Leopold-Franzens-University Innsbruck

Institute of Computer Science
Research Group Quality Engineering

MASTER THESIS

Handling Uncertainty in Software Projects A Controlled Experiment

Jakob Pinggera

Supervisor: Dr. Barbara Weber

Innsbruck, April 7, 2009

"To be uncertain is to be uncomfortable,
but to be certain is to be ridiculous."

Chinese proverb

Abstract

Handling uncertainty is essential for the success of software development projects. Literature suggest that especially agile approaches to software development are well suited for highly volatile environments, as they provide means for dealing with uncertainty. Unfortunately, empirical evidence investigating the influence of unforeseen changes on the success of software development projects, is still rare. This thesis thesis picks up this demand and investigates the influence of unforeseen events on software development projects. As the comparison of real software projects is rather time and cost intensive a simulator deploying a journey as metaphor for software development projects has been implemented, which was utilized by students for planning journeys as part of a controlled experiment. The gathered data clearly indicates that the outcome of software development projects, measured as business value, is negatively affected by unforeseen changes. In turn, the question whether events have a negative influence on the overall success of a journey (measured as the implementation of all essential features) remained inconclusive due to the limited number of participants.

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 me with tons of inspirations and feature requests, and gave me helpful feedback with regard to functionality and usability of the Alaska Simulator. 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 **DI. Stefan Zugal** and **DI. Michael Schier** for providing me with helpful advice and always being available for conducting inspiring discussions which were essential for the fast and successful extensions implemented in the Alaska Simulator.

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

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

Date

Signature (Jakob Pinggera)

Contents

Chapter 1

Introduction

The discipline of software engineering emerged in the 1940s, when the main challenge programmers were facing was scarce hardware resources. Making the devised code runnable on available computers was an elementary task of every programmer by that time. The assumption that once more powerful machines are available, programming would no longer be a problem [Dij72], was rather common at that time. Tough more and more powerful machines became available and computing power became cheaper than ever before the problem was not solved at all, as the demand for more complex programs increased dramatically. For the first time software development was more expensive than the hardware necessary to run programs.

> *"The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he had to dream about them and, even worse, he had to transform such dreams into reality! Is it a wonder that we found ourselves in a software crisis?"*
>
> *Edsger W. Dijkstra [Dij72]*

At the *NATO Software Engineering Conference* in Garmisch-Partenkirchen in 1968 this issue of systematic software development was addressed and the term *Software Engineering* was coined [NR69]. The urgent need for profound development methodologies is furthermore underlined by a quote from Ronald Graham who participated at the conference: *"We build systems like the Wright*

brothers built airplanes - build the whole thing, push it off the cliff, let it crash, and start over again." [NR69]

During the following years several approaches for systematic software development have been proposed that can be categorized in *plan-driven* (e.g., Waterfall Model [Roy70], V-Model [Hes08] or the Spiral Model [Boe88]) and *agile* (e.g., eXtreme Programming [Bec99], Scrum [Sch04] or Lean Software development [PP06]) process models.

One important aspect of the various software development approaches is how *uncertainty* is addressed in order to ensure the successful completion of the project. Todays market demands for high quality products, being delivered in time for a prize known before starting the project. These demands conflict with software development projects as several aspects are unknown in the initial phases of the development process making it difficult to create accurate estimates. Barry Boehm first stated that uncertainty reduces during the development of the application as more information becomes available [Boe81]. Consequently, plans tend to be more inaccurate the earlier they are developed in the software development project's life cycle [Boe81].

> *"Agile Planning shifts the emphasis from the plan to planning."*
>
> *Mike Cohn [Coh06]*

Agile approaches incorporate uncertainty handling in the development process by creating a coarse initial plan which is subsequently refined when more information becomes available. As uncertainty decreases with the project progressing decision making is deferred to later stages of the development process. Consequently, planning can be seen as an activity, rather than an artifact describing the project like in plan-driven approaches [Coh06]. Therefore, agile approaches are expected to deal well with uncertainty.

1.1 Research Objectives

Changing requirements, incomplete understanding of the product's domain and business opportunities provide several sources of uncertainty within software de-

velopment projects. Nevertheless, empirical evidence investigating the applicability of how uncertainty is addressed in the various planning approaches is still scarce. This thesis investigates the influence of unforeseen changes on the success of agile software development projects. For this purpose, an application deploying a journey as a metaphor for software development projects was implemented, which enabled us to conduct experiments investigating the planning behavior of students. The conducted experiment focuses especially on the influence of unforeseen events on the planning behavior in an agile environment. For evaluating the influence of unforeseen changes, the gained *business value* as well as the *overall success* is taken into account.

The goal of this thesis is to investigate the influence of unforeseen changes on software development projects, measured in terms of gained **business value** and **successful completion** of the project.

1.2 Research Method

The research method used in this experiment is based on literature providing guidelines for setting up and conducting experiments [FP97, Bro90, KPP+02]. The aim of the conducted experiment is the investigation of the influence of unforeseen events on the success and outcome of software development projects. For this purpose the metaphor of travelling was used, as several similarities can be identified when comparing journeys to software development projects (cf. Chapter 2). The applied research method is depicted in Figure 1.1.

1. **Implementation.** As this thesis is based on the work done in two previous master theses [Sch08, Zug08], the implementation was based on their previous developments. Several extensions were included in the *Alaska Simulator* in order to provide the functionalities necessary for conducting the experiment described in this thesis.

2. **Experiment.** To retrieve the data needed for the evaluating the impact of unforeseen events on software development projects an experiment was

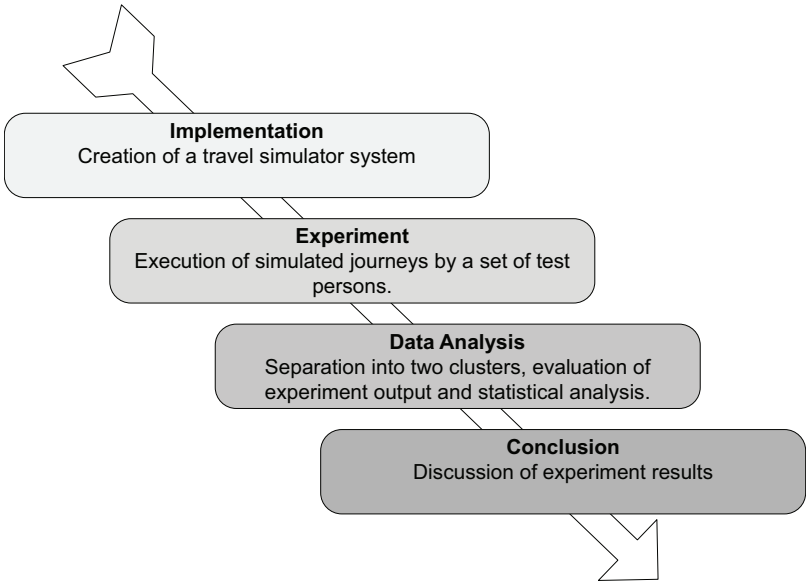

Figure 1.1: Visualization of the Applied Research Method [Sch08]

conducted at the Management Center Innsbruck. Students were asked to plan journeys to different locations with varying levels of unforeseen events (cf. Chapter 4).

3. **Data Analysis.** This step focuses on the validation and analysis of the data obtained in the experiment. For inferring conclusions profound statistical methods are utilized (cf. Section 4.4).

4. **Conclusion.** Finally, the experiment's results are discussed and their applicability for software development projects is investigated (cf. Section 4.6). Furthermore, future work based on findings of this experiment is identified.

1.3 Related Work

Empirical studies can be interpreted as simple tests comparing expectations to the experimenter's observations [EPV00]. This practice is widely used in other

research disciplines, but underused in the context of software engineering. Consequently, software engineering researchers tend to not validate their research by conducting appropriate experiments [ZW97]. This might be due the individualities of software development projects, which make the comparison of different projects difficult. Consequently, creating adequate experiment designs remains a challenging activity [Bas96].

In previous master theses similar experiments have been conducted investigating the differences of plan-driven and agile planning approaches [Zug08] and the influence of decision deferring techniques in plan-driven planning approaches [Sch08]. The results obtained in [Zug08] indicate that agile approaches are potentially superior but require well trained personal in order to bring them to their full potential. In contrary, the experiment conducted in this thesis focuses on the influence of run-time changes on agile planning approaches.

Another experiment making use of the Alaska Simulator investigating the usability of declarative process-aware information systems (PAIS) is described in [WRZW09]. Users had to deal with different levels of constraints restricting their possibilities. Interestingly, the outcome was not significantly affected by a higher number of constraints [WRZW09]. Based on these findings I am conducting an empirical experiment investigating the impact of unforeseen events on the success of software development projects.

An empirical study conducted at the University of Southern California investigates the usability of *Schedule as an Independent Variable* (SAIV) [YBY+07]. This approach, based on the Spiral Model [Boe88], utilizes dropping of less important features in order to avoid missing deadlines [BPHB02]. The experiment focused on coping with the uncertainty of cost in 8 software development projects at USC [YBY+07]. The rather low number of projects investigated is due to difficulties when comparing software development projects [Bas96]. In order to tackle this issue my approach deploys a journey as a metaphor for software development projects (cf. Chapter 2) that facilitates the gathering and comparison of data (cf. Chapter 4).

Another series of experiments investigates the accuracy of effort estimates and reasons why a strong over-confidence about the estimates could be ob-

served [JTM04]. Based on the findings of the experiments a set of evidence-based guidelines about how to assess software development cost uncertainty was published [Jør05]. In contrast to my thesis the experiments focus on the accuracy of estimates and how they might be improved using feedback mechanisms [GJ08] while this thesis investigates the impact of unforeseen events on the outcome of the project.

1.4 Overview

The remainder of the thesis is structured as follows.

Chapter 2 describes the concepts important for the thesis, explaining agile principles that are especially relevant in the context of my experiment. Furthermore, the *Journey Metaphor* is introduced and the parallels to software development projects are described.

Chapter 3 contains information about the *Alaska Simulator* and other applications being part of the *Alaska Toolset*. In addition, architectural decisions, made during the development of the Alaska Simulator, are outlined and utilized third party frameworks are briefly described.

Chapter 4 explains the experiment's design and covers it's execution and the obtained results illustrated in this thesis.

Chapter 5 concludes the thesis with a summary.

Chapter 2

Concepts

This chapter describes the concepts underlying the Alaska Simulator and explains why travelling can be used as a metaphor for planning a software development project. Section 2.1 describes different ways of planning focusing especially on agile approaches and Section 2.2 covers requirement analysis and planning. Section 2.3 explains how software projects are influenced by certain restrictions. Finally, Section 2.4 investigates the danger of unforeseen events for the overall success of software development projects. In each of the sections first a general description is provided before describing its realization in the Alaska Simulator.

2.1 Planning Approaches

Planning is a crucial aspect when dealing with software development projects. On the one hand, plans are needed by the management in order to make decisions and on the other hand, plans are important to guide the development process and therefore ensure that all required functionalities are implemented [Coh06].

According to Royce the development process can be divided into the following phases [Roy70].

- **Requirement Analysis and Specification.** The customer describes the intended behavior of the application creating requirements for successfully developing the product.
- **System Design an Specification.** After gathering the requirements the design of the application is created.

- **System Implementation.** The afore created design is actually implemented in this phase.

- **Integration and System Testing.** In order to guarantee a working system the application has to be tested and integrated into the customer's working environment. Therefore, the interoperability with legacy systems has to be ensured.

- **Delivery, Deployment and Maintenance.** The product is delivered to the customer. To ensure the trouble-free usage of the application maintenance work has to be performed.

Planning is essential for the success of a software development project, but especially creating the initial plan turns out to be a challenging activity as not all information about the project is available [Coh06]. Figure 2.1 depicts the cone of uncertainty [McC98] indicating that decisions made during the life-cycle of software projects are always based on decisions made in earlier stages of the development process [Boe81]. Therefore it is *theoretically impossible* to estimate a project accurately in the early stages of the project [McC98]. In the feasibility phase of the development process the schedule estimate is typically as far off as 60% to 160%. Consequently, a project which is scheduled for 20 weeks may last only 12 weeks, but on the other hand may take as long as 32 weeks [Coh06].

