: Journeys and corresponding validation classes

availability is at least above zero. Another important subject of inspection is the geographic compatibility of `Actions`, meaning that the end `Location` of a preceding `Action` must be equal to the start `Location` of the succeeding one. Besides that, a `Plan` is only judged as valid when every day of the journey contains an `Accommodation` to spend the night at. Finally, all global constraints like having mandatory `Actions` included in the `Plan` or finishing the journey at a specific `Location` have to be satisfied. In the case of invalidity, a `ValidationResult` is returned which contains one or more `Failures` explaining the validation result and providing hints about the source of error. This in return allows the highlighting of faulty elements in the calendar which dramatically eases error handling for the user.

Services

Services are a crucial element of Alaska's core, since they simulate the environment of a journey. The `IAvailabilityService` determines whether a given `Action` is available at a certain point of time. The `IDurationService` informs about an `Action`'s duration which can vary within a certain time frame and the `ICertaintyService` calculates with which certainty the maximum business value will result from executing an `Action`. Furthermore, the `IConstraintService` is used for performing generic checks on the travel plan, the `IBookingService` is utilized to perform bookings which guarantee the availability of its subject, the `IEventService` coordinates the creation of random `Events` and the `IWeather-Service` is used to obtain `Location` dependent forecasts of the weather.

All services can be accessed by using the `ServiceProvider` and are unique for each instance of `Game`. In order to be able to restore a service's state after loading a saved game, its data has to be persisted in some way. By implementing the interface `IPersistableService`, each service can define which data should be stored by itself and the actual storage procedure happens in the persistency layer.

3.2.3. Persistency Layer

The persistency layer is the connection between the business logic layer and the underlying storage device. During the application's startup, an `IGameConfigService` is created which searches at a predefined location for game configurations which have the form of ZIP files. Such a file typically contains the actual configuration stored in a XML format and associated images referenced by the configuration. The `IGameConfigService` unmarshals the configuration and creates a user interface model of it which is equipped with a reference to the ZIP file to enable load on demand behavior for images. During a journey, all user inputs relevant for reconstructing a voyage are being logged which is done by the `PromLogger` making use of a `IPromWriter`. Depending on the chosen planning methodology, the `AgileCalendarLogger` or the `CalendarLogger` makes use of the `PromLogger`'s basic logging mechanism and produces logging entries for `Action` creations, `Action` movements, occurrences of `Events`, etc. The logged information is further enriched with data describing modifications of service's states and conforms to the Mining XML [vdAvDG+07] format. Based on logged information, a journey can be restored step by step: An `IPromReader` analyzes the input and the unmarshalled data to one of `IJourneyRestorer`'s subclasses `AgileCalendarJourneyRestorer` and `CalendarJourneyRestorer`, depending on the type of journey. This is a powerful feature and enables the user to restore his journey until a certain point of time and to try some alternatives from there on.

Currently there exist two implementations for the `IPromReader` and `IPromWriter` – one supporting persisting data to files in XML format and the other allowing their storage in a relational database. Figure 3.12 illustrates how the persistency layer classes are related to each other. Furthermore, figure 3.13 provides an overview of possible information flows within the persistency layer.

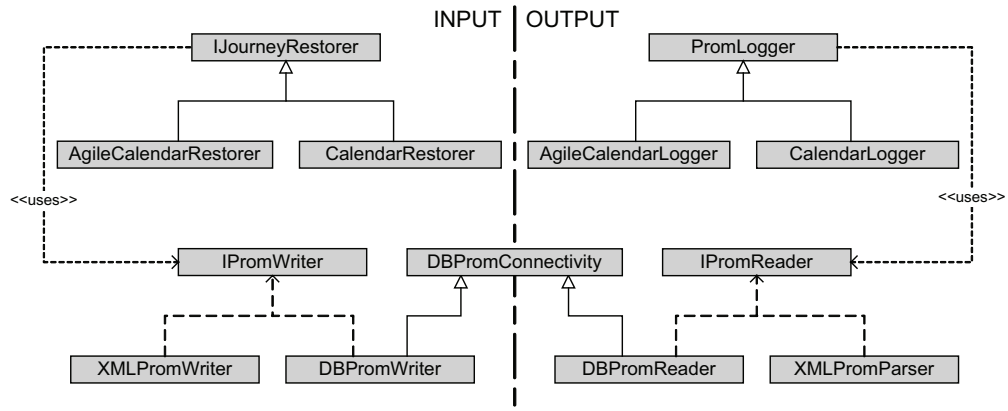


Figure 3.12.: Class diagram of the persistency layer

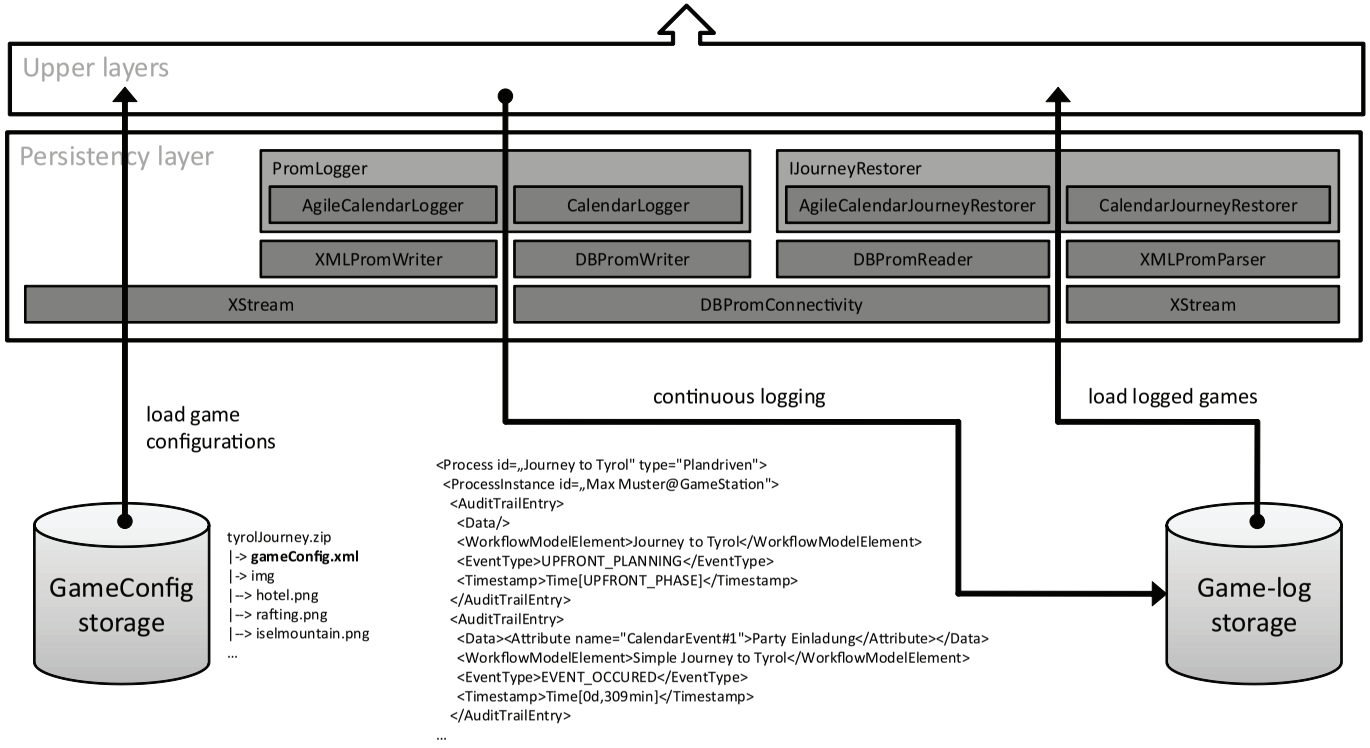


Figure 3.13.: Flow of information in the persistency layer

Chapter 4.

Experiment

"The great tragedy of Science – the slaying of a beautiful hypothesis by an ugly fact."

Thomas Huxley

In this chapter, the experiment and its results are described in detail. We begin our discussion in section 4.1 with an explanation of basic terminology frequently used in subsequent sections. Afterwards, we give an outline of the experiment's design (section 4.2) followed by section 4.3 describing the experiment's execution and the collection of relevant data. Subsequently, in section 4.4, the data is analyzed, the results' meaning is explained (section 4.5) and in section 4.6, a risk analysis is conducted and combined with proposals for risk mitigation. At the end of the chapter, the experiment's results are extensively discussed (section 4.7).

