RWTH Aachen

Department of Computer Science
Technical Report

Empirical Studies for the Application of Agile Methods to Embedded Systems

Dirk Wilking

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2008-19

RWTH Aachen · Department of Computer Science · December 2008

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Empirical Studies for the Application of Agile Methods to Embedded Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des akademischen
Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Inform. Dirk Wilking
aus Osterholz-Scharmbeck

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr.-Ing. Ulrik Schroeder

Tag der mündlichen Prüfung: 13.11.2008

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Dirk Wilking
Lehrstuhl Informatik 11
wilking@informatik.rwth-aachen.de

Aachener Informatik Bericht AIB-2008-19

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

Agile Methods are a collection of software engineering techniques with specific differences to traditional software engineering processes. The main differences consist of rapid, cycle based development phases setting the focus of attention on feedback of the source code being developed. The results taken from user feedback, software reviews, or other forms of software evaluation are used as a basis for changes which comprise for example corrections of the user interface or the adaption of the software architecture. Based on single techniques taken from Agile Methods, their application to embedded systems software engineering is empirically evaluated in this thesis.

The experiments and studies which have been conducted comprise the techniques of *refactoring*, *short releases*, and *test driven development*. The results hint to inconclusive effects. For example it could be shown that a constant effort for functional work is achieved by using the short releases technique, but its impact on the resulting software remains difficult to assess. For refactoring a reduced consumption of memory was found, but this effect was created by an overhead for applying the refactoring technique itself.

The effect of agile techniques appears to be inferior to individual software development ability of participants in terms of factor strength. Consequently, the second part of the thesis aims at creating variables for the purpose of experiment control. Variables comprise *C language knowledge* and *viscosity* measuring a participant's level of reluctance to change a fragment of source code.

An additional experiment consists of the replication of the N-version programming experiment by Knight and Leveson. The original experiment of independence between two program failures has been extended by an additional factor of hardware diversity. By using different hardware platforms, it has been expected to create mutual independent failures which is not approved by experimental observations.

Acknowledgments

I would like to thank Prof. Dr. Stefan Kowalewski for supporting the overall and sometimes very ambitious aim of this thesis. Prof. Dr. Ulrik Schroeder gave kind advice concerning the human centered approach. I would like to thank Prof. Dr. Dr. Wolfgang Thomas and Prof. Dr. Horst Lichter for participating in the dissertation committee.

The members of the embedded software chair must be mentioned for all the fruitfull discussions and great cooperation. Especially the first three Ph.D. students, namely Dr. Bastian Schlich, Dr. Falk Salewski, and Daniel Klünder were a strong point of motivation.

Finally, I would like to thank all my diploma students. Umar Sherwani, David Schilli, Axel Janßen, Sven Abeln, and Ahmad Afaneh amongst others had a strong influence on the results and direction of this thesis.

Contents

I Introduction 1

1 Motivation 3

2 Thesis Synopsis 5
- 2.1 Research Question . . . 5
- 2.2 Research Methodology . . . 6
- 2.3 Thesis Outline . . . 7
- 2.4 Bibliographic Notes . . . 8

II Experiments on Agile Techniques 9

3 Overview of Agile Methods related Experiments 11

4 Planning of Embedded Software Projects 13
- 4.1 The Technique of Short Releases . . . 13
- 4.2 Design of the Experiment . . . 14
  - 4.2.1 Variables and Measurement . . . 14
  - 4.2.2 Hypothesis . . . 15
  - 4.2.3 Procedure . . . 15
  - 4.2.4 Participants . . . 16
  - 4.2.5 Threats to Validity . . . 16
- 4.3 Analysis . . . 17
  - 4.3.1 Quality of the Survey Data . . . 17
  - 4.3.2 Group Differences . . . 17
  - 4.3.3 Main Hypothesis . . . 19
  - 4.3.4 Explorative Analysis . . . 19
- 4.4 Experiment Results . . . 24

5 Refactoring 27

5.1 The Technique of Refactoring . . . 27
5.2 Design of the Experiment . . . 28
5.2.1 Variables and Measurement . . . 28
5.2.2 Hypotheses . . . 29
5.2.3 Procedure . . . 29
5.2.4 Participants . . . 30
5.2.5 Threats to Validity . . . 31
5.3 Analysis . . . 32
5.3.1 Main Hypothesis . . . 32
5.3.2 Analysis of Additional Variables . . . 35
5.3.3 Experiment Power . . . 37
5.4 Experiment Results . . . 38

6 Test Driven Development 39
6.1 Design of the Experiment . . . 39
6.2 Reasons for Failure . . . 40

7 Summary of Findings 41
7.1 Problem of the Human Factor . . . 41
7.2 Results of Experiments . . . 41

III Experiments on the Human Factor 43

8 Overview of Human Factor related Experiments 45

9 Rasch Models 47
9.1 The Dichotomous Rasch Model . . . 47
9.2 The Linear Logistics Test Model . . . 48
9.3 Logit as Parameter Units . . . 49
9.4 Benefits and Drawbacks of the Rasch Model . . . 50

10 C Knowledge 53
10.1 The Concept of C Knowledge . . . 53
10.2 Design of the Experiment . . . 54
10.2.1 Variables and Measurement . . . 54
10.2.2 Hypothesis . . . 54
10.2.3 Procedure . . . 55
10.2.4 Participants . . . 55
10.2.5 Threats to Validity . . . 56

10.3 Analysis . . . 58
10.3.1 Test Revision . . . 60
10.3.2 Assessing Validity . . . 61
10.4 Experiment Results . . . 61

11 Viscosity **65**
11.1 The Concept of Viscosity . . . 65
11.2 Design of the Experiment . . . 65
11.2.1 Variables and Measurement . . . 66
11.2.2 Hypothesis . . . 67
11.2.3 Procedure . . . 68
11.2.4 Participants . . . 70
11.2.5 Threats to Validity . . . 72
11.3 Analysis . . . 73
11.4 Experiment Results . . . 75

12 Uncertainty **77**
12.1 Uncertainty within Software Engineering . . . 77
12.2 Relation to Agile Methods . . . 78
12.3 Design of the Meta Study . . . 78
12.3.1 Variable and Measurement . . . 79
12.3.2 Procedure . . . 80
12.3.3 Participants . . . 82
12.3.4 Threats to Validity . . . 83
12.4 Analysis . . . 83
12.4.1 Overview . . . 83
12.4.2 Non-Productive Effort . . . 84
12.5 Results of the Study . . . 87
12.5.1 Uncertainty in Processes . . . 87
12.5.2 Threats to Validity of Variables . . . 88
12.5.3 Relation to Higher Level Software Engineering . . . 89

13 Summary of Findings **91**

IV An Experiment on Dependability **93**

14 N-Version Programming with Hardware Diversity **95**
14.1 N-Version Programming . . . 95
14.2 Design of the Experiment . . . 96

14.2.1 Variables and Measurement 96
14.2.2 Hypothesis 96
14.2.3 Procedure 97
14.2.4 Participants 98
14.2.5 Threats to Validity 98
14.3 Analysis 99
14.3.1 Independence of NVP with Forced Diversity 99
14.3.2 Replication of the NVP Experiment 100
14.3.3 Assessing the Strength of Factors 102
14.4 Experiment Results 103

V Measurement Tools 105

15 Code Evolution Framework 107
15.1 Example of an Analysis 107
15.2 Code Evolution Data Collector 108
15.3 Code Evolution Data Analyzer 109
15.4 Source Stepper 111
15.5 Module Interface for Data Analysis 112

16 Progress Measurement Environment 115

17 Disturber 119

VI Conclusion 121

18 Lessons Learned 123
18.1 Lessons Learned for Empirical Work 123
18.2 Lessons Learned for Software Engineering 124
18.3 The Black Matter of Software Engineering Experimentation . . 125
18.4 Effect Strength 126

19 Results 129
19.1 Agility and Embedded Systems 129
19.2 Variables 130
19.3 The Perfect Experiment 132

A Basic Methodical Approach 135
A.1 Box Plots 135

A.2 Hypothesis Testing 136
A.3 Test Artifacts 136
A.4 Resampling 137
A.5 T-Test 138
A.6 U-Test 138

List of Tables

2.1 Short summary of experiments 8

3.1 Overview of agile experiments 12

4.1 U-Test to assess initial differences in performance 19
4.2 U-Test concerning the treatment influence on the different variables 19

5.1 Power calculation of a difference in means of 12 seconds for different sample sizes N 37

9.1 Coding of correct and incorrect answers 47
9.2 Excerpt of a q-matrix for the LLTM 49

10.1 Item parameters and fitness values 58
10.2 Item parameters and fitness values for revised test 61

11.1 Questions and the according item numbers, resembling η parameter used in the result section 67
11.2 Different models compared 74
11.3 Resulting parameters concerning the twelve questions were put in order by easiness 74
11.4 Resulting parameters concerning libraries. Parameters are given relative to first library (MFC) 76

19.1 Variable assessment as used within the thesis 131

List of Figures

4.1 Accumulated data points per group and date - one data point is an entry consisting of a specific task, a duration and a type . . . 18
4.2 Overall fraction of different types of work in percent 20
4.3 Mean and normalized functional part 21
4.4 Mean absolute working time in minutes and normalized fraction of architectural work by treatment. 22
4.5 Mean normalized fractions of defect by treatment 23
4.6 Mean normalized fractions of change by treatment 24

5.1 Box plot of mean fixing time of each participant divided by treatment group, 6 data points per group 32
5.2 Box plots for changed LOC per version categorized by 6 data points per treatment 33
5.3 Box plots for fraction of development time compared to first version per modification categorized by 6 data points per treatment 34
5.4 Bootstrap simulation of mean memory difference 35
5.5 Accumulated occurrences of refactoring techniques for 6 participants 36

9.1 Item characteristic curves of items one and eight form the C knowledge experiment depicting the ogive function 51

10.1 Histogram of the number of years the participants were programming 56
10.2 Frequency of background categories for participants 57
10.3 Goodness of fit plot for two separated groups by median of person parameter 59
10.4 Boxplots for parameter estimates of C knowledge versus years of programming 62

11.1 Background of participants 70
11.2 Libraries known to participants 71

11.3 Goodness of fit plot for β parameters of the basic Rasch model based on randomized groups 72
11.4 Goodness of fit plot for η parameters of the LLTM based on randomized groups 75

12.1 Example 1 of changes in lines of code - organizational information stripped due to length of the line 79
12.2 Example 2 of changes in lines of code - organizational information stripped due to length of the line 80
12.3 Steps to analyse similar lines in a file 81
12.4 General data for each study 84
12.5 Percentage of effort deleted in each study 85
12.6 Events of deleting source code over all studies 86
12.7 Plot of deleted effort and relative project time 87
12.8 Comparison of the effort spent on productive (final) and non-productive (deleted) lines 88

14.1 The simulation process 100
14.2 Histogram of independent model simulation 101
14.3 Strength assessment of different factors of influence 102

15.1 Accumulated complexity/McCabe metric over time taken from TDD experiment 108
15.2 Structure of stored experiment data 109
15.3 Graphical user interface of the source stepper applications . . . 112
15.4 Effort spent on different aspects in N-version experiment 113

16.1 Change of functionality by version for two exemplary implementations 115
16.2 Changes of instructions and branches for two exemplary implementations 117

A.1 Overview of box plot elements 135

Part I

Introduction

1 Motivation

The principles of the software engineering process are subject to frequent changes concerning the general paradigm. As pointed out by Boehm [9], the process of creating software was considered as crafting in the nineteen-sixties, formally definable in the nineteen-seventies, productivity oriented in the nineteen-eighties, concurrent vs. sequential in the nineteen-nineties, and agile and value based since the year 2000. The aim of this thesis is to increase findings for agile methods in the special area of embedded software engineering and thus follows the course of current paradigms as defined by Boehm.

Agility in Software Engineering is focussed on a multitude of aspects. Regarding embedded systems, continuous feedback of the software product's functionality, early testing, and focus on simple design appear as viable effects for the development process. Especially the design and implementation of sophisticated architectures like AUTOSAR for the automotive domain or CORBA for distributed systems might benefit from these effects. In order to evaluate agile methods for embedded systems, three different techniques were compared using an experiment for each. The reason for this approach originated from the internal structure of agile methods being based on single techniques. The most important representative of agile methods is Extreme Programming introduced by Beck [7]. It consists of twelve individual and different techniques. As each technique is of different importance, only techniques relevant to embedded systems were chosen. Thus the techniques of refactoring, test driven development and short releases were regarded to have a high relevance as explained in Section 2.1. The benefits of agile methods are linked to the individual techniques and are rarely subsumed for the entire family of methods. Refactoring as introduced by Fowler [28] has the aim of increasing and maintaining a good internal software structure with a simple design. This is supposed to reduce the effort needed for changes and to increase maintainability of the software. Test driven development is regarded to increase software design quality due to usage of structures before they are created. Another benefit is thought to be an increase in reliability of the resulting software. Especially the creation of a test collection with a sufficient code coverage is in support of the embedded systems need to fulfill the quality of reliability for a given functionality. Short releases are considered to be one of the main differences

to traditional software development methods. The aim of this technique is to gather rapidly and continuously feedback from users and stake-holders of a software project in order to increase overall perceived quality. This might support the aspect of configurability often found for embedded systems software development. By identifying solutions dynamically during the project lifetime their design quality might be reviewed during the course of the project. The aim would be to find a small software design allowing to implement the required number of configurations.

Combining all these single effects into one paradigm would result in a reasonable strong software engineering methodology. The required precondition is that every single technique has the benefits that are propagated. This leads to an individual assessment of techniques which was done within this thesis.

The execution of single experiments revealed problems in the area of noise-induced errors. Although experiment control and design became rigid, results suffered from these disturbance effects. The origin of these effects was supposed to be generated by specific human traits which were not measured directly. The definition and measurement of participant related variables lead to a human centered approach during the second part of the thesis. A general shift to a human centered paradigm guiding software engineering as proposed by Cockburn [19] was omitted, but measuring the human factor for the purpose of control appeared inevitable. Although "changing" this factor is not possible for software engineering projects in general, controlling this aspect to reduce noise-induced errors is regarded critical for experimentation. Finally, when assessing the effect strength of techniques proposed by software engineering and the characteristics of experiment participants, the question is raised if the the former surpasses the later.

2 Thesis Synopsis

2.1 Research Question

Agile methods are a generic term for a collection of software development methods which focus on short planning and cycle based execution [2]. The methods are a controversial topic that have been evolved over the last two decades (cf. Abrahamsson et al. [3] for a short history). Well-known methods belonging to this family are *Extreme Programming* [55], *Scrum*, the *Crystal Family* of methodologies and *Feature Driven Development*. An important agile method is Extreme Programming consisting of twelve single techniques to be applied within a project. The techniques are:

- On-Site Customer: Collaborative software development with the customer
- Planning Game: Repeated planning meetings "in the large"
- Metaphor: List of terms taken from the customer's domain
- Short Releases: Small steps in terms of functional software increase
- Testing: Collection of regression tests
- Simple Design: Implementation of non-complex software architectures
- Refactoring: Continuous structural change to maintain subjectively good design
- Pair Programming: Two programmers work simultaneously on one machine supervising each other's works
- Collective Ownership: Every developer is responsible for every source code aspect and no individual responsibilities exist
- Continuous Integration: Source code changes are gathered in small iterations in a source code repository

- Coding Standards: Source code layout and naming regulations are gathered and maintained
- 40 Hour Week: This technique tries to enforce productive work by reducing meetings and controlling overtime

Williams et al. [90] state that empirical evaluation of single techniques is an acceptable approach to increase overall knowledge of agile methods. According to this proposal a subset of these techniques is used in this thesis.

In general, agile methods are a rather well covered area concerning empirical evaluations, although other sources speak of a need for even more experimentation [3]. The technique that received the most research attention is pair programming as reported by [1]. Because of this attention and a rather small effect of the pair programming technique as shown by a meta-analysis of Müller et al. [64], it was omitted in this thesis. Short releases, refactoring and test driven development were the techniques I considered to have the strongest effect. In addition the application of these techniques could be controlled rather easily. Other techniques like *on-site costumer* or the *planning game* were considered as strong effects, but creating a realistic environment to test them would have been difficult as they require long projects with bigger project groups. As this environment was not feasible, the direct programming techniques presented above were evaluated.

2.2 Research Methodology

The decision to work empirically was mainly influenced by Tichy [82]. In order to evaluate software engineering process related techniques, a measurement driven approach appeared as viable. During the second part of the thesis, the focus on person related measurement made it necessary to "borrow" methods and artifacts (as proposed by Singer et al. [79]) from other disciplines which are mainly human and social sciences. Experiment design, treatment, size of participant groups, control group, effect strength and so on are all concepts which were taken from human science and accordingly are rarely found in computer science. Nevertheless the generic concept of experimentation catches hold in all sciences and only variable acquisition was directly influenced by human sciences. The acquisition particularities comprise for example the Hawthorne effect describing a change in human behavior in an experimental situation. Human behavior tends to support the experimenter in showing a significant difference, which in turn makes the study invalid. Other examples taken from human science experimentation are the use of randomization in

order to factor out individual differences in performance which otherwise might blur a given treatment effect. Apart from a variety of computer science articles describing experimentation for software engineering in detail, a good source for software engineering experimentation is given by Bortz and Döring [12].

One complaint about software engineering experimentation is that students are not considered representative for individual software developers. As explained in [80], this type of participants is sufficient for evaluating basic effects or an initial hypothesis which was the aim of this thesis. Höst et al. [42] state that at least last-year software engineering students have a similar assessment ability compared to professional developers. In [15] no general difference could be found for a different programming expertise between these groups. According to these findings students are considered sufficient for the evaluation of basic effects. Looking at this from a different point of view, an effect that only affects the group of professional or expert developers can be regarded as too specialized for an application in software engineering. Incorporating the average programmer into software engineering experimentation appears as a more holistic approach.

2.3 Thesis Outline

The main aspect of this thesis is divided into three distinct experiment families which appear as separate parts. Agile methods related experiments are presented in Part II of the thesis. It comprises the short releases experiment (Chapter 4), the refactoring experiment (Chapter 5), and the test driven development related experiment (Chapter 6). Part III presents experiments which are related to assessing the human factor consisting of the C knowledge experiment (Chapter 10), the viscosity experiment (Chapter 11) and the uncertainty related experiment (Chapter 12). Part IV contains the experiment about N-version programming (Chapter 14). This experiment is not directly related to agile methods but presents a replicated experiment and is included for the purpose of reference. For each of these parts an individual summary is given consisting of a collection of findings and their interpretation. The experiments are presented in chronological order in Table 2.1 with N being the number of participants. The domain the experiment belongs to is given, additionally.

A brief overview of the created tools is given in Part V. They are presented in this thesis because of the tacit knowledge they contain and because of the additional variables they generate within experiments. The thesis is closed by Part VI comprising an interpretation of the results for the application of agile methods in embedded system software engineering, empirical work in general

Name	Domain	Type	N	Chapter
Planning Horizon	Agile Methods	Experiment	28	4
N-version Progr.	Dependability	Experiment	24	14
Refactoring	Agile Methods	Experiment	12	5
Test Driven Development	Agile Methods	Experiment	24	6
C Knowledge	Empirical variable	Experiment	151	10
Uncertainty	Agile Methods	Meta study	51	12
Viscosity	Empirical variable	Experiment	63	11

Table 2.1: Short summary of experiments

and additional observations that are based mainly on experience gathered within the experiments and lab courses that were executed.

Each experiment is presented using the same template. It consists of a background description, an experiment design, an analysis, and a section for result interpretation. The structural presentation of the experiments is loosely based on the template suggested by Wohlin et al. [91].

2.4 Bibliographic Notes

The experiments presented in this thesis were published on different conferences and in a journal. The N-version programing experiment was published in [74]. A slightly more detailed description of the refactoring experiment was published in [88].

Regarding the human factor related experiments, the variable for C knowledge and more details on the underlying Rasch model are given in [89]. A more thorough tool description for the code evolution framework described in Chapter 15 can be found in [87].

Part II

Experiments on Agile Techniques

3 Overview of Agile Methods related Experiments

The following three experiments are directly related to specific techniques of the Agile Methodology. Although a considerable number of techniques exist, the most promising techniques were chosen as described in Chapter 2.1. At first the technique of short releases was tested as this was thought to be the main difference between traditional and agile paradigms. The main treatment of this experiment was to induce frequent planning and result phases compared to one long phase of planning and development. 28 students participated in this experiment which was executed during a lab course. This experiment is described in Chapter 4.

Consequently the presumably second strongest factor i. e. refactoring was empirically evaluated. For this experiment twelve developers were divided into two groups with one group having to apply a controlled refactoring to its program while the second group had to execute a placebo treatment consisting of an additional documentation. Details and results of this experiment are presented in Chapter 5.

Finally the test driven development technique was assessed within another experiment. The aim here was to test if the overall reliability was influenced by this method. In addition to the twelve student taking part in this experiment, it was planned to reuse results from earlier experiments in order to identify a change in reliability. Due to organizational problems concerning the time frame and place of this experiment, only a few students successfully finished this course. The resulting number of six measurement points was insufficient to perform an inference statistical test. Although this experiment failed, details and problems are described in Chapter 6 in order to enhance quality of further studies of that technique.

Table 3.1 presents the properties of each experiment which are important for a statistical assessment. Regarding the size of treatment group N_t, the refactoring experiment was the only experiment with very few students. The number of students for the test driven development experiment was higher because of the existing data being reused. In order to cope with small groups in experiments, the inference test used during the statistical analysis was changed.

Name	N_t	Variable	Source	Inference test
Planning Horizon	7	Effort	Survey	U-Test
Refactoring	6	Time	Measurement	Resampling
Test Driven Development	12	Failures	Test environment	Resampling

Table 3.1: Overview of agile experiments

Resampling (or *bootstrapping*) is a method which reuses the observed results and therefore is able to omit certain assumptions. For example a t-Test has the precondition that the observed data follows a normal distribution which is difficult to assure for small experiment sizes. Due to the fact that resampling easily scales up to bigger sized experiments, this method was used since the second experiment. Another quality assessment of an experiment is given by the measurement variable and its source. Again the planning horizon used a rather rough main variable based on a survey for effort estimation. The refactoring experiment used a stopwatch time measurement while the test driven development experiment used a hardware test environment to count the failures of a system.