2.1.1 Agile Planning

There exist several approaches dealing with the afore mentioned uncertainty. Plan-driven approaches, including Waterfall Model [Roy70] or V-Model [Hes08], may be followed which try to create a plan as detailed as possible in order to cope with uncertainty. Unfortunately, unforeseen changes might invalidate the plan created in the initial phase of the software development process after a few months [Coh06]. According to [PP06], it might be feasible to create a plan which includes all requirements of the product, but it is impossible to anticipate all possible changes up-front. Starting the development in a chaotic way without

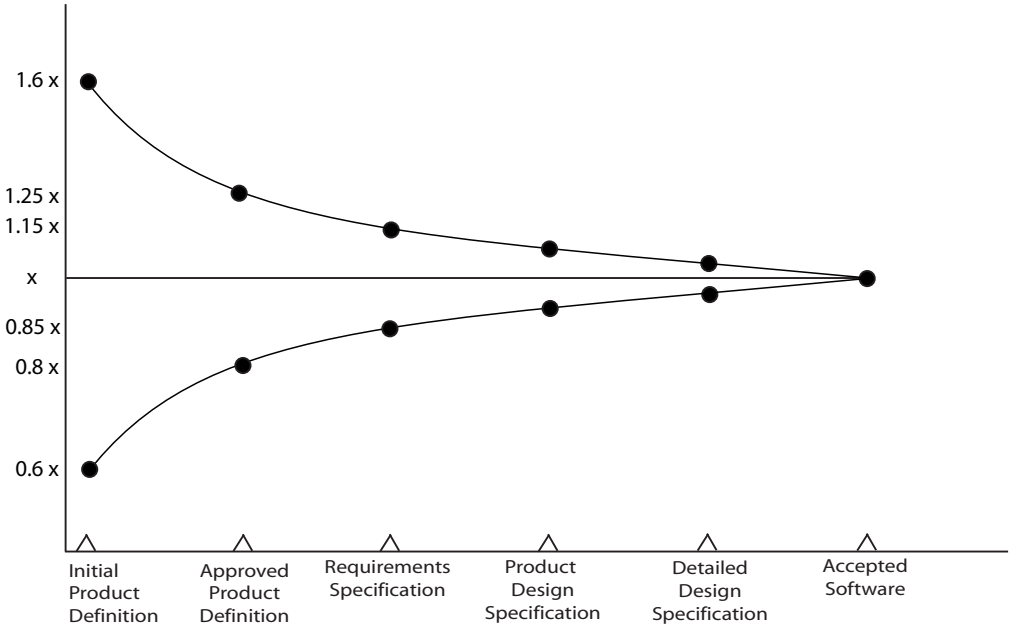

Figure 2.1: Cone of Uncertainty [Coh06]

creating any plan is certainly not an option as basic questions questions like "When will you be done?" cannot be answered [Coh06].

> *"No plan survives contact with the enemy."*
>
> *Field Marshal Helmuth Graf von Moltke*

Agile planning tries to shift the emphasis from the plan to planning. *"A plan is only a snapshot of how one believes the project may unfold in the future"* [Coh06]. For this purpose an iterative approach is used, which executes the different phases of development several times, in so called *iterations*. After ending an iteration the plan is revised and changed if necessary. It might be possible that the customer discovers that extending a feature provides more business value than implementing another one. Consequently, the plan for the next iteration is altered and some time for including the new functionalities is assigned. Other features may become obsolete or will be postponed to later iterations. Furthermore, the knowledge gained when executing the previous iteration reduces

the project's uncertainty (cf. Fig. 2.1) and therefore increases the accuracy of the plan. This additional knowledge may be on the customer's side, like in the example before, but also developers increase their knowledge about the product, technologies and legacy systems [Coh06].
After each iteration a runnable piece of software should be available to customers [Coh06]. This does not mean the system is going to be used in production, but at least the customer is able to investigate the current developments and react on possible problems by altering the plan for the next iteration accordingly.

2.1.2 Journey Phases

Like software development projects, journeys can be divided into several phases. For example, before actually starting to travel a rough plan has to be created defining locations the traveller is planning to visit including the attractions the traveller is most interested in. After actually starting the journey another phase is entered in which the initial plan is continuously refined and actions are executed.

Planning Phase

Before actually travelling to the desired destination a rough initial plan incorporating the most important actions and taking into account constraints that are limiting the users' possibilities is created. The amount of money available, the maximum duration or some interesting attractions are influencing the initial plan of the journey (cf. Section 2.3).

Travelling Phase

When arriving at the destination the travelling phase is entered making it possible to execute actions, travel along routes to different locations and stay for the night at accommodations. Furthermore, the traveller can always return to the planning phase in order to update the plan according to changing circumstances (e.g., unforeseen events). For example, it might be possible that an action becomes unavailable or that an action's duration changes (e.g., getting stuck in a traffic

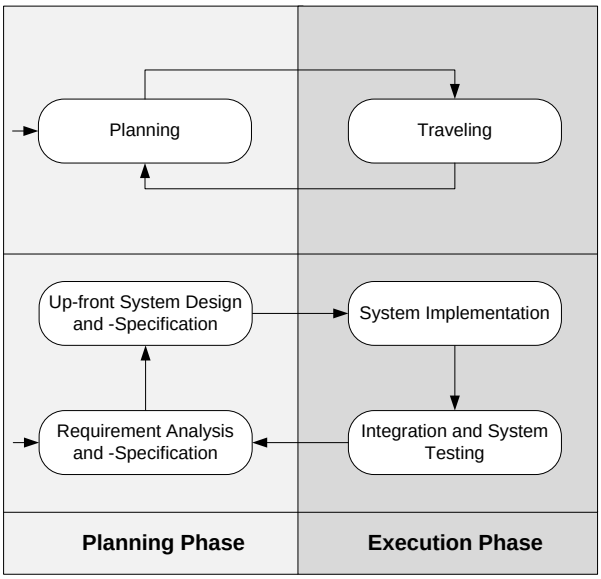

Figure 2.2: Phases in Journeys and Software Projects adapted from [Zug08]

jam) (cf. Section 2.4). Like in agile software development the project's planning and execution are closely interwoven.

Figure 2.2 illustrates how the phases in the journey metaphor can be mapped to software development. The planning phase comprises Requirements Analysis - and Specification as well as System Design, while the travelling phase maps to System Implementation and Integration and System Testing. In the context of both travelling and software projects the execution and planning phases are closely interwoven as constant planning is necessary in order to update the plan according to unforeseen events and uncertainties.

2.2 Requirement Analysis and Planning

An essential step for the success of a software project is the communication between customers and developers in order to clarify the demanded features of the application. For this purpose *User Stories* may be utilized providing a written

description of intended functionality [Coh04]. After creating the initial stories of the project *prioritization* should be performed determining the implementation order.

2.2.1 Prioritization in Software Engineering

Richard Koch stated that 80 % of the value is delivered by 20 % of the features [Koc04]. Therefore, it seems to be feasible to start with the features that are promising the highest revenue. For this purpose, the principle of prioritization was introduced which focuses on the completion of the features with the highest business value gain [PP06]. Consequently, the customer is expected to prioritize the requested functionalities in order to retrieve the important 20 % of the software product as fast as possible [Coh06]. The following factors are influencing a feature's prioritization.

Value and Knowledge

On the one hand, the value of a feature is the financial benefit a company gains by implementing the functionality. May it be by selling the product to customers or increasing the productivity of it's employees when developing in-house software. On the other hand, gain of knowledge, which could possibly speed up the further development process and reduce uncertainty, has to be taken into account when talking about the value of a feature [Coh06].

Cost

As money is always a factor when dealing with software projects the developing costs have to be considered when prioritizing features. The customer has to decide whether a feature's expected costs are are outweighing it's projected benefit. For evaluating the costs an abstract estimate, denoted as *story points*, can be used. Story points do not refer to a specific amount of time, as the actual duration for implementing a feature heavily depends on the implementation speed of the team, which is, especially at the beginning of a project, unknown. On the other hand, it has to be pointed out that the relation between the estimated times of different features has to be consistent, e.g., a story assigned 4 story points has to take twice as long as a feature assigned 2 points [Coh06].

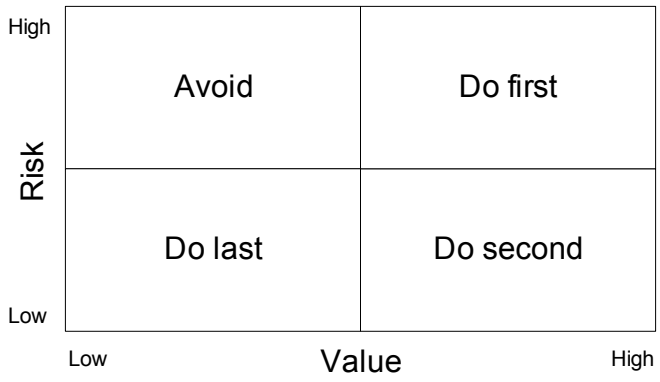

Figure 2.3: Classification in High / Low Risk and Value [Coh06]

Risk

According to [Coh06] risks can be categorized into schedule risks, cost risks and functionality risks. Basically there is a trade-off whether to start with the features endangering the overall success the most in order to reduce uncertainty (cf. Fig. 2.1), or assigning the features which promise the highest benefit to the first iteration. Figure 2.3 illustrates a strategy proposed by Mike Cohn in which features with high risk and priority should be implemented first. Second, safe features providing a high benefit should be integrated. Afterwards features which provide only limited value and do not endanger the overall success should be developed. Features that endanger the overall success and do not have any or only a low influence on the overall value of the product should not be implemented at all [Coh06].

2.2.2 Prioritization in the Journey Metaphor

In the context of travelling user stories are represented by **Activities**, **Routes** and **Accommodations**, subsequently referred to as **Actions** [Zug08]. The amount of time per journey and also per day is limited. In order to optimize the travel experience the traveller has to select the actions promising the highest benefit or being essential for the successful completion the journey (cf. Section

2.3). At the end of each day the traveller has to find an accommodation to stay for the night.

As a result, the concept of prioritization is well applicable for travelling as the traveller should strive for selecting the most promising actions out of the pool of available ones. The same influencing factors as defined in the context of software engineering [Coh06] can be identified.

Value and Knowledge

The value received when executing an action is the gained *business value*. Additionally, constraints influence the value of an action as some activities may be mandatory and have therefore to be executed (cf. Section 2.3).

- **Business Value.** When travelling decisions have to be made regarding what attractions the traveller is going to visit and what is not considered as being important enough to spend time with. The overall goal of a journey is to optimize the amount of business value obtained. Therefore, each action has a prior known maximum business value. The actually gained business value is determined when executing the action based on the weather influence and the inherent distribution. On the one hand, each action has a specific **weather influence** describing how strong the weather affects the gained business value. Some actions are rather weather independent (e.g., visiting a museum) whereas the outcome of others highly depends on the current weather situation (e.g., hiking). Furthermore, each action has an assigned **inherent distribution**, which describes the distribution of the business value without taking the weather into account (for details see [Zug08]). Additionally, it is possible that unforeseen events influence the outcome of an action. For example, spotting Grizzly Bears when travelling through Alaska might increase the business value of the respective action (cf. Section 2.4).

- **Weather.** The business value obtained when executing an action is influenced by the weather at the action's location. In the up-front planning phase the weather is unknown. After actually starting the journey a weather forecast is available for the rest of the journey. The accuracy of

the forecast is improved the closer the traveller gets to a specific date. This behavior models the process of knowledge gain during a software project. The increasing accuracy of the weather forecast can therefore be compared to the reduced uncertainty described by the cone of uncertainty (cf. Figure 2.1).

Cost

Each action has a specific prize that has to be paid immediately when the action is booked or executed. Besides, the traveller can also visit attractions that are for free (e.g., hiking).

Risk

The obtained value of an action is influenced by several factors like the current weather situation, the weather influence, the inherent distribution, cancelation fees or run-time events.

- **Reliability.** As it is rather difficult for the user to estimate the expected outcome of an action the concept of reliability was introduced. The reliability of an action takes all influencing factors, except the weather, into account and calculates a value between 0.0 and 1.0, where 1.0 indicates that the action is completely independent from the current weather situation, whereas a low reliability represents a high weather dependency. Consequently, the user can estimate the outcome of an action during run-time as it is calculated using the current weather and the reliability of the action.

- **Availability.** Furthermore, it is possible that actions are not available. For example, some attractions might be closed due to maintenance or just be overbooked. The availability of an action represents the possibility of an action being available. Similar to the reliability the availability is encoded using values from 0.0 to 1.0, where 1.0 indicates that the action can be executed for sure. Some actions can be booked, which ensures that the action can be executed. On the contrary, actions have to be paid immediately upon booking and in case of canceling booked actions the paid money is

only partly refunded depending on the time between the scheduled execution and the cancelation of the action.

The risk classification depicted in Figure 2.3 can also be applied to journeys. Mandatory actions can be interpreted as high-risk high-value actions that are essential for the success of the journey, which should therefore be appropriately secured by including buffers in the journey plan. Actions which provide a high business value, but are not essential for the overall success may be considered as low-risk high-value actions. Actions which provide a rather secure but small gain of business value should therefore be used to fill the gaps between the afore mentioned activities. High-risk low-value actions which do not offer a high business value at a rather high risk should not be executed at all.

In a nutshell several similarities can be identified between prioritization in software development projects and travel planning. The principle of prioritization is therefore well applicable for the Alaska Simulator.

2.3 Project Specific Limiting Factors and Basic Conditions

During the software development process several restrictions limiting the implementation of user stories have to be taken into account. Some of them are rather obvious like a limited budget or deadlines which reduce the number of user stories that can be implemented in the given time. Furthermore, a company may impose several restrictions which influence the development process. For example, it may not be allowed to include open source frameworks into the application, or extensive documentation and testing can be required in order to ensure a high quality level.