4.1. Basic Terminology

Various design guidelines about software experiments can be found in literature [Bro90],[PSS81]. Important expressions essential for understanding the experiment's design have to be explained before demonstrating the actual experiment process. In order to clarify terminology, a scenario of an example experiment adapted from [Bor77] will be used. In this, the experimenter wants to examine which impact the use of election advertising has on townsmen's votes. In subsections 4.1.1 to 4.1.6, all important concepts will be explained in detail at which

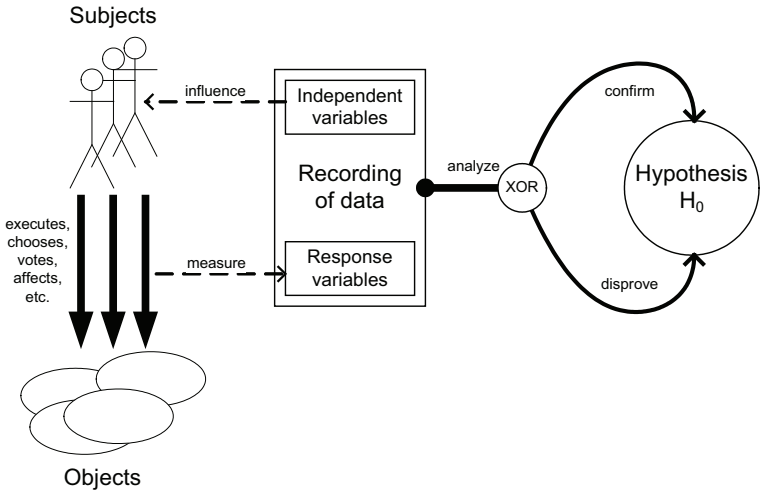


Figure 4.1.: Basic concepts of an experimental setup

the overall interplay of them is illustrated in figure 4.1. Subjects, which are influenced by independent variables, manipulate or act on a set of provided objects. The consequences of the subjects' actions are measured by the experimenter to obtain the response variables. In a second step, he analyzes the data and decides whether to accept or reject a stated hypothesis H_0.

4.1.1. Subjects

Subjects can be persons, physical objects as well as non-physical things like opinions or style trends. They serve as source for experimental data by actively producing input or by passively providing processible information. In connection with the election scenario, all citizens entitled to vote are in fact subjects of the experiment.

4.1.2. Objects

Objects as well as subjects can take diverse forms. They form the task, problem or exercise with which the subjects are confronted. Because of this, the choice

of objects highly influences data produced by subjects. As example, the election together with a set of electable parties form the experiment's object.

4.1.3. Independent Variables

Independent variables, also referred to as factors, are experimental variables which are manipulated by the experimenter. They are control parameters allowing him to conduct the experiment multiple times in parallel with each having different factor level configurations. Based on that, the experimenter can group his results to gain information about the impact of the parameter's variation. Generally, only a small set of independent variables are selected to be controlled in the experiment. The amount of election advertising is such an independent variable which can take several magnitudes.

4.1.4. Response Variables

A parameter value measured during or after the experiment is called response variable. Such a variable usually depends on independent variables and may be influenced by other response variables. Generally speaking, the response variable is essentially a measure of the effect of change in an independent variable and is determined when the participants of the experiment apply the factor levels to an object. Analysis of an experiment is primarily conducted based on the combination of independent and response variables. Considering again the election scenario, the response variable is the election's result – more precisely the number of votes a monitored party receives for a certain election given a certain amount of money spent on advertisements.

4.1.5. Experimental Designs

Probably the most important concern in experiments is the possibility to be in full control over the experimental situation. Such experiments are called randomized or true experiments. Unfortunately, the experimenter is often not free to determine the subjects by himself or to control independent variables (see

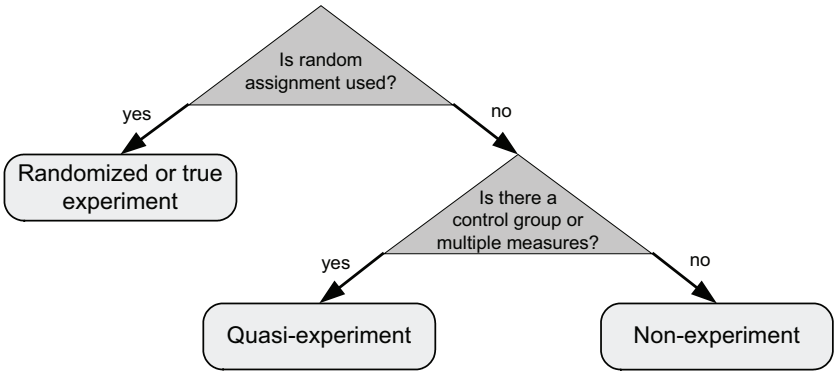


Figure 4.2.: Three different types of experimental design

figure 4.2). As a result, the assignment of single subjects to groups is not randomized anymore. Such experimental situations form part of quasi-experimental designs. The third category are non-experimental designs in which there are neither multiple goups nor multiple waves of measurements [Tro00].

Due to randomization, true experimental designs deliver the most significant results. As mentioned before, the experimenter is often confronted with situations prohibiting randomized group assignments. As a result, he has to use a quasi-experimental design. The important thing to mention here is that the significance of results obtained from such experiments strongly depends on whether the groups are balanced or not. If the distribution of subjects among different groups is well-balanced, quasi-experimental designs deliver results being nearly as significant as results obtained from true experiments. Finally, a non-experimental design is the least meaningful experiment design. Still, the experimenter is sometimes forced to establish such an experiment – just imagine a study about the effects of a natural disaster like a tornado by interviewing survivors. You will only have one comparison group (townsmen who are still alive in the affected city) as long as the tornado does not devastate a second city. A further example could be that a university professor wants to compare levels of success of different forms of student courses. He compares lectures, tutorials and a combination of both, but is unable to perform his experiment at only one university because of administrative reasons. Therefore, he is forced to compare results from differ-

ent universities. Unfortunately, they provide different forms of courses offered to students coming from diverse semesters. This in fact prohibits the professor from having full control over the choice of subjects

4.1.6. Hypotheses

An experiment is generally conducted to produce data which is then statistically analyzed [Ste99]. Based on the results, a postulated hypothesis is either accepted or rejected with a certain amount of significance. Hypotheses are simplified, unambiguous models of reality [PKB05]. A scientist usually makes an assumption, formulates it as hypothesis and tries to falsify it. In this experiment, two statistical hypthesis are postulated which make an assumption about the distribution of a random variable. When considering a distribution's mean value μ, one distinguishes between the null hypothesis H_0 and the alternate hypothesis H_1, which state the following:

$$H_0\text{: } \mu = \mu_0 \quad \text{or} \quad \forall i, j \in \{1..n\} : \mu_i = \mu_j \tag{4.1}$$

$$H_1\text{: } \mu \neq \mu_0 \quad \text{or} \quad \exists i, j \in \{1..n\} : \mu_i \neq \mu_j \tag{4.2}$$

The chosen level of significance γ can be seen as point of reference and limits the probability of making errors. In this context, two types of wrong decisions can be made: the type I error rejecting a null hypothesis when it is actually true and the type II error failing to reject a null hypothesis when the alternative hypothesis is true.

4.2. Experiment Design

This section describes the experiment's design and identifies its subjects, objects, independent variables etc. An overall summary can be found in table 4.1 as well as figure 4.3.

The Alaska application, already mentioned in previous chapters, is used in the experiment. **Subjects** are students of a Bachelor Computer Science program at the University of Innsbruck. In order to mitigate learning effects in connection

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

Table 4.1.: Elements of this experiment's design

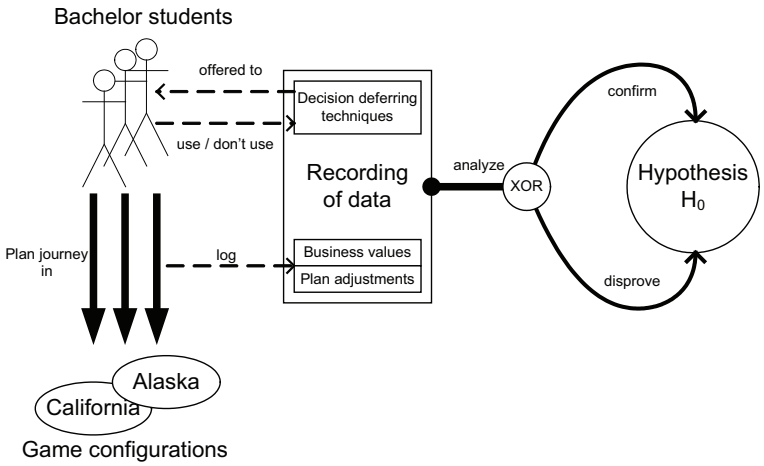


Figure 4.3.: This experiment's setup

with a second experiment being executed in parallel and playing a role in [Zug08], two travel scenarios are used. From the view of the simulator application, travel scenarios are `GameConfigs` (see chapter 3) and define the general framework of a journey. This comprises among other concepts a set of locations combined with possible actions, constraints which have to be taken into account, and events which might happen unexpectedly.