4 Planning of Embedded Software Projects

4.1 The Technique of Short Releases

Regarding the different development methods related to Agile Methods [2], similarities between the approaches can be found. The most important similarity consists of short releases meaning that short implementation phases are continuously re-iterated by an intermediate organizing phase. A characterization of this organizing phase leads to a similar significance as the general planning phase of an arbitrary software development. For Agile Methods the target time frame and the specification focus has a length of one (the next) development cycle. The length of these cycles differs strongly between different types of Agile Methods and ranges from two days (feature driven development) to approximately four weeks (Extreme Programming) as depicted in [17].

The experiment presented in the following was executed in order to assess one of the basic differences of agile and traditional methods: short releases of the agile approach compared to long phases of waterfall based methods. The waterfall method is taken as a reference because it is considered as the most prominent representative for traditional methods. The main purpose was to assess the impact of the different planning approaches on the outcome of a software project. The underlying reason to start such an experiment was based on the assumption that errors which were discovered late in a project threaten the project's success. But as stated in Lippert et al. [55], this model of rising costs for changes in a project at a later project stage might be wrong.

The overall aim of the planning phase of a waterfall approach is to build a software architecture and a schedule for the entire lifetime of the project. This approach respectively allows to control the time and the programming tasks for multiple programmers in a project. The Agile Methods' approach of planning consists of a reflection of the system created so far and a shorter planning focused on the next cycle. The intended time frame in conjunction with the different focus is considered as the difference in project execution and named planning horizon in the following. This rather informal term comprised

the main differences in the treatment of the experiment.

4.2 Design of the Experiment

In order to run the experiment, a lab course consisting of 28 students was taken as a platform (cf. [16]). The students were randomly divided into two groups. The first group used a traditional approach, where the design of the system architecture and the implementation planning constituted the initial phase. The second group was asked to deliver a usable and useful system source code every two weeks. The method for assessing the influence of the design horizon on the software development was to compare the time expended for the lab course by the different groups.

4.2.1 Variables and Measurement

Two types of main data collection devices were used. The first one was a log book which had to be filled out biweekly by the students (cf. [41] and [92] for similar approaches). Here a rough description of the work was required with a precision level of minutes. In addition to the general description in a single log book entry, the task's type had to be documented. The possible choices were a functional, a planning, an architectural, a failure removal, or a non-function oriented change task. The overall development time could be calculated from these entries.

The controlled variable during this study was the planning time which was changed in length. The group with a short planning horizon was asked to omit a long time planning and to provide newly programmed source code every two weeks. The other group was asked to provide an architectural description of the system in UML together with a time schedule to program the different modules. While the first group progress was checked directly every two weeks, the second group provided UML diagrams and Gantt diagrams only once at the beginning. After that, no further control of development was made for this group.

Although a multitude of environments and principles for data collection exists (cf. [6], [18], [26], [90]), the approach taken in this work was different. In order to show a general influence of the effect, the data collected dealt with the parameters of the experiment. Therefore this log book oriented data collection can be regarded as a special measurement tailored for this experiment.

4.2.2 Hypothesis

The early creation of a system architecture and a time schedule in the long planning horizon group were assumed to lead to an increased development effort during the late phases of the project. The reason for this assumption was expected to be the focus on non-critical aspects of the development. The main functional part of the system was assumed to be done late in the project and this was thought to impose major changes in architecture, modules and the system in general. Overall this leads to the more general hypothesis of:

$$H_1 : t_L \neq t_S$$

Here t_L denotes the overall development time of the long planning horizon group and t_S the overall development time of the short planning horizon group. According to the principal of falsification, the hypothesis used in the inference test used the counter hypothesis H_0 being

$$H_0 : t_L = t_S.$$

4.2.3 Procedure

The system that had to be developed by the students consisted of a digital signal processor (DSP) and an ultrasonic sensor as used in an automotive environment. The purpose of the system was to provide pre-crash warning with automatic breaking. The functionality to be developed included the following tasks:

- Accessing an ultrasonic range sensor
- Computing the level of danger of collision based on the measured range
- Communicating the range to another DSP using a predefined protocol

The DSP was a C5416 by Texas Instruments. The language used in the project was C++ and the development environment was the standard IDE provided by Texas Instruments DSPs (Version 2.2). The location that was used to develop the software was either the computer pool or the system was developed at home. In both cases no direct monitoring took place. Because of an insufficient number of hardware systems and a lack of working space a simulator was developed and given to the students. This simulator focused on the communication and sensor aspects of the main task. In conjunction with the instruction set simulator from Texas Instruments this environment simulator was able to provide a complete development environment.

4.2.4 Participants

The experiment itself was embedded into a main study period lab course at the RWTH Aachen University. The project lasted 15 weeks. The participants for this experiment were composed of 28 main study period students working in teams of two.

The participation was not controlled directly, instead the students took part in an election procedure allowing them to choose their favorite course.

Ethical issues, discussed for example in [22] and [76], were countered by using the data after the lab course had finished[1], by keeping the results anonymous and by voluntary participation[2].

4.2.5 Threats to Validity

The communication between students was a clear threat as the groups' results could become dependent among each other. This concerns architecture design as well as the approach chosen to develop the software.

The usage of different practices of software engineering, like test driven development, pair programming etc. had to be controlled in order to increase the experiment's internal validity. Except for UML and Gantt chart based modeling no special practices were allowed during the development. The usage of UML and Gantt charts were regarded as a low threat.

The data collection was not controlled directly due to the high number of participants and the length of the experiment. This threat was encountered with a detailed introduction to the logbook using an example allowing the students to use the logbook as it was intended to be used. In addition the logbook artifact was tested during a preparation phase.

There was a considerable number of students who quit the course. Two long planning horizon groups stopped working after about three quarters of the lab course had passed. In addition one long and one short planning horizon group were reduced to only one programmer and therefore have been removed from the data analysis as well.

The language C++ is a threat when programming an embedded system. The reason for that is that C is regarded as the lingua franca for embedded systems. The language C++ was used during this experiment because object oriented languages allow a better design representation of the software architecture using the UML. This was considered an aspect supporting software design in general. The team size with only two developers is very small which is a threat. The

[1] The logbook sheets were collected in an unaccessible box.
[2] One group did not join the experiment.

advantage on the other hand is having an increased number of measurement points supporting the conclusion's validity. Because of this, the team size can be regarded as a trade off. The experience of the developers is a threat as well, although the degree of an advantage in experience is unknown.

4.3 Analysis

As a first analysis step the quality of the biweekly provided survey data is checked. This is followed by tests which are applied on the initial surveys filled out by the participants. The initial surveys were executed in order to control equality of previous knowledge of both treatment groups.

The most important hypothesis was that both treatment groups needed the same effort to finish their project. This is considered the null hypothesis that must be falsified to show an effect.. In addition the biweekly surveys were used to identify differences in the type of the work. The different types of work were regarded as a unique concept. Therefore the final analysis step provides an explorative assessment of these variables.

4.3.1 Quality of the Survey Data

In order to assess overall measurement quality Figure 4.1 depicts the difference in the log book collection mechanism using single log book entries as data points. It presents an overview of the mean data points for each treatment group in a two weeks time frame. The precision is led by the short planning horizon group having about twice the number of data points as the long planning horizon group. This imposes a threat to the internal validity as it is different for the two treatment groups. The overall precision of the data can be expressed using the sum of the points divided by the number of the participating groups for each treatment. It is 88.3 data points per short planning horizon group and 38.8 for the long planning horizon group. Consequently the data collection itself is difficult to use and must be subject to control.

4.3.2 Group Differences

In order to describe the background concerning programming and software engineering experience an initial survey was carried out. Table 4.1 shows the result of a non-parametric U-Test of this survey (cf. Appendix A.6 for details). The test focuses on the equality of the central tendency (or median) of a sample. The Mann-Whitney U-Test is used for independent measurements and the

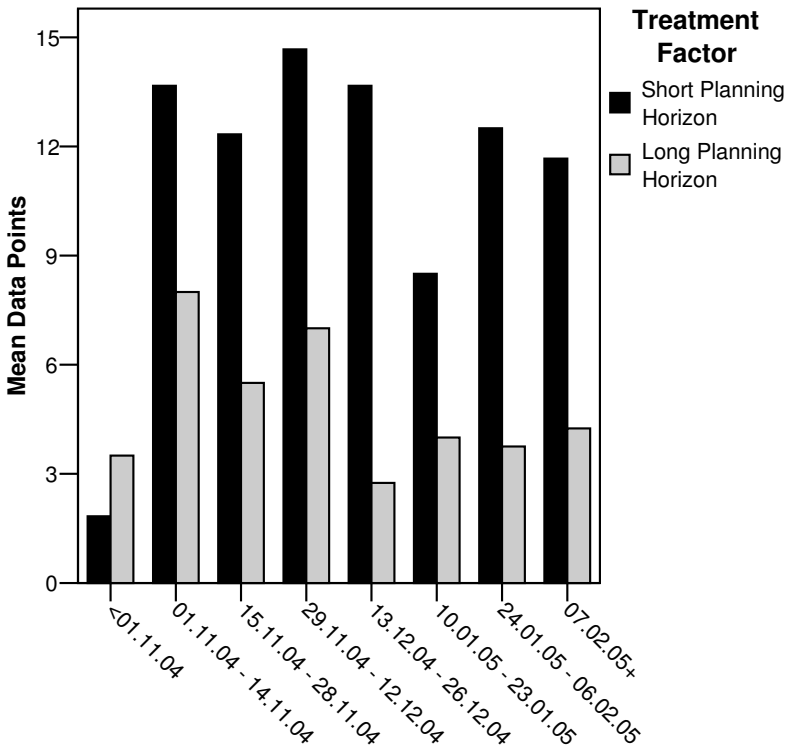

Figure 4.1: Accumulated data points per group and date - one data point is an entry consisting of a specific task, a duration and a type

Wilcoxon W-Test is used if the two measurements are dependent. The values are based on rank sums were each value is transformed to its position in the occurrence of values. The lowest value is transformed to a value of one while the next higher value is transformed to the rank value of two (and so on). By building the sum of values it is possible to identify significant differences for example when one group receives too many low ranked values. The z-value transforms the test statistic to a distribution with a mean of zero and a variance of one. The significance finally indicates the likelihood of the test statistic with values less than 0.05 considered as significant. Two tailed tests aim for equality while one tailed tests aim at directed hypotheses.

The table points to a significant difference (0.04) for the variable "Number of Years of Programming Activity". This indicates an unbalanced general programming knowledge between both groups although randomization was used during treatment assignment.

	Good Programmer	Project Work is new to him/her	Knows many Programming Languages	Is not good at C++	Is familiar with Extreme Programming / other Agile Methods	Doesn't know how a SW Project should be carried out	Number of Years of Programming Activity
Mann-Whitney U	30,000	46,500	44,000	38,000	34,000	25,500	24,000
Wilcoxon W	108,000	82,500	122,000	74,000	112,000	61,500	60,000
Z	-1,512	-,120	-,388	-,815	-1,110	-1,822	-2,055
Asymp. Sig. (2-tailed)	,131	,905	,698	,415	,267	,068	,040
Exact Sig. [2*(1-tailed Sig.)]	,181[a]	,910[a]	,792[a]	,473[a]	,305[a]	,082[a]	,069[a]

a. Not corrected for ties.

b. Grouping Variable: Treatment

Table 4.1: U-Test to assess initial differences in performance

4.3.3 Main Hypothesis

The hypothesis of a different overall development time is tested with a U-Test as shown in Table 4.2. The variable T_{sum} is used as the sum of development time for each group. The table shows a non-significant (0.286) difference for this variable. Therefore h_0, which is the hypothesis that both groups needed the same development time, has to be accepted. Accordingly the main hypothesis of this experiment could not be shown.

Test Statistics[b]

	Tsum	NTFunction	NTPlanning	NTArchitecture	NTDefect	NTChange
Mann-Whitney U	7,000	5,000	10,000	7,000	1,000	4,000
Wilcoxon W	17,000	26,000	20,000	28,000	11,000	14,000
Z	-1,066	-1,492	-,426	-1,066	-2,352	-1,711
Asymp. Sig. (2-tailed)	,286	,136	,670	,286	,019	,087
Exact Sig. [2*(1-tailed Sig.)]	,352[a]	,171[a]	,762[a]	,352[a]	,019[a]	,114[a]

a. Not corrected for ties.

b. Grouping Variable: TreatmentFactor

Table 4.2: U-Test concerning the treatment influence on the different variables

Additional work, which was expected to appear in some long planning horizon groups, did not occur. On the other side an advantage of a long planning phase did not have an influence on the overall development time either.

4.3.4 Explorative Analysis

Due to the explorative nature of this study, the area of interest was covered with more variables than actually needed by the main hypothesis. The greater

detail of the time variable is discussed in the following part.

NT Variables

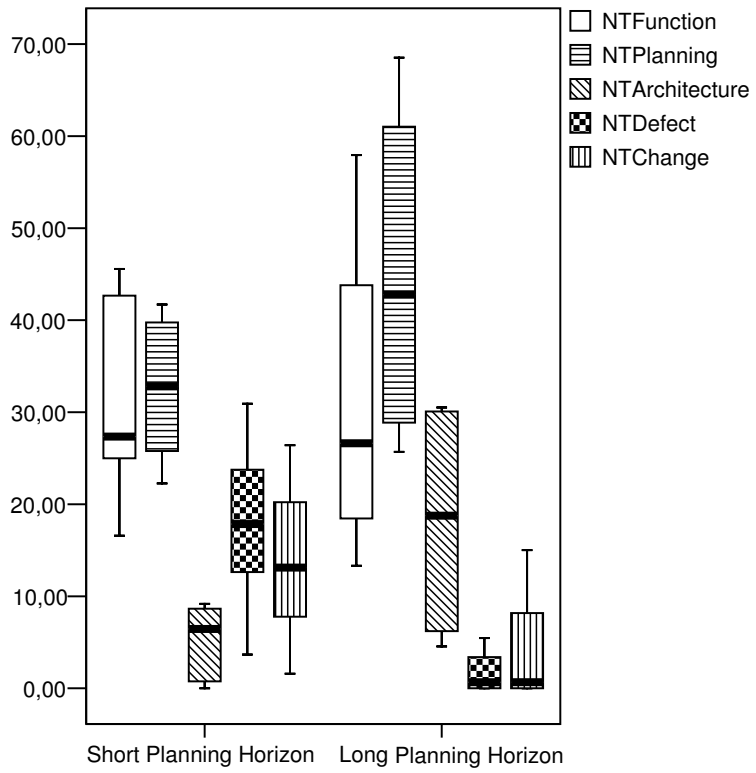

Figure 4.2: Overall fraction of different types of work in percent

The main variables of the survey based data collection were post processed in order to focus on the type of work for each treatment. The T in each variable indicates that the type of work was multiplied by the time needed for the specified task. The N refers to the normalization step, which computed the ratio of the given type of work (in percent) for the two weeks time frame. The result is a fraction describing the effort for each group and type of work. The meaning of the variables is:

- **NTFunction** represents the functional fraction of the work like programming algorithms or peripherals. As for the other variables, it represents the percentual part of work within two weeks.
- **NTPlanning** is the part of planning time used during a two weeks cycle. This includes the time needed to read documents.

- **NTArchitecture** is related to a programming task which changes the general architecture of the system without changing functionality.
- **NTDefect** comprises the process generally considered as defect removal: finding the cause of the failure and fixing the fault.
- **NTChange** indicates the development part of the work that was used to do non-functional and non-architectural programming. This comprises commenting, renaming, and general organizational changes.

Analysis of NT-Variables

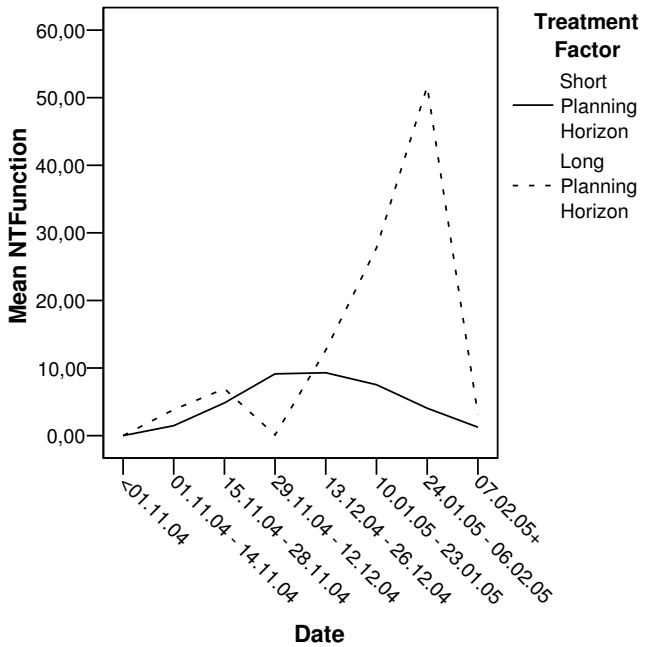

Figure 4.3: Mean and normalized functional part

In order to evaluate the fraction of each work type concerning the overall development process, Figure 4.2 provides a box plot of the different data points. The influence of the different treatments reveals a higher maximum planning fraction for the long planning horizon group which could have been expected. In addition the architecture is a more important task for the planning group which again is caused by the treatment of the according group. The variance for the short planning horizon group seems to be smaller, although the reason for

this might be found in the precision of the data points as discussed in Section 4.3.2.

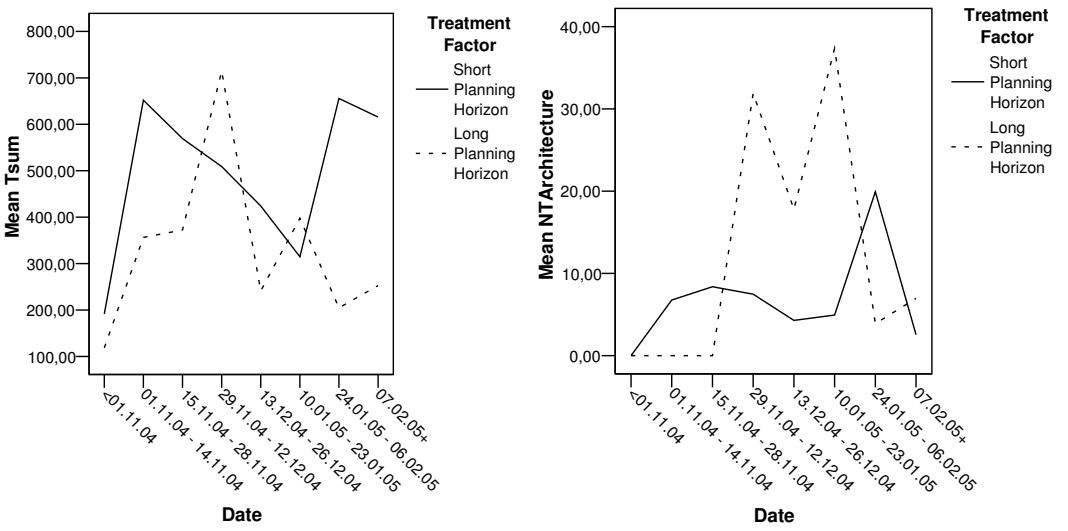

Figure 4.4: Mean absolute working time in minutes and normalized fraction of architectural work by treatment.

The U-Test on the NT-variables presented in Table 4.2 was chosen because the assumption of normal distribution could not be met for the given variables. The table shows a significant difference for the NTDefect-variable. This variable indicates that the fraction of work used for defect removal was significantly different for each treatment. One interpretation of this difference is a more dependable system which is caused by a superior planning and architecture. The result may have been a reduced number of failures for the system. The other reason consists of the overall system quality. It is likely that some groups did a more challenging yet more powerful system than others leading to a different number of system defects. One last aspect is the lack of precise data as mentioned in Section 4.3.1 which means that the real fraction of defect related development is not shown precisely enough for both treatments.

One solution to the problem of precision is a final system quality test. It was suggested in [64] as a control device to assure equality of programming results. Coming too late for this experiment, the proposal was included only informal, but has been established for a different experiment environment as described in [73].

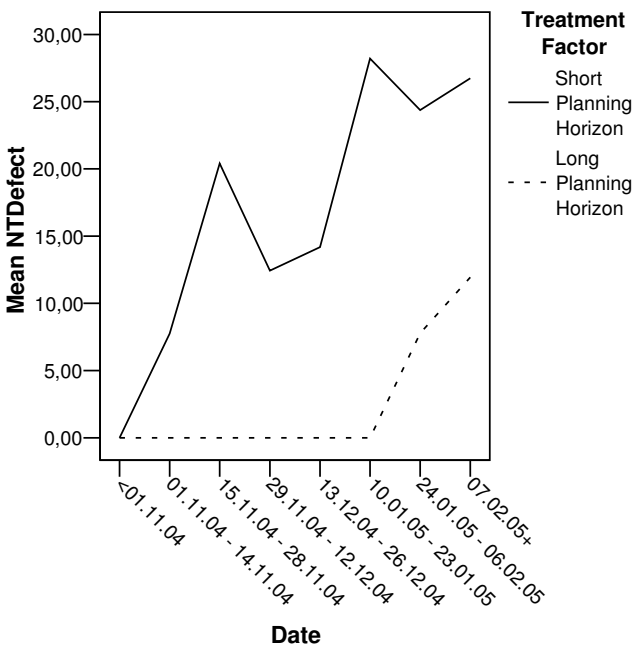

Figure 4.5: Mean normalized fractions of defect by treatment

Analysis of the Development Progress

The mechanism of collecting data regularly over a period of time allows to investigate general development progresses and differences caused by the two treatments. One effect which is assumed to be a problem of long planning horizon approaches is shown in Figure 4.3. A peak of the functional part of the work becomes evident at the end of the course. This indicates an effect, where just before the absolute deadline, a major part of the needed functional work is done using a waterfall method.