Additionally, some project specific restrictions may exist. For example, some user stories may require the earlier implementation of other features on which the new functionality can be based on. Besides, some features are essential for

the usability of the application, whereas other may be omitted if time or money is scarce.

2.3.1 Factors Limiting the Traveller's Possibilities

In the context of travelling also some limiting factors can be identified. Similar to a software project resources, like money and time, are restricted. Additionally, constraints may be utilized to model interrelations between actions. For example, in order to take part in the advanced rock climbing course one has to complete the beginners class first. Mandatory actions can be defined, which are essential in order to successfully complete the journey. Constraints can roughly be divided into *Execution Constraints* and *Termination Constraints*, but there are certain types which belong to both groups.

Termination Constraints

Termination Constraints can be used to define conditions, which have to be fulfilled in order to successfully complete the journey (e.g., the journey has to end at at specific location). The following list contains all types of Termination Constraints supported by the Alaska Simulator.

- **Co-Requisite Constraint.** A Co-Requisite Constraint requires the user to execute two actions along with each other. Consequently, it is not possible to execute only one of the actions during the journey. On the other hand, it does not matter how often the actions are invoked.
- **Journey Completion Day.** This constraint restricts the day when the user finishes the journey. The reason for such a constraint could be a booked flight for a specific day.
- **Journey End Location.** The journey has to be ended at a specific location, because the flight is departing there.
- **Minimum Execution Constraint.** Minimum Execution Constraints can be used to define activities, which have to be executed. For example, when travelling to San Francisco the traveller has to see the Golden Gate Bridge.

- **Response Constraint.** Response requires that every time Activity A is executed Activity B has to be executed after it [AP06]. The implementation of Response is rather relaxed, as there can be other actions between the execution of A and B. Furthermore, executing A several times followed by a single execution of B is also permitted.

Execution Constraints

Whereas Termination Constraints are responsible for prohibiting the termination of the journey, *Execution Constraints* are used to prevent the execution of actions. Therefore, whenever one tries to execute an action all execution constraints are checked and, if necessary, the execution is prohibited.

- **Budget Constraint.** The Budget Constraints restricts the amount of money available for the journey.
- **Execution Time Constraint.** Sometimes it is necessary to restrict the execution time of an action. For example, a tour starts only every two hours, or is only available on Wednesdays. Execution Time Constraints can therefore be used to restrict the starting time and/or day of actions.
- **Maximum Execution Constraint.** This constraint can be used to set an upper bound for the number of times an action can be executed.
- **Mutual Exclusion Constraint.** Mutual Exclusion Constraints prevent the execution of two actions within a journey. Depending on which action was executed first the execution of the other is prevented.
- **Prerequisite Constraint.** Prerequisite Constraints make sure that whenever action B is executed action A has to be executed previously. For example, before skiing the ticket for the ski lift has to be bought.
- **Time-limited Non Response Constraint.** After executing action A one has to wait a certain amount of time before he is able to execute action B. For instance, rock climbing the day after taking a winery tour is not

allowed.
As this constraint is not symmetric, the waiting time does not apply when executing activity A after activity B.

A-n*B-C Constraint

The A-n*B-C Constraint requires the user to execute action A first. Afterwards action B needs to be executed n times before finally invoking action C.
This constraint cannot be put into one of the categories mentioned above, as it is required to permit the execution of illegal actions during runtime and at the same time prohibiting the successful termination of the journey in case of a constraint violation.

- **Properties of n.** The number of executions of action B can be fixed number like [3], but also a range like [1..3]. Additionally, it is possible to define open ranges like [1..], which means that action B has to be executed at least once. Open ranges can also be defined the other way around like [..4], which indicates that action B can be executed at most 4 times. In the later case not executing action B is also valid.

- **Execution Day.** All actions have to be executed at the same day of the journey. Therefore, it is not possible to execute action A on day one and the other actions covered by the constraint on another day.

- **Single execution of C.** A single execution of action C, without preceding invocations of actions A and B, does not satisfy the constraint.

The Alaska Simulator supports the user when dealing with constraints by high-lightning all errors in the journey plan and providing a meaningful message indicating how the problem might be resolved.

2.4 Project Dynamics and Unforeseen Events

In order to ensure the success of software projects a deliberate dealing with risks is essential. While ignoring risks very likely leads to failure, trying to excluding all risks is neither possible nor promising [dL03]. Particularly, when dealing with highly competitive business fields and changing requirements the ability to adapt to changes is crucial for the overall success of the project. If the development process is lacking flexibility a company may not be able to react on new business opportunities and therefore loosing customers to competitors [PP06].

2.4.1 Deferring Commitment in Software Engineering

Whenever dealing with important decisions developers should strive for deferring commitment to the last responsible moment [PP06]. The cone of uncertainty depicted in Figure 2.1 indicates that knowledge increases throughout the development process and therefore reduces uncertainty. If decisions are deferred to later phases of the development process uncertainty is reduced and more accurate decisions are possible. This does not imply that all decisions should be deferred. The primary goal should be to make decisions reversible, so that they can be easily changed [PP06].

> *"In preparing for battle I always found that plans are useless, but planning is indispensable."*
>
> *Dwight D. Eisenhower*

It has to be pointed out that not all decisions can be made reversible without having additional costs. In such scenarios developers have to weigh the additional costs of making the decision reversible against the danger of changing the plan, after committing to an option, at higher costs. If it is extremely unlikely that another option is chosen it may be better to wait as long as responsibly possible and making a commitment to one of the options. On the other hand, in case of high uncertainty, it might be feasible to live with increased costs but gaining a higher degree of flexibility by including options at critical points.

The principle of deferring commitment partly conflicts with the principle of prioritization, as Figure 2.3 suggests that most critical features should be implemented first in order to reduce uncertainty. On the contrary, the principle of deferring decisions indicates that in case of uncertainty commitment should be deferred as long as responsibly possible [PP06]. The appropriate strategy for a specific feature might be discovered by investigating if uncertainty can be reduced by deferring decision to a later stage in the development process when more knowledge is available. For example, if a customer is unsure about the business value gained by a specific feature it might be wise to defer commitment as long as possible and start with other feature to build some knowledge about the application's domain.

One approach for deferring commitment is *set-based design* which is an appropriate technique for making high-impact, irreversible decisions [PP06]. Particularly, if there are fixed deadlines which cannot be moved under any circumstances, set-based design is well suited. For this purpose, several different implementations are developed and the decision which one to use is deferred as long as possible [PP06]. The following example taken from [PP06] illustrates a usage scenario for set-based design.

In order to deliver the best possible solution at fixed deadline three development teams are formed. Team A implements the simplest solution possible in order to solve the problem. Team B is assigned the task of developing a preferred solution, whereas Team C creates the optimal solution for the problem. *"When the deadline arrives, there will be a solution, and it will be the best one possible at the time.* [PP06]*"* If only Team A is done with the work use their solution, but if Team B has completed their solution already, use the better one, but only if Team C has not completed their optimal solution yet.

A suspicious mind may ask why not moving people from Team A to Team B in order to increase the chance of having the better solution ready in time. This is certainly an option, but it has to be taken into account that failing the deadline is not an option. Consequently, the resources invested into Team A, ensuring that there will be a solution, does not seem to be waste even if their solution is never used [PP06].

Due to the high amount of resources necessary for using set-based design this may only be feasible for high-impact decisions, which are even more costly to reverse.

2.4.2 Deferring Commitment in the Journey Metaphor

In context of planning a journey the principle of deferring commitment can also be applied in various situations. As long as an action is not booked or executed it is possible to reschedule the activity without having any additional costs. In case of booked actions one may reschedule the action, but has to pay cancelation fees (cf. Section 2.2.2). Therefore, booking an action can be interpreted as committing to a specific option. There are several reasons why deferring decisions can be beneficial in the context of travelling.

Weather Uncertainty

Some actions are strongly influenced by the current weather situation (cf. Section 2.2.2). Therefore, commitment should be deferred as long as responsibly possible (e.g., by not booking low availability actions before the booking deadline). Besides, flexibility can be increased by incorporating options in the journey plan (e.g., by scheduling a weather resistant action in parallel and choosing the more promising one when arriving at the location and more detailed weather information is available).

Unforeseen Events

When travelling similar situations like in software development projects can occur. For example, it might be possible that traffic jams or closed roads delay the arrival at the destination. Consequently, the traveller should be well prepared and react appropriately on the changed circumstances by canceling less important actions and rescheduling the more important ones.

Furthermore, a traveller may get to know about an attraction he was not aware of previously, which is more interesting than the other ones available at the location. As a result, the user might want to change the plan to incorporate

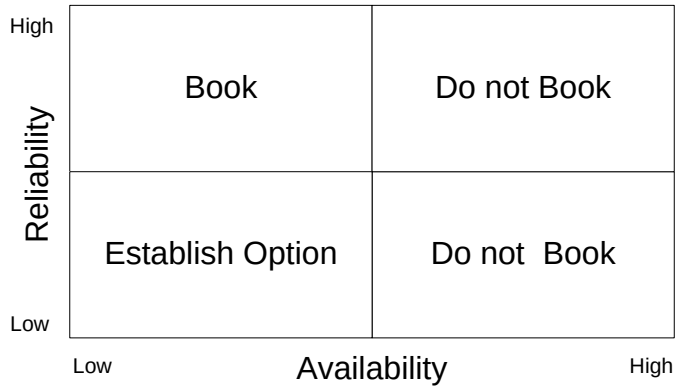

Figure 2.4: Decision Deferral Strategies

the newly available activity. Consequently, the gained knowledge (e.g., changed circumstances or new action) can be used to create a better plan for the rest of the journey without having additional costs. Committing to an option earlier might prevent the traveller from using the gained knowledge as costs for reverting the previously made decision have to be considered.

On a journey the traveller has to decide whether it is worth making a commitment by booking an action or taking a higher risk pays off by the gained flexibility. Figure 2.4 classifies actions according to their *availability* and *reliability* (cf. Section 2.2) and shows strategies on how to best deal with particular classes of actions [Zug08]. By following this strategy it is ensured that highly available or unreliable actions are not booked. Booking unreliable actions increases costs for reverting decisions as cancelation fees have to be paid if the plan needs to be changed. On the other hand, booking highly available actions introduces unnecessary early commitment while providing no valuable benefit and reducing flexibility. Consequently, highly reliable actions with low availability should be booked as it is ensured that by executing the action a high business value is generated. Nevertheless, the possibility of unforeseen events which may invalidate the plan and force the customer to cancel booked actions has to be considered. Consequently, the danger of delays due to unforeseen events should be tackled by including buffers in the journey plan.

Actions with low availability and low reliability are more difficult to handle and may therefore be addressed by using principles from Set-based design. Of course it is not possible to execute several actions in parallel, but it is still feasible to plan in parallel. If the business value that can be obtained is high enough and the action's outcome highly depends on the current weather situation it may be backed up with another action which is not influenced by the weather. Before executing the action the user can inspect the current weather and execute the more promising one for the current situation. In case of nice weather the high gain of the weather dependant action is most likely the better alternative, whereas in case of bad weather the lower, but secure business value gain of the alternative action should be the preferable choice (cf. Section 2.2).

In summary, deferring commitment is an appropriate principle for fostering the possibility of successfully completing a software project and maximizing the outcome of a journey. Consequently, this principle seems to be well applicable for the Alaska Simulator.

Chapter 3

Architecture

This chapter explains the architectural decisions made when implementing the Alaska Toolset and describes the most important third party frameworks used. Section 3.1 describes the interplay of the different applications being part of the Alaska Toolset. Subsequently, Section 3.2 focuses on the Alaska Simulator, the main application of the Alaska Toolset. Finally, Section 3.3 sketches the frameworks used and explains their interrelations.

3.1 Alaska Toolset

Subsequently, the applications being part of the *Alaska Toolset* are described and their interrelations are outlined.