Both `GameConfigs` are defined by the experimenter and form the experiment's **objects**. They differ in their design because the first configuration taking the traveler to California does not allow precise estimations about action's duration whereas the Alaskan journey, as the second configuration, confronts the traveler

with a lot more unforeseen events. The degree of complexity concerning the number of existing locations and actions is approximately the same. Besides that, scenario 1 is characterized by more stable weather conditions whereas Alaska basically only allows to take one reasonable route in order not to lose too much business value. These circumstances force us to analyse results originating from unequal configurations separately.

This experiment's **independent variable** is the traveler's choice whether to use decision deferring techniques in the plan-driven simulation or not. The two **response variables** are the overall business value of a finished journey and the number of performed travel plan adaptation steps.

To clarify how we obtain results crucial for this thesis, we have to explain the overall experiment process including the part of Zugal [Zug08], Alaska's co-author: Zugal's *independent variable* is the applied planning methodology which is controlled by the experimenter as follows: The subjects are randomly divided into two groups of equal size. During the experiment, each group has to plan two journeys in both scenarios, but uses different planning techniques (see figure 4.4). Group A plans a journey in travel scenario 1 (California) by using the plan-driven approach, before planning a second journey in travel scenario 2 (Alaska) with the help of agile concepts. Group B also starts with travel scenario 1 (California) but makes use of the agile planning mode first whereas the second planning session dealing with travel scenario 2 (Alaska) is plan-driven.

Focusing on this thesis' experiment, only half of the gained results are of concern. Data obtained from journeys planned by using the agile approach is discarded, whereas data from plan-driven journeys is taken under consideration (results of group A in first run and results of group B in second run – see figure 4.4). The interest primarily lies on differences between pure plan-driven journeys and journeys making use of decision deferring tour packages.

Defining the student's decision of whether to utilize such concepts or not as fully controllable independent variable of the experiment would give us the possibility to obtain two equally sized samples. Because of the experiment's setup, this is not possible and the subjects can make the decision on their own accord-

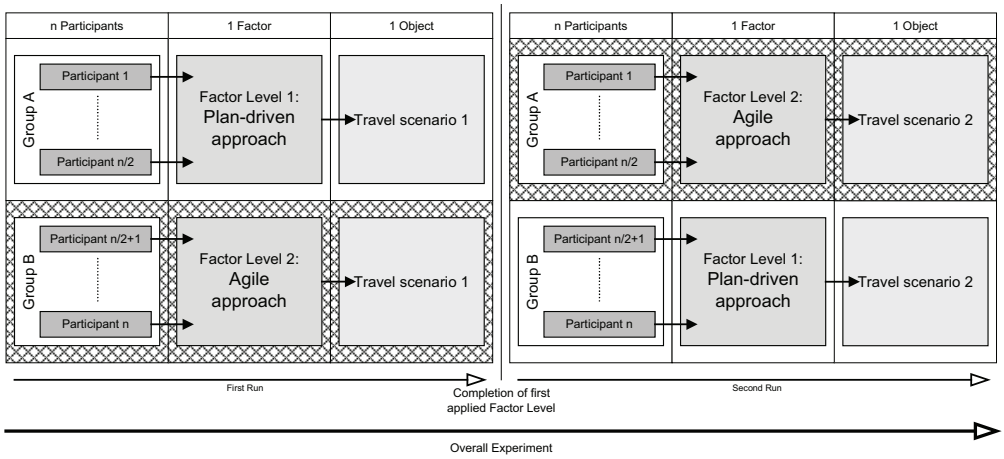


Figure 4.4.: Two phases of the experiment

ing to their preferred planning behavior. As we will see later on, this does not mitigate this experiment's expressiveness due to the near-balance of groups.

As a consequence, this thesis' experiment can be seen as **unbalanced single factor experiment with repeated measurement** investigating the effects of the *adoption of advanced planning concepts* like tour packages on the two previously mentioned response variables (*business value* and *absolute change frequency*). The central question of how these indicators behave depending on the use of decision deferring techniques leads to the formulation of two null hypotheses:

- H_0^{BV}: Using advanced techniques like Late Modeling and Late Binding yields no significant difference in resulting business value when compared to strict plan-driven methodologies.

- H_0^{CF}: Using advanced techniques like Late Modeling and Late Binding yields no significant difference in frequency of project plan changes when compared to strict plan-driven methodologies.

In order to be able to derive these values, we make use of logging information provided by Alaska and analyze the devolution of each journey in the samples.

This instrumentation procedure has very low risk because everything is done automatically – the user does not have to fill out questionnaires and is not explicitly involved in the production of data to be analyzed. Furthermore, it is not feasible for a user to manipulate logged data, because the latter is collected in a remote protected database. Finally, for data analysis, we use well established statistical methods and standard metrics ([RMHL91], [She00], [Bor77], [Hog06]).

Mathematically seen, the experiment consists of n subjects S_i with $1 \leq i \leq n$, divided into two groups each of size $\frac{n}{2}$. Each subject S_i plans two journeys using the agile and the plan-driven approach respectively as depicted in figure 4.4. To countervail learning effects, two different travel scenarios T_C (California) and T_A (Alaska) are used. The data collection procedure gives us $2n$ results $R_j^{T_C,Agile}$, $R_j^{T_A,Agile}$ as well as $R_j^{T_C,Plan}$ and $R_j^{T_A,Plan}$ with $j \in \{1, \ldots, \frac{n}{2}\}$ in the form of journey logs. Now we discard results obtained from agile journeys, omit the second superscript for convenience and further refer to n as the number of students planning a plan-driven journey within one run. Doing so, we end up with two remaining result sets $\{R_1^{T_C}, \ldots, R_n^{T_C}\}$ and $\{R_1^{T_A}, \ldots, R_n^{T_A}\}$.

4.3. Experiment Execution

The experiment took place in April 2008, was split up into four sessions taking place during three days and its total duration was 12 hours. The participants were 56 students of the Bachelor program in Computer Science at the University of Innsbruck and attended the courses *Software Development and Project Management* as well as *Advanced Topics in Software Engineering*. The latter is a course offered to both Bachelor and Master students which forced us to discard Master students' data in order to avoid having a too high variance among experiences of subjects.

Each session began with a presentation summarizing the concepts of plan-driven and agile planning techniques. The subjects also were informed about the experiment's process and its scientific connection to this thesis. Furthermore, we explained that the analysis of their own planning behavior is the experiment's main goal and that different techniques of dealing with uncertainty will be avail-

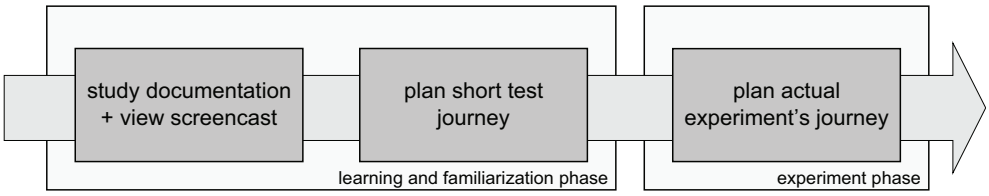


Figure 4.5.: A familiarization phase prepended to the actual experiment run

able. Finally, we illustrated the idea of taking a journey as metaphor for a software project and underlined the importance of planning in project management.

Subsequently, we explained how to set up the Alaska simulator and randomly assigned each student to one of two different groups. As depicted in figure 4.5, the subjects first studied a screencast specific for the approach they had to follow and then planned one or more test journeys in the first travel scenario – **California**. At this, *Learning and familiarization phase* lasted for 20 minutes whereas the actual Experiment phase lasted twice as long. After finishing the first round and a short break, the subjects were asked to repeat the whole procedure but use the complementary approach in a new planning scenario – **Alaska**.

4.4. Data Analysis Procedure

The data analysis procedure can be split up into two steps with the first one focusing on the data's validity and the second one analyzing the data using well-established statistical methodologies.