An interesting aspect, especially regarding the main hypothesis, is the overall work that was done by both groups. The difference between the two treatments is shown on the left in Figure 4.4. It shows that the mean work time in minutes for the short planning horizon group was much higher in the end than for the long planning horizon group which is a surprising result. The reason for this unexpected observation is that the focus on functional work is considered to reduce unexpected work during the end phase of a project.

In order to obtain a better assessment considering the higher amount of work, the according fraction of work types is shown on the right in Figure 4.4. The fraction of the architectural work as shown in this Figure points out greater effort for the short time planning group. Here the mean value of 20% indicates

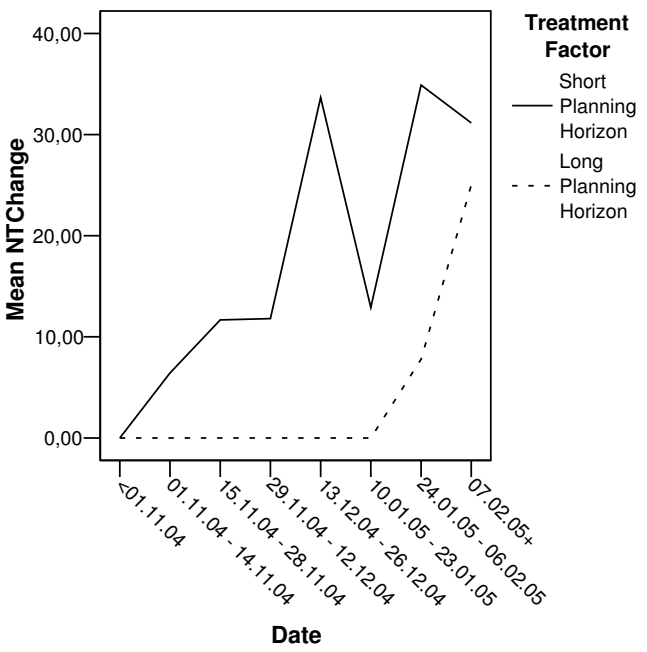

Figure 4.6: Mean normalized fractions of change by treatment

major changes right before the end of the course. The assumption of additional work is supported by Figures 4.5 and 4.6. The amount of non-architectural change programming and defect caused programming appears to be much higher in the short time planning group. Although these effects are an often reported and well known effect ("embrace change"), this might be a reason for the additional work for the short time planning group before the deadline of the lab course.

4.4 Experiment Results

The hypothesis of both groups having a different development effort could not be shown. Thus, neither a short planning horizon nor a long planning horizon changes the time needed to develop a system significantly. Concerning the explorative analysis for this experiment, differences in the types of programming work were found. Differences in functional or defect related development appeared. Although not part of the hypothesis, it is very likely that these differences are imposed by the different methods and that an according effect could be shown.

Technically this experiment is regarded weak concerning environment and

control. Especially the underlying assumption, that short and long planning composed the only factor of influence for the development time is regarded false in retrospect. Experiment control was low as gathering UML and Gantt diagrams is not really countable. Differences in quality as well as detail level could not be controlled. Especially the lack of control of the final version is critical. As only the functioning sensor was sufficient for a running system, neither reliability nor other non functional aspects of the source code were tested. This is regarded highly problematic as differences especially in these areas were enormous and thus the source code of the different groups was not really comparable.

Nevertheless the kind of data collected in this experiment is regarded interesting. As types of programming tasks were gathered, their frequency of occurrence and average length can be considered potentially different for both kinds of software projects. The final question of the impact of different types of work regarding project success could not be answered.

5 Refactoring

5.1 The Technique of Refactoring

Refactoring first mentioned by Opdyke [65] is an important technique on its own, although it is considered part of the Extreme Programming methodology. It can be described as continuous review of the source code with a restructuring of source code according to implicit design rules. The steps to change source code are described in Fowler [28]. Another important aspect is when to apply a refactoring. It is described to be best executed when a developer identifies a source code fragment which does not follow the implicit design rules. The discrepancy in design is subjective in nature and described as "smell" by [28]. After identification of the problem the catalog of changes is used to correct the source code design. The steps to achieve this comprise for example renaming of variables, extraction of new methods or even extraction of complete classes.

According to Mens and Tourwé [60], refactoring is assumed to affect positively non-functional aspects, presumably extensibility, modularity, reusability, complexity, maintainability, and efficiency. Negative effects consist of additional memory consumption, higher power consumption, longer execution time, and lower suitability for safety critical applications.

Research on refactoring mostly includes technical aspects like computer aided application of refactoring. One area of research is refactoring on non object oriented programming languages ([37, 54]). Another common area of interest is the computer aided support of design flaw identification ([58, 70, 77, 84]). The area consists of the actual application of a refactoring step as this is syntactical non trivial ([30, 59]). Empirical work on this topic is rather uncommon. One example for an empirical evaluation is the influence of refactoring on changeability as evaluated in [33] reporting a lower change effort. Other empirical results provide a taxonomy for bad smells as presented in [58].

The results reported by experience reports provide a mixed picture of refactoring. Non-satisfactory results are reported by [67]. The reason for this was given as bad tool support and in addition to this the size of a legacy system being refactored. Code evolution analysis [48] investigates code clones (copied code fragments) as one of the main artifacts minimized by refactoring. For

some code clones no refactoring was available and additionally it was reported that not all code clones should be refactored. One successful application of refactoring is reported in [32]. Here programming performance was increased. A secondary nevertheless interesting aspect mentioned was the compliance to the design principle of information hiding after having finished the refactoring.

5.2 Design of the Experiment

For the refactoring experiment, twelve students had to develop a program based on the same requirements specification. Six students were asked to apply refactoring to their software. The other six students continuously had to document their source code. The later step is regarded a placebo effect treatment to omit the effect that students behave in an experiment conform way (Hawthorne effect [69]). Additionally the disturbance for a person that was created by refactoring was also applied to the control group by using this treatment. Group assignment was done randomly.

Two hypothesis based on non functional aspects were tested. Maintainability was thought to be higher in refactored source code. This was tested by randomly inserting errors in the source code and measuring the time a participant needed to fix the error (thus classified as corrective maintainability [5]). The second hypothesis was that modifiability was higher in the refactored code due to its better internal structure. This was tested by adding requirements to the software and measuring the times and lines of code (LOC) needed to meet these requirements. For both hypotheses the differences in metrics were not used within an inference statistical test. Although this is proposed in [46], the distribution and scale types are regarded as problematic for testing purposes.

5.2.1 Variables and Measurement

The main independent controlled variable was the assignment of a participant to the refactoring or documentation group. The execution of refactoring as well as documentation was done with a tool named disturber (cf. Chapter 17). Using this tool every participant was interrupted every twenty minutes and reminded to execute the treatment task. In case of refactoring a list of design flaws (smells) had to be checked. For the documentation treatment it was needed to document the new software parts.

The dependent variables were lines of code, development time for a new requirement (modifiability) as well as the time to identify and correct errors in the source code (maintainability). The two relevant variables for modifiability

could be measured using the code evolution framework as described in Chapter 15. It gathered login and logout information of participants as well as source code changes written during the time of implementation. For the randomly induced errors of the maintainability test, a simple tool was written which randomly removed lines of code. This removal of source code lines resulted in syntactical or semantical errors. 15 syntactical and 10 semantical errors were created in each version. The time needed to fix them was measured using a stopwatch and the measurement was supervised by a member of the chair.

5.2.2 Hypotheses

As maintainability was assumed to be better in the refactoring group, the time t_{main} needed to find and correct errors induced into the source code was thought to be lower in the group $t_{main_{Ref}}$ compared to the documentation group $t_{main_{Doc}}$. As the opposite had to be rejected, the hypothesis H_0 was formalized as

$$H_0 : \bar{t}_{main_{Ref}} \geq \bar{t}_{main_{Doc}}.$$

The resulting alternative hypothesis was

$$H_1 : \bar{t}_{main_{Ref}} < \bar{t}_{main_{Doc}}.$$

A similar hypothesis was used for modifiability. Only the meaning of t changed, as $\bar{t}_{mod}$ described the mean time to develop additional requirements for the refactoring group. Consequently, the H_0 hypothesis for modifiability was

$$H_0 : \bar{t}_{mod_{Ref}} \geq \bar{t}_{mod_{Doc}}.$$

and H_1 was given as

$$H_1 : \bar{t}_{mod_{Ref}} < \bar{t}_{mod_{Doc}}.$$

5.2.3 Procedure

In this experiment, video tutorials were used to explain details about the environment and the microcontroller to the participants. Two different videos were shown for the treatment consisting of a video about refactoring and documentation. The advantage of using this way of training was that each participant received the same amount of attention and no person or method was favored. After this initial tutorial step, a general survey was executed in order to assess person ability. Questions in this survey were based on experience, language and microcontroller knowledge. To avoid any motivation effect, refactoring was

named reorganization during the active experimentation procedure. Based on a requirement list, programming was done. Each participant had to work in a separate room. This lead to a non continuous development with students working during different times a week.

The main task consisted of a reaction and a memory game to be programmed. The reason for selecting this task was the low domain knowledge required to start development. Different types of hardware needed to be programmed as buttons, LCD, LEDs and hardware interrupts were part of the requirements specification. An acceptance test was executed checking the functionality and requirements as programmed by the participants. In this early experiment the test was manual in contrast to the later available automatic tests used in other experiments as described in Chapter 16.

The refactoring treatment was based on a subset of steps applicable to the programming language C. This was needed as C is regarded a more common and realistic language for embedded systems[1] and C itself has no direct support for object oriented features needed to apply all steps. Thus only non object oriented refactoring steps could be chosen for the treatment. As a special refactoring C macro refactoring was added as discussed in [29] and [30].

The application of refactoring (and the documentation treatment) was controlled by the disturber tool (cf. Chapter 17) which lead to a constant treatment execution every twenty minutes. The controlled interruption was done to remind, assure and consequently control treatment execution. Although the periodic application of refactoring is not natural as refactoring is executed based on a developer's decision, the rigid control was needed because participants were supposed to be new to the technique. As the main reason to start a refactoring is based on a programmer's subjective view, this had to be controlled via checklists based on smells described in [28].

The chair's standard development environment consisting of an ATMEL ATmega16 microcontroller was used for this experiment. The development environment was WINAVR 2 together with ATMEL AVR Studio 3. As it was not part of the main experiment, a header and a C file was provided for the LCD programming.

5.2.4 Participants

Participants of this experiment were twelve computer science students from the RWTH Aachen University. All of them were in the main study period and their participation in the experiment was paid on basis of a regular student contract.

[1] According to a survey on embedded.com over 60 % of embedded developers use C.

The experiment lasted 40 hours and the overall execution of the experiment took three months. The reason for this time span was the number of rooms which had to be used exclusively leading to organizational issues. The project related data were saved on individual network drives and inaccessible for other participants. Based on the type of room interruption from other members of the chair occurred but the frequency was acceptable. Questions and feedback to the participants were handled via e-mail and a specific instant messaging server. Messages and experiment data were archived which had to be accepted and signed by the participants.

5.2.5 Threats to Validity

Although the more detailed validity examination proposed by Wohlin et al. [91] was used in the published experiment description [88], the following text contains the standard validity description of internal and external validity.

The most important threat to internal validity consists of the checklist which is used for refactoring. As it is artificial and not based on the developer's subjective view, the realistic execution of this technique is threatened. The same is true for the periodical application every twenty minutes, which does not resemble a "natural" application. On the other hand a detailed theory regarding the effects of refactoring does not exist. The main idea of "once and only once" suggested by the inherent term "factor" as explained by Fowler [28] is the most precise effect description that can be given. Thus it is not known how unrealistic a checklist is compared to intuition. The usage of maintainability and modifiability contains additional threats to internal validity. First both non functional aspects are described as one of a multitude of affected factors. The modifiability test of adding requirements may be in support of the refactoring technique thus creating a rather artificial setting. One last problem are the few participants consisting of six students per group. This low number would have required an exceptional factor strength of the refactoring treatment. This aspect is discussed in Section 18.4 in detail and tested in the analysis Section 5.3 of this experiment.

The external validity is lowered by the use of the C language as a major part of refactoring steps cannot be applied without object orientation. Here a trade off situation occurred, as the realism of using C for embedded systems is regarded higher. Additionally the principle of "good" micro design is considered as a language independent aspect. The test of adding requirements is regarded realistic, as requirements elicitation and requirements stability are problematic on their own.

5.3 Analysis

5.3.1 Main Hypothesis

Maintainability

The maintainability test consisted of 15 syntactical and 10 semantical errors that were induced into the source code of each participant. The time to locate the error in the source code and to correct it was measured in seconds. Figure 5.1 depicts the mean correction time for each participant of each group as a box plot. Although the refactoring treatment had a slight advantage concerning reaction, results were not significant when tested with a bootstrap inference test for a level of significance of $\alpha = 0.05$. Thus better maintainability is not regarded as different between the refactoring and the documentation treatment.

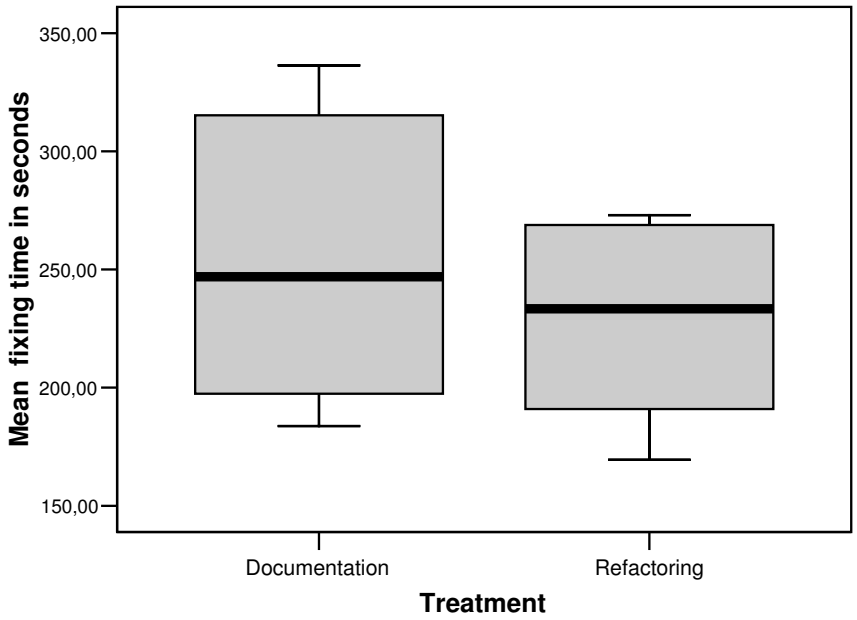

Figure 5.1: Box plot of mean fixing time of each participant divided by treatment group, 6 data points per group

Modifiability

The effect of new requirements to a project was measured using the lines of code metrics (including comment lines). Three kinds of lines of code were

combined: added lines, changed lines and deleted lines. Additionally the time being measured needed to fulfill each new set of requirements. Due to differences in participant's performance, ten completed version were available for the first requirement addition (version 1.1), while nine participants completed version 1.2 and 1.3.

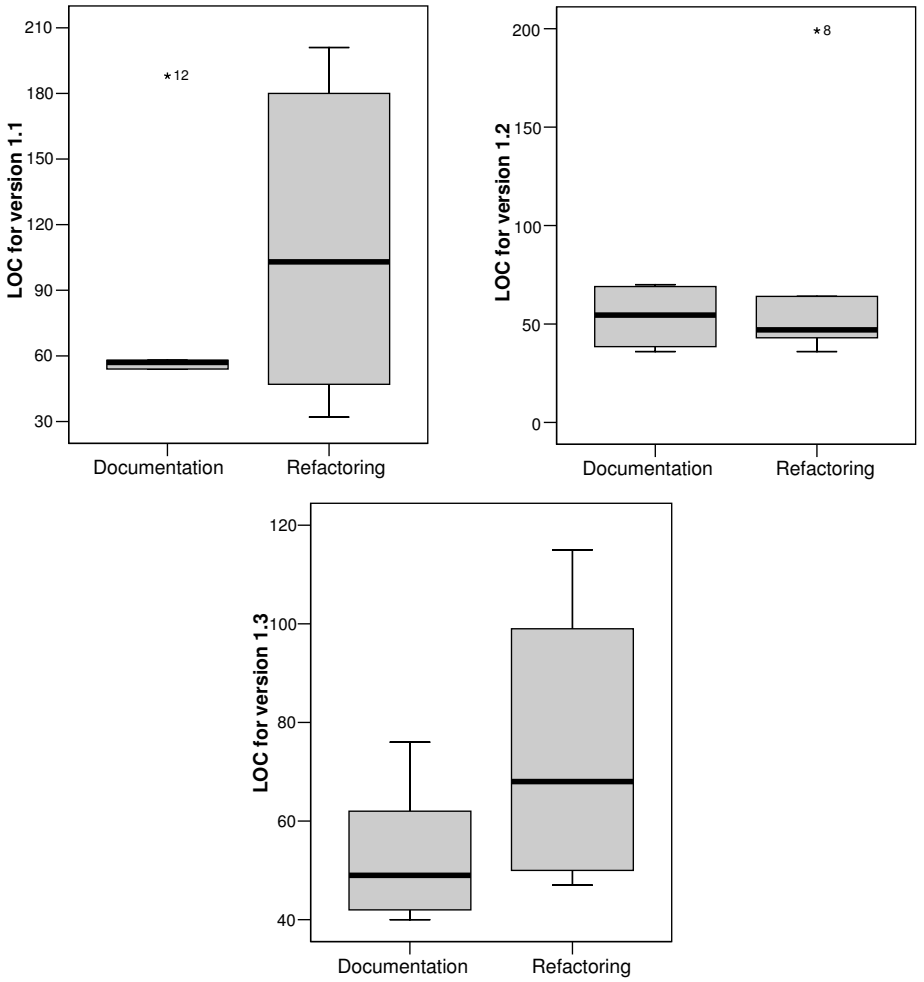

Figure 5.2: Box plots for changed LOC per version categorized by 6 data points per treatment

The change of lines of code for the different versions is shown in Figure 5.2. Obviously the initial hypothesis for modifiability contradicted the observed results as for two new versions, more lines of code were created for the refactoring

group. The median of changes of the refactoring group surpassed the median of changed lines for the control group nearly every time.

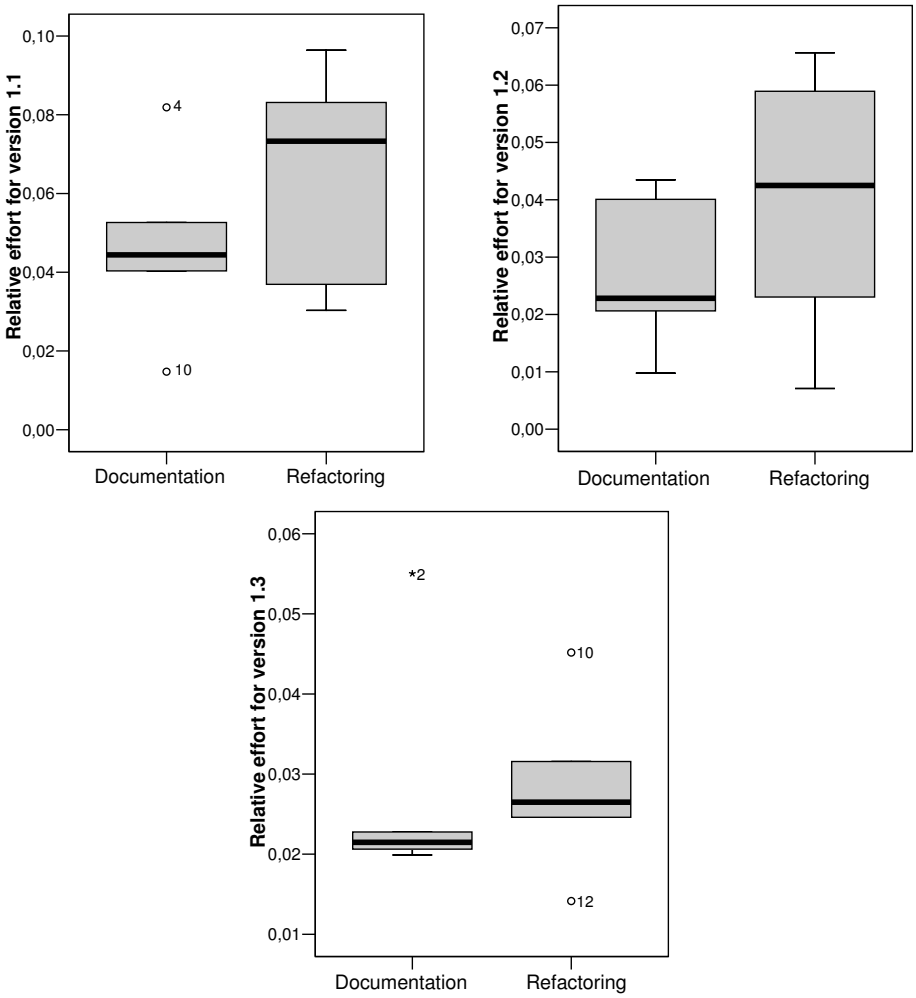

Figure 5.3: Box plots for fraction of development time compared to first version per modification categorized by 6 data points per treatment

The results found above are supported by the time measurements as shown in Figure 5.3. Although both measurements naturally correlate (more lines of code take longer to be programmed), additional effort seemed to be required when applying the refactoring technique. Regarding the difference of time, all three additional versions needed more time to be finished by the refactoring group.

Summing up, refactoring does not appear to support maintainability or modifiability. Apart from an influence on non functional aspects, side effects regarding development effort seem to occur when refactoring is applied. For a more detailed interpretation, see Section 5.4.

5.3.2 Analysis of Additional Variables

As additional variables were collected, an analysis of the results was done more deeply. As hypotheses were not formulated in the beginning, these results had to be considered indicators of possible effects.

The first observation was that the overall size of the resulting source code was not influenced by treatment. The size of the projects varied from 745 lines to 2214 lines of code.