- **Alaska Simulator.** The *Alaska Simulator* is the main application of the Alaska Toolset offering the possibility of conducting experiments. It has been developed by Michael Schier [Sch08], Stefan Zugal [Zug08] and considerably extended as part of this thesis.
- **Alaska Configurator.** The *Alaska Configurator* can be utilized for creating journey configurations executable with the Alaska Simulator. The development of the Alaska Configurator was funded by the e-learning initiative of the University of Innsbruck [Ele08].
- **Alaska Data Analysis.** The *Alaska Report Generator* was developed as part of a Bachelor thesis at the University of Bolzano [Tri08] which can be

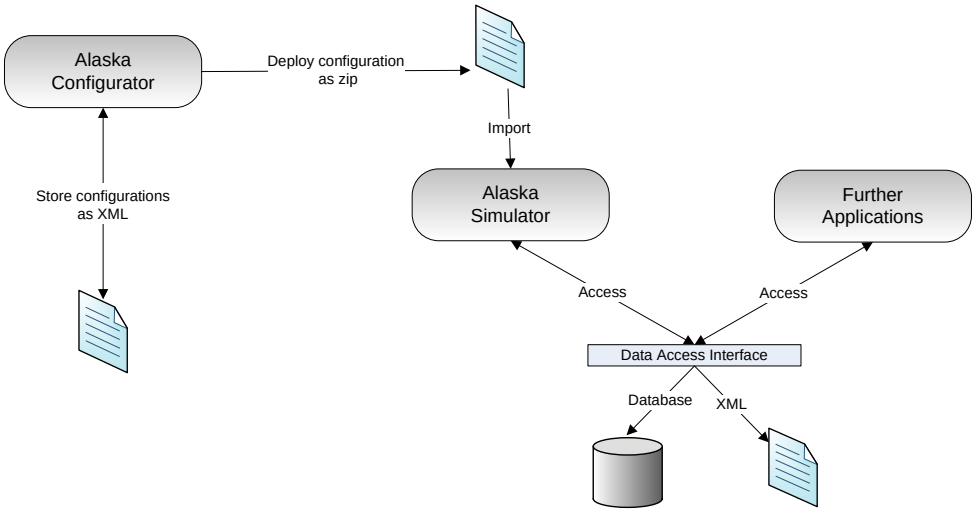

Figure 3.1: Alaska Toolset adapted from [Zug08]

used for creating reports using basic statistical metrics, e.g., mean business value and cost. More sophisticated evaluations are provided by the *Alaska Data Evaluator*, which was also funded by the the e-learning initiative of the University of Innsbruck [Ele08].

Alaska Life Cycle

Figure 3.1 describes how the different applications of the Alaska Simulator play together. The Alaska Configurator allows for the creation of journey configurations, which are persisted using XML documents. When deploying the configuration all information is put into a zip file that can be loaded by the Alaska Simulator, which enables the user to plan and execute journeys using two different planning approaches: *Plan-driven* [Sch08] and *Agile* [Zug08]. All relevant planning steps are logged to a datastore, like a MySQL database or an XML file. The logged journeys are accessed by data analysis applications like the Alaska Report Generator [Tri08] or the Alaska Data Evaluator.

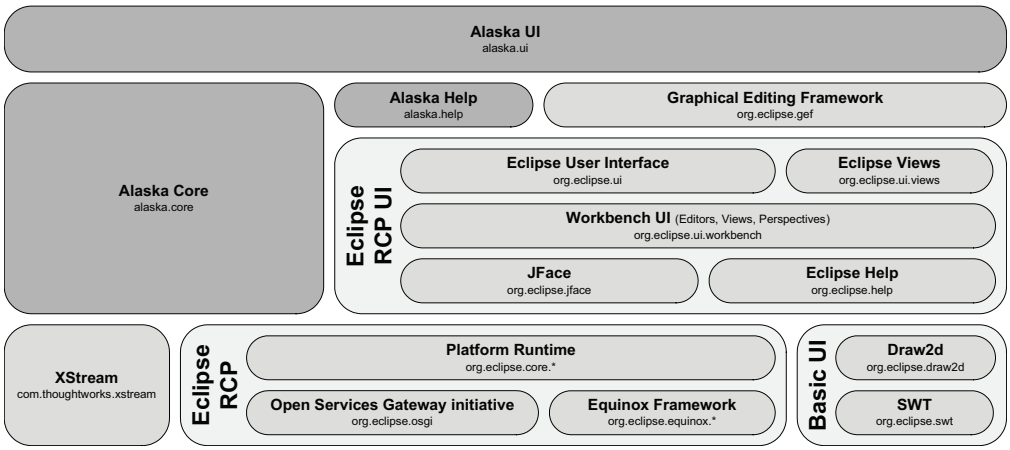

Figure 3.2: Plug-in Composition of the Alaska Simulator [Sch08]

3.2 Extensions to the Alaska Simulator

This section describes the extensions which were made to the Alaska Simulator in this thesis to be capable of conducting the desired experiment. The Alaska Simulator is based on the Eclipse Rich Client Platform (cf. Section 3.3.1) and is consequently built as a set of plug-ins, which are depicted in Figure 3.2. The plug-ins implemented as parts of various master theses are illustrated in dark grey (Alaska Core, Alaska UI and Alaska Help). Third party plug-ins that are utilized by the Alaska Simulator are shown in a slightly brighter color. The remainder of this section focuses on the Alaska Core and Alaska UI plug-in as those were the ones which had to be modified when implementing the extensions necessary for conducting the experiment.

3.2.1 Alaska Core

The *Alaska Core* plug-in contains the business logic of the application and implements the concepts elaborated in Chapter 2. Subsequently, the proxy concept, being necessary for changing actions at run-time, is explained before making use of its advantages when explaining the services included in the Alaska Simulator.

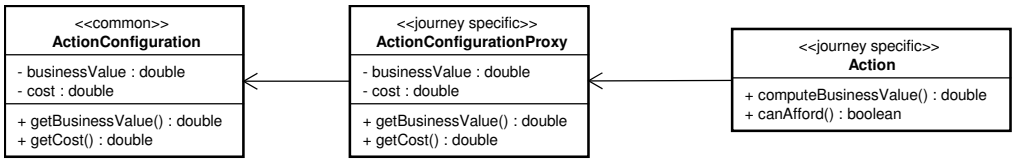

Figure 3.3: Proxy Caches a Configuration [Zug08]

Configuration Proxies

The mechanism of configuration proxies available in previous versions of the Alaska Simulator had to be extended in order to incorporate more sophisticated functionalities necessary for conducting the experiment being part of this thesis (e.g., extended event system and improved business value calculation).

A journey consists of locations, constraints, events, accommodations, routes and activities. `ActionConfiguration`'s contain information about the cost, maximum business value and availability of the associated action. All these properties can be altered by unforeseen events during journey execution. Thereby, it has to be ensured that these changes only affect one journey instance even when several journeys are opened at the same time. If this event would modify the `ActionConfiguration` directly the changes would be reflected in all journeys of the same configuration. Consequently, a mechanism providing run-time equivalents of action configurations, so called `ActionConfigurationProxy` objects, had to be introduced. Changes made by events are only stored in the proxy objects and do not affect the actual configuration. This structure can be interpreted as a slightly altered version of the *Proxy Pattern* [GHJV94] .

An action configuration proxy object acts as a template for the actual `Action` object which is inserted in the journey plan. Action objects contain information about a specific instance of the action configuration template like execution time and date or the actually generated business value. Figure 3.3 depicts the interrelations between configuration, proxy and action objects.

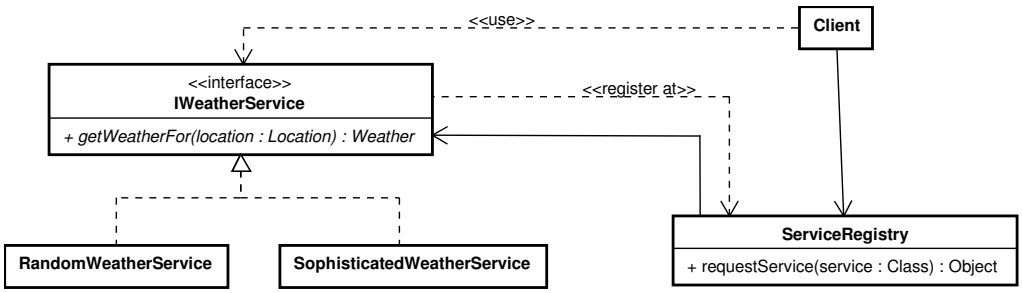

Figure 3.4: Services as Easily Exchangeable Parts of Functionality [Zug08]

Services

The Alaska Simulator's business logic contains several algorithms calculating an action's properties. In order to avoid dispersing the calculations across the application services were utilized encapsulating the functionalities and preventing so called *Shotgun Surgery* [Fow99]. Furthermore, the exchangeability of services facilitates the decoupling of certain functionalities and fosters modularization [Erl05].

Consequently, *Interfaces* were created defining the functionalities provided by specific services. As depicted in Figure 3.4, the implementation can be exchanged without modifying the clients of the service. To further extend the usability of this concept the Alaska Simulator provides a `ServiceRegistry` which ensures that for a specific task only a single service is used by registering an instance of the service. Client classes can use the service registry to obtain a reference to the requested service, giving them the possibility to invoke methods defined in the service's public interface. It is important that developers only use methods defined in the public interfaces of services as exchangeability is reduced otherwise.

Event Service

In order to provide enhanced event functionalities like duration changes, event expiration (cf. Section 2.4) and synchronization of events accessing the same action the *Event Service* had be considerably extended as the previous version of the Alaska Simulator only contained rudimentary support for events.

Event Properties All types of events share some common properties which are described subsequently.

- **Event Trigger Mechanisms.** Events can be attached to locations, routes and activities. In case of routes and activities, the event may occur whenever the corresponding action is executed. For example, an accident may occur when travelling from one location to another. On the contrary, events attached to locations can only occur at the traveller's current location. For example, the user arrives at the destination and spots an advertisement about an exciting attraction he was not aware about previously.
- **Event Probability.** Each event has a specific probability indicating how likely an event is to occur. A probability of 1.0 ensures that the event occurs whenever the corresponding event trigger is fired, whereas 0.0 prevents the execution at all. Nevertheless, events can only occur once. Even if the probability is 1.0, an event is executed only the first time the corresponding condition is fulfilled.
- **Event Expiration.** It is possible that a road is closed due to an accident, but after removing the destroyed vehicles the the road is reopened for traffic. The Alaska Simulator supports this concept by providing an optional expiration timeout for events. If no expiration time is specified the effects of the event will not be reverted until the traveller's plane departs at the end of the journey.

As a result of an event increasing the duration of the currently executed action it is possible that the user needs to reschedule the following actions due to conflicts within the journey plan. From time to time, it may be possible that such a duration increase happens when executing the last action of the day and the user arrives delayed at the accommodation for the night. For modelling this scenario two different mechanisms were implemented in the Alaska Simulator.

- **Minimum Sleep Amount.** This approach defines a minimum number of hours one has to sleep every night. If the arrival at the accommodation

is further delayed and the user cannot get an appropriate amount of sleep he has to stay in bed longer the next morning. Consequently, the traveller needs to adjust his plans for the next day.

- **Business Value Reduction.** The traveller is forced to get up the next morning even if still being tired. As a result, the business values of all actions which are executed the next day are reduced depending on the amount of sleep he had. For example, being late for about half an hour does not decrease the business value significantly, whereas missing a few hours of sleep reduces the business value dramatically. For this purpose the business value service had to created providing means for reducing an action's business value.

Event Core Structure Figure 3.5 depicts the structure of the event mechanism. The abstract `Event` class defines functionalities for executing and expiring events, which have to be implemented by it's subclasses that are representing the different types of events. `NewActionConfigEvent`s are utilized whenever new actions should become available at run-time. Expiration of those events is handled by setting the availability of the corresponding action to 0.0. On the contrary, `ChangeActionConfigEvent`s are responsible for changing values of existing actions during the journey. For example, the business value or the cost of an action might be increased. In order to offer the possibilities of changing action durations another subclass had to be introduced, which was capable of changing the action's duration. `ChangeTemporalActionEvent`s can only be applied to temporal actions (e.g., routes and activities). Accommodations cannot be altered by this event type as their duration is always the same as the night's duration.

Event Synchronisation By incorporating expiration timeouts the problem of event synchronization arose as several events can modify the same action. Previously, an event altered the values stored in the `ActionConfigurationProxy` which were only changed in case another unforeseen event was being executed accessing the same `ActionConfigurationProxy`. With the introduction of ex-

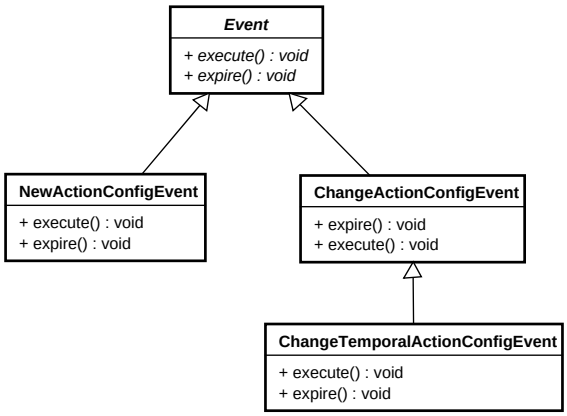

Figure 3.5: Event Core Structure

pirable events a mechanism for resetting the values to the ones previously set by another event had to be created.

Two possible scenarios illustrating the synchronization of events modifying the same values of an action are depicted in Figure 3.6. Scenario a) contains two nested events. Event 1 is executed changing the values of the corresponding action. Afterwards Event 2 is executed changing the action's properties again. After the expiration of Event 2 the values defined in Event 1 should be used before resetting them to the initial ones after expiring Event 1. Scenario b) depicts two overlapping events. Similar to the previous scenario, after executing both events, the values of the action should be set to the ones defined in Event 2. In contrast to scenario a) Event 1 expires first not affecting the values of the corresponding action. After the expiration of Event 2 the values are reset to the initial ones defined in the action's configuration. Furthermore, it might be possible that events only affect specific values. For example Event 1 in scenario a) might change only the business value of an action whereas Event 2 changes the action's cost. Consequently, the business value of the action should be the one defined in Event 1 whereas the cost of the action should be set to the value defined in Event 2.