4.4.1. Data Validation

During the experiment, all data sets are stored in a central database. To guarantee **data consistency** we discard data originating from test phases by considering logged timestamps. Furthermore, we exclude duplicates (loaded games) as well as multiple games created by the same host.

Data plausibility is analyzed by considering the journey's state at the end of the game and by estimating the students' seriousness. We define the first condition to be valid if all actions in the game plan have been executed, which is the case in all data sets. The second condition is violated if the business value of the journey under consideration is a lower outlier (smaller than median minus three times the inter quartile range) and is judged by the experimenter as futile. Only one data set had to be eliminated by this filtering which was already predictable by studying the student's behavior during the experiment.

4.4.2. Data Analysis

As mentioned before, we discard data from journeys planned using the agile approach and take only results of plan-driven journeys of the first and second test run into consideration denoted by

$$\begin{aligned} &\{R_1^{T_C}, \ldots, R_n^{T_C}\} \\ &\{R_1^{T_A}, \ldots, R_n^{T_A}\} \end{aligned} \tag{4.3}$$

This data is again split up into two sets of unequal size using presence $R_i^{T_*,+}$ and absence $R_i^{T_*,-}$ of tour packages respectively as differentiator where m_C denotes the number of test persons which make use of tour packages in the first run (California) and m_A denotes the amount of test persons in the second run (Alaska) making use of such packages. As a consequence, $m_C, m_A \in \{1, \ldots, n\}$

$$\begin{aligned} &\{R_1^{T_C}, \ldots, R_n^{T_C}\} \Longrightarrow \{R_1^{T_C,+}, \ldots, R_{m_1}^{T_C,+}\}, \{R_1^{T_C,-}, \ldots, R_{n-m_1}^{T_C,-}\}, \\ &\{R_1^{T_A}, \ldots, R_n^{T_A}\} \Longrightarrow \{R_1^{T_A,+}, \ldots, R_{m_2}^{T_A,+}\}, \{R_1^{T_A,-}, \ldots, R_{n-m_2}^{T_A,-}\} \end{aligned} \tag{4.4}$$

From these results which have the form of log entries in a central database, we extract desired data which is the gained business value $BV_i^{T_*,*}$ as well as the number of project plan adaptations $CF_i^{T_*,*}$ during the game:

$$\begin{aligned}
\{R_1^{T_C,+},\ldots,R_{m_1}^{T_C,+}\} &\Longrightarrow \{BV_1^{T_C,+},\ldots,BV_{m_1}^{T_C,+}\},\{CF_1^{T_C,+},\ldots,CF_{m_1}^{T_C,+}\}\\
\{R_1^{T_A,+},\ldots,R_{m_2}^{T_A,+}\} &\Longrightarrow \{BV_1^{T_A,+},\ldots,BV_{m_2}^{T_A,+}\},\{CF_1^{T_A,+},\ldots,CF_{m_2}^{T_A,+}\}\\
\{R_1^{T_C,-},\ldots,R_{n-m_1}^{T_C,-}\} &\Longrightarrow \{BV_1^{T_C,-},\ldots,BV_{n-m_1}^{T_C,-}\},\{CF_1^{T_C,-},\ldots,CF_{n-m_1}^{T_C,-}\}\\
\{R_1^{T_A,-},\ldots,R_{n-m_2}^{T_A,-}\} &\Longrightarrow \{BV_1^{T_A,-},\ldots,BV_{n-m_2}^{T_A,-}\},\{CF_1^{T_A,-},\ldots,CF_{n-m_2}^{T_A,-}\}
\end{aligned} \tag{4.5}$$

Our hypotheses H_0^{BV} and H_0^{CF} are now analyzed based on two-sided t-Tests for two independent samples [RMHL91]. In doing so, we can decide whether the means of data sets statistically differ from each other. We define our error probability $\alpha = 0.05$ and get as level of significance $\gamma = 0.95$. Focusing on the business value, this means that we reject H_0^{BV} after successfully conducting two two-sided t-Tests with data from the first and second run when

$$|T_{BV^{T_*}}| > T_0 = t(1-\frac{\alpha}{2}, n-2) = t(0.975, 26) \cong 2.056 \tag{4.6}$$

with

$$T_{BV^{T_*}} = \sqrt{\frac{m_k \cdot (n-m_*)}{n}} \cdot \frac{\overline{BV^{T_*,+}} - \overline{BV^{T_*,-}}}{s_{BV^{T_*}}} \qquad * \in \{C, A\} \tag{4.7}$$

At this, $\overline{BV^{T_*,+}}$ and $\overline{BV^{T_*,-}}$ are the sample mean business values of the test run using scenario $*$. $s_{BV^{T_*}}$ is the weighted standard deviation and is defined as follows:

$$\overline{BV^{T_*,*}} = \frac{\sum_i^{l^{T_*,*}} BV_i^{T_*,*}}{l^{T_*,*}} \tag{4.8}$$

$$s_{BV^{T_*,*}} = \sqrt{\frac{\sum_i^{l^{T_*,*}} \left(BV_i^{T_*,*} - \overline{BV^{T_*,*}}\right)^2}{l^{T_*,*}-1}} \tag{4.9}$$

$$s_{BV^{T_*}} = \sqrt{\frac{(m_*-1)s^2_{BV^{T_*,+}} + (n-m_*-1)s^2_{BV^{T_*,-}}}{n-2}} \tag{4.10}$$

Testing of H_0^{CF} is performed analogously.

To guarantee the legitimacy of the t-Test, we have to check whether the samples emanate from a normal distribution. This is done using the Kolmogoroff/Smirnov Test [She00] at which we use a confidence interval of 5% which is approximately defined as $K_0 = \frac{1.36}{\sqrt{n}} = \frac{1.36}{\sqrt{28}} \cong 0.257$. The task here is to show that

$$K_x \leq K_0 \tag{4.11}$$

at which

$$K_x = max\left(\bigcup_{i=1}^{n} \left\{\left|S(x_i) - \Phi[\overline{x}, s_x](x_i)\right|, \left|S(x_{i-1}) - \Phi[\overline{x}, s_x](x_i)\right|\right\}\right) \tag{4.12}$$

Here, S denotes the relative cumulative distribution function of the input data x_i and Φ is the cumulative distribution function of the normal distribution using mean value and standard deviation of the input data as estimation for the actual distribution.

Furthermore, both samples which are to be compared must have the same variance. This is tested using the two sample F-Test [Bor77] and we have to check whether

$$\frac{\max(s_x^2, s_y^2)}{\min(s_x^2, s_y^2)} = F \leq F_{0,*} = F\big(1 - \frac{\alpha}{2}|m_* - 1, n - m_* - 1\big) \tag{4.13}$$

where $F_{0,*}$ is the $1 - \alpha/2 = 0.975$ quantile of the Fisher-distribution with $m_* - 1$ and $n - m_* - 1$ as degrees of freedom and therefore depends on m_*, the number of subjects using decision deferring techniques in scenario T_*. Besides that, x and y represent the sample data the variances of which are to be compared. Preconditions for applying the F-Test are that the samples emanate from a normal distribution which has already been checked using the Kolmogoroff/Smirnov Test and that both samples are independent from each other which is the case due to the experiment's setup. Additionally to the previously described test series, we use statistical measures like the *median*, the *interquartile range*, the *expected*