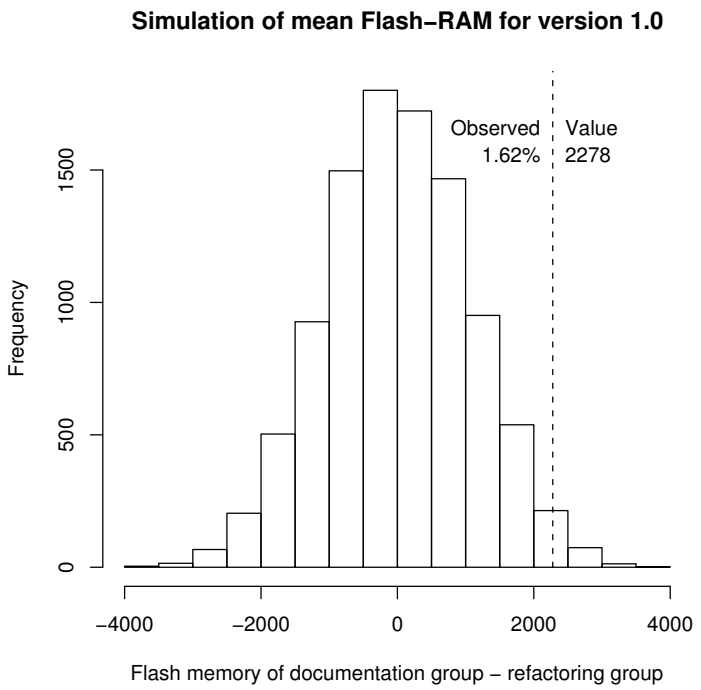

Figure 5.4: Bootstrap simulation of mean memory difference

One interesting difference was observed in the memory consumption between both treatment groups. This value was reported after every compilation and thus could be extracted from the finished versions of each participant. One

memory type of the microcontroller used was a flash memory. This type of memory is used to store program code and constants of the program. In order to compare the differences, a bootstrap simulation [23] was executed (cf. Appendix A.4 for details). The simulation created random groups based on the original data and computed statistics for both groups. By doing this, the likelihood of the observed differences could be assessed comparing this value with the multiple simulated results. Figure 5.4 displays simulated differences as a histogram and marks the observed value. As the value of 2278 only occurs rarely (1.62%), the difference is thought to occur non randomly and accordingly may be caused by refactoring.

The main reason to use a bootstrap technique was the low number of participants and the unknown distribution of the observed values making t-tests and u-tests invalid. Especially small experiments benefit from this inference statistical test [83].

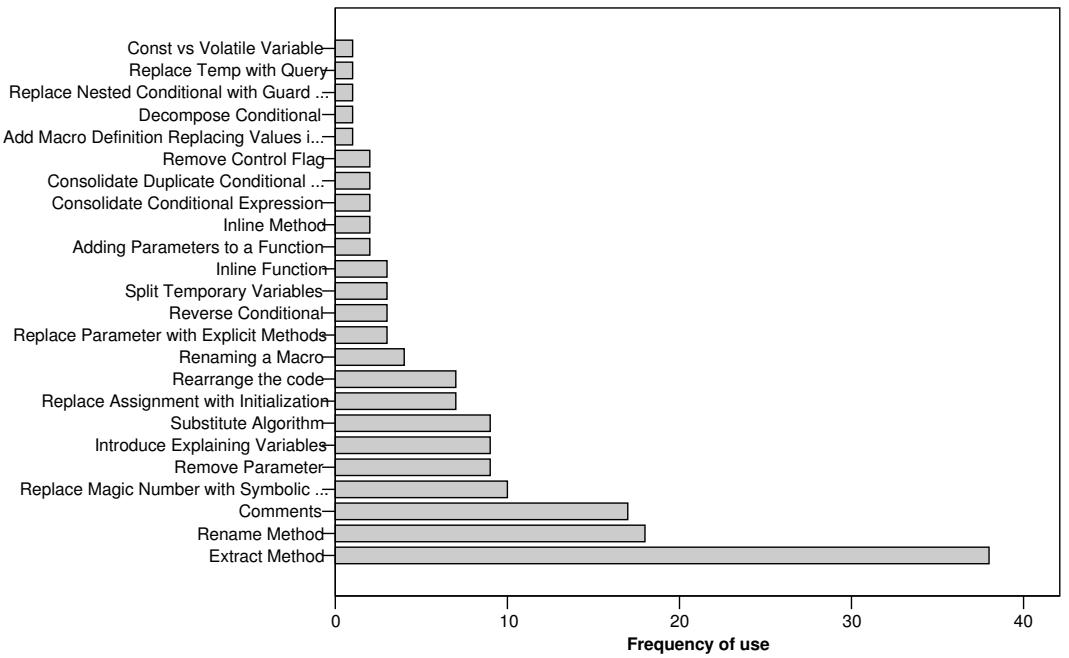

Figure 5.5: Accumulated occurrences of refactoring techniques for 6 participants

The checklists used for refactoring constitute another interesting source of data. As each student filled out the type of refactoring being executed, the frequency of each refactoring could be counted. This is shown in Figure 5.5. Regarding single refactoring techniques, the "extract method" principle

appeared to be the most important refactoring technique. Following that, correct naming of variables as well as adding source code comments were important. All three refactoring techniques aggressively change semantic aspects of source code. The technique of changing a number to a symbolic constant may be important only for embedded systems as single values may be used more often to represent a certain state of the system. The list is regarded as an initial hint on the importance of refactoring for embedded systems.

5.3.3 Experiment Power

As the number of experiment participants with $n = 6$ is regarded very low, a power analysis as described by Cohen [20] is executed. Experiment power describes the probability of rejecting the null hypothesis or it describes the probability of showing an effect at all. The two factors of effect strength and number of participants influence experiment power as described in detail in Section 18.4. The importance of experiment power is considered high as pointed out by [61, 62]. In order to assess power a bootstrap simulation is executed again. As described by Efron and Tibshirani [24] a bootstrap power calculation samples (with replacement) a higher number of participants using the original data of the experiment. The results of a bootstrap are shown in Table 5.1 and present the power p for different sample sizes having a mean fixing time of twelve seconds or more. The basic idea here is to accept th assumption that there is a difference and to test when the increase in group size leads to a significant difference. Starting with 48 participants the experiment power appears to be appropriate. The other factor to reflect is the effect strength which is a value of twelve seconds in this case. The relevance of a difference of twelve seconds is questionable as no other measurements of this type can be used for comparison and overall relevance of this value cannot be evaluated properly. Regarding this type of power analysis, a continuous review of power is proposed in [78] suggesting to stop acquiring new participants when power is appropriate.

N:	6	12	24	48
$p(d \geq 12)$	0.68	0.74	0.83	0.91

Table 5.1: Power calculation of a difference in means of 12 seconds for different sample sizes N

5.4 Experiment Results

A direct increase of maintainability and modifiability could not be shown with this experiment. This may have been caused by the rigid control of the refactoring technique. The enforced application of refactoring every twenty minutes as well as the definition of the refactoring types using a predefined list may have been too artificial and restrictive. On the other hand the effect strength of refactoring is considered to be important. If only applicable in longer and bigger projects, problems might occur as even within this experiment, a certain amount of development overhead appeared. Although these tests were not based on inference tests as no hypothesis was made in the beginning, the additional amount of work in the refactoring groups can be explained by the treatment asking for redevelopment in certain situations. The question is if refactoring pays off at the same amount as it increases development effort. For embedded system projects, which in general are thought to be short, the payoff created by refactoring may not occur at all and thus its benefit appears at least as doubtful.

The only argument in favor of refactoring is the smaller amount of memory needed to store the program text. In case of limited memory this method seems to decrease the size of programs due to better micro design. This effect may become relevant as reducing memory becomes interesting in the area of mass production. Despite these good aspects an assessment of program runtime performance cannot be given with this experiment. It is very likely that increased encapsulation slows down program execution as more function calls are needed.

One general criticism to this experiment might be the use of refactoring without unit tests ([31, 63]). This has been discussed with several practitioners during the planning phase of the experiment. The problem of applying unit testing is that this method itself is thought to have an effect. Thus evaluating whether a difference was caused by refactoring alone would not be possible. Accordingly unit testing was not included in the treatment of this experiment.

6 Test Driven Development

The following experiment describes the last experiment of the agile family. Unfortunately an analysis was not carried out as the number of participants as well as the internal validity was found out to be insufficient. Consequently problems and hints to increase quality of future experiments on test driven development are presented in the following.

6.1 Design of the Experiment

The main idea of this experiment was to reuse results created by the N version experiment as described in Chapter 14 and to compare these "natural" developed programs with a group of developers who applied test driven development. The reason to reuse the programs was that the N version experiment environment (cf. Chapter 14) allowed to assess program failures with a sophisticated test environment. In addition the intended number of twelve students could be directly compared with twelve existing programs developed in the earlier experiment. This would have lead to 24 participants who compared to the twelve developers of the refactoring experiment is a considerable number.

The preparation mainly focused on the development of a unit test environment. Although a multitude of these environments exists, non was suitable for a memory of only 1 kByte. The reason for this was the internal usage of strings which were used to identify and explain the results of the unit test environment. This became problematic as allocation of strings is memory intensive and technically not feasible for very small systems. Accordingly libraries were rewritten in order to use single bit flags to indicate success or failure of a single test.

Other aspects of the experiment were similar to the N version experiment. For example an initial survey was executed. The planned time for this experiment was the same as in the N version. The documentation was similar with an additional chapter explaining the special experiment treatment. Experiment details mainly resemble those described in Section 14.2.

6.2 Reasons for Failure

Three main reasons for failure were identified:

- Organizational problems
- Technical problems
- Insufficient control of important variables

Organizational problems occurred as this experiment was not executed directly at the laboratories of the chair, but rather at an external facility. The problem was that participants had difficulties in reaching that place or, which was worse, did not appear at all. The initially planned number of twelve participants was directly reduced to nine. Additionally, participants tended to give up the course as examinations had to be done. In the end one participant was not able to finish the experiment task leading to a final number of seven students producing usable experiment results.

The main technical problem was the unit test environment which required to supervise a certain memory address (of an array) to check for a test failure. Although one LED was used to provide a visual clue whether a test problem happened, the ease of use of the environment was low. In addition the number of tests was limited by using four bytes of memory for each test (function pointer and result indication flag). Although this lead to an amount of 80 bytes used up by the environment, this represented nearly ten percent of the available random access memory (RAM) of the system which can be regarded a considerable part of the resources.

The last problem was an unpredictable aspect of control. As the students were asked to write tests before developing production code, this had to be controlled during the execution of the experiment. Emphasis was laid on the (technical) environment as well as on a tutorial of how to use it. The actual control of a correct test driven development procedure was not done. The result was a low number of tests, culminating in a very low statement coverage of the unit tests. The statement test coverage was found to be very different between participants reducing the already low number of results even further. At this point data analysis was stopped and the experiment regarded as a failure.

7 Summary of Findings

7.1 Problem of the Human Factor

Apart from individual experiment's results which are discussed in the following, the most important finding was an important but subjective factor. It can be described best as human factor or, to be more precise, the impact of a participant's software development ability on the variables measured in an experiment. A good example for this is the experiment on refactoring where the requirements were given and the time to fulfill them was measured. Although it was natural to measure the time needed for programming the experiment task (devices for time measurement exist and are simple to utilize), the assumption of refactoring having a positive influence on development time did not hold at all. Participants which were perceived as "good" needed less time while developers with a multitude of questions seemed to take longer to finish.

Measurement of the human factor on the other hand was difficult. Perceiving students as fast or slow was based on the feature completeness of their work or because they had challenging questions for the tutor. Both aspects are difficult to measure and although this factor appeared in all experiments, neither it was thought to measure it nor was it possible to do so. The consequence of these findings was the focus on measuring human variables for software engineering as described in Part III.

7.2 Results of Experiments

In retrospect the experiment assessing the impact of a different planning phase suffered from a low variable quality and a general inexperience with experiment execution. This is problematic as the agile technique of short releases is thought to have a strong influence on software projects. Even though experiment quality might be increased by using better control and more sophisticated variable design in the future, the question of the area of effect of short releases is still unclear. Even if development focus is shifted towards functional system aspects, the resulting difference of that change is not easy to predict in terms of variables. One good design may be to fix the development time to a constant

value and measure the amount of fulfilled functional aspects while controlling non functional requirements. This leads to the assumption that failure rates of agile projects are lower which is difficult to define within experiment variables. Another aspect is that the control of a project itself is simpler within an agile project as the project's status is updated more frequently. The relevance of a better project control is questionable.

The refactoring experiment reaches a comparatively higher quality regarding variables and participants. Concerning variables the errors induced in the source code together with a time measurement are considered as very objective and precise. Moreover paying participants enhances experiment control making the results more realistic. The only problem with that experiment is the low number of participants available. In addition the effect strength of applying refactoring appears to be low, especially when compared with the overhead generated. Accordingly refactoring is considered not to be relevant for embedded systems software development.

Summing up the observed impact of agile techniques is regarded as low. During the execution of experiments, disturbance variables seemed to have a greater influence on dependent variables than the controlled variables of the experiment. Consequently a quantification of disturbance variables was tried to achieve as described in the experiment family of the next part.

Part III

Experiments on the Human Factor

8 Overview of Human Factor related Experiments

Regarding the influence of the participant's abilities and knowledge, this aspect is regarded as a disturbance variable to experiment measurement. Unfortunately the impression is raised that the effect size of a person's ability is an order of magnitude higher than the effect of agile techniques themselves. This observation is supported by Grant and Sackman [35] who describe the human factor in having a difference around 28:1, although this factor is reduced to a more trustworthy value of three as reported in a meta analysis executed by [68]. Consequently controlling the disturbance variables using randomization during treatment assignment did not prevent measurement inaccuracy. Moreover the small participant size increased the effect of disturbance variables as randomization may not have lead to equal participant groups in terms of ability.

Regarding the human factor in software engineering, the number of sources for this topic is scarce. One part of research based on the human factor is presented in Karn and Cowling [45], where personality types in projects were identified with a Myers Briggs Type Indicator. A further aspect of human centered research in software engineering is the cognitive aspect found for example in numerous works by Wang (cf. [85, 86]). In this area, software comprehension and reading techniques are important categories. The different approaches are represented in John et al. [43] again. In general, the human factor in software engineering is only covered lightly with several directions of interests. *Human-computer interaction* (HCI) or education related research, where the human factor is much more present, are omitted here as they do not focus on the software engineering process.

As most guidelines for experimentation in software engineering originate from the domain of social and human sciences, the measurement for the experiments described in the following was based on the Rasch model [27] taken from psychometrics. The reason to use this model is its proclaimed psychologic adequateness as it is based on a probabilistic answer scheme. The probability of a correct answer is only based on the person ability and the item difficulty. Other strong reasons to use the Rasch model based measurement are the good interpretability together with a multitude of sophisticated quality control

mechanisms. A detailed description of the two Rasch models used within experiments is given in Chapter 9.

In general, the main problem of determining human related variables does not consist of the model the variables are based on, but on their meaning for the software engineering process. The main question is whether language knowledge, programming experience or general intelligence constitute important person based variables or not. The first experiment thus tried to define a measurement for the variable C knowledge as described in the first experiment in Chapter 10. This was regarded as a conservative decision as domain knowledge to define such a variable certainly exists within computer science. Following this rather simple definition the more interesting variable of *source code viscosity* was experimentally evaluated. Chapter 11 describes details of this variable, which was not only aimed to measure a person's ability, but also allowed to rate and compare the viscosity of different fragments of source codes.

The last study related to human behavior presents a meta analysis of source code progress combining the course of all experiments into one analysis. It was calculated how often lines of code were added and removed later. Lines not appearing in the final version were considered wasted effort. The overall loss of programming time could be assessed because the amount of time which was used to create these wasted lines of code could be estimated. Additionally, the probability that a line of code appears in a final version and the frequency of change of a line of code can be calculated, too. This data acquisition is used to describe the uncertainty which is regarded as an inherent part of a software engineering process.

9 Rasch Models

9.1 The Dichotomous Rasch Model

When working with person based variables one way of variable measurement consists of creating a question based survey to measure the ability of a person concerning a specific variable. As human sciences use *item* as a synonym for *question*, this special term is retained in the following. Rasch scaled variables use the dichotomous values of one and zero to denote correct or incorrect answers to an item. An excerpt of a raw data answer encoding taken from the experiment assessing C knowledge is shown in Table 9.1. Rows represent answers given from one participant while columns represent one item of the test.

Participant	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}	i_{11}	...
71	0	0	0	0	0	0	1	0	0	0	0	
131	0	0	0	1	0	0	0	0	0	0	0	
34	1	0	0	1	0	0	0	0	0	0	0	
114	1	0	0	1	0	0	0	0	0	0	0	
126	1	1	0	0	0	0	0	0	0	0	0	
134	1	0	0	1	0	0	0	0	0	0	0	...
9	1	0	0	1	0	0	0	0	1	0	0	
37	0	1	1	0	0	0	0	0	1	0	0	
70	0	0	1	1	1	0	0	0	0	0	0	
84	0	1	0	1	0	0	1	0	0	0	0	
20	1	1	0	1	0	1	0	0	0	0	0	

Table 9.1: Coding of correct and incorrect answers

Under the assumption that the Rasch property holds for the data, the row sum can be interpreted as the person ability. The reason for this is that a person with a higher ability in the measured variable will be able to answer more items correctly. The same is given for the sum of a column indicating the easiness of an item. The more persons were able to answer correctly, the easier an item is considered to be.

In order to describe the probability of a correct answer p_1, the item difficulty σ_i and person ability Θ_v are needed. The probability of a correct answer of a person v and an item i (a specific cell in Table 9.1) thus is calculated with

$$\log \frac{p_1}{p_0} = \Theta_v - \sigma_i$$

with p_0 as probability of a wrong answer. Accordingly p_1 is given as

$$p_1 = \frac{\exp(\Theta_v - \sigma_i)}{1 + \exp(\Theta_v - \sigma_i)}$$

representing the simplest dichotomous Rasch model.

Algorithms and tests based on this model are described by Fischer and Molenaar [27] and Wright and Masters [93]. As the algorithms themselves are not in the scope of this thesis, they are not presented here. Only the applications purpose and outcome are described when used for an experiment.

9.2 The Linear Logistics Test Model

Regarding the families of Rasch models the linear logistics test model (LLTM) splits the item parameter σ into additional, more detailed parameters. For example in a mathematical test multiplications or additions may be part of a single item difficulty. An example item might ask for the value of x in $2 + 3 * 4 = x$. The LLTM allows to model the fact that item difficulty is made up of the difficulty to perform an addition and to perform a multiplication. The addition and multiplication receive new basic parameters η_j each which are estimated during the parameter calculation. In general mapping of item difficulty σ_i is accomplished by a weighted sum of all basic parameters η_j, which is

$$\sigma_i = \sum_{j=1}^{h} q_{ij}\eta_j - c.$$

The value of h denotes the number of different basic parameters while c is used as a standardization constant (the zero sum property). This general mapping of basic parameters is controlled by the values of q_{ij}. These values are gathered in a (q) matrix which directly provides the basic parameters influencing a single item i. For the viscosity experiment the Table 9.2 shows an excerpt of the matrix used within the experiment.

Each row contains two parameters of influence. The viscosity experiment uses parameters to model the difficulty of different source code fragments which

q_1	q_2	q_3	q_4	s_1	s_2
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0
0	0	0	1	1	0
1	0	0	0	0	1
0	1	0	0	0	1
0	0	1	0	0	1
0	0	0	1	0	1

Table 9.2: Excerpt of a q-matrix for the LLTM

are presented during the experiment and its uses parameters to model question difficulty which are repeated for each source code fragment. First the left part of the table specifies the used question. The repeated questions are indicated by a returning parameter of one for the first column. The right part is used to incorporate the difficulty of the source code. Each question is executed for a source code fragment only once after which the source code (and its parameter) are changed. One constraint for the q-matrix is the linear independence of columns, one question and one source code parameter must be removed.

9.3 Logit as Parameter Units

As the results of the Rasch model based studies are given as parameters based on the scale of this model, a short explanation is given in the following. Parameters of a Rasch model based variable are given in the logit unit. This unit is used to describe the likelihood of a correct answer in the form of

$$Logit : \log \frac{p(X_{vi} = 1)}{p(X_{vi} = 0)}$$

with $p(X_{vi} = 1)$ representing the probability of a correct answer and $p(X_{vi} = 0)$ as the probability of an incorrect answer for a given item i and person v. The log function serves as a symmetrical projection as the original values only range between zero and positive infinity. For example, a probability (of a correct answer) of 0.25 is represented by the logit value of -1.1, a probability of 0.75 is represented by 1.1 logits while a probability of 0.5 has a logit value of 0.

9.4 Benefits and Drawbacks of the Rasch Model

The dichotomous Rasch model has the advantage of a simple principle to create variables accompanied with a variety of post mortem analysis tests. Creating a test consists of finding variable related questions of rising difficulty. Tests allow to check if a given parameter set is sufficient to explain the data or to check if items have a different difficulty within a group of participants. It is possible to identify single items which do not have a sufficient model fit to the Rasch model. By excluding such items the overall model fit can be increased. Examples for this case would be a difficult item that has been answered correctly too often by low ability participants. Comparing different models, additional statistics can be used to assess overall model conformance to the data. One of these statistics is the Akaike Information Criterion (AIC) given as

$$AIC = 2(n_p - \log(L_0))$$

with L_0 being the likelihood of the model and n_p as the number of parameters. This statistic devaluates not only models with a low likelihood but also models with a high number of parameters.

The Bayes Information Criterium (BIC) is another model statistic which directly includes the number of participants N with

$$BIC = -2 \log L + (\log N) n_p.$$

This criterium takes into account that for a high number of possible different answer sets, additional parameters have a very strong effect on the statistic. The BIC prevents over parameterization by increasing the influence of having a higher number of participants. Consequently for a given set of possible models the best models in terms of precision and parameters can be chosen as described in Chapter 11.