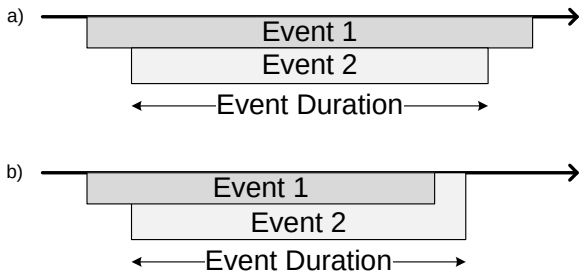

Figure 3.6: Event Synchronisation Scenarios

Furthermore, the extended capabilities of events affected other parts of the application. For example, the availability of actions was pre-calculated on a per-day basis. Events changing an action's availability simply forced a re-calculation of the availability. This practice worked well as long as event expiration was not an issue. After extending the event mechanism also the availability service needed to be adapted in order to assign the previously calculated availability to an action after expiring the corresponding event.

Constraint Service

The *Constraint Service* is responsible for validating the journey plan and preventing the execution of actions that are violating constraints of the journey. Consequently, when executing an action all *Execution Constraints* applicable for the current journey have to be evaluated. When finishing a journey, all *Termination Constraints* have to be evaluated, investigating if the journey was completed successfully (e.g., all constraints were fulfilled).

Constraint Core Structure As illustrated in Figure 3.7, all constraints implemented in the Alaska Simulator share a common abstract base class called `Constraint`, offering methods for validating the journey plan, testing if all conditions for executing an action are fulfilled and evaluating if a journey can be completed successfully. Furthermore, a method checking if the given action is influenced by the constraint was implemented, which is utilized when displaying

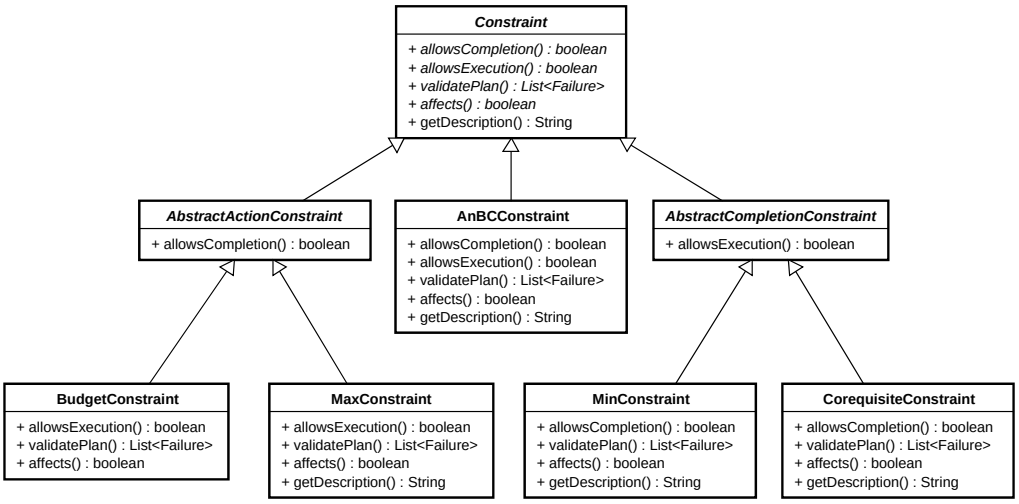

Figure 3.7: Constraint Core Structure

additional information about actions to the user. Additionally, each constraint offers a human readable textual description explaining the restrictions deployed by the constraint. For this purpose, a default description can be passed when invoking the constructor of the `Constraint` class or subclasses may override the `getDescription` method.

Two abstract classes extending the Constraint class are provided representing default implementation for Execution and Termination constraints. A default implementation for `allowsCompletion` always returning true as action constraints are only used for checking execution rights of actions is provided by the `AbstractActionConstraint` class. The `AbstractCompletionConstraint` class provides a default implementation for `allowsExecution` as execution right checks are not addressed by this constraint type. In order to add an additional constraint, one of the afore mentioned classes has to be extended and the abstract methods have to be implemented. Figure 3.7 depicts only a small subset of the constraints available in the Alaska Simulator (cf. Section 2.3).

Weather Service

The *Weather Service* was extended to give the experimenter more control about the environmental circumstances of the journey. In contrast to the previous implementation the experimenter can define a specific weather seed, which will be used to calculate the weather for all locations of the journey, when creating the journey configuration. For this purpose, the `IWeatherSeed` interface was created and incorporated in the journey configuration. Two base implementations, one providing a random seed and another one calculating predictable seeds for all locations were implemented.

Furthermore, overwriting the weather seed when conducting the experiment is possible by providing a specific password to subjects. This can be useful when conducting an experiment where users execute a specific journey several times (cf. Chapter 4). As learning effects about the weather conditions might compromise the experiment, different weather seeds for the same configuration can be provided by simply changing the used password.

Business Value Service

When introducing the concept of *Business Value Reduction*, which reduces the business value of all actions of a day if the traveller arrived delayed at the accommodation the day before, the original business value calculation had to be replaced by a more sophisticated one.

As the service oriented structure had proven to work well within other parts of the application we decided to introduce another service capable of calculating business values of actions. The *Business Value Service* has information about the current weather situation at all locations and about additional reductions affecting the outcome of the calculation. Business Value Reductions are represented by classes implementing the `IBusinessValueConstraint` interface, which is offering the possibility to extend the Alaska Simulator by adding additional classes manipulating the business value of actions. For this purpose, a class implementing the interface has to be created and registered with the Business Value

Service. Like every other service, the Business Value Service should be accessed via the Service Registry.

3.2.2 Alaska User Interface

This section contains information regarding the graphical user interface of the Alaska Simulator. The user interface was developed as a separate plug-in that makes use of the Alaska Core (cf. Section 3.2.1). In order to create a convenient user interface heavy use of the functionalities, e.g., views, provided by the Eclipse Rich Client Platform [CR08] was made (cf. Section 3.3.1). Additionally the user interface was created using the Graphical Editing Framework (GEF), which provides support for the development of rich graphical representations of models [MDG+04] (for details see [Zug08]).

Constraint Overview

The increased amount of constraints attached to a journey demanded the creation of an overview mechanism for the user. Eclipse offers several technologies facilitating user interface creation. As all constraints share the same base class, offering a method for obtaining a constraint's human readable description, it was no problem to create a table containing all constraints of a journey. For this purpose a JFace viewer was utilized making filtering and sorting of the constraints rather convenient as only the corresponding classes had to be extended [CR08]. The viewer was embedded into an Eclipse view, making full use of Eclipse's user interface.

Event Overview

Similar to the Constraint Overview another view containing information about all unforeseen events happened during the journey was implemented. Sorting and filtering mechanisms provide means to quickly retrieve information about the journey. By making use of the *Observer Pattern* [GHJV94] the overview is always updated as soon as new unforeseen events occur.

3.3 Third Party Frameworks

This section shortly introduces frameworks used during the development of the Alaska Simulator and explains how they relate to each other. Section 3.3.1 describes the Eclipse Rich Client Platform, which is used as a basis for the application. Subsequently, Section 3.3.2 and 3.3.3 focus on testing frameworks used to guarantee the quality of the software. Finally, Section 3.3.4 illustrates how the afore mentioned testing frameworks relate to each other and why they were used.

3.3.1 Eclipse Rich Client Platform

Originally Eclipse was designed as an Integrated Development Environment (*IDE*) for Java applications. In recent years a framework for building custom applications was extracted from the original Eclipse IDE [ML06]. Nowadays, the Eclipse Rich Client Platform (RCP) is a generic platform that can be utilized for building and executing applications [ML06]. The advantage of developing a RCP application is the reusability of frequently used concepts and the therefore shortened development time. Furthermore, the quality and compatibility of future versions of the framework is guaranteed by the Eclipse development team.

Eclipse Rich Client Platform Architecture

The architecture of the Eclipse Rich Client Platform is based on the concept of *plug-ins*. Consequently, developers can extend the functionalities by developing and integrating additional plug-ins. When starting the application all present plug-ins are loaded and installed accordingly. As every Eclipse Rich Client application can be interpreted as a set of plug-ins, modularization is fostered. Consequently, parts of the application can be reused by adding the plug-in and its prerequisites to the run-time environment.

The Eclipse architecture is depicted in Figure 3.8. At the bottom the platform runtime is shown, which takes care of the installed plug-ins necessary for running the application. On top the Standard Widget Toolkit (*SWT*) is responsible for basic user interface components. Those components are wrapped by, so called

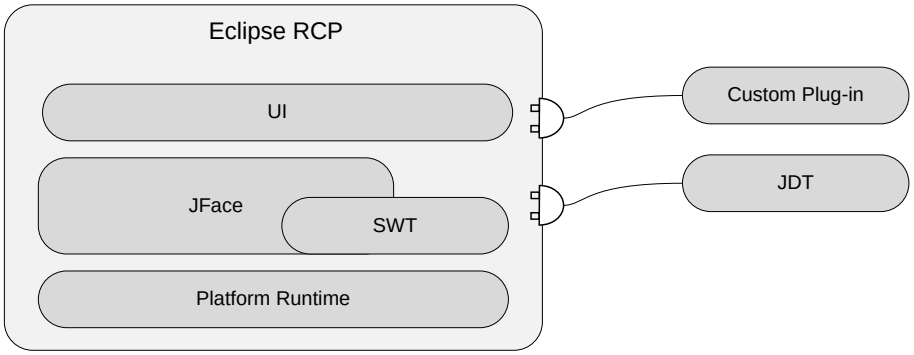

Figure 3.8: Eclipse Rich Client Platform's Architecture [Zug08]

Viewers, within the *JFace* layer. Viewers provide high-level support for manipulating graphical components [CR08]. Even more sophisticated user interface components, e.g., perspectives, editors or views, are provided by the *UI* layer (for details see [CR08]). Custom plug-ins, like the Alaska Simulator, may use the functionalities provided by the afore mentioned layers of the Eclipse Rich Client Platform. The Java Development Environment (*JDT*) can be interpreted as a RCP application itself, as it is based on the previously described components.

Drawbacks

Several drawbacks exist when using the Eclipse Rich Client Platform. Especially for beginners the learning curve is rather steep as many concepts have to be understood in order efficiently develop applications. After mastering the initial difficulties the advantages, like included packaging of the application (for details see [DFK+04]), outweigh the increased effort in previous development stages.

Another drawback is the amount of code included in auxiliary plug-ins. Consequently, the actually developed plug-ins may only be a small portion of the deployed application.

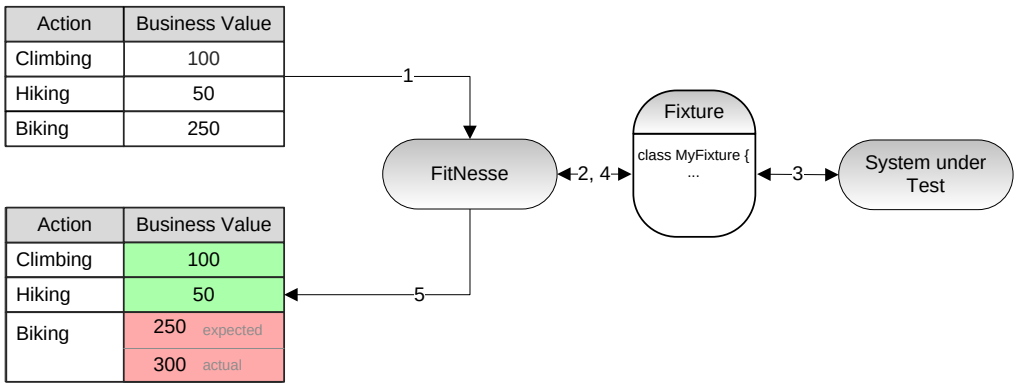

Figure 3.9: Interaction of FitNesse with Tests and System Under Test [Zug08]

3.3.2 FitNesse

FitNesse is an automated acceptance testing framework, which fosters the collaboration between customers and developers [MC05]. It compares expected outcomes, represented as tables, to the actual results returned by the *system under test*. Consequently, developers are ensured that the implemented functionalities are implemented correctly and furthermore a contract of acceptance is created defining what behavior is necessary in order to completely implement a demanded feature [MC05].

FitNesse is based on the Fit framework, which actually offers the possibility to extract tables from an HTML page and execute the desired functionalities, utilizing *fixtures* as glue code between the HTML pages and the system under test. The resulting outcome is compared to the expected values defined in the HTML table, and written to a result page. FitNesse extends Fit by providing a Wiki that enables the user to easily create, maintain and execute tests.