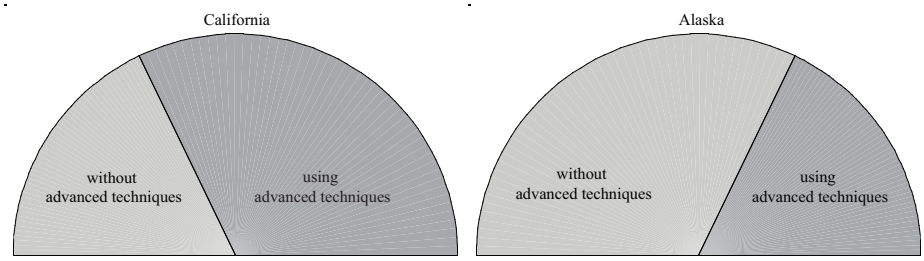


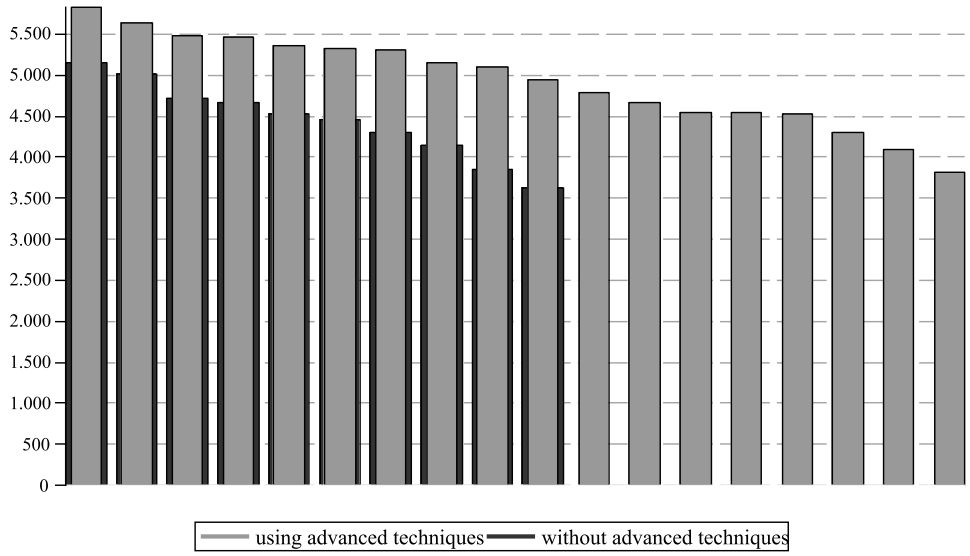
Figure 4.6.: Distribution of subjects in both travel scenarios

value and the *standard deviation* to construct Box-Whisker plots to enable a graphical interpretation of measured results.

4.5. Experiment Results

We analyzed the results of **56 subjects** of which **28** did planning in travel scenario T_C and **28** did planning in scenario T_A. At this, both groups were using the plan-driven approach. As mentioned before, the decision about using packages as advanced technique for design decision deferral was made by the subjects themselves. The resulting distribution is visualized in figure 4.6. When traveling through California, $m_C = 18$ students used packages and $n - m_c = 10$ students favoured the strict plan-driven approach whereas in Alaska, only $m_A = 10$ students used packages and the remaining $n - m_A = 18$ students did their planning without them. The reason for this distribution can only be guessed. We suppose that the fact that in Alaska many actions had to be booked in advance deterred subjects from using travel packages due to the increased level of complexity.

Obtained results are illustrated in figure 4.7 and 4.8. A table of all measured values can be found in appendix A. These two plots show the gained business values and the sums of project plan adaptations respectively at which light-grey results are obtained using decision deferring techniques and dark-grey results come from strictly plan-driven travelers. In this context, plan adjustments are defined as the sum of pre-planning steps (action scheduling, sequence selections,



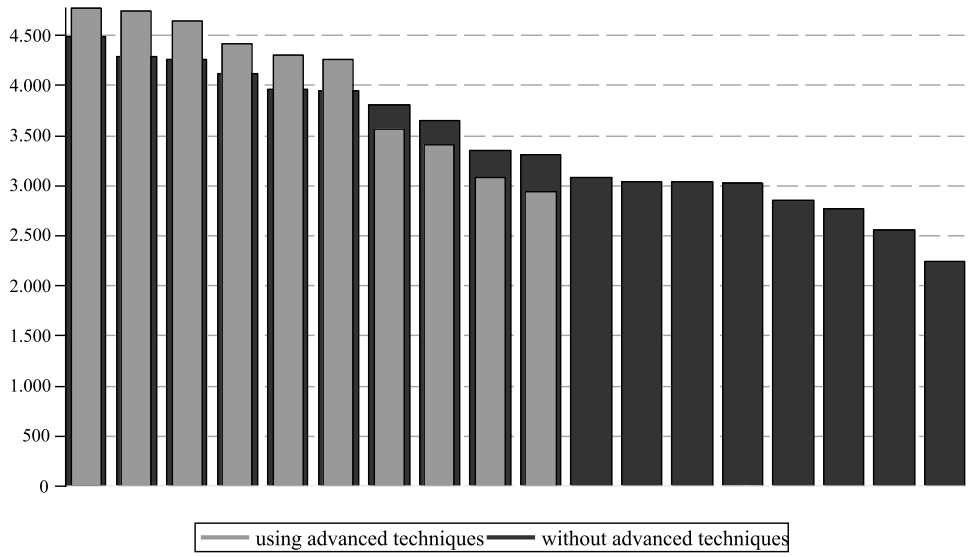
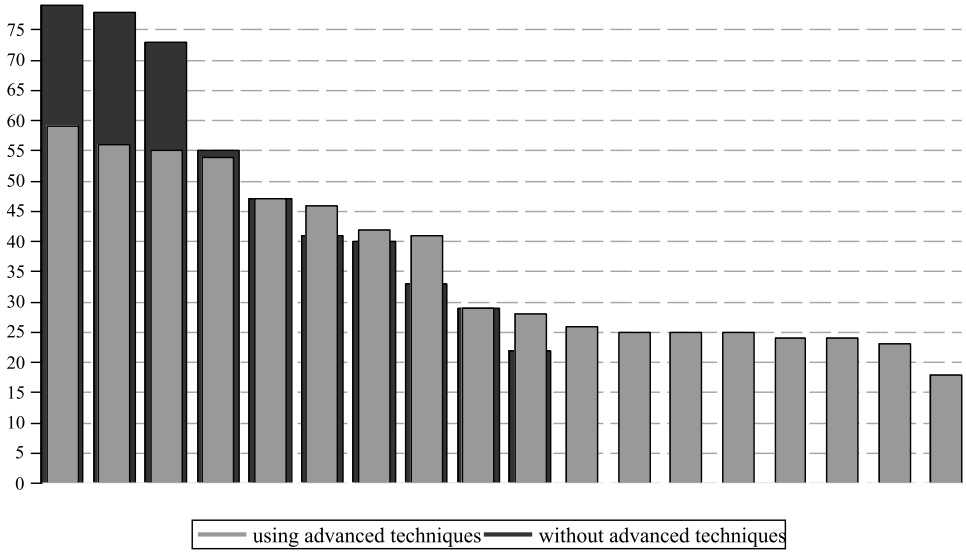


Figure 4.7.: Business value results from both travel scenarios



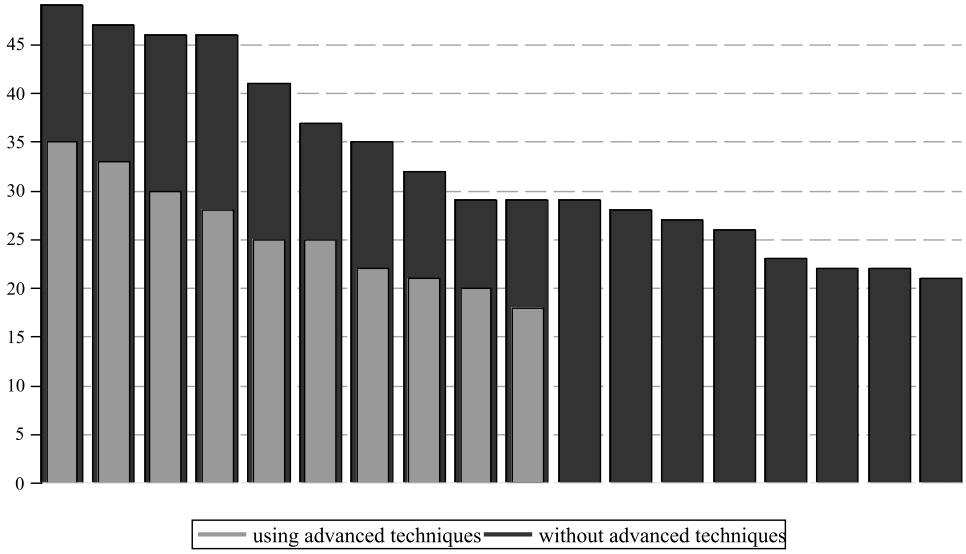


Figure 4.8.: Project plan adaptation results from both travel scenarios

Applied test	Test value	Threshold	Condition	Result
K/S-Test	$K_{BV^{T_C}} = 0.081$	$K_0 = 0.257$	$0.081 \leq 0.257$	**accept**
	$K_{BV^{T_A}} = 0.144$	$K_0 = 0.257$	$0.144 \leq 0.257$	**accept**
	$K_{CF^{T_C}} = 0.182$	$K_0 = 0.257$	$0.182 \leq 0.257$	**accept**
	$K_{CF^{T_A}} = 0.163$	$K_0 = 0.257$	$0.163 \leq 0.257$	**accept**
F-Test	$F_{BV^{T_C}} = 1.413$	$F_{0,C} = 3.722$	$1.413 \leq 3.722$	**accept**
	$F_{BV^{T_A}} = 1.083$	$F_{0,A} = 2.985$	$1.083 \leq 2.985$	**accept**
	$F_{CF^{T_C}} = 2.148$	$F_{0,C} = 3.722$	$2.148 \leq 3.722$	**accept**
	$F_{CF^{T_A}} = 2.886$	$F_{0,A} = 2.985$	$2.886 \leq 2.985$	**accept**
t-Test	$T_{BV^{T_C}} = 2.427$	$T_0 = 2.056$	$2.427 \leq 2.056$	**reject**
	$T_{BV^{T_A}} = 2.270$	$T_0 = 2.056$	$2.270 \leq 2.056$	**reject**
	$T_{CF^{T_C}} = 2.195$	$T_0 = 2.056$	$2.195 \leq 2.056$	**reject**
	$T_{CF^{T_A}} = 2.204$	$T_0 = 2.056$	$2.204 \leq 2.056$	**reject**