Another reason for using this model is its psychological adequateness presented in [72]. Instead of a linear function to describe the likelihood of a correct answer, the Rasch model is based on an ogive curved function. An example for this type of function is given in Figure 9.1. The left item is the easiest item of the C knowledge experiment while the right *item characteristics curve* (ICC) shows item eight having a medium difficulty. The reason to use this type of function is that the person parameter has the most influence on the correctness of the answer in the middle of the function. For the outer regions the probability of correct answers converges one and zero respectively. The reason for the convergence is that even very simple questions can be accidentally answered incorrectly or that very difficult questions by chance may be

answered correctly. Thus the Rasch model provides a good representation of the participants' underlying answering behavior.

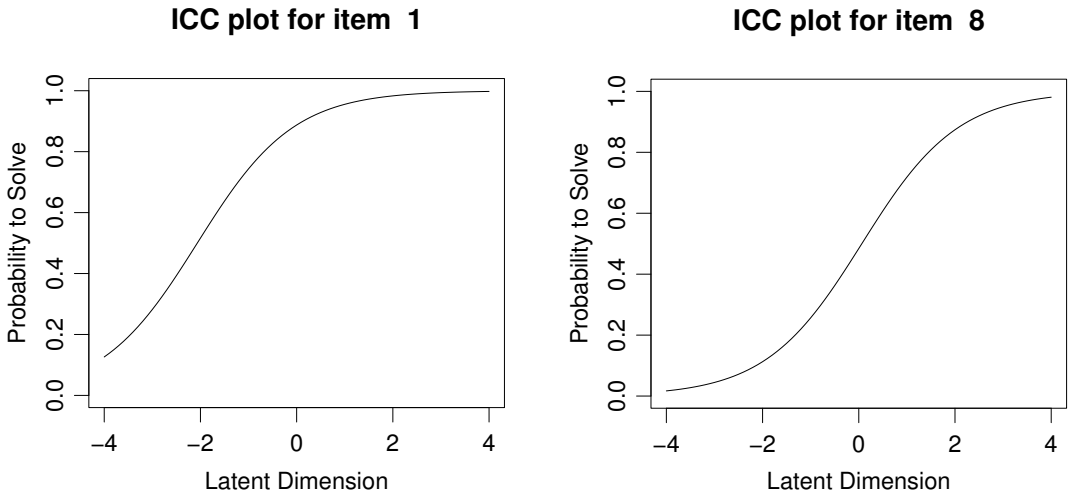

Figure 9.1: Item characteristic curves of items one and eight form the C knowledge experiment depicting the ogive function

One of the drawbacks of Rasch models is the vast number of derived models. Although the simple Rasch model is a special case of all other models, they are regarded as derived as they add additional parameters allowing a more complex model to be tested against the data. The potential of having more types of models is problematic as the degrees of freedom increase. This makes it difficult to assess models with higher degrees of freedom with a better model fit compared to simpler models with a weaker fit. The different families of Rasch models allow for example to use ordinal answer categories, integration of qualitative and quantitative measurement or a three factorial design instead of two parameters (cf.Rost [72] for an exhaustive discussion of models).

10 C Knowledge

10.1 The Concept of C Knowledge

Experience is often mentioned as an implicit factor influencing software development as noted for example by Broy and Rausch [13]. This rule of thumb is somewhat problematic as a person might have developed software systems for a long time while not being really good at it. Even if this rule is true in general, measuring just the number of years a person has spent on developing systems is too imprecise for experimentation. Indirectly, this inaccuracy of the experience variable appeared throughout earlier experiments. As every experiment participant provided a variable of development experience (e. g. number of years, programming languages, libraries used), some correlation regarding development time or other experiment related variables should have appeared. In fact every plot or inference test yielded no hint on a correlation between experience based variables and dependent variables. As an increase in experience is thought to have an effect in general, the quantification of this variable based on single survey questions is simply regarded insufficient. Thus a better quantification of person ability was necessary.

As described in the overview chapter, quantification of "C knowledge" is regarded a conservative decision. Gathering variable related questions appeared feasible, as relevant questions related to the programming language C emerged during practical work. Additionally syntactical as well as semantical particularities of this language are easily found. As the underlying principle of variable generation was new, this domain represented a trade off between concept relevance and feasibility.

Other variables to start with comprised general library knowledge or general programming language knowledge. Both concepts should cover a large area of knowledge and, although more valuable in the end, their definition appeared too ambitious as a first experiment. Finally, another question to bear in mind is which person related variable makes up the most important factor for software engineering experiments. As this includes abstract concepts about a person like viscosity (see Chapter 11), ingenuity, or doggedness, these aspects were out of scope for the first Rasch based experiment.

10.2 Design of the Experiment

10.2.1 Variables and Measurement

Measuring C knowledge was based on technical aspects of the programming language C. For example details of the preprocessor, pointers, the difference between call by reference and call by value, dynamic memory allocation, function pointers, operator precedence were part of the experimental survey. All examples were based on small source code fragments with according questions regarding the functionality of the code. Answers to these fragments were made in an open form to reduce the probability of correct guessing and to increase discriminatory power of single questions. An example for a source code fragment is:

```
Please write down the output of this program.

int digit = 100;
int *No;
No = &digit;
printf("%d", No);
```

In this case dereferencing is the main aspect of the medium difficult question (item 10). The address of variable "digit" is saved to "No". Printing the value of "No" would result in an unknown address as it is not shown in the code fragment and normally done automatically by the compiler. When this was mentioned in the answer of a participant the item was considered as correctly solved. A drawback of this open answering mechanism was that every answer had to be processed manually as an automation was not feasible. Additionally some answers were difficult to judge as correct or incorrect which was partly countered by a strict definition of correct answers.

10.2.2 Hypothesis

As this experiment aimed at creating a variable itself, no real hypothesis could be given as this would require variables on its own. For variables certain aspects are of interest and these were tested within this work. The variable properties of interest were:

- Reliability
- Validity
- Ease of Use

Variable reliability describes the equality of values when the measurement is repeated under the exact same circumstances. Validity for variables describes how well the variable measures the intended concept. Ease of use comprises length in minutes needed to fill out the survey.

10.2.3 Procedure

The experiment started with a pretest consisting of 40 questions that was executed with members of the chair. Redundant questions were identified and removed. As the test was regarded too difficult a subset of rather easy questions was chosen for the final test consisting of 17 questions.

The final questionnaire was executed as an online survey. It was posted on different bulletin boards. Apart from the C knowledge related questions, additional questions were added to assess the background of each participant. These questions aimed on the external validity of the experiment asking for the personal background and prepared a correlation test to the years of programming that participants were asked to fill out, too. The survey was executed anonymously. At the end of the experiment, a prize draw of €50 was done. This was thought to increase motivation to fill out the questionnaire. Filling out the survey took 22 minutes on average.

10.2.4 Participants

As the questionnaire was posted on bulletin boards, the composition of participants was highly influenced by the choice of bulletin boards. As the Rasch model directly includes a person parameter to model differences in ability, one aim was to include professional developers as well as non C programmers. All boards used the German language. The following boards were used to post the questionnaire:

- mikrocontroller.net
- c-plusplus.de
- chip.de
- computerbase.de
- informatik-forum.at
- dormitories of the RWTH Aachen University

The reason to include a bulletin board was based on the community it represented. For example "mikrocontroller.net" was regarded C language oriented as most mikrocontrollers are programmed in C. The bulletin board "informatik-forum.at" is regarded a general technical website consequently comprising participants of medium ability. The non-technical boards like the dormitory boards were assumed to be visited by all kinds of students subsequently providing participants with a low ability in term of C knowledge.

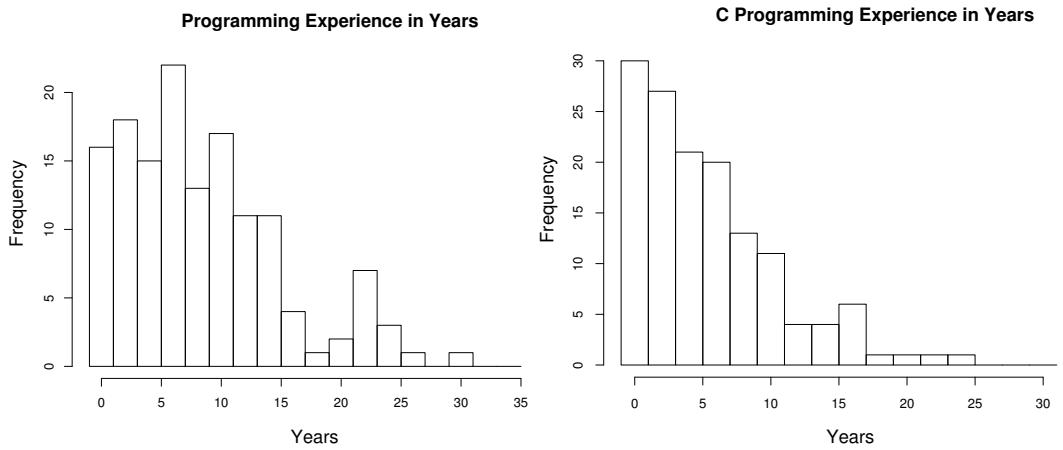

Figure 10.1: Histogram of the number of years the participants were programming

Figure 10.1 provides an impression of the participants' programming background. From the 151 persons that were analyzed in the study a considerable fraction had programming experience of a few years. Compared to general programming knowledge experience with C appears to be lower.

Another valuable background information is the participants' occupation shown in Figure 10.2. As most bulletin boards for this survey were chosen because of their specific background, the majority of the participants were students of a technical specialization. Roughly a third had a professional background increasing the external validity of this study.

10.2.5 Threats to Validity

One problem of internal validity is the lack of control during the execution of the online experiment. A participant may have used additional sources or help to fill out the survey. The risk of such behavior was reduced as it was directly stated that the study needed participants with different ability and thus lack

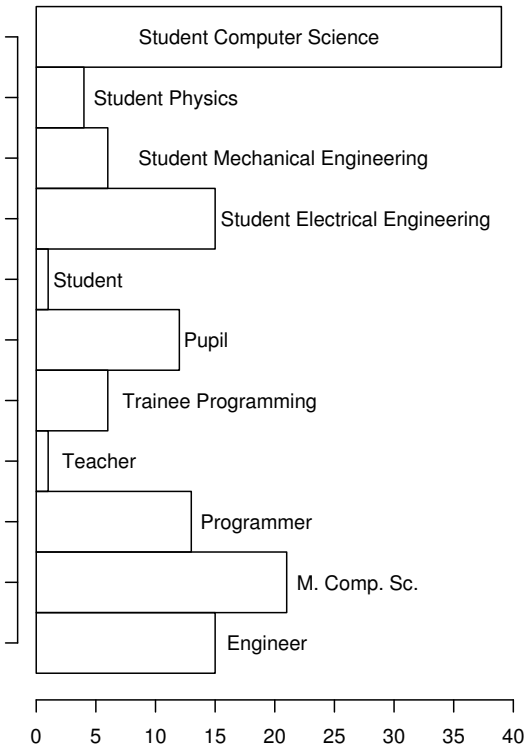

Figure 10.2: Frequency of background categories for participants

of knowledge was not regarded problematic, but beneficial. In order to test correctness of question results, all examples were tested with a C compiler.

External validity of the study depends on the principle measurement of the variable C knowledge. If the questions made for this experiment are not regarded as relevant for C, the variable does not meet its expectations. This is directly circumvented with using real C source code as part of the test questions. In addition some well known problems like pointer problems and semantics issues are used which were thought to be present in everyday problems. Using frequent problems would have increased the validity of each item, but no data describing the frequency could be found. The choice of participants might have been used to influence the overall results, but as described in Section 10.2.4 the resulting background of participants is regarded to cover different levels of language knowledge. Regarding the actual selection of participants no direct selection or control could be achieved as a high number of participants appeared to be preferable.

10.3 Analysis

Parameter estimation was done using the program Multira[1] and eRm package [39] of the statistics language R. Regarding the analysis in R, an additional script for parameter estimation was developed based on the UCON algorithm [93]. Results of the different programs were equal when comparable algorithms were used. Results of the MULTIRA program are shown in Table 10.1.

Item	Difficulty	Standard Error	Infit t	Outfit t
Item 1	-2.09	0.298	0.857	0.916
Item 2	-2.09	0.298	-0.413	-0.108
Item 3	-2.001	0.292	-1.152	-0.470
Item 4	-1.606	0.268	3.065!	2.277!
Item 5	-0.349	0.218	0.222	0.3
Item 6	-0.302	0.217	-2.009	-0.709
Item 7	-0.256	0.215	3.130!	3.279!
Item 8	-0.074	0.211	-0.216	-0.580
Item 9	0.057	0.208	2.373!	1.854
Item 10	0.311	0.204	-1.927	-2.311
Item 11	0.393	0.203	-2.446	-2.400
Item 12	0.555	0.201	-0.557	-1.088
Item 13	0.674	0.200	-2.127	-2.283
Item 14	0.831	0.190	0.797	0.210
Item 15	1.649	0.201	-1.530	-1.608
Item 16	1.729	0.202	-0.604	-0.645
Item 17	2.57	0.224	-0.858	-1.015

Table 10.1: Item parameters and fitness values

Infit and outfit statistics in Table 10.1 are used to describe model conformance of individual items. Both represent chi-square statistics assessing unexpected patterns of answers and observations respectively (cf. [93]). Fit statistics with a negative value are accepted in general, as their discriminatory power is better than required by the Rasch model leading to an item in the direction of a Guttman scale. According to Bond and Fox [10] only values higher than two of the t-standardized statistics are problematic. Their misfit to the model is regarded too low. Problematic values are indicated with an exclamation mark

[1] http://www.multira.de

in the according fit statistics column. One way to handle this kind of items is to simply remove them as presented in the next section.

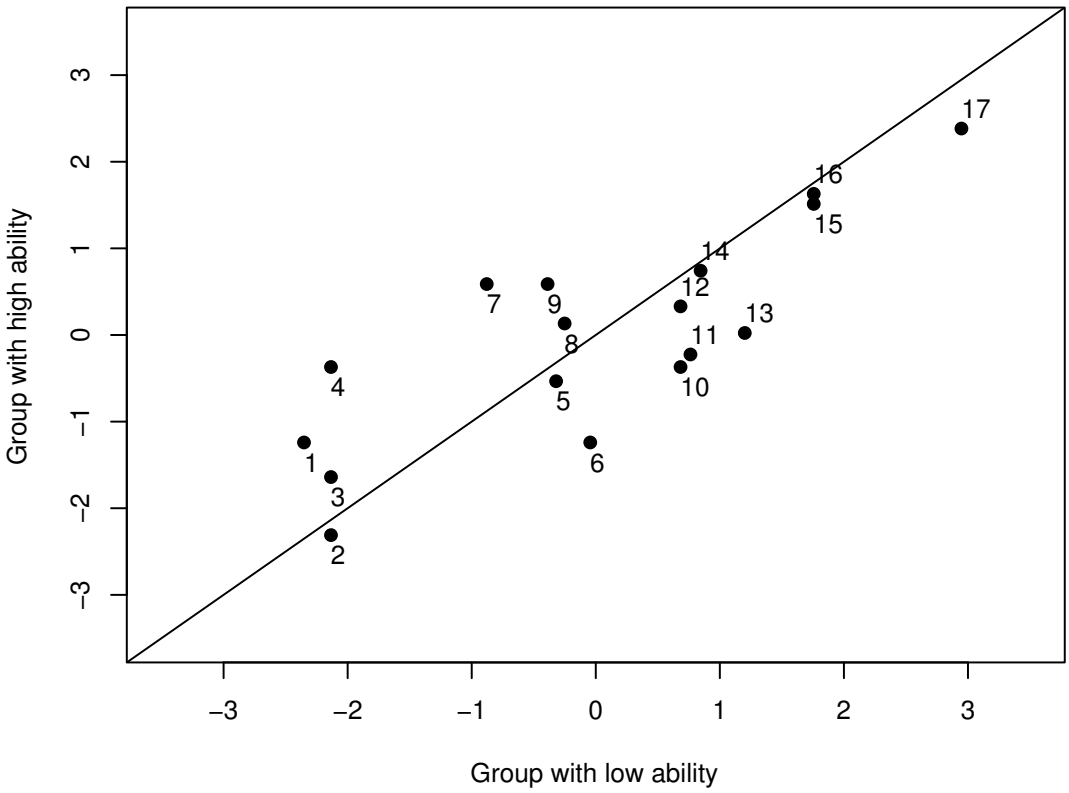

Figure 10.3: Goodness of fit plot for two separated groups by median of person parameter

Another hint on item fitness is provided by a goodness of fit plot as shown in Figure 10.3. This plot is based on two artificial groups based on the person parameter. One group is made up of participants with above median values, while the other group consists of below median participants (median values itself are assigned to the first). New parameter values are calculated for each subgroup and both values are used as a coordinate for the original item. Ideally difficulty is the same for both artificial groups and accordingly points are expected to be situated on a line. The results of the plot strengthen the findings of the fitness statistics indicating a misfit of item four and item seven.

10.3.1 Test Revision

Three items of the original test were removed in order to increase overall variable quality. The reason for low item quality can only be guessed. First, some questions may have been grounded on accurate reading instead of C knowledge. The rather easy item four is a good evidence for having a bad item fit:

```
Please compute the variable solution.
In which order was the term computed?

int solution;
solution = 8 / 4 * 2;
```

As the question aimed at operator precedence, the simple solution of a left to right execution appeared only loosely linked with overall C knowledge ability. Consequently, the question had to be removed. A different problem consisted of a general code interpretation confusion as revealed by item nine:

```
Please write down the value of v.

void funct( int *x ){
  *x = 5;
  x = (int *)malloc(10000);
  *x = 10;
}

MainProgram:
int v = 8;
funct(&v);
print-out of v
```

One of the main problems of this code was the address a value was written to. Due to the variable `x` changing its address, the global value is not overwritten in the last step of the function. As the code is slightly longer, correct solutions are based on a participants ability to follow and memorize some instructions. As this may represent a problem (or variable) itself, the question had to be removed, too.

Table 10.2 shows the results of a parameter estimation with three misfitting items being removed. The parameters do not indicate a bad item fit and thus results were accepted. A slight problem with the second parameter estimation is the lack of easy items. Accordingly parameter values between -1 an -3 should be added in future revisions of the test.

Item	Difficulty	Standard Error	Infit t	Outfit t
Item 1	-2.542	0.324	1.729	1.354
Item 2	-2.542	0.324	0.312	0.918
Item 3	-2.435	0.316	-0.657	0.092
Item 5	-0.517	0.231	1.123	1.648
Item 6	-0.463	0.230	-1.239	-0.398
Item 8	-0.206	0.223	0.636	0.85
Item 10	0.226	0.215	-0.984	-1.699
Item 11	0.318	0.214	-1.940	-2.034
Item 12	0.498	0.211	0.004	-1.156
Item 13	0.630	0.21	-1.241	-1.993
Item 14	0.805	0.209	1.810	0.678
Item 15	1.709	0.21	-1.272	-1.423
Item 16	1.797	0.211	-0.071	-0.102
Item 17	2.722	0.233	0.353	-0.496

Table 10.2: Item parameters and fitness values for revised test

10.3.2 Assessing Validity

The implicit assumption of an increase in ability with years of development existed. Therefore it was tested if a participant's "years of C programming" values (as provided by the participant in the survey) influenced the measured variable of C ability. Figure 10.4 shows boxplots of grouped participants with respect to the number of years of C programming. The asymptotic curve fits the psychological assumption of the Rasch model well. Consequently variable validity is increased by this observation.

10.4 Experiment Results

In general the test is too easy as the person parameter mean value is 0.61. As extreme values consisting of all correct or all wrong answers are not used for the parameter estimation, the range from the lower 3 percent to 93 percent is covered by the test. The final test consists of 14 items and the average execution time is predicted to be 16 minutes. The variable that can be obtained by the test is interval scaled. On the ground of 136 persons' answers values can be interpreted when compared to other persons and as questions are based on C language related aspects, the variable value is considered to be meaningful.

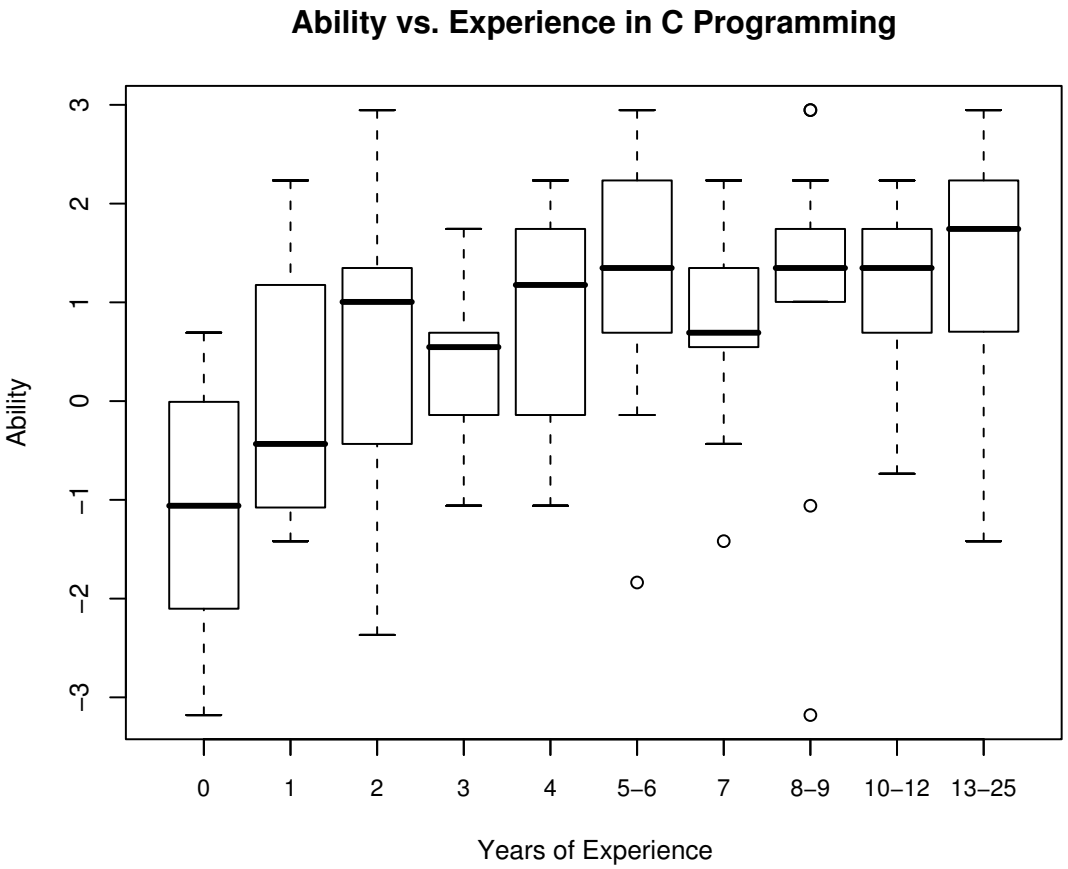

Figure 10.4: Boxplots for parameter estimates of C knowledge versus years of programming

The test can be regarded a reliable and comparable way to assess the concept of the C knowledge ability of experiment participants. One scenario of usage is a pretest allowing to select participants who meet a minimum level of C language knowledge. Another way to use this variable is to test for a correlation with variables like programming performance or software metrics. For example a hypothesis might test if a high knowledge of the C language allows to create less complex programs as indicated by a lower cyclomatic complexity of result programs. The variable can be used for purposes of control as well as an independent variable for an experiment.