Test Execution

Figure 3.9 depicts the execution of FitNesse tests and the interplay with the system under test using *fixtures*. When the user triggers the test execution within the FitNesse Wiki an ordinary HTTP request is sent to a web-server, provided

by the FitNesse framework, which contains information about which tests should be run (1). The web server uses Fit to parse the HTML pages and to determine the tests to be executed. In order to invoke the functionalities of the system under test *fixtures* have to be implemented. Fixtures are simple classes, specific for the application under test, subclassing existing fixtures provided by the Fit framework. Those classes are capable of identifying the request (2) and triggering the desired functionalities in the system under test and returning the retrieved results (3). The obtained results are interpreted by FitNesse that is again making use of the Fit Framework (4). Finally, HTML pages containing information about the results of the tests are created and returned to the user (5). In the example shown in Figure 3.9 the expected business values of actions are compared to the ones returned by the system under test. The business values of Climbing and Hiking are correct, whereas the actual business value for Biking is 50 lower than the expected one.

3.3.3 JUnit

JUnit is a framework intended for testing Java applications, especially focusing on unit tests [Bec04]. It is freely available and seamlessly integrated in several development environments. Tests are represented by Java classes invoking functionalities provided by the system under test. By using special assert statement the results of computations can be compared to the expected values. Consequently, JUnit is mostly used for testing the correct behavior of single or small sets of classes.

3.3.4 Interplay of Testing Concepts

The testing concepts described in the previous sections work very well together as JUnit focuses on testing the behavior of single classes whereas FitNesse utilizes a more high-level approach to evaluate larger parts of the application.

During the development of the Alaska Simulator JUnit tests were used in order to ensure the correct behavior of single classes. As customers are normally not aware of the Java syntax it is not feasible to use JUnit tests to clarify the

behavior of applications. Furthermore, developers cannot expect customers to be aware of the concept of classes. Consequently, another approach has to be used for discussing functionalities with customers. FitNesse provides a convenient mechanism as tests are represented using tables. As a result customers and developers can collaborate creating test cases which are defining the correct behavior of the application. Furthermore, FitNesse tests can be interpreted as a contract of acceptance [MC05], as passing all tests ensures that all functionalities have been implemented.

In a nutshell, JUnit and FitNesse work perfectly together as one approach is designed for testing small portions of the application, whereas the other one uses a more abstract view providing a representation that can also be understood by business people.

Chapter 4

Experiment

This chapter describes the experiment for evaluating the impact of unforeseen events on software projects. Section 4.1 wraps up the basic terminologies which are subsequently used to explain the experiment. Section 4.2 illustrates the experiment's setup whereas Section 4.3 focuses on the execution of the experiment. Section 4.4 outlines the data analysis procedure and presents the experiment's results. Section 4.5 discusses risks threatening the validity of the experiment. The chapter is concluded by Section 4.6 providing a discussion of the results.

4.1 Basic Terminology

Before describing the actual experiment some basic terminologies will be explained (cf. Sections 4.1.1 to 4.1.5) and put into context (cf. Section 4.1.6).

4.1.1 Objects

Objects are physical entities on which the experiment is run [JM01]. It does not matter if those objects are material or immaterial (e.g., exercises or errors).

4.1.2 Subjects

Subjects can be persons but may also be non-physical things, like opinions, which apply techniques on objects [JM01]. The goal of the experiment is the identification of varying results after adapting parameters defined in the experiment's

environment. When performing experiments in the context of software engineering it is likely that subjects influence the outcome of the experiment, whereas in other disciplines the result is not changed by the subject [JM01]. For example, when testing different fertilizers the subjects are the people who apply them. It is likely that the influence of the subject is rather insignificant for the growth of the plant. Software engineers on the other hand change the outcome and have therefore to be specially addressed when designing the experiment [JM01].

4.1.3 Independent Variables

Experiments assume that there are not any other factors influencing the outcome but the independent variables, which are controlled by the experimenter. Independent variables are also referred to as factors [JM01]. Therefore, the effect of changing the independent variables by running the experiment several times with different factor levels can be derived. The set of independent variables should be kept rather small as an increase of independent variables complicates the experiment's analysis. An experiment with only one single independent variable is called a single factor experiment [KL99].

4.1.4 Response Variables

The outcome of an experiment is represented by the response variables or dependant variables [JM01]. Response variables depend on the independent variables and are utilized to measure how applying the different factor levels on the independent variables influences the result of the experiment. The experimenter should strive for identifying statistical significant differences in the response variables when applying the different factor levels to the independent variables.

4.1.5 Hypotheses

In case of an experiment the hypotheses represent the results expected by the experimenter [TD07]. For an experiment being well defined two mutual exclusive hypotheses have to be defined. In other words, only one of the hypotheses can be fulfilled.

- **Alternative Hypothesis.** The alternative hypothesis represents the experimenter's prediction of the experiment's outcome. The abbreviation used for the alternative hypothesis is H_1.

- **Null Hypothesis.** The null hypothesis is contrary to the alternative hypothesis. It is commonly abbreviated using H_0.

Hypotheses can be divided into *one-tailed* and *two-tailed* hypotheses.

- **One-tailed Hypothesis.** Additionally to the dependant variables the direction of change which is expected for the different factor levels has to be defined. For example, in case of a numeric response variable the experimenter has to define if the variable is going to increase or decrease.

- **Two-tailed Hypothesis.** Two-tailed Hypothesis do not specify a direction. Only the fact that the response variable is going to change significantly is stated.

4.1.6 Interrelations

This section combines the terminologies defined in the previous sections and illustrates their interplay (cf. Figure 4.1). *Subjects* taking part in the experiment perform actions on the experiment's *objects*. The experimenter tries to find out how the *independent variables* influence the subjects when they are performing their tasks, by observing changes in the *response variables*. The experiment's null hypotheses can be rejected if the experimenter can identify statistical significant dependencies between the independent and the response variables.

4.2 Experiment Design

This section explains the subjects, objects, selected variables and hypotheses of the experiment described in this thesis, which was conducted at the Management Center Innsbruck.

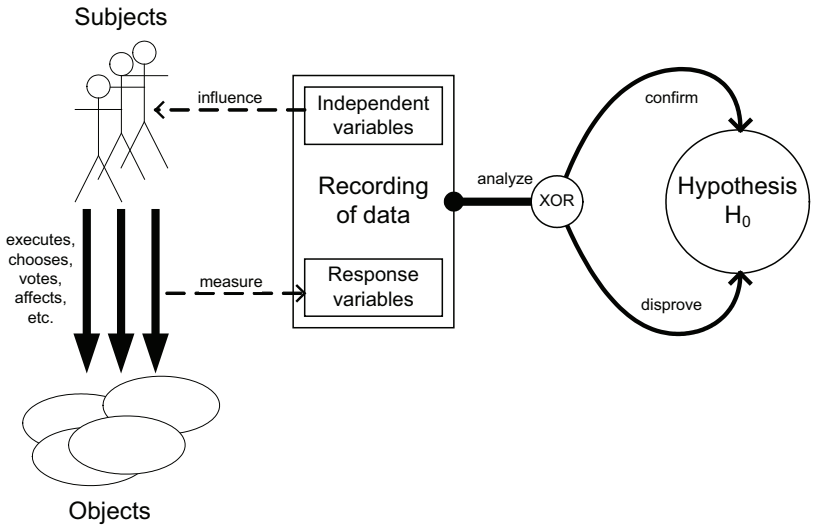

Figure 4.1: Experiment's Concepts [Sch08]

- **Subjects.** Students of the "Agile Softwareentwicklung und Projektmanagement" class were chosen as **subjects**.

- **Objects.** Two configurations, subsequently referred to as Configuration California and Configuration Alaska, were provided, which contained actions to be executed and constraints restricting the user's possibilities. Furthermore, events may required the user to react on unforeseen changes during run-time. Each configuration was available in two variants which were only differing in the number of unforeseen events. Variant A did not contain any events, whereas Variant B contained several unforeseen events. Except the changing number of events variants A and B were identical for each configuration.

- **Factor and Factor Levels.** The number of unforeseen events in a journey can be considered the factor of the experiment with the factor levels "No Events" and "Events". Variants A are corresponding to the factor level "No Events" and variants B to the factor level "Events".

- **Response Variables.** For evaluating the planning behavior the gained *business value* (cf. Section 2.2), on the one hand, and the *overall success* of the journey, on the other hand, is taken into account and can therefore be classified as the experiment's **response variables**. A journey is considered being successful if the traveller manages to fulfill all constraints and is able to get back to the starting location in time. In case of a missed flight or constraint violations the journey is not considered being successful.

Hypotheses

Subsequently, the hypotheses of the experiment are described.

- H_0^{BV}: The number of unforeseen events does not decrease the business value.
- $H_0^{Success}$: The number of unforeseen events decreases the number of successful journeys.

The corresponding alternative hypotheses are defined as follows.

- H_1^{BV}: The number of unforeseen events decreases the business value.
- $H_1^{Success}$: The number of unforeseen events decreases the number of successful journeys.

Group Assignment

Students were randomly divided into two groups (cf. Figure 4.2). In the first run each student had to plan a journey to California. Group 1 was assigned the task of planning a journey using Variant B containing several unforeseen events, whereas Group 2 used Variant A without any events. After completing the journey the groups were swapped and the students were asked to start their journey to Alaska. Therefore, every student had to plan a journey with lots of events and one without any events. It has to be pointed out that the students did not know which variant of the configuration they were using.

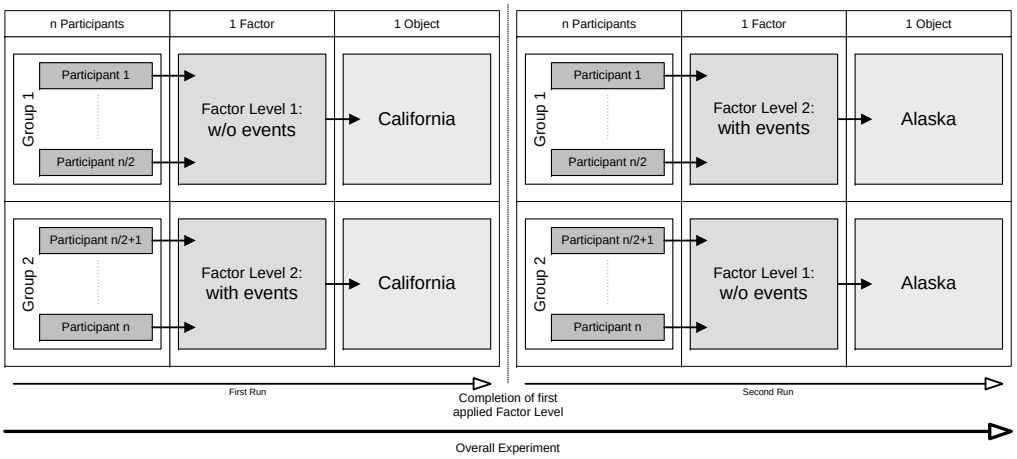

Figure 4.2: Experiment's Phases adapted from [Sch08]

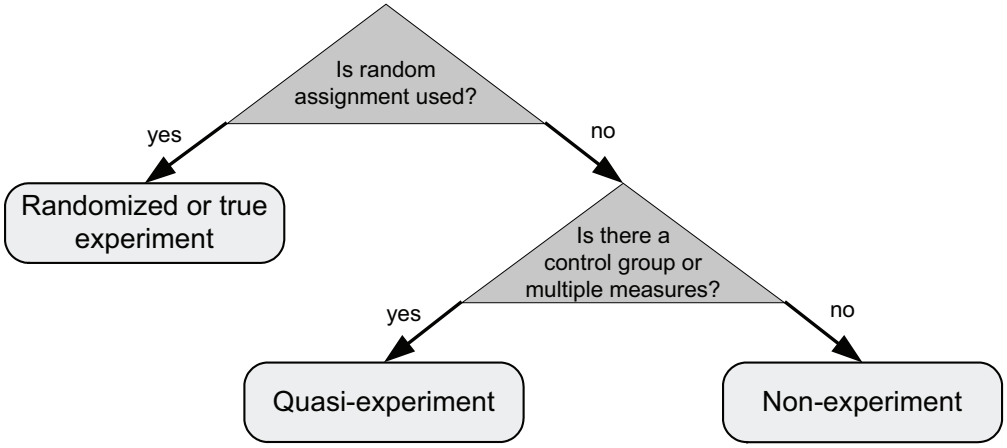

Figure 4.3: Classification of Experimental Designs [Sch08]

Experimental Design

Figure 4.3 depicts the classification of experiments according to [TD07]. If the subjects can be assigned to groups randomly the experiment is called *randomized*, which provides the best internal validity and should therefore be used whenever possible. *Internal validity* of an experiment is the approximate truth about inferences regarding causal relationships [WRH+00]. In other words, it indicates whether the response variables are solely influenced by the independent variables or if there are other manipulating factors.
Whenever a random assignment of groups is not possible, but multiple measurements or groups do exist a *quasi-experiment* is conducted. Experiments without groups or multiple measurements are called *non-experiment*.

The experiment described in this thesis can be considered being a **randomized single factor experiment with repeated measurement**, which addresses the influence of unexpected events on the planning process and the resulting decrease of business value and danger for the overall success.