Table 4.2.: Results of experiment analysis' test series

sequence booking operations, etc.) as well as modifications of the plan (unbooked action cancellations, shifting of actions, etc.) and cancellations of booked actions during the journey. At this, all three addends are weighted with w_1, w_2, w_3 fulfilling the condition $w_1 < w_2 < w_3$. This function produces a sensible measurand for plan adjustments because it takes their temporal and financial impacts on the overall journey into consideration.

We can now start our test series. In doing so, we distinguish between **four data sets**: Business values $BV_i^{T_C,*}, BV_i^{T_A,*}$ and project plan adaptation frequencies $CF_i^{T_C,*}, CF_i^{T_A,*}$ of both the Californian and the Alaskan travel scenario. First, we conduct the **Kolmogoroff/Smirnov Test** as well as the **F-Test** to check whether data satisfies preconditions of the t-Test. Their results are listed in table 4.2. In all four cases, the K/S-Test accepts the null hypothesis which states that there is no statistical evidence (95% certainty) against the assumption that the data emanates from a normal distribution. Considering the results of the F-Test, we also get four positive results. In the case of the F-Test, this means that again there is no statistical evidence (95% certainty) that the compared samples' variances differ significantly from each other.

This information legitimates the use of the **t-Test** which we apply next to the experiment's data. As table 4.2 depicts, **all four test runs reject their null hypotheses** with 95% certainty. As a consequence, we can **reject the experiment's two main hypotheses H_0^{BV} and H_0^{CF}** as well.

$$reject\left(H_0^{BV^{T_C}}\right) \wedge reject\left(H_0^{BV^{T_A}}\right) \Longrightarrow reject\left(H_0^{BV}\right) \tag{4.14}$$

$$reject\left(H_0^{CF^{T_C}}\right) \wedge reject\left(H_0^{CF^{T_A}}\right) \Longrightarrow reject\left(H_0^{CF}\right) \tag{4.15}$$

To represent variance, mean value and quantiles of the experiment's data graphically, two box-whisker plots are created as depicted in figure 4.9 and 4.10. In a box plot, a box indicates the interquartile range IQR which is bounded by the 25^{th} and the 75^{th} percentile. The vertical line in its middle is the sample's median whereas the dot stands for the sample's mean value. The lines at both ends of the box are called whiskers and end at the sample's minimum and maximum respectively when there are no outliers. Outliers are values which are more than $1.5 \cdot IQR$ away from the median and are represented by single dots.

Figure 4.9 clearly underlines the statement made in section 4.2, claiming that both game configurations are not equal. The average traveler gains 30.8% more business value in Californian than he gets in Alaska. Besides that, Californian travelers who use advanced techniques reach a 11.1% higher business value than their strict plan-driven colleagues, and considering the Alaska travel scenario, the difference is even 16.9%.

In figure 4.10, we notice that the variance of change frequencies in California is much higher than the variance in Alaska. The reason for this phenomenon is an event defined for the Californian travel scenario occurring quite often, which causes a route closure. As a consequence, many actions have to be reordered and changed to create a valid modified travel route. Generally, it can be observed that journeys making use of advanced techniques perform significantly less plan adaptations than journeys without such concepts. In the Alaskan travel scenario, this can be explained by events causing quite a lot of actions to be unavailable. In such a case, if the event affects an action inside a package, a traveler who

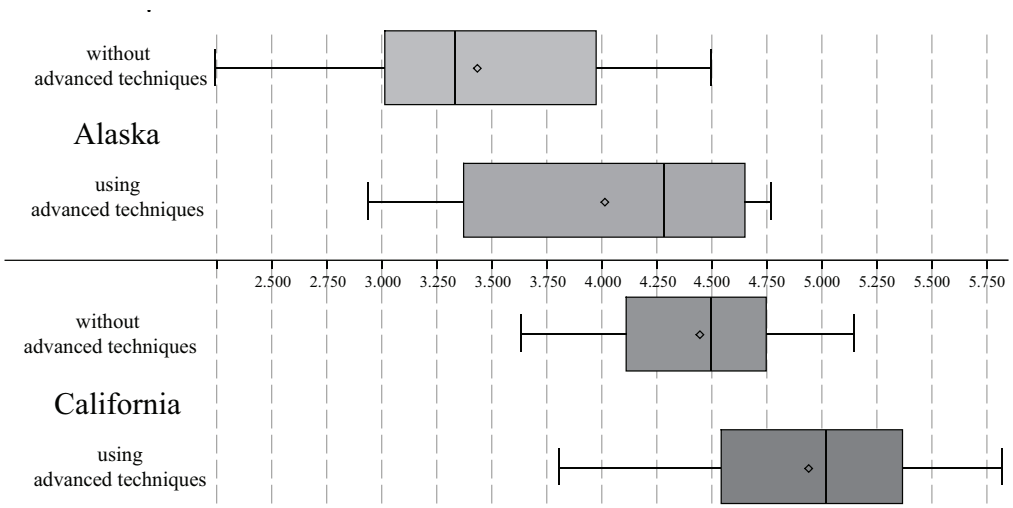


Figure 4.9.: Box-Whisker plot of gained business values in both travel scenarios

applied the advanced approach can simply change the package sequence whereas under certain circumstances, the strict plan-driven traveler has to perform several adaptation steps depending on the significance of the unavailable action. In numbers, strict plan-driven travelers need 27.3% more changes in Alaska and 38.3% more changes in California than travelers making use of tour packages .

4.6. Risk Analysis and Mitigation

This section discusses risks threatening the validity of our results and countermeasures we took.

Validity of measurements is an important concept, independent from whether it is in a testing situation or in an experimental situation [CS63]. Focusing on the latter, validity is related to the control of secondary variables. Such secondary variables can have a significant influence on the measured parameters and may mitigate the assumption that response variables are primarily dependent on the independent variable. In the worst case, drawn conclusions become invalid. In experimental situations, one distinguishes between **internal** and **external validity**. Internal validity is given when assumptions and claims made

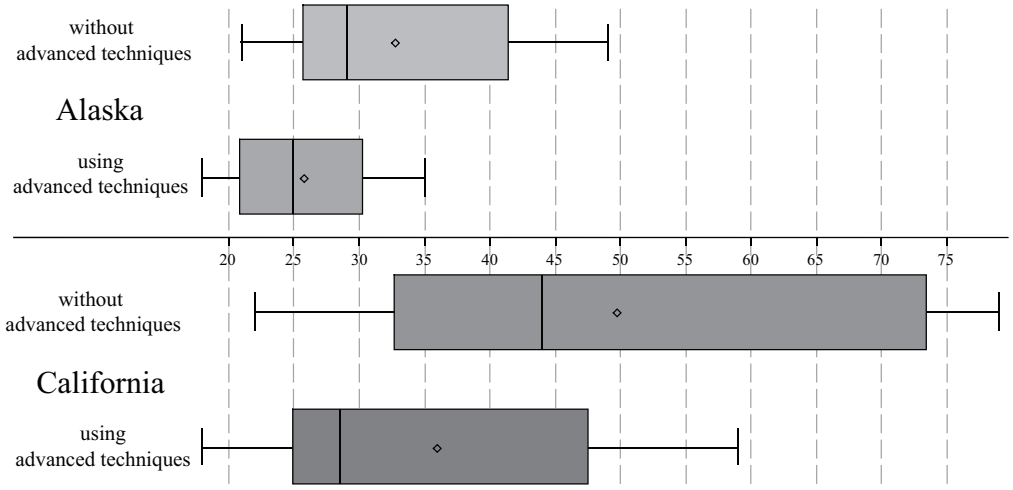


Figure 4.10.: Box-Whisker plot of project plan change frequencies in both travel scenarios

about the experiment in connection with dependencies between control and response variables are correct. On the other hand, external validity deals with the generalization of the experiment's statements and investigates the question whether implications can be established between the examined sample and the targeted population.