Regarding the process of variable creation and experiment execution the level of difficulty of the entire process can be regarded low. The reason for this is that knowledge about the language C is given and that creating tests to assess that level frequently occur in a university environment. The new aspect that

a researcher is forced to perform is the test for Rasch model compliance. A successful test allows to use the resulting variable as interval scaled. Therefore interpretation of the distance between two variable values is possible leading to a meaningful interpretation of the "strength" of a variable value. Finally the Rasch model compliance test resembles a quality test for the measured variable.

11 Viscosity

11.1 The Concept of Viscosity

Viscosity as a conceptual, human related aspect of software engineering is first mentioned by Green [36] and rediscovered by Rosson [71]. It is described as the resistance of the programmer to local changes of the source code. As this definition directly includes the programmer as a main factor, the relation to the human aspect for software engineering is given. Wikipedia[1] on the other hand uses a different definition for the aspect of software engineering viscosity. It is based on an object oriented source code and describes the ease of adding design preserving code. The source code which can only be changed by "hacking" is regarded as having a high viscosity.

The physical definition of viscosity which is the base of the metaphor is the resistance of a fluid to shear stress. As a matter of coincidence the use of the η parameter for fluids is the same as the parameters used for basic parameters of the LLTM used below.

Another concept mentioned by Green is *premature commitment* that describes the situation of a developer taking a decision before the consequences can be foreseen. A problem caused by premature commitment may arise for compatibility or software design issues when the developers are forced to make a decision without a proper technical knowledge base. A different concept is *role expressiveness* which represents the ease of discovering program parts, their role and purpose. These human related variables are normally presented as a whole theoretic base and while all of them appear as sound entities of a low-level programming theory, only viscosity is quantified in the following experiment.

11.2 Design of the Experiment

The viscosity variable definition resembles the experimental evaluation presented in Chapter 10. The study itself is executed as an online survey and only the

[1] http://en.wikipedia.org/wiki/Viscosity_(programming)

type of model as well as the item definition significantly deviates from the first experiment.

11.2.1 Variables and Measurement

The basis of the source code of this experiment was made up of popular libraries like Microsoft Foundation Classes or Trolltech's Qt. The following libraries were used:

- Microsoft Foundation Classes (MFC)
- Windows Forms library of the Microsoft .NET framework (Forms)
- Trolltech Qt library (Qt)
- CORBA library (CORBA)
- Microsoft COM library (COM)

Initially it was planned to reuse source code fragments that were created in other experiments. The main problem with this source code was that the meaning and differences within the fragments were difficult to describe and present. After that interfaces for open source embedded applications were planned to be taken, but again their unknown internal nature and rather low relevance did not seem to support variable assessment. Popular libraries were chosen because of easy access and, as practical knowledge may exist for a wider audience, because of a better assessment of variable validity from external reviewers. In general tutorials from libraries were chosen as the small size was appropriate for an online survey. This did not allow a general viscosity assessment of the entire library, but library specific usage and design particularities were included in the tutorials.

Another decision highly influenced the generic design of the survey. As the experiment for assessing a C knowledge variable only focused on creating a person related variable. In the case of viscosity it appeared as a reasonable aim to create special parameters describing the difference in viscosity for the different libraries. Accordingly, it was not only intended to create person parameters that may be used within experiments, but to create generic questions leading to comparable parameters quantifying the difference in viscosity between different source code fragments. These parameters thus represent a special, semantic oriented assessment that is based on human perception and regarded complementary to software metrics.

The twelve generic questions used in the study are shown in Table 11.1. Each statement allows a dichotomous answer and was designed with the intention to cover different levels of viscosity.

1. The code complexity is appropriate for this task.
2. Some aspects are not clear and must be looked up.
3. Very intuitive - it looks like I have written the code.
4. The sequence of actions is confusing.
5. There are unneeded intermediate steps.
6. The code is uncomfortable to use.
7. Names and expressions are self-explanatory.
8. Important aspects can be found fast.
9. The code is not understandable at all.
10. The effort required to work with this code is low.
11. Complex parts are hidden or separated.
12. Some things must be memorized to use this code.

Table 11.1: Questions and the according item numbers, resembling η parameter used in the result section

During the design of questions, the basic idea of viscosity was integrated into the statements by means of time. Thus, it was assumed that the more effort was needed to use a library, the more reluctant a person is to use that code. This culminated in the statement "The code is not understandable at all" describing a perceived infinite effort to use the code fragment. In order to increase participant attention and positive influence on viscosity, statement 1, 3, 7, 8, 10, and 11 were inverted.

11.2.2 Hypothesis

The intention of this experiment was again to create a human based variable which allowed to assess source code as well. Accordingly no hypothesis in terms

of variable relation was made. The problem of a Rasch based study is whether the item of the test is model conform or not. This was tested using individual item fitness, χ^2 likelihood based model conformance tests, and in case of the LLTM and under given Rasch model conformance, BIC statistics (cf. Section 9.4) were used for comparison.

11.2.3 Procedure

In order to depict the source code used in the study, the source code fragment used within the CORBA library is shown below.

```
#include "baro.h"

int main(int argc, char *argv[])
{
  try
  {

    BARO_ORB_var orb=BARO_ORB_init(argc, argv);

    const char* refFile="TestObject.ref";
    ifstream in;
    in.open(refFile);
    if(in.fail())
    {
      cerr<<argv[0]<<":can't open'"<<refFile<<"':"
      strerror(errno)<<endl;
      return 1;
    }

    char s[1000];
    in>>s;

    BARO_Object_var obj=orb->string_to_object(s);
    assert(!BARO_is_nil(obj));

    TestObject_var testObject=
                        TestObject::_narrow(obj);
    assert(!BARO_is_nil(testObject));
```

```
    testObject->testFunction();
  }
  catch(BARO_SystemException& ex)
  {
    OBPrintException(ex);
    return 1;
  }

  return 0;

}
// end of program
```

This library was regarded problematic as object oriented design and expressiveness of certain steps was rather unclear. In addition, the source code shows that the occurrence of the word CORBA or any hint on the original library was removed in order to omit motivation effects of participants. A subjectively better source code in term of viscosity was the FORMS library tutorial as shown in the following:

```
using namespace System::NWindow;

__gc class Hello : public NForm
{
  public:
  Hello()
  {
    Text="Hello World";
    m_p=new Button();
    m_p->Text="Click";
    m_p->Top=120;
    m_p->Left=100;
    m_p->Click += new EventHandler(this, button_click);
    this->Controls->Add(m_p);
  }

  void button_click(Object* sender, EventArgs* e)
  {
    MessageBox::Show("Hello World!");
  }
  private:
```

```
  Button *m_p;
};

int main()
{
  Application::Run(new Hello);
  return 0;
}
// end of program
```

As both example are very short in terms of code, their respective design and meaning is considered very different. One aim of the study was to quantify these differences as presented in the results Section 11.3.

11.2.4 Participants

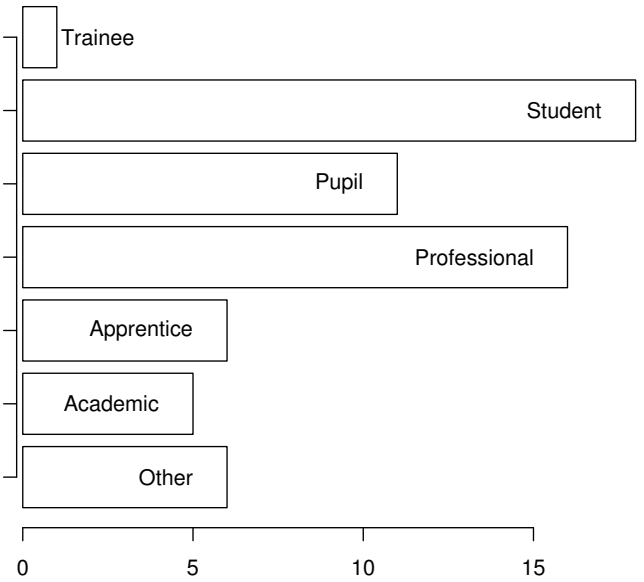

Figure 11.1: Background of participants

The same principles of participant selection as presented in Section 11.2.1 were applied in the viscosity experiment. Again a mix between very able and programming newcomers was tried to achieve by an appropriate selection of bulletin boards. In order to increase external validity, both English and German bulletin boards were used for this study. English programming communities are represented by:

- Codecall
- Codecomments
- Devshed
- Cprogramming

and German programming and technical communities were made up of

- c-plusplus
- pcwelt
- java-forum

63 of the 103 received answers were completed and used within the study.

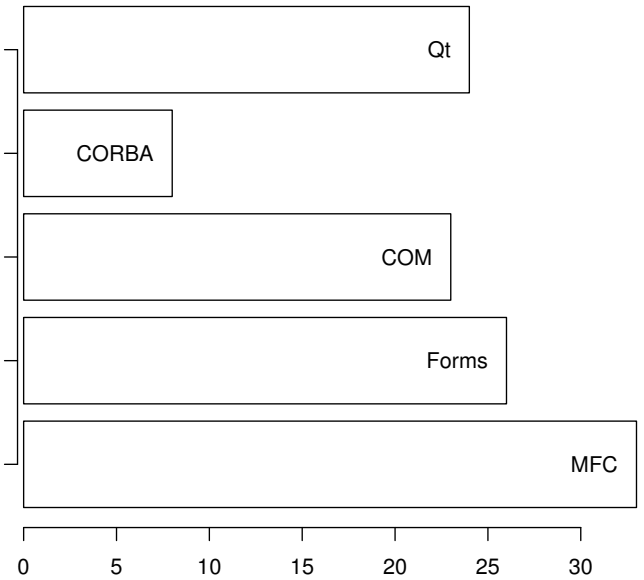

Figure 11.2: Libraries known to participants

The survey included questions related to the background of a participant which are shown in Figure 11.1 and 11.2. About a quarter of the participants were professional programmers in terms of occupation. Similar to the first Rasch based study, the majority of persons had an educational background which was expected for this kind of survey. Regarding the library knowledge the previous knowledge appeared considerable as about half the participants indicated to have used MFC before. Other libraries like Qt, COM, and Forms were known,

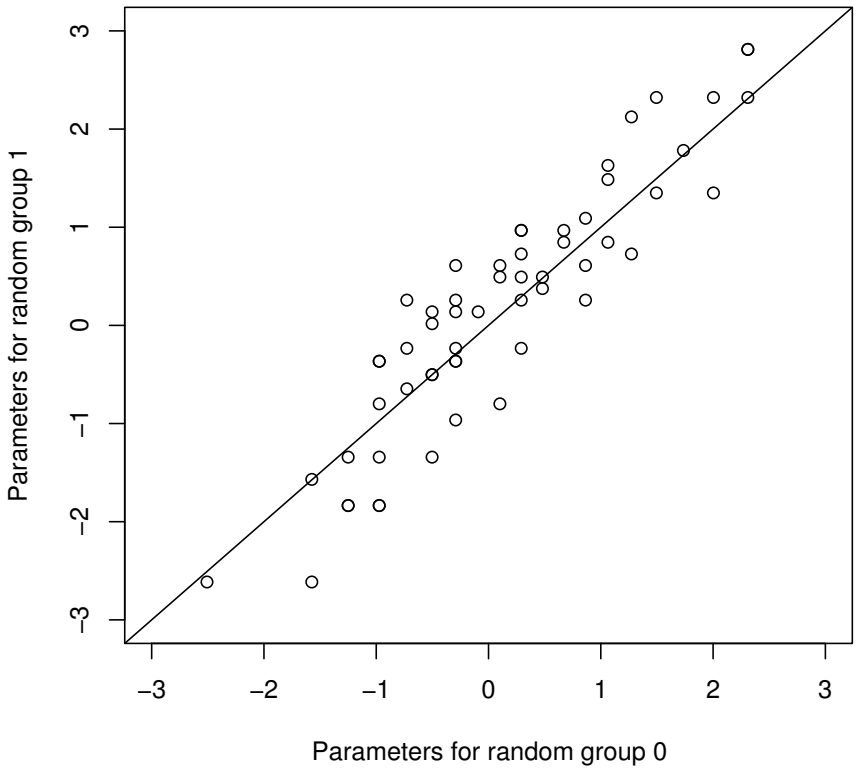

Figure 11.3: Goodness of fit plot for β parameters of the basic Rasch model based on randomized groups

too. Although this is problematic as it may have imposed motivation aspects, the answers of Figure 11.2 reflect the relevance of the chosen libraries.

Motivation to participate was increased by a €50 prize draw.

11.2.5 Threats to Validity

One of the main difficulties of internal validity concerns the source code. This code may have been modified in order to suit the Rasch model requirements or to artificially create differences in viscosity. This was countered by using the preexisting source code from tutorial oriented sources. The source code had to be changed in case of direct references to the original library. These named references were removed. As the selection process itself may have influenced the results, it is more difficult to argue for neutrality within this process. The informal guidelines to select a library were based on the aspects of a tutorial, that it should present the basic application of a library and that functionality and size was appropriate for an online survey. As no comparable source code

existed beforehand, this was the best way to ensure the equality of source code fragments.

Regarding external validity the usage of everyday libraries together with preexisting code was regarded to increase realism. As only a small fraction of the library source code is covered examples must be considered as hints to overall viscosity.

11.3 Analysis

The first test was aimed at assuring the Rasch model conformance for all items that were included in the survey. For twelve questions applied on five source code fragments, this resulted in 60 parameters with 59 being estimated. The programs used for calculation comprised Multira[2] and the Rasch model package of the R statistics language eRm (cf. [39]). Overall the test was slightly too easy with a resulting mean person parameter of -0.42. In order to provide a rough description of the parameters Figure 11.3 displays a goodness of fit plot. Similar to the C knowledge experiment parameters which are distant from the middle line are considered problematic as their difficulty is too different within both groups. For the parameter calculation values do not appear problematic on first sight. Here must be noted that random groups had to be created as it was difficult to find groups were all parameters could be estimated. The reason is that items with all zeros or all ones are removed from the parameter estimation.

Person homogeneity could be tested by creating two artificial groups and comparing their likelihoods using a χ^2 test. Groups are created by separating persons based on the median of the person parameter leading to a low scoring and a high scoring group. The calculated χ^2 value is significant only when the difficulty of parameter is really different between both groups. With a p-value of 0.82 no significant difference was found and person homogeneity was considered given. Individual item fit statistics indicated five items with an overfit which was accepted as having items with high discriminatory power was desirable.

Subsequently the design of the experiment consisting of twelve questions and five source code fragments was tested using the LLTM as described in Section 9.2. The basic 59 parameter models were tested against models consisting of only the questions as parameters, only the different source code fragments as parameters and both parameters together leading to a 4, an 11 and a 15 parameter models respectively as shown in Table 11.2. The results of the BIC statistic show that using a model that includes both source code viscosity

[2]http://www.multira.de

Model	# Par.	log L	AIC	BIC
Rasch model	59	-1788.348	3695	3821
Source Code	4	-2087.610	4183	4192
Questions	11	-2033.998	4092	4118
Code/Quest.	15	-1838.728	3707	3739

Table 11.2: Different models compared

difference as well as question difference as a parameter is superior to the other models. The AIC in this case was slightly inferior, but as a high number of items and persons were used, the BIC can be regarded as more relevant. Accordingly the 15 parameters LLTM could be regarded a legal substitution for the basic Rasch model.

η	Expected	Logit value	Std. Error
9	Very easy	-1.75	0.16
4	Very easy	-0.75	0.13
5	Easy	-0.67	0.13
6	Easy	-0.34	0.12
7	Difficult	0.02	0.12
10	Difficult	0.19	0.12
11	Difficult	0.32	0.12
5	Difficult	0.33	0.13
2	Easy	0.78	0.12
3	Very difficult	1.25	0.13
12	Difficult	1.44	0.13

Table 11.3: Resulting parameters concerning the twelve questions were put in order by easiness

The question based parameters as estimated by the LLTM are shown in Table 11.3. Values are given in logits (cf. Section 9.3). Regarding the range covered by parameters, the values between 0.3 and 0.8 show a slight redundancy in that area, while no values can be found between -0.75 and -1.75. Thus this can be regarded as a certain shortcoming of the test. Note that expected validity given in the table is discussed in the results' section.

Similar to the question based parameters the source code fragment based parameters are shown in Table 11.4. All parameters are given relative to the difficulty of the MFC library. Given a difference in parameter values from -1.2 to 0.9, the influence of the source code was rather strong. This is regarded a

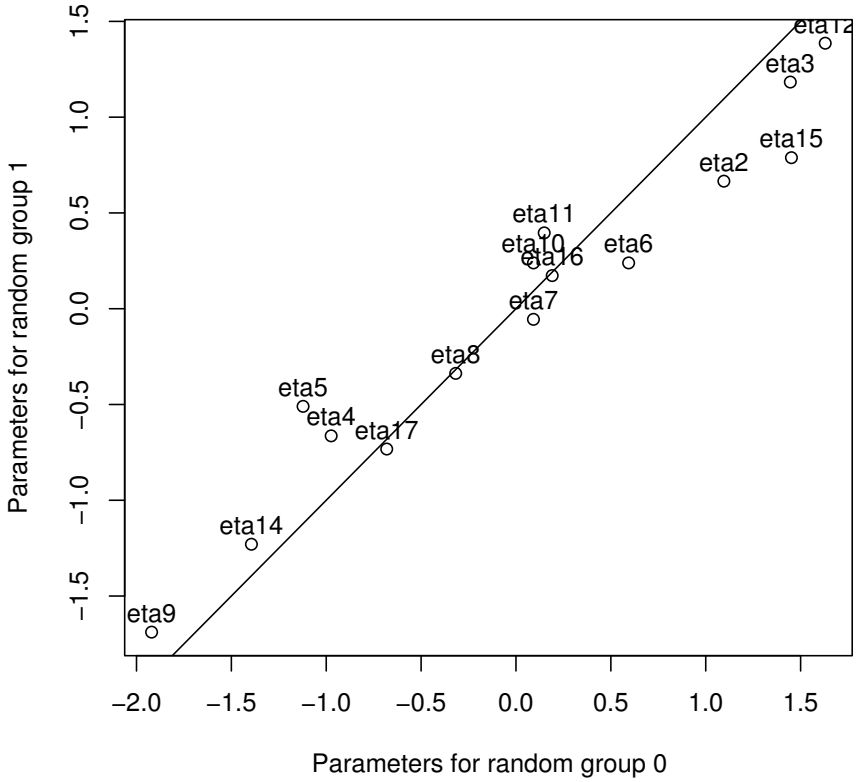

Figure 11.4: Goodness of fit plot for η parameters of the LLTM based on randomized groups

hint on the relevance of the variable as reluctance to use is obviously different between the libraries. One problem is that both middleware libraries (COM and CORBA) have a very high viscosity. Thus it was tried to reduce model parameters to the only aspect of middleware and user interface library, but this reduction is worse concerning AIC and BIC.

As the execution of a χ^2 was regarded problematic due to low response number and high number of possible answers (no χ^2 distribution given, cf. Rost [72]), a goodness-of-fit plot was made as shown in Figure 11.4. Apparently the η parameter did not show serious deviation in terms of difficulty differences between both groups.

11.4 Experiment Results

Regarding the outcome of the experiment a positive result of the study was that the source code related parameter values met an expected value. This

η	Library	Expected	Logit value	Std. Error
14	Forms	Very easy	-1.27	0.13
15	COM	Very difficult	0.97	0.12
16	CORBA	Very difficult	0.18	0.12
17	Qt	Easy	-0.72	0.12

Table 11.4: Resulting parameters concerning libraries. Parameters are given relative to first library (MFC)

can explained by the example of the CORBA and the FORMS source code as shown in Section 11.2.1. As indicated in Table 11.4 the viscosity value of both parameters was considered to be antithetic and the results of the analysis showed a parameter value supporting the expectation. While this is not considered to reflect the quality of an inference statistical test, the extreme logit difference of 2.3 between the two parameter values gives a strong hint on its applicability as a variable for software engineering in general. Together with the property of absolute scale of an LLTM parameter variable quality is strengthened. Another benefit of the given test is the data base of parameters which may be used to compare newly gathered data. As in addition person parameters can be used to predict experiment outcomes or to test correlation with other variables the test is regarded successful in creating the intended variable.

12 Uncertainty

During the execution of all software experiments involving actual programming of source code, all increments made between two compilation steps were gathered by a tool running in the background (see Chapter 15). The aim was to execute a meta study loosely based on the principle ideas of code evolution (cf. [34], [47], [53]).

12.1 Uncertainty within Software Engineering

While executing several software engineering experiments and lab courses, the task of implementation appeared as a rather chaotic element. Especially the time needed to finish requirements appeared as probabilistic in nature. It was not possible to find a related variable allowing to predict development times at all.