Mathematical View

n subjects S_i $1 \leq i \leq n$ are split up into two groups with size $\frac{n}{2}$. Each group has to plan a journey to Alaska and California, one with several events and the other one without unforeseen events. The students do not know if they are executing a journey with or without events. $2n$ results $R_j^{T_A,Event}$, $R_j^{T_A,NoEvents}$, $R_j^{T_C,Events}$ and $R_j^{T_C,NoEvents}$ with $j \in \{1, \ldots, \frac{n}{2}\}$ are retrieved by conducting the experiment.

Data Measurement

Whenever a significant planning step (e.g., adding an action, executing an action) is performed by the user the system logs it to a central database. Therefore, no additional user interaction is necessary for gathering data and one cannot influence the measurement process. Likewise, the database server is appropriately secured. In order to guarantee the correct timing of the experiment a password protection was implemented, which enforces the correct starting of journeys. Consequently, the measuring process can be considered as secure. The design of the experiment is wrapped up in Table 4.1.

Design terminology	Corresponding element
Subjects	Students at the Management Center Innsbruck
Objects	Two configurations of the Alaska Simulator
Independent variable	Events / No Events
Response variables	Business value and overall success
Experiment design type	Randomized single factor experiment with repeated measurement
Hypotheses	The number of events does not decrease the business value and journey success

Table 4.1: Experiment's Design

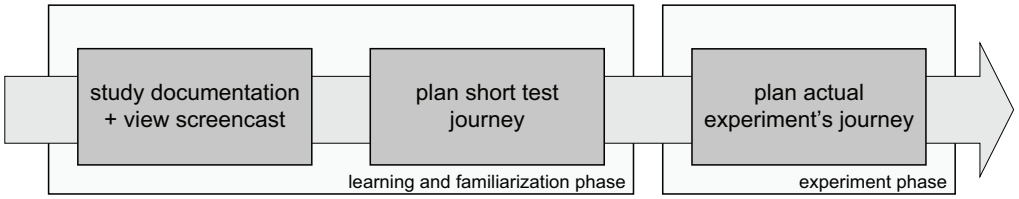

Figure 4.4: Experiment Run Structure [Sch08]

4.3 Experiment Execution

The experiment took place on December 6th 2008 at the Management Center Innsbruck. The experiment had to be split up into two sessions, each following exactly the same protocol. 32 students attending the "Agile Softwareentwicklung und Projektmanagement" course were participating in the experiment.

As all students were taking part in the "Agile Softwareentwicklung und Projektmanagement" course, some prior knowledge about planning and agile principles can be assumed. Therefore, only a brief theoretical introduction had to be given to explain the setup and goals of the experiment and why a journey is an appropriate metaphor for managing a software development project (cf. Fig. 4.4).

Learning and Familiarization Phase

The following phase was intended to familiarize the students with the Alaska Simulator. For this purpose screen-casts and a simple journey configuration

were used. The students had about 30 minutes for watching the screen-casts and investigating functionalities of the Alaska Simulator.

Experimental Phase

The experimental phase depicted in Figure 4.2, consisted of two runs. Group 1 was presented with Configuration California A without any events, whereas Group 2 had to plan a voyage using Configuration California B containing several unforeseen events. Afterwards Group 1 had to plan a journey to Alaska (Configuration Alaska B) with runtime events and Group 2 was working on the journey to Alaska without events (Configuration Alaska A).

Each run was divided into two parts. First, the students had about 25 minutes to inspect the configuration and investigate their details like the number of constraints and available actions. Second, the students were asked to stop their current activities and restart planning. After another 20 minutes the students had to finish their journey, which was used for analysing the student's planning behavior.

4.4 Data Analysis Procedure

This section focuses on the validation and analysis of the data retrieved in the experiment.

4.4.1 Data Validation

The logging mechanism formerly described in Section 4.2 ensured that all relevant planning steps were recorded in a central database, saving the corresponding host for identifying the journeys. The recorded data was filtered in order obtain only relevant journeys. For example, journeys created during the familiarization phase (cf. Section 4.3) were omitted.

One threat for the validity of the experiment is the compliance with the experiment's setup. This issue turned out to be severe in this experiment, as some students collaborated when planning their journeys. Others did not plan a sec-

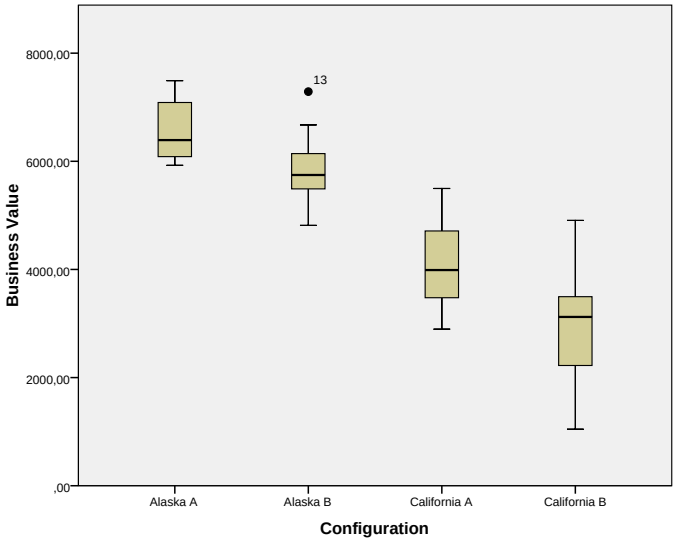

Figure 4.5: Business Value per Configuration

ond journey after finishing the familiarization phase (cf. Section 4.3). Therefore, all data of those students had to be discarded.

When performing an experiment with students one has to consider limited motivation and different levels of experience as influencing factors. Fortunately, our result set contained no journeys with recognizably low business values. The box plot depicted in Figure 4.5 shows no lower outliers (smaller than median minus three times the inter quartile range). It has to be pointed out that this plot also contains journeys which failed to fulfill all constraints.

4.4.2 Data Analysis

This section describes the data analysis procedure of the experiment. The first part focuses on a descriptive analysis of the data by calculating several metrics like mean and standard deviation. Afterwards procedures for testing the hypotheses are outlined before presenting the obtained results.

Descriptive Analysis

After discarding the flawed data 38 journeys from 19 subjects remained in the database. Table 4.2 shows descriptive metrics of the data. Due to differences between the configurations the business values obtained using Configuration Alaska are higher than the ones from Configuration California.

Scenario	Events	N	Fail	Min	25% Q.	Mean	75% Q.	Max	Stdv.
Alaska	No	9	0	5925	6026	6581	7105	7492	573
Alaska	Yes	10	4	4815	5478	5883	6275	7289	693
California	No	10	1	2895	3457	4114	4827	5497	827
California	Yes	9	6	1045	1861	2826	3564	4907	1179

Table 4.2: Descriptive Statistics

By investigating the average business values it can be observed that the journeys with events have lower business values than the corresponding configurations without events. Furthermore, the number of erroneous journeys, e.g., journeys for which the user did not succeed in fulfilling all constraints (listed in the Fail column), increases with the appearance of events. However, it has to be investigated whether these differences are statistically significant.

Hypotheses Testing

This section describes actions that have to be performed in order to test the validity of the hypotheses and presents the obtained results.

Business Value Hypothesis In order to evaluate the business value hypothesis H_1^{BV} dealing with decreased overall business values due to unforeseen events, we need to test whether there is a significant difference in the mean business values between both journeys. As the Alaska(T_A) and California(T_C) configurations have different average business values the evaluation has to be done separately. Therefore $R^{T_A,Event}$ is compared to $R^{T_A,NoEvents}$ and $R^{T_C,Event}$ to $R^{T_C,NoEvents}$.

For evaluating the significance of the differences the t-test is utilized, which indicates if the means of two groups differ (for details see [TD07]). In order to perform a t-test it has to be ensured that all samples follow a normal distribution.

For this purpose the Kolmogorov-Smirnov is used to compare the samples to a normal distribution. Furthermore, the samples have to be tested for equality of variances. SPSS performs the Levene test for this purpose (for details see [Sta08]).

In a nutshell, the following tests are performed to test the business value hypothesis H_1^{BV}.

1. Kolmogorov-Smirnov test for normal distribution of
   - $R^{T_A,Event}$
   - $R^{T_A,NoEvents}$
   - $R^{T_C,Event}$
   - $R^{T_C,NoEvents}$
2. T-Test (including Levene test) for
   - $R^{T_A,Event}$ - $R^{T_A,NoEvents}$
   - $R^{T_C,Event}$ - $R^{T_C,NoEvents}$

Business Value Hypothesis Evaluation The results of the Kolmogorov-Smirnov test are summed up in Table 4.3. As the significance for all configurations is greater than 0.05 it is safe to assume that the data follows a normal distribution, and it is therefore possible to perform a t-test.

Group	Significance	Normal Distribution
$R^{T_A,Event}$	0.931	**yes**
$R^{T_A,NoEvents}$	0.918	**yes**
$R^{T_C,Event}$	0.983	**yes**
$R^{T_C,NoEvents}$	0.993	**yes**

Table 4.3: Results of the Kolmogorov-Smirnov Test

The results of the t-test are shown in Table 4.4. In order to perform the t-test SPSS automatically executes the Levene test to check the equality of variances. As illustrated in Table 4.4 (column Sign. L.) both values are greater than 0.05 indicating that the variances are equal.

The third column contains the significance value of the t-test. As this value is smaller than 0.05 the null hypothesis H_0^{BV} can be rejected at a confidence level of 95% implying that the difference of the samples' means is between the lower and upper bound with a certainty of 95% [Zar98]. As the lower and upper bounds of the confidence interval are positive numbers it can be expected that the business values of journeys without unforeseen events are significantly higher than the ones obtained in journeys containing unforeseen events.

			95% Conf. Interval		
Scenario	**Sign. L.**	**Sign. T.**	**Lower**	**Upper**	**Result**
Alaska	0.982	0.030	78.28	1317.79	**reject**
California	0.313	0.013	310.99	2265.84	**reject**

Table 4.4: Results of the T-Test

Success Hypothesis In order to evaluate the success hypothesis $H_1^{Success}$ dealing with decreased success values due to unforeseen events, we need to test whether there is a significant difference between the success rates of both journeys. A journey is considered being successful if all constraints are fulfilled and the journey's end location is reached in time. As by counting the number of successful and unsuccessful journeys only nominal data can be obtained the t-test is not applicable. Consequently, tables with the structure illustrated in Table 4.5 are constructed for each configuration, which can be investigated using the Chi Square or Fisher's exact test [Fle81]. Due to the limited amount of samples

	No Events	**Events**	**Total**
Unsuccessful	N1	N2	N1+N2
Successful	N3	N4	N3+N4
Total	N1+N3	N2+N4	N1+N2+N3+N4

Table 4.5: Success Table Structure

the Chi Square test would lead to imprecise results, whereas Fisher's exact test calculates the exact significance and can therefore also be used for small sample

sizes and unbalanced tables [Fle81].
For evaluating the success hypothesis the following steps have to be applied.

1. Create success tables (cf. Table 4.5) for
   - $R^{T_A,Event}$ - $R^{T_A,NoEvents}$
   - $R^{T_C,Event}$ - $R^{T_C,NoEvents}$
2. Perform Fisher's exact test on both tables

Success Hypothesis Evaluation In order to perform Fisher's exact test success tables had to be created for Configuration Alaska and Configuration California (cf. Tables 4.6 and 4.7).

	No Events	**Events**	**Total**
Unsuccessful	0	4	4
Successful	9	6	15
Total	9	10	19

Table 4.6: Success Table Alaska

	No Events	**Events**	**Total**
Unsuccessful	1	6	7
Successful	9	3	12
Total	10	9	19

Table 4.7: Success Table California

The result of Fisher's exact test is shown in Table 4.8. The null hypothesis $H_0^{Success}$, which indicates that there is no significant difference between the journeys with or without events can clearly be rejected for Configuration California, as the significance is less than 0.05. For Configuration Alaska the null hypothesis cannot be rejected. Consequently, it cannot be shown that there is a significant difference in the number of successful journeys depending on the number of unforeseen events. This might be due to the relatively low sample size and should therefore be revalidated in another experiment with a higher number of participants.

Scenario	Significance	Result
Alaska	0.054	**not reject**
California	0.017	**reject**

Table 4.8: Results of Fisher's Exact Test

4.5 Risk Analysis and Mitigation

This section describes risks endangering the experiment's validity, and countermeasures taken to mitigating them. The validity of an experiment can be categorized into **internal** and **external** validity.

4.5.1 Internal Validity

The internal validity of an experiment deals with the observed relationship between treatment and outcome [WRH+00]. It has to be ensured that this relation is causal and is not influenced by factors the experimenter has no control about. All other influencing factors should be reduced in order to make only an insignificant impact on the outcome. Consequently, only the treatment influences the experiment's result [WRH+00].