4.6.1. Internal Validity

In the context of this experiment, a threat to internal validity is that the planning experience of participating students may differ individually. Students who have already worked in a software development project and who have come in touch with planning activities may be able to produce better results than their colleagues with a lack of planning knowledge. Obviously, experience in the field of journey planning can also be of use. This issue can be relativized by taking the results from a sufficiently large number of students, which is guaranteed in this experiment.

Further, problems regarding the use of Alaska may occur, which as well might have an impact on the results. As a countermeasure, we prepend a short learning and familiarization phase to the actual experiment phase during which the

subjects study the program's documentation, are introduced to its functionality by a screencast specific for the particular approach and can plan a short test journey (see figure 4.5 on page 82). This should prevent the subjects from misunderstanding functionality and should give them deeper insights into a simulated journey's devolution. Furthermore, useful tooltips are available during runtime as well as Alaska's extensive help system.

Another issue which may violate internal validity is the meaningfulness of provided test scenarios. A possible problem might be that routes cannot be taken because they simply last too long or negative events occur too frequently prohibiting the traveler from finishing his journey. As a countermeasure, the used scenarios have been carefully designed and tested by the authors as well as test persons to minimize this kind of threat.

4.6.2. External Validity

Considering external threats, it has to be taken into consideration that the participants are students and not professionals having several years of experience in the field of project management. However, we can weaken this argument by referring to other works stating that results of student experiments are transferable and can provide valuable insights into an analyzed problem domain [Hou99],[Run03].

Second, one might put a journey as meaningful paradigm for a software development project into question. Because of several parallels already discussed in chapter 2 and the fact that a journey's devolution indeed behaves quite similar to that of a software project with regard to limiting factors and unforeseen incidents, we think that this metaphor is legitimate. Furthermore, to mitigate the threat that a game configuration is examined to which no appropriate counterpart in the software project world can be found, we decided to use two different configurations, as discussed in section 4.2 on page 77.

In addition to that, we can also mitigate the concern that our journeys are not generalizable. The main arguments against it are the high degree of dissimilarity of both game configurations as well as the strong correlation of their results.

4.7. Discussion

Our results indicate that **the use of travel packages enable travelers to reach a higher overall business value outcome**. We think that the reason for this phenomenon lies in the increase of flexibility caused by the use of packages. They allow the traveler to react to bad weather conditions and unforeseen events. Especially in the Alaskan travel scenario, these decision deferring techniques proved to be advantageous.

One main reason for this can be found in Alaska's weather characteristics. At most locations, the weather behaves very unstable and hardly any trend is cognizable. This causes the weather to be very hard to forecast and consequentially, close estimations about it are possible not until the day before or even not until the same day. As a result, the standard deviation of possible business values of heavily weather dependent actions is high which prohibits the traveler to fully rely on the calculated expected business values. Considering the situation in California, the player is not confronted with that unstable weather conditions and gets more significant weather trend information.

A second, even more profound reason, is the fact that Alaska's configuration contains five events of which two can cause conflicts in the travel plan. Furthermore, they can occur after each execution step with a probability of 5%. At this, these two events only affect activities on which no other actions' executability depends. This makes such actions dropable or replaceable without further having to worry about the travel plan's consistency as a whole. In contrast to that, though the Californian travel scenario has a lower number of events (four), one of them turns out to be severe. A route can get closed with a probability of 10% per execution step which results in most cases in a cascade of drop operations because subsequent actions cannot be reached on time due to the unavailability of short alternative routes. In such a situation, even the use of decision deferring packages cannot satisfyingly compensate conflicts because their duration is limited in this scenario to one day being too short to comprehend an alternative route. Furthermore, all packages have equal start and end locations also prohibiting their adoption in such a case.

Besides an increase of journeys' business value, **packages cause a reduction of the number of project plan adaptations**. This can again be explained by referring to the advantages of flexibility that comes along with the adoption of decision deferring mechanisms. Comparing both travel scenarios, we expected that the difference would be bigger in Alaska than in California. After scanning the data samples, we detected that more subjects using the strict plan-driven method took the critical route than those who made use of packages. Precisely speaking, during 28 games, this negative event occured 23 times. In addition to that, in 17 games the route was chosen at which 7 times strict plan-driven players got stuck wheres other subjects were only confronted with this problem 3 times. The impact of the previously mentioned event on the results may also explain their relatively high variance in California.

To conclude the discussion, we summarize the main results: The use of design decision deferring techniques in scenarios which are characterized by **incomplete knowledge** and the **presence of uncertainty**

$\Longrightarrow$ **does** have an impact on the overall success of a journey (software project) measured in positive travel experience of a traveler (business value of an implemented system). The higher the degree of uncertainty is, the more beneficial Late Binding and Late Modeling mechanisms turn out to be.

$\Longrightarrow$ **does** have an impact on the absolute frequency of travel plan adaptations (project plan changes and corrections). Again, the higher the degree of uncertainty is, the more plan adaptations are needed in strict plan-driven methodologies. Late Binding as well as Late Modeling techniques can mitigate unforeseen environmental changes and as a consequence lower the number of required plan adjustments.

Due to many parallels between journeys and software projects which were discussed in chapter 2, we are convinced that the results listed above can also be applied to the management of software projects. We think that decisions on e. g. a feature's implementation or on the use of a certain technology should be deferred if a certain level of uncertainty is in play. The more uncertainty exists and the less information is available, the more should a far-sighted manager put

a focus on flexibility and the disposability of alternatives. As our experiment attests, such practices can enhance a software projects success and at the same time, they reduce the number of steps needed to adjust a corrupted project plan.

Chapter 5.

Summary

This thesis investigates the impact of the adoption of design decision deferring techniques in plan-driven software projects. More specifically, we inspect deviations in a project's overall business value which is defined as the cumulative sum of implemented features' values for the customer. Besides that, we investigate how the absolute frequency of project plan adjustments is influenced by the use of previously mentioned techniques.

We investigate whether the adoption of such techniques generally results in a higher project's business value due to an increase of flexibility. Furthermore, considering the number of project plan adaptations, we think that the use of design decision deferring techniques allows the reduction of project plan adaptations at which we again refer to flexibility coming along with such mechanisms as a main reason.

While looking for sources of statistical material to verify our thesis, we soon realize that we cannot investigate real world software projects for several reasons. First of all, software projects in general last too long which would drastically exceed this master thesis' time frame. Another argument against inspecting software projects is the fact that no company develops a system twice which prohibits us from collecting complementary data samples.

As a consequence, we decided to consider journeys instead of software projects because we think that the planning behavior of these tasks is very similar. As discussed in detail in chapter 2, both scenarios demand for wise and foresighted

planning behavior and both quite regularly have to struggle with unforeseen events and insufficient knowledge about environmental parameters.

In a next step, *Alaska* ist developed, a travel simulator allowing subjects to plan a virtual journey in different scenarios where they are confronted with similar problems as those mentioned before. Design decision deferring techniques are implemented in the form of Late Binding and Late Modeling packages which represent ideas like interfacing and polymorphism in Object Oriented systems as well as approaches in Workflow Management systems. For generalization purposes, we defined two unequal scenarios – two journeys to California and Alaska respectively. The former does not allow precise duration estimations of tasks and contains a severe event forcing the traveler to make many adjustment steps whereas the latter is characterized by very unstable and unpredictable weather conditions.

We performed the experiment during three courses at the University with subjects being Bachelor students of Computer Science. The obtained results confirm our assumptions and underline the benefits of adopting design decision deferring techniques. Based on this, we are convinced that not only travelers can benefit from such methodologies but also software project managers as explained at the beginning of this section. *The more uncertainty there is in software projects, the more flexibility is demanded of the project's plan* at which previously mentioned techniques provide one alternative.

Appendix A.