In order to analyze the course of these software projects in detail, two different approaches have been taken. First the principle of software evolution and its analysis methods as described in [38, 50, 51, 94] have been applied on the level of implementation. It was done by collecting every source code change between two compile steps as presented in [87]. As a difference to the results in Lehman [51] the laws that have been identified could not be applied directly on the data collected during implementation. The main reason is that the laws of software evolution focus on the entire life cycle of software, while the collected data only represent the implementation phase.

As a second approach the focus was laid on the source code in detail. One technique on this level is the source code analysis as summarized in [8]. Using this kind of analysis the syntactical parts of a programming language such as the control-flow graph, the call graph or the abstract syntax tree are described. This analysis is static in nature e. g. compared to software evolution the time aspect is missing. In addition the interpretation of results is rather difficult. Therefore a simple quantification regarding lines of code and effort based on measured time was used in the meta-study.

In general, the relevance of the implementation phase often is regarded as low. For example [9] gives a summary of different high level aims during the decades

of software research which comprise formal methods, software reuse, and time-to-market amongst others. Implementation is rarely used as foundation for evaluation. Often the task of implementation is presented as deterministic and linear as indicated by the waterfall model (cf. [81]).

Uncertainty is identified in [75] as part of an estimation, but research on that topic is described as scarce. It is normally linked to a pre-implementation phase as assumptions play an important role [44]. In [52], Lehman argues that program behavior might be inadmissible during future execution, even when admissible behavior has been observed. In other words a one-time satisfactory execution does not imply anything on future execution. As one source of uncertainty the given effort to maintain assumptions might not be invested by an organization. A solution to cope with uncertainty consists of the use of probabilistic Bayesian networks as provided for effort estimation in [66] and for defect estimation in [25].

12.2 Relation to Agile Methods

Although being part of the human factor family of experiments, the relevance of uncertainty for Agile Methods is given indirectly. When executing short development iterations, rapid feedback of development decisions can be obtained. Thus decisions which are based on uncertain knowledge or assumptions can be tested rapidly. In other words short releases as a development technique may cope with uncertainty much better than traditional planning based waterfall models. The question is whether uncertainty exists within software engineering and if the effect of uncertainty is important enough to justify its incorporating in software development methods.

12.3 Design of the Meta Study

The underlying data in this study originates from a collection of source code executed during six experiments and lab courses. This collection of atomic changes for each executed compilation normally was an analyzed version by version with the aim of gathering variables like cyclomatic complexity, lines of code or number of functions. This step based view was changed to a line based view: how long does a line of code exist, how often is it changed and how much time is spent on the changes? The resulting analysis focuses on the lowest level of programming and highlights the process of implementation.

12.3.1 Variable and Measurement

Figures 12.1 and 12.2 present examples of sequential changes to source code. Each line shows the file and its line number together with the changed line. After that the collection index is given in round brackets followed by the weighted effort, which normally is increased in very small amounts. The first example shown in Figure 12.1 illustrates one aspect observed often for changed lines. It consists of trying out different values for variables. Here the variable *soll* ("desired value" in English) is tested using different values. The effort for each change is rather small and the cumulative effort for the line is given as 5.7 minutes. One interesting aspect here is the variable *vorzeichen* ("leading sign" in English) and the change of the leading sign for index 139, giving a hint on a problem the group might have had. Another problem of detecting similar lines is shown in the first to second line, where the variable *UpFast* was found to be similar with *DownSlow*, although the former line of code continued to exist. In fact in this case lines were swapped at this index due to higher similarity of the following characters. Because of the high similarity it is difficult to rate this as a measurement error or not. Finally the changes for the index time 86 to 93 consist of changed white spaces which are accounted as changes, too.

```
uebung.c:128:case UpFast:   soll = 4000; break;(55)|1.5
uebung.c:131:case DownSlow: soll = -2000; vorzeichen = -1; break;(77)|2.5
uebung.c:131:case DownSlow: soll = -1000; vorzeichen = -1; break;(81)|3.0
uebung.c:132:case DownSlow: soll = -1000; break;(86)|3.0
uebung.c:164:case DownSlow: soll = -1000; break;(88)|3.0
uebung.c:132:case DownSlow: soll = -1000; break;(92)|3.3
uebung.c:168:case DownSlow: soll = -1000; break;(93)|3.3
uebung.c:168:case DownSlow: soll = -2000; break;(98)|3.6
uebung.c:168:case DownSlow: soll = -1250; break;(115)|3.8
uebung.c:168:case DownSlow: soll = -2000; break;(119)|4.2
uebung.c:141:case DownSlow: soll = -1500; break;(129)|4.4
uebung.c:177:case DownSlow: soll = -1400; break;(137)|5.0
uebung.c:178:case DownSlow: soll = 1400; break;(139)|5.3
uebung.c:178:case DownSlow: soll = -1400; break;(140)|5.7
```

Figure 12.1: Example 1 of changes in lines of code - organizational information stripped due to length of the line

In Figure 12.2 another type of change is shown. Here syntactical errors might be the problem which is indicated by the cast operator in the second line. In addition a change of a variable name is shown. Assumptions about the reasons of change are difficult, but it is assumed that each change at least had a reason to be executed.

```
task2_12.c:57:new_speed = new_speed * (1 - err_ratio);|0
task2_12.c:57:new_speed = (int)new_speed * (1 - err_ratio);|1
task2_12.c:57:_speed = (int)new_speed * (1 - err_ratio);|1
task2_12.c:57:_speed = (int)_speed * (1 - err_ratio);|1.5
task2_12.c:57:_speed = _speed * (1 - err_ratio);|1.8
```

Figure 12.2: Example 2 of changes in lines of code - organizational information stripped due to length of the line

12.3.2 Procedure

Regarding the execution of the meta-study an analysis tool had to be developed in order to identify changed lines of code. The procedure of identification started with a comparison of two successive files that were saved as consecutive versions during the data collection. Two tasks had to be performed then: first corresponding lines of code had to be detected in both versions. The second task was to detect changes on a code line. This mainly comprised a comparison of the first version's code line to all lines of code of the following version. Based on a similarity metric the best line was identified. As the detection of the same line and the detection of change to a line cannot be solved unambiguously, heuristics were used on both tasks.

Having detected a change to a code line the effort was computed. This was done based on the difference in time between the two consecutive versions. Thus if the difference between both versions was five minutes the overall effort applied on the changes was assumed to be five minutes. Furthermore this was weighted based on the length of a single line of code compared to the overall number of changed characters. Thus each line only received a fraction of the effort based on its new line length.

Additional data could be extracted like the number of changes for each line or points of time with a high amount of changes. In addition lines which could not be found were regarded as deleted. Together with the effort spent on these lines the lost effort can be quantified and is regarded as non-productive effort.

Algorithmic Approach

As the main course of the analysis has been outlined in 12.3.2 this section gives a short view on the main algorithms used. Before the actual analysis for a file can be made, some initial steps must be checked. First the addition, removal, and renaming of files must be detected. Especially the renaming of a file is important, otherwise a new file would be detected while the old file and its

Figure 12.3: Steps to analyse similar lines in a file

lines will be considered deleted lines. This would result in serious measurement errors later on. Another primary step is the computation of overall effort spent on the new version and the filtering of unneeded project and header files.

As depicted in the sequence of Figure 12.3 the first step to analyze a file is to mark the commented lines in a file. This step is necessary as commented lines in the final version which did not start as a comment are considered as unproductive lines. As well it was interesting to find out how often commenting took place. The marking of unchanged lines is mainly used to speed up later steps and to prevent the usage of similar lines which actually represent unchanged lines to be processed later. The update of changed lines is executed on the remaining lines which could not be located directly (e. g. unchanged) in the corresponding versions. Lines are compared and the most similar line is considered to be the changed line. If the line is not found, it is automatically marked as ceased to exist at the date index. If it is a new line it is added to the pool of lines. The following step of computing the weighted effort increases the effort applied to a line based on the overall effort and its length in characters. Finally newly commented lines have their comment counter increased.

In order to forecast the line position of unchanged lines which appear after a changed section of lines, an external algorithm library for the longest common subsequence problem (cf. [40]) was used. This algorithm was used again to detect renamed files based on the fraction of changed lines. Regarding similarity metrics a multitude of algorithms exists. As their precision does not depend on their approach but rather on parameterization of the results, the Levenshtein (edit-distance like functions in [21]) algorithm was used as basis.

Problems and Measurement Error

The problem to identify added, deleted and changed lines is important as a correct solution does not exist. The underlying reason is that lines may be ambiguous. For example, a developer might decide to remove some lines of code as their solution quality was low. While starting from scratch certain variables get the same, generic name as before, for example the name "index". The analysis would find corresponding lines, although the developer removed all lines and added new ones. Another example can be constructed based on the

execution of the similarity metric. A line might have been changed drastically by exchanging one short named variable with a new longer named one. This would result in a serious increase of the metric. If a completely different line would have changed its variable to the initial name, this variable would be taken as similar based on the comparison of the raw similarity value. Thus the true changes with the intention of the developer cannot be restored perfectly.

The detection of similar lines was especially error prone for very short lines. Here even one changed character leads to very different meaning and totally different lines which were often undetected. Thus only lines with a minimum length of five characters were included in the analysis.

It is rather difficult to give a quantitative description of the error in the measurement. The reason is that the correct solution can only be guessed by observation.

12.3.3 Participants

Except for the *refactoring* experiment all studies were executed during a lab course where all groups developed at the same time. All projects were executed using an ATMEL ATmega16 microcontroller. The programming language was C and no external libraries were used. The sources of data used in this meta-study are:

- *cpld* This lab course data was gathered from eight groups and taken from the N-version experiment described in Chapter 14.
- *fpga* As a successor of the *cpld* experiment the seven projects contained in this study developed the same task.
- *refactoring* The ten participants were taken from the experiment described in Chapter 5.
- *safety* The nine groups included in this lab course consisted of two students each. They developed a crane controller and had to apply additional techniques to increase safety.
- *tdd* This study consisted of seven developers which used the technique of test driven development throughout the project. The implementation task was the same as in the N-version experiment.
- *0607* In this study nine groups of two participants replicated the task of *fpga*.

12.3.4 Threats to Validity

The most important threat to the internal validity consisted of the automated analysis. Here simple errors like off by one index errors had an enormous effect on data precision. The main reason for this threat was the length and complexity of the analysis program. Additionally heuristic values for similarity of lines were found by comparing several parameter values. The problem with this value was that lax values detected similar lines which were obviously not related to each other, but sharp values on the other hand tended to miss several changes. Thus intermediate values were taken. In order to control the detection of similar lines log files were created. These files indicated the time of the change, the new line and the effort spent on that line. This data was used to ensure a sufficient similarity detection. Nevertheless precision was lost due to the implementation of heuristics to detect corresponding lines and their changes. Additionally this meta-study did not execute an actual hypothesis test, but post-hoc observations within the data. The main reason was that data collection was done to follow the course of an implementation in general. Thus no real factors could be checked within the collected data. Accordingly results should be considered as starting point for a hypothesis test.

Regarding the external validity the environment of a lab course was problematic. Lab courses were a learning situation and major problems regarding the programming had to be solved with the tutor. Thus the implementation was not free, but comparing this to real life development, the use of feedback either from internet sources or from colleagues might be seen as similar behavior.

12.4 Analysis

12.4.1 Overview

The general project duration for the studies is shown as boxplots in Figure 12.4. Especially the data collection in *0607*, *cpld*, and *fpga* shows a rather short programming phase of around ten hours. The studies of *refactoring*, *safety*, and *tdd* with raw development times of 30 hours are more representative concerning realistic development times. A notable difference is the variance in working time. The rule of thumb of a factor of three as mentioned in [68] appears as a very good factor to describe differences between development performance. The right diagram of the figure depicts the percentage of lines that was created and could be found in the final version. Only about 60 percent of the created lines of code are taken over to the final version.

On of the most problematic issues of the implementation phase is shown

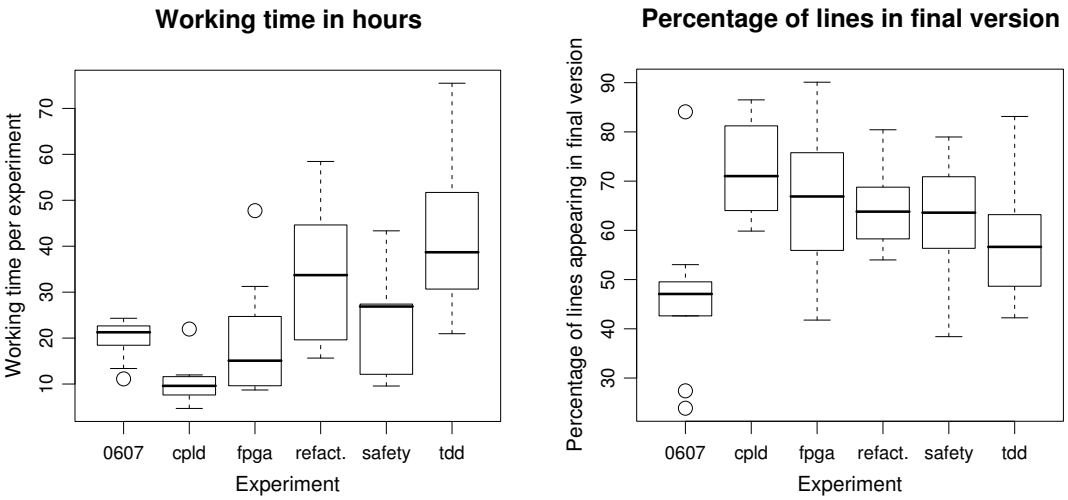

Figure 12.4: General data for each study

in Figure 12.5. Here, the percentage of time not appearing in the lines of code of the final version is shown for all studies. Around 40 to 50 percent of the time spent in the implementation phase of a project is lost during development. This certainly does not mean that all lines deleted that way are wrong approaches and just wasted. Especially for embedded systems, a certain part of the development is likely to be spent on aspects which need some exploration. Most often participants tried to use a part of the hardware which had to be derived from a static description of hardware units. Thus these lines represent the learning of programming an aspect. Direct productivity in terms of writing down the correct and complete functional source code does not occur.

12.4.2 Non-Productive Effort

The loss of effort was assumed to be linked to single events where a part of the code was removed at once. Figure 12.6 presents these events from all studies accumulated in two diagrams. The loss events are measured between two consecutive compile steps as provided by the analysis. They are given as relative percentage of the entire project time (left diagram) and as relative percentage of the project time until that time index (right diagram). The later view was selected to reflect that the decision of removing the lines of code was done during the running implementation. Thus the relative effort deleted was considerably higher than the comparison with the entire project effort. All groups were integrated in the diagrams as the studies themselves

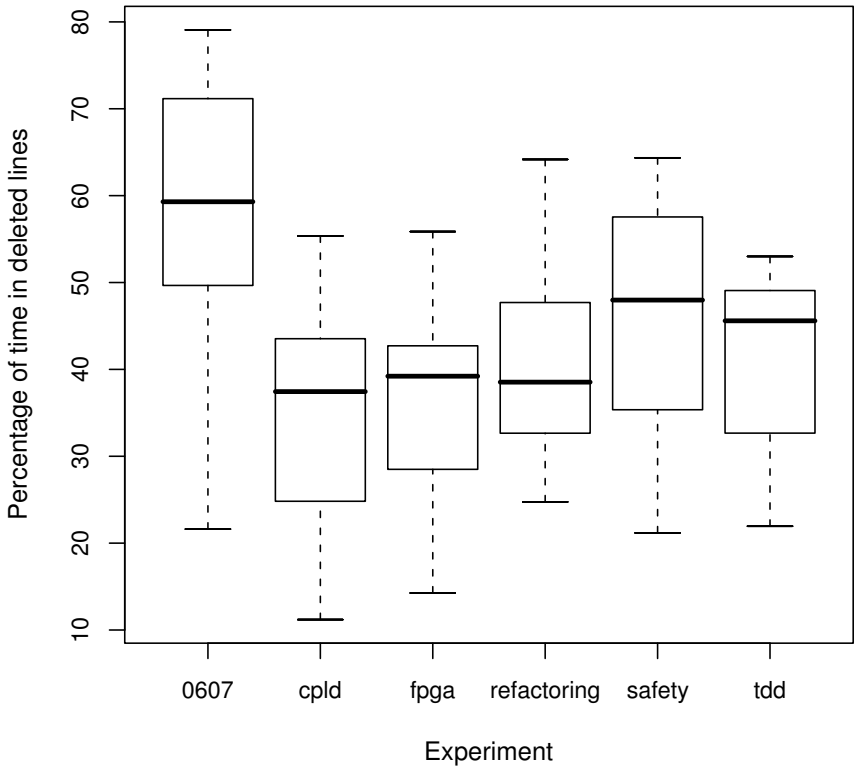

Figure 12.5: Percentage of effort deleted in each study

are not regarded as a factor. Single measurement points in which deleted lines were detected are depicted as loss events in the diagrams. When compared to the entire project, about five percent appear as a normal value for deleting effort. Unsurprisingly values are much higher when compared to the effort done until that index of time. Developers seem to accept losses of 20 to 60 percent. Regarding the maximum values the point in time causing that removal was important. For example when done during the initial state of a development, bigger changes were simpler to achieve due to the low absolute number of lines in the project.

Regarding the losses one observation in the data was that apart from peak values of removal, in general losses did not appear in consecutive points of time. Rather, a certain distance between indices can be found. Thus deleted lines were regarded as unique events in the analysis above.

The general loss of lines was tested for an influence on project time. In order to do this, the project time of each study was normalized. The group with the longest development time was assigned the value one and the group with

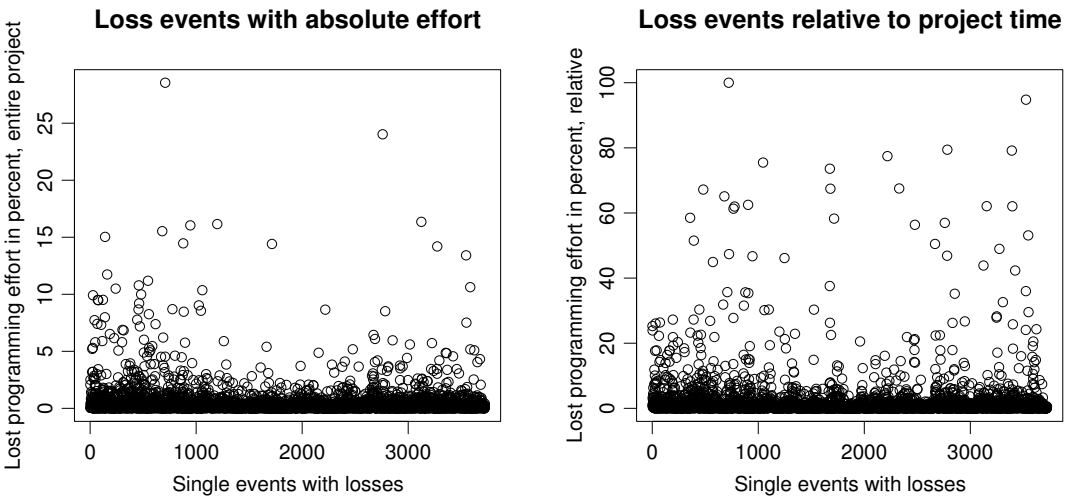

Figure 12.6: Events of deleting source code over all studies

the lowest effort was assigned a value of zero. Other groups were arranged based on the relative project time compared to these two groups. This is depicted in Figure 12.7 in the abscissa. The ordinate presents the overall loss of development time in percent of the entire project effort. One point represents one development team of the experiments included in the study. No relation among these variables can be found, so in particular losses are not an inherent part of long projects.

One last aspect to be tested was the relation of productive versus non-productive effort. Here the mean effort per productive line was plotted against the mean effort per deleted line for each experiment team as shown in Figure 12.8. The line represents equal mean effort between productive and non productive lines. The diagram indicates that the mean effort for a deleted line generally is higher when compared to productive lines. As most points are near the line, the difference is not regarded serious. Nevertheless the assumption that work on productive lines was more thorough could not be shown. Rather it appears that unproductive lines demand more attention. Because of missing variables it could not be checked if experienced users spent less time on unproductive lines than unexperienced developers.

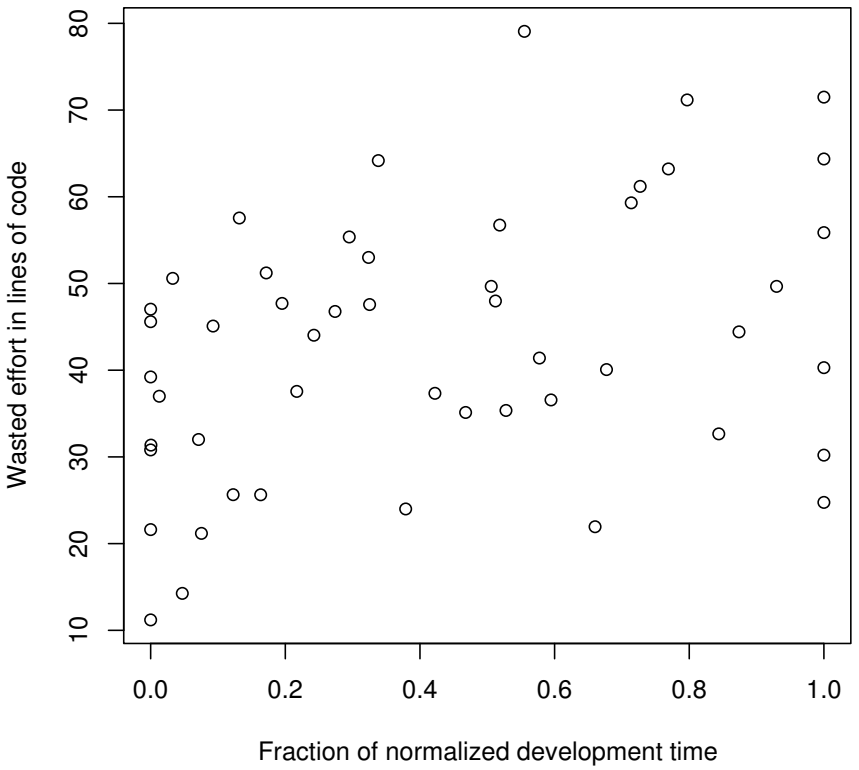

Figure 12.7: Plot of deleted effort and relative project time

12.5 Results of the Study

12.5.1 Uncertainty in Processes

One of the main aspects mentioned by [75] as uncertainty formulated as trial and error hypothesis in [87] is supported by this meta-study. Development effort which does not appear in the final version is not an exception, but part of the majority of implementation procedures. One interesting point is that the overall development time is not influenced by the degree of non-productive effort. Even fast projects had a considerable amount of deleted effort as shown in Figure 12.7.