Experiment Setup

In order to guarantee internal validity the experiment's setup has to be followed by all students. Due to the small groups some of the mistakes endangering the internal validity could be tackled during the experiment. Unfortunately, some students still failed to obey the setup. Consequently, all journey of those students were removed and not taken into account when evaluating the experiment (cf. Section 4.4).

Motivation

Another threat that may be considered is student motivation, which might influence the outcome as a result of careless planning. If subjects are not interested in planning a journey the gained business value will most likely be below the

average and therefore negatively influence the overall results. Fortunately, motivation did not seem to be a severe problem during the experiment as there were no journeys indicating careless planning. A lack of motivation could only be recognized by not obeying the experiment's setup and skipping one phase. Respective journeys were removed from the data set.

Planning Experience

Different planning experience levels can influence the outcome of the experiment. As all students are taking the same course at the Management Center Innsbruck a common prior knowledge can be assumed.

Usability

The usability of the planning tool may also influence the gained business value. If the user interface facilitates confusion instead of supporting the subject by providing useful information the outcome may be affected. It is hard to measure the usability of an application, but we are confident that the Alaska Simulator is performing well. So far about 300 journeys have been planned by users of all ages ranging from 7 to 70 years leading to a steady improvement of the user interface.

Data Measurement

Data measurement was not a major concern for the internal validity of the experiment, as all planning steps were automatically logged to a central database (cf. Section 4.4). The correctness of the logging mechanism was enforced by automated tests [MC05].

4.5.2 External Validity

External validity deals with generalizing the results outside the scope of the experiment [WRH+00]. In the context of the Alaska Simulator it has to be ensured that the results are applicable for software engineering.

Sample Size

The biggest concern in terms of external validity may be the rather small sample size. Unfortunately, some journeys had to be removed from the data, as the students did not obey the experiment's setup.

As a result of the rather small sample size we are planning a replication of the experiment with a higher number of subjects in order to verify the results obtained from this experiment.

Journey Metaphor

Using a metaphor and an artificial environment instead of a real project constitutes a potential threat to the generalization of our results. Chapter 2 describes the similarities of planning a journey and developing a software project. In either case resources are limited and additional constraints may increase the difficulty of the planning process. Furthermore, unforeseen events provide additional challenges on the way to a successful and in terms of business value optimized project. Therefore, we are confident that the journey metaphor is applicable for software development projects.

Planning Experience

The results may not be considered as representative for a software project where mostly experts are involved as students do not have enough prior knowledge in the field of planning. Although all students attended the "Agile Softwareentwicklung und Projektmanagement" class at the Management Center Innsbruck and therefore some prior knowledge in agile concepts can be assumed, they clearly were no experts in planning. However, experiments with students can give valuable insight into a problem domain [Run03] [Hou99].

4.6 Discussion

The results presented in Section 4.4 clearly indicate that the business value null hypothesis H_0^{BV}, which states that unforeseen events do not decrease the gained

business value of a journey, can be rejected. The null hypothesis $H_0^{Success}$, in turn, insisting that there is no increase of unsuccessful journeys when dealing with events could not be rejected. This might be due to the relatively small sample size making it difficult to identify significant differences. Nevertheless it can be derived that unforeseen events are negatively influencing the outcome of journeys by decreasing the gained business value.

Manual analysis of the journeys showed that some events were more dangerous for the success of a journey than others. Especially events that occurred in combination with mandatory actions were endangering the overall success of the journey. For example, a traffic jam on a route to a location which contained only one mandatory activity turned out to be a severe danger. The difficulty was further increased by a restricted execution time and reduced availability of the action. Therefore, the action could not simply be moved and huge parts of the plan had to be rearranged in order to fulfill the imposed constraints and use the remaining time as effectively as possible. Consequently, constraints affecting an action have to be taken into account when evaluating the impact of unforeseen events on the plan.

Previous experiments have shown that the number of constraints does not have a significant influence on the journey's outcome [WRZW09]. This seems to be confirmed by the results obtained in this experiment as only one out of 18 journeys without unforeseen events was not successful. The complexity of configurations used in this experiment can be compared to the complex ones utilized in [WRZW09] (cf. Table 4.9). Subjects provided with tool support to verify their plans against the constraints had no major difficulties in successfully completing their journeys as long as there were no unforeseen events providing additional challenges. In a nutshell, the success of a journey is mostly endangered by a combination of constraints and unforeseen events whereas constraints without unforeseen events do not affect the outcome.

In context of software development projects this might indicate that an increase of requirements does not endanger the success, whereas additional unforeseen events, like problems with legacy systems or technologies, put the overall success at risk. Proper planning is necessary to cope with unknown dangers. Especially

when dealing with important features of the product countermeasures have to be taken to be able to deal with the unexpected.

Scenario	Experiment	Constraints
Alaska	Events	12
Alaska	Constraints	12
California	Events	9
California	Constraints	10

Table 4.9: Comparison of Configurations with [WRZW09]

Planning Behavior

The efficient usage of agile principles for planning journeys of software development projects depends largely on personal strengths of the planner [Zug08]. The agile user interface offers lots of possibilities for planning a journey. It is possible to plan and book all actions in advance and make therefore use of a plan-driven approach, whereas it is also possible to skip the pre-planning phase and drop directly into the execution phase in a more or less chaotic way. The agile planning approach can be considered as being somewhere in between [Zug08]. Manual analysis showed that this turned out to be true for this experiment as well. Even tough all students had prior knowledge about agile principles and were encouraged to make use of them, some still tried to pre-plan the complete journey. Those students updated their plan only if forced by unforeseen events. Some of them mitigated the danger of events by using huge buffers. Unfortunately, they did not move their actions at run-time and therefore did not bring their journeys to the full potential by having lots of unused time in their plan.

When confronted with problems at run-time the personal strength of the user came into play. Some of them performed exceptional well when rearranging their plan to fulfill all constraints and still get the most business value out of the remaining time. On the contrary, some students seemed not to be able to keep up with the complexity of the planning process. Some of them tried to reduce complexity by focusing on mandatory activities and not taking business value optimizations into account. As a result, they still managed to complete the journey successfully, but their business value was inferior. Others tried to

take all factors into account, but failed due to planning mistakes to accomplish all mandatory goals of the journey.

When developing software products agile principles promise a higher gain, but also have a higher demand for well trained personal in order to cope with the increased complexity [Zug08]. Those findings seems to be confirmed by the experiment conducted in this thesis, especially focusing on unforeseen events. As long as well-trained employees, who are capable of creating a flexible plan upfront and reacting appropriately on unforeseen changes, are involved in the project a successful and satisfying outcome is likely. If those employees are missing it may be better to stick with a plan-driven approach including buffers in the plan in order to cope with uncertainty.

Chapter 5

Summary

This thesis described an experiment conducted at the Management Center Innsbruck investigating the influence of unforeseen events on the gained business value and the overall success of software development projects (measured as the implementation of all important features).

For this purpose, the *Alaska Simulator* has been developed deploying a journey as metaphor for software development projects. We consider this feasible as many parallels between journeys and software development projects can be identified (cf. Chapter 2). In order to prevent learning effects two journey configurations were provided, each available in two variants representing the different factor levels of the experiment. Students were undertaking a journey of each configuration switching to the other factor level in the second run, which allowed us to to compare the outcome of journeys affected by unforeseen events to the ones without events (cf. Chapter 4).

The results of the experiment showed that significantly reduced business values could be identified when users were presented with unforeseen events indicating that the outcome of software development projects is negatively affected by unforeseen changes. Manual analysis of the data revealed parallels to the results obtained in [Zug08]. When confronted with changes due to events several students were not able to cope with the high complexity of the agile planning approach leading to lower overall business values and unsuccessful journeys. In terms of successful journeys, however, no significant differences could be identified. This might be a result of the relatively small sample size, which complicates

the obtainment of statistically significant results. Consequently, we are planning another experiment following a similar setup with a higher number of subjects in order to obtain more significant data in terms of overall journey success. Additionally, some more effort might be invested in developing additional mechanisms supporting the user's decision making.

List of Figures

List of Tables

Bibliography

[AP06] W.M.P. van der Aalst and M. Pesic. Specifying, discovering, and monitoring service flows: Making web services process-aware. BPM Center Report BPM-06-09, BPMcenter.org, 2006.

[Bas96] Victor R. Basili. Editorial. *Empirical Software Engineering Journal*, 1(2), 1996.

[Bec99] Kent Beck. *Extreme Programming Explained: Embracing Change*. Addison-Wesley, 1999.

[Bec04] Kent Beck. *JUnit Pocket Guide*. O'Reilly Media, 2004.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21:61–72, 1988.

[BPHB02] Barry Boehm, Dan Port, LiGuo Huang, and Winsor Brown. Using the Spiral Model and MBASE to Generate new Acquisition Process Models: SAIV, CAIV, and SCQAIV. *CrossTalk - The Journal of Defense Software Engineering*, January 2002.

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

[Coh04] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

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

[CR08] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-Ins: Building Commercial-Quality Plug-ins.* Addison Wesley Pub Co Inc, 2008.

[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.

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

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

[Ele08] E-Learning at the University of Innsbruck. http://www.uibk.ac.at/elearning/, 2008. Last visited: 19.9.2008.

[EPV00] Dewayne E.Perry, Adam A. Porter, and Lawrence G. Votta. Empirical Studies of Software Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 345–355, 2000.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall International, 2005.

[Fle81] J. L. Fleiss. *Statistical Methods for Rates and Proportions.* Wiley, New York, 1981.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Object Technology Series. Addison-Wesley, June 1999. With contributions by Kent Beck, John Brant, Willima Opdyke, and Don Roberts.

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

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GJ08] Tanja M. Gruschke and Magne Jørgensen. The Role of Outcome Feedback in Improving the Uncertainty Sssessment of Software Development Effort Estimates. *ACM Trans. Softw. Eng. Methodol.*, 17(4):1–35, 2008.

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

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

[JM01] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation.* Springer, 2001.

[Jør05] Magne Jørgensen. Evidence-Based Guidelines for Assessment of Software Development Cost Uncertainty. *IEEE Transactions on Software Engineering*, 31(11):942–954, 2005.

[JTM04] Magne Jørgensen, Karl Halvor Teigen, and Kjetil Moløkken. Better sure than safe? Over-confidence in judgment based software development effort prediction intervals. *The Journal of Systems and Software*, 70(1–2):79–93, February 2004.

[KL99] Fred N. Kerlinger and Howard B. Lee. *Foundations of Behavioral Research.* Wadsworth Publishing, 1999.

[Koc04] Richard Koch. *Das 80/20-Prinzip. Mehr Erfolg mit weniger Aufwand.* Campus Verlag, 2004.

[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.

[MC05] Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[McC98] Steve McConnell. *Software Project Survival Guide*. Microsoft Press, 1998.

[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.

[ML06] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform – Designing, Coding and Packaging Java Applications*. Addison Wesley, 2006.

[NR69] Peter Naur and Brian Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO Science Committee*. NATO Scientific Affairs Division, January 1969.

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

[Roy70] Winston W. Royce. Managing the Development of Large Software Systems. In Winston W. Royce, editor, *WESCON*, volume 8, pages 1–9, August 1970.

[Run03] Per Runeson. Using Students as Experiment Subjects - An Analysis on Graduate and Freshmen Student Data. In *EASE'03 - 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.

[Sch08] Michael Schier. Adoption of Decision Deferring Techniques in Plan-driven Software Projects. Master's thesis, University of Innsbruck, 2008.

[Sta08] NIST/SEMATECH e-Handbook of Statistical Methods. http://www.itl.nist.gov/div898/handbook/, Juli 2008.

[TD07] William Trochim and James P. Donnelly. *The Research Methods Knowledge Base.* Atomic Dog Publishing, 2007.

[Tri08] Hannes Tribus. Extending the Research Prototype Alaska (Agile and Traditional Planning Simulator) through Flexible BIRT-based Experiment and User Protocols, 2008. Bachelor Theisis, Univ. of Bolzano.

[WRH+00] Claes Wohlin, Per Runeson, Martin Hösrt, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering.* Kluwer Academic Publishers, 2000.

[WRZW09] Barbara Weber, Hajo A. Reijers, Stefan Zugal, and Werner Wild. The Declarative Approach to Business Process Execution: An Empirical Test. In *International Conference on Advanced Information Systems*, June 2009.

[YBY+07] Da Yang, Barry W. Boehm, Ye Yang, Qing Wang, and Mingshu Li. Coping with the Cone of Uncertainty: An Empirical Study of the SAIV Process Model. In Qing Wang, Dietmar Pfahl, and David M. Raffo, editors, *Software Process Dynamics and Agility, International Conference on Software Process, ICSP 2007, Minneapo-*

lis, MN, USA, May 19-20, 2007, Proceedings, volume 4470 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 2007.

[Zar98] Jerrold H. Zar. *Biostatistical Analysis*. Prentice Hall International, New Jersey, 1998.

[Zug08] Stefan Zugal. Agile versus Plan-Driven Approaches to Planning - A Controlled Experiment. Master's thesis, University of Innsbruck, October 2008.

[ZW97] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental Validation in Software Technology. *Information and Software Technology*, pages 735–744, 1997.