Data of the Experiment

Business Value				Change Frequency			
$BV_i^{T_C,+}$	$BV_i^{T_C,-}$	$BV_i^{T_A,+}$	$BV_i^{T_A,-}$	$CF_i^{T_C,+}$	$CF_i^{T_C,-}$	$CF_i^{T_A,+}$	$CF_i^{T_A,-}$
5304.1	4530.6	4411.7	2553.9	25	79	25	37
5094.8	5012.7	3565.4	2772.2	42	73	20	35
4525.8	4458.7	4766.1	3944.1	46	33	28	46
3807.1	4292.8	4740.0	3036.2	55	41	35	46
4795.1	4724.8	3403.5	3078.2	23	29	30	47
5482.1	5147.4	4307.3	3804.6	54	78	33	32
4942.8	4136.5	2937.1	4113.0	56	55	21	28
4091.3	3843.4	4645.7	3358.3	41	40	25	49
4664.6	3629.1	3078.2	4265.8	29	22	18	29
5359.5	4657.7	4264.4	4492.6	24	47	22	41
4305.8			3963.3	28			23
5455.7			3650.6	25			29
5820.3			3310.1	18			22
5631.9			2849.2	25			22
5329.7			4288.3	47			21
5154.7			3029.5	26			27
4549.7			3032.9	24			29
4550.0			2244.3	59			26

List of Figures

List of Tables

Bibliography

[ABCP02] Steve Adolph, Paul Bramble, Alistair Cockburn, and Andy Pols. *Patterns for Effective Use Cases*. Addison Wesley Professional, 2002.

[Bar07] Liz Barnett. Agile Survey Results: Widespread Adoption, Emphasis on Productivity and Quality. *Agile Journal*, 8:17–24, 2007.

[Bec00] Kent Beck. *eXtreme Programming Explained*. Addison-Wesley Longman, 2000.

[BF00] Kent Beck and Martin Fowler. *Planning eXtreme Programming*. Addison Wesley Longman, 2000.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Innovative Technology for Computing Professionals*, 21:61–72, 1988.

[Bor77] Jürgen Bortz. *Statistik für Sozialwissenschaftler*. Springer Berlin, 1977.

[Bro90] Krishan D. Broota. *Experimental Design in Behavioural Research*. John Wiley & Sons, 1990.

[BS02] Kurt Bittner and Ian Spence. *Use Case Modeling*. Addison Wesley Longman, 2002.

[BT03] Barry W. Boehm and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley Professional, 2003.

[Coc04] Alistair Cockburn. *Crystal Clear – A Human-powered Methodology for Small Teams*. Addison Wesley Professional, 2004.

[Coh06] Mike Cohn. *Agile Estimating and Planning*. Prentice Hall Professional, 2006.

[CS63] D.T. Campbell and J.C. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Co., 1963.

[CT06] Yinong Chen and Weitek Tsai. *Introduction to Programming Languages: Principles, C, C++, Scheme and Prolog*. Kendall/Hunt Publishing Company, 2006.

[CV98] Gerry Coleman and Renaat Verbruggen. A quality software process for rapid application development. *Software Quality Journal*, 7:107–122, 1998.

[Dau07] Berthold Daum. *Rich Client Entwicklung mit Eclipse 3.3 – Anwendungen entwickeln mit Eclipse RCP, SWT, Forms, GEF, BIRT, JPA*. dpunkt.verlag, 2007.

[deM86] Tom deMarco. *Controlling Software Projects: Management, Measurement and Estimates*. Prentice Hall, 1986.

[DFK+04] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse*. Addison Wesley Professional, 2004.

[dG06] Szabolcs Mattias de Gyurky. *The Cognitive Dynamics of Computer Science – Cost-effective, Large Scale Software Development*. Wiley & Sons, 2006.

[Dij72] Edsger Wybe Dijkstra. The humble programmer. *ACM Turing Award Lecture*, EDW340:859–866, 10 1972.

[DK02] Sarv Devaraj and Rajiv Kohli. *The IT Payoff – Measuring the Business Value of Information Technology Investments*. Prentice Hall Financial Times, 2002.

[dL03] Tom deMarco and Timothy Lister. *Waltzing With Bears: Managing Risk on Software Projects*. Dorset House Publishing Company, 2003.

[dR05] Jim des Rivières. API first. `http://www.eclipse.org/eclipse/development/apis/API-First.pdf`, 2005.

[FB07] Bjorn Freeman-Benson. Eclipse Development Process. `http://www.eclipse.org/projects/dev_process/development_process.php`, 2007.

[FP97] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Thomson Computer Press, 2 edition, 1997.

[GHJ94] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[Hes08] Wolfgang Hesse. *Das V-Modell XT*. eXamen.press. Springer Berlin Heidelberg, March 2008.

[HHJHS97] J. Hagemeyer, T. Herrmann, K. Just-Hahn, and R. Striemer. Flexibilität bei Workflow-Management-Systemen. In *Software-Ergonomie '97 Usability Engineering: Integration von Mensch-Computer-Interaktion und Software-Entwicklung*, pages 179–190, 1997.

[Hog06] Robert Hogg. *Probability and Statistical Inference*. Pearson, 2006.

[Hou99] F. Houdek. *Empirical-based Quality Improvement: Systematic Use of external Experiments in Software Engineering*. Logos-Verlag, 1999.

[Hum89] Watts S. Humphrey. *Managing the Software Process*. Addison-Wesley Professional, 1989.

[Hum94] Watts Humphrey. *A Discipline for Software Engineering*. Addison Wesley Professional, 1994.

[Hum99] Watts Humphrey. *Introduction to the Team Software Process*. Addison Wesley Professional, 1999.

[Jal02] Pankaj Jalote. *Software Project Management in Practice*. Addison Wesley Professional, 2002.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley Professional, 1999.

[Jon91] Capers Jones. *Applied Software Measurement*. McGraw-Hill, 1991.

[Kan84] Noriaki Kano. Attractive quality and must-be quality. *The Journal of the Japanese Society for Quality Control*, 1:39–48, 1984.

[KPP+02] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[McC96] Steve McConnell. *Rapid Development: Taming Wild Software Schedule*. Microsoft Press, 1996.

[McC04] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[MDG+04] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Phillipe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, 2004.

[MSWW03] Michele Marchesi, Giancarlo Succi, James Donovan Wells, and Laurie Williams. *eXtreme Programming Perspectives*. Addison Wesley Professional, 2003.

[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.

[NW04] Steve Northover and Mike Wilson. *SWT – The Standard Widget Toolkit*. Addison Wesley Longman, 2004.

[Par71] David Lorge Parnas. Information Distribution Aspects of Design Methodology. In *IFIP Congress '71*, pages 339–344, 1971.

[PC86] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[PKB05] Manfred Precht, Roland Kraft, and Martin Bachmaier. *Angewandte Statistik*. 7. Oldenbourg, 2005.

[PP06] Mary Poppendieck and Tom Poppendieck. *Implementing Lean Software Development*. Addison Wesley Longman, 2006.

[Pry02] Kristi Pryma. Agile provides middle ground. *IT World Canada*, 5:15–19, May 2002.

[PSS81] Alan J. Perlis, Frederick Sayward, and Mary Shaw. *Software Metrics: An Analysis and Evaluation*. MIT Press, 1981.

[RMHL91] Hans-Christian Reichel, Robert Müller, Günter Hanisch, and Josef Laub. *Lehrbuch der Mathematik 8*. öbv&hpt, 1991.

[Roy70] Winston W. Royce. Managing the Development of Large Software Systems. *Institute of Electrical and Electronics Engineers*, 8:1–9, 1970.

[Run03] Per Runeson. Using Students as Experiment Subjects: An Analysis on Graduate and Freshmen Student Data. In *Proceedings 7th*

International Conference on Empirical Assessment & Evaluation in Software Engineering, pages 95–102, 2003.

[Sch04] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004.

[SG05] Andrew Stellman and Jennifer Greene. *Applied Software Project Management*. O'Reilly, 2005.

[She00] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2000.

[Ste99] James Stevens. *Intermediate Statistics: A Modern Approach*. Lawrence Erlbaum Associates Inc., 1999.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*, volume 3. Addison Wesley Professional, 2000.

[Tro00] William M.K. Trochim. *The Research Methods Knowledge Base*. Atomic Dog Publishing, 2 edition, 2000.

[Ull07] Christian Ullenboom. *Java ist auch eine Insel – Programmieren mit der Java Standard Edition*, volume 6. Galileo Press, 2007.

[vdAvDG+07] W.M.P van der Aalst, B.F. van Dongen, C.W. Günther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. Process Mining with ProM. In *Proceedings of the 19th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC)*, 2007.

[Zug08] Stefan Zugal. *Agile versus Plan-driven Approaches to Planning – Results from a Controlled Experiment*. Master Thesis, Department of Computer Science, University of Innsbruck, 2008.