The effort spent on deleted lines appears as static value when counted after a project has been finished. Problems only arise when a project is not finished yet but is in the state of execution. The reason is that no line of code can be regarded as final and thus, even completed modules may not be regarded as finished. Accordingly the main goals are not to reduce the lost effort to a low value and increase productivity in general. For purposes of control and

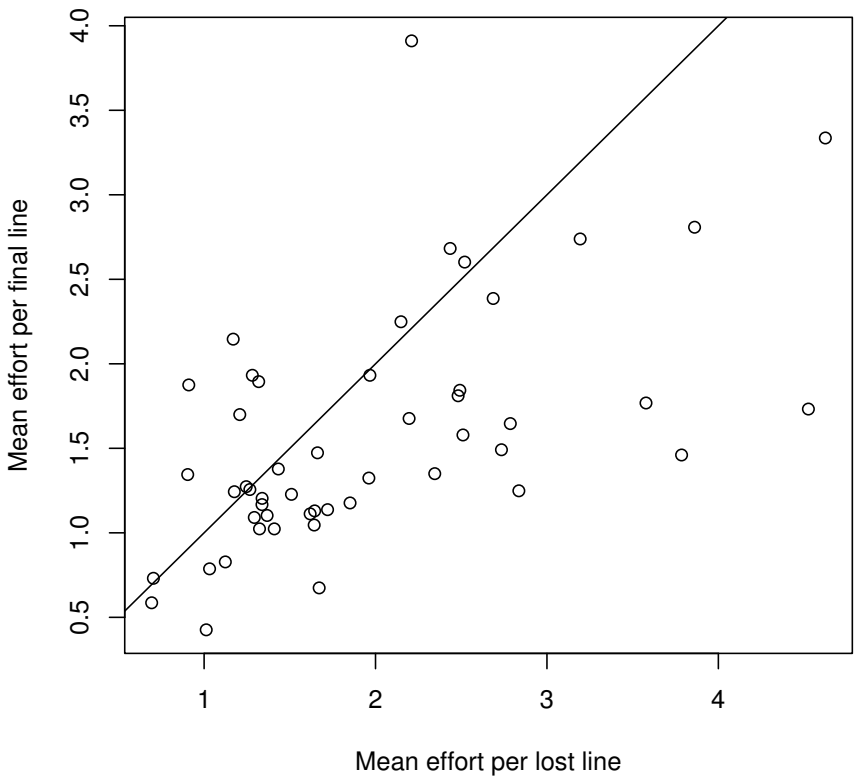

Figure 12.8: Comparison of the effort spent on productive (final) and non-productive (deleted) lines

measurement the most important aspect is to identify areas which are likely to change. This can be named as maturity of regions of code.

Another observation is that the removal of lines may occur in the form of peaks. These peaks of losses, which may be reported as anomalies [14], represent major changes in the source code. Based on the data of this meta study, the frequency of peaks is considerable as shown in Figure 12.6.

12.5.2 Threats to Validity of Variables

Regarding a project's state of completeness, uncertainty is a major problem because functions or modules which have been regarded as finished may be changed or even deleted later during a project. This is a special threat to the LOC metric. As it is not certain that a line of code is in the final version of a program, this metric is assumed imprecise when used for forecasting during the lifetime of a project. Another aspect related to uncertainty is the measurement

of progress of a project. The problem was that even though lines of code were added and projects became large, features were not fulfilled. A good hint on this problem can be observed in the data provided by the Progress Measurement Environment in described Chapter 16. For example Figure 16.2 indicates that the participant on the right side added a huge amount of instructions while overall functionality was unchanged. This can be considered as a horizontal implementation finishing an architectural framework before executing the program even once. As this horizontal development is difficult to quantize, the aspect of uncertainty imposes an additional threat on top of this. Even when measurable features are finished, future versions might remove lines leading to the feature being removed. Thus when lines of code are added, something is done, but it is difficult to assess its direct value for the system.

One last point is that uncertainty in implementation may be used to intentionally blur a project's current status. An example of this is when participants finish a programming experiment early, the true status may be hidden easily within the code. This makes progress determination very difficult as a variable as the intention of participants must be considered carefully.

12.5.3 Relation to Higher Level Software Engineering

The most important aspect regarding uncertainty in the implementation phase is whether the problem of lost effort is transferable to higher level software engineering or not. One assumption is that a project schedule is threatened as some estimation for tasks may be incorrect. Tasks can be longer than expected, but the opposite of an oversize estimation must be considered, too. Accordingly the degree of precision of software engineering process planning is influenced by the implementation. For programming problems which cannot be solved at all, the relation is obvious, as an assumption of the original planning is broken. The emerging thought is whether the relation is unidirectional, e. g. high level software engineering only influences implementation, or bidirectional. The observed uncertainty in implementation may not be covered well by a linear project schedule with absolute timings (as given by Gantt charts).

Regarding the collection and analysis of single lines of code, a data collection on this low level may be seen as a microscope view on the software engineering processes. Thus the aim is to increase knowledge about each line like how often it is called in a program (static analysis), how old it is, how well it is covered by tests, and so on. The aim then is to find factors influencing the probability that this line is changed or deleted. In addition factors from higher level software engineering must be attached to each line. This includes the part of the project schedule being executed as well as the part of the architectural design that is

developed. Information like this allows to map lines of code, implementation effort, and the amount of unproductive lines to aspects of higher level artifacts. It might be possible to observe dependencies among different architectural parts and among scheduled tasks which were thought to be finished or scheduled for later execution. When regarding software engineering experimentation, a difficulty exists in gathering this kind of project data, as techniques to reliably identify the architectural part or the part of the schedule a participant works on are not known to me.

13 Summary of Findings

The result of human factor related experiments are two variables which are regarded as relevant concepts of embedded systems software engineering: the variable of C knowledge as a main language of this domain and the variable of viscosity describing the resistance of a developer to change code. Both variables can be measured very economically as they are based on a few dichotomous questions which can be created as part of the preparation for an experiment. Additionally both variables were measured and the results were gathered. This allows an interpretation of new values as average, uncommon or even extreme. Moreover the variables are at least interval scaled allowing to evaluate differences among persons. Regarding validity the C knowledge variable has an asymptotic behavior over time which is considered adequate for this type of variable. For the viscosity variable validity is strengthened by expected differences in viscosity for known libraries. Both variables are expected to measure their conceptual idea as questions for variables are based on variable related concepts. For example the meaning of viscosity is based on questions of effort to work with a given source code. In the case of high effort, viscosity of change is considered to be high, too. Consequently these variables excel when compared to ordinary survey variables devoid of any additional background.

The application of the viscosity variable might suffer from a need for an additional layer of human interaction in order to gather variables. The reason is that in order to assess source code viscosity, no automated test environment can be used, but an additional test with participants must be executed. This is regarded a considerable overhead. This intermediate layer of human related measurement to create a software engineering variable might be relevant, as automatic methods do fail on the level of semantic interpretation. An experiment that utilizes this variables and shows their relevance is to be executed, though.

The meta-analysis concerning uncertainty in the implementation phase is considered as a small hint on a control problem for software engineering experiments. The correct assessment of progress is considered problematic in itself leading to a need for new variables as introduced in Chapter 16.

Part IV

An Experiment on Dependability

14 N-Version Programming with Hardware Diversity

14.1 N-Version Programming

One of the first experiments executed examined the effect of N-Version Programming (NVP) with an additional factor of diverse hardware platforms. NVP describes a general technique of failure reduction by developing multiple program versions for the same requirements. By gaining mutual independence of software failures, the overall failure rate is thought to be reduced as single incorrect program results can be identified directly when compared to the results of other programs. Using voter mechanisms the correct program results can be detected and overall program execution may continue without producing a failure. Redundant software structures of NVP are recommended by IEC 61508 pointing out the importance of this safety related mechanism.

One of the problems of NVP is the assumption of statistical independence from failure occurrence. The property of independence must be given as the identification and correction of single versions failing does not work when version fail dependently. As experimentally shown by Knight and Leveson [49], independence of failures in software programs is not given. One way to cope with this finding is a method called "forced diversity" as introduced by Littlewood and Miller [56]. The main idea is to create additional constraints among software programs in order to diversify results. As for embedded systems a multitude of platforms and development environments exist, an experiment was executed using the different platforms of a microcontrollers(MCU) and a complex programmable logic device(CPLD). The difference between both platforms is enormous, as MCUs are imperatively programmed, sequential executed systems while CPLDs are parallel systems programmed on hardware level. Using both systems the question of dependent failures was evaluated.

14.2 Design of the Experiment

The experimental principle is based on an experiment by Knight and Leveson [49] who executed the first experiment on N-Version Programming. The main difference in design is that an additional factor for the two hardware platforms had to be included.

14.2.1 Variables and Measurement

The basic variable used within the inference statistical tests were failure rates. Failures were defined as *the manifestation of a fault will produce errors in the state of the system, which could lead to a failure* as proposed by Anderson and Lee [4]. These rates were estimated by a test environment which tested whether certain functional requirements were met. The main functional requirement consisted of a frequency measurement with sending the results over a CAN bus interface. Details about the environment can be found in the published paper [74] and in a technical report hosted at the chair.

The independent variable controlled in the experiment was the assignment of a participant group to one of the two hardware platforms and assignment was done randomly.

14.2.2 Hypothesis

Although not directly visible independent N-Version Programming represents a considerable effect. For N independent failure rates p_1, p_2, ..., p_N the probability of no failure P_0 is given as

$$P_0 = (1 - p_1)(1 - p_2) \cdots (1 - p_N).$$

Each new independent version would considerably reduce the probability of having no failure. In the case of dependent failures P_0 actually is higher, though this is gained by sacrificing independence which is needed for NVP. A single failure probability P_1 can be calculated by summing up all single failures:

$$P_1 = \frac{P_0 p1}{1 - p_1} + \frac{P_0 p_2}{1 - p_2} + \cdots + \frac{P_0 p_N}{1 - p_N}$$

Each summand represents other versions having no failure ($\frac{P_0}{1-p_i}$ for a given version i) with the according version failing (thus $\frac{P_0 P_i}{1-p_i}$). Accordingly the overall probability of a combined software failure of two ore more versions failing at the same input can be calculated as

$$P_{more} = 1 - P_0 - P_1.$$

This basic modelling of independence for multiple errors is broadened by adding repeated tests. The aim is to assess the likelihood of a certain number of observed multiple failures with respect to the independent model presented above. In order to assess the likelihood of k observed the observed value is compared with simultaneous failures (two or more versions exhibiting a failure for the same input) for n test cases executed. As alternative events are used in this case (P_{more} and $1 - P_{more}$), a binomial distribution must be used with

$$P(K = x) = \binom{n}{x}(P^x_{more}(1 - P_{more})^{n-x}).$$

Since n is sufficiently large in the experiment a normal approximation is used for the binomial distribution. The aim of this replacement is to use z values which allow to compare event occurrences using a standardized normal distribution. The z value is used as a generic representation of a normal distributed value and is calculated by dividing the difference to the mean by the standard deviation. Thus z is calculated by

$$z = \frac{K - nP_{more}}{\sqrt{nP_{more}(1 - P_{more})}}$$

and has a mean of zero and a standard deviation of one. The resulting z value can be used to assess the likelihood of the observed event for the given model by using standard statistics tables. The results of this original experiment design are presented in Section 14.3.2.

In addition to the independence based test for a homogeneous group, the independence between two groups had to be tested. The hypothesis was that making independent failures on both platforms was calculated by the product of single failure probabilities $p(F_{MCU})$ and $p(F_{CPLD})$ with F being a failure event of the according platform. Thus, the property

$$H_0 : p(F_{MCU} \cap F_{CPLD}) = p(F_{MCU}) \cdot p(F_{CPLD})$$

had to be tested. The procedure and results of the resampling approach are presented in Section 14.3.1.

14.2.3 Procedure

The experimental task consisted of a speed measurement for an automotive prototype vehicle. Four different input signals had to be processed and sent via CAN bus. The frequency measurement on one hand needed to measure the

rectangular shaped frequency signals and on the other hand required a special calculation in order to transform the measurement results into a comparable format. Additional requirements comprised safety and time delay aspects.

The development for the MCU group was done using ATMEL ATmega16 microcontrollers clocked with 6 MHz was programmed with ATMEL AVR Studio. The CPLD group used Xilinx CoolrunnerII CPLDs clocked with 1,8 MHz and the Xilinx ISE as development environment. All groups were issued the according development boards which contained buttons and LEDs for additional testing purposes.

The experiment was executed within a lab course which took 13 weeks with a three hours weekly appointment. The average development time was 19.5 hours. In the end an acceptance test was executed in order to assure functionality and comparability of versions.

14.2.4 Participants

The lab course consisted of 26 students. Participants were not chosen randomly but were assigned to the course by a university course selection system. Working in groups was necessary as the number of participants exceeded the number of development boards. Consequently twelve groups with a minimum of two and a maximum of three students were formed. Ten groups passed the final acceptance test for both platforms leading to 20 versions available for analysis.

14.2.5 Threats to Validity

Internal validity suffered the most from the setting of the lab course. The room was very small for 26 students, so communication between groups could not be prevented. Slower groups might have copied ideas from faster, and more successful groups. Another problem was the involvement of the experimenter as a teacher. As certain solutions and development programs had to be found, the experimenter was responsible to help students. Thus a certain level of dependence might have been induced by that way. Regarding the test environment, dependences might have been exploited by choosing certain frequencies with the test environment. This was partly circumvented by using random frequencies.

One of the main drawbacks when using students as participants in software engineering experiments is the lower external validity. As having a sufficient number of persons in an experiment is another challenge, students as participants had to be accepted. The general task together with the tools can be described as realistic. One of the reasons for a realistic setting was a list of written

requirements which had to be analyzed and a defined software quality which was made up by the acceptance test.

14.3 Analysis

The following sections present the analysis of the results taken from the test environment made for this experiment. At first the effect of an additional factor for NVP is tested. After that the original analytic approach is replicated.

14.3.1 Independence of NVP with Forced Diversity

Due to the analytic approach of Knight and Leveson being factor-less, a different approach was chosen. In order to test independence between two groups the method of resampling (cf. [24]) was used. Basically resampling (or bootstrapping) is a simulation that reuses the observed data in order to assess the likelihood of a given statistic. As an example an observed difference in treatment group median values can be compared to 1000 simulation based medians. If the difference is sufficiently unlikely (e. g. less then 5% of the simulated values are as high as the observed value) a random generation of median difference can be ruled out and treatment is regarded as having caused the difference.

The simulation process used for assessing dependence within groups is informally sketched in Figure 14.1. It starts by randomly selecting one version of the MCU group M_i and one version of the CPLD group C_j. By selecting only one version from a group, the dependence that exists within one group is circumvented. Having created randomly selected pairs for each of the 15000 input frequencies of the test environment, the number of simultaneous failures can be counted which is considered to be the observed value. Based on the known overall failure rate of each version M_i and C_j the observation of failure is replaced by a probabilistic simulation step. This simulation step only uses the failure rates and thus is considered independent. An error for both versions thus is created independently and again this value is counted. By comparing the observed and the simulated simultaneous failures, it is possible to assess whether independent occurrence of failures happens or not. If the number of observed failures is not significantly higher or lower than the simulated error independence will be given. In case of dependence failures more often occur for both versions simultaneously. Accordingly the single simulation step consisting of 15000 pairs is repeated multiple times. As the number of simulated failures was often lower than the observed failures, a very high number of repetitions was

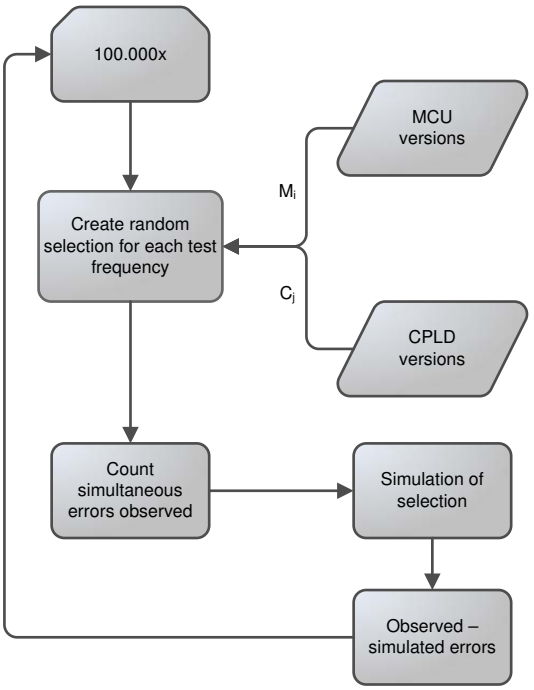

Figure 14.1: The simulation process

necessary finally leading to utilization of the RWTH cluster. Using the cluster for the resampling calculation was possible because simulations themselves are independent and can be executed on multiple machines at once.

Each single simulation step described above resulted in a difference of observed and simulated errors. These results were accumulated in a histogram as shown in Figure 14.2. The abscissa shows the differences of observed and simulated failures. Obviously a positive value indicates that more simultaneous failures were observed than simulated. The case of having more observed failures than simulated errors rarely appeared. In fact only 127 of 100000 simulations resulted in the same number or more observed failures. Accordingly the independence property can be regarded as an inappropriate model for describing the data. Finally the hypothesis of an independent model must be given up.

14.3.2 Replication of the NVP Experiment

The original experiment of Knight and Leveson compared the observed failures to an analytic, independence based model as described in Section 14.2.2. Rejection of independence basically is the same as presented above, with the only difference

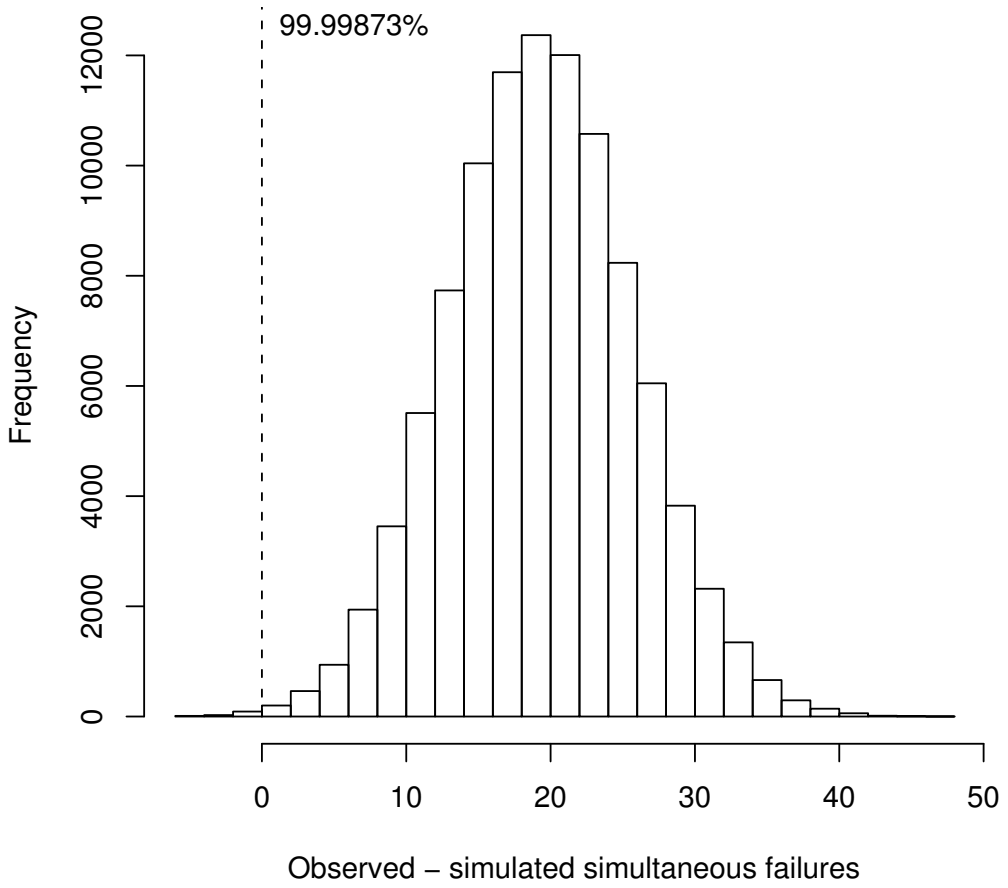

Figure 14.2: Histogram of independent model simulation

that instead of creating a population based on resampling, the observed value is calculated as a z value allowing to directly assess probability of occurrence. Thus by counting the k cases with more than one failing version and calculating P_{more} based on single failure properties, z can be calculated. The values for k in this experiment were $k_{MCU} = 1673$ and $k_{CPLD} = 294$ with $P_{moreMCU} = 0.086$ and $P_{moreCPLD} = 0.012$. Thus z-values are 11.141 for the MCU group and 8.295 for the CPLD group. The z value for 99 percent is only 2.33 and accordingly the observed value for k with the respective probability P_{more} are very unlikely. Finally the independence property does not hold and results support the findings of Knight and Leveson.

14.3.3 Assessing the Strength of Factors

Different platforms can be regarded at least as a factor influencing dependency and, when multiple versions are developed and a voting mechanism is available, overall safety. Using the principle of resampling presented above, the frequency of two versions failing simultaneously can be described for different constellations of pairs. First the best performance is considered to be given for the independent model. Three other pairs are possible: one version taken from the MCU group, the next version taken from the CPLD group and a pair consisting of one of each group. Each pair configuration is repeated 10000 times for the 15000 test inputs. The results are gathered in a histogram as shown in Figure 14.3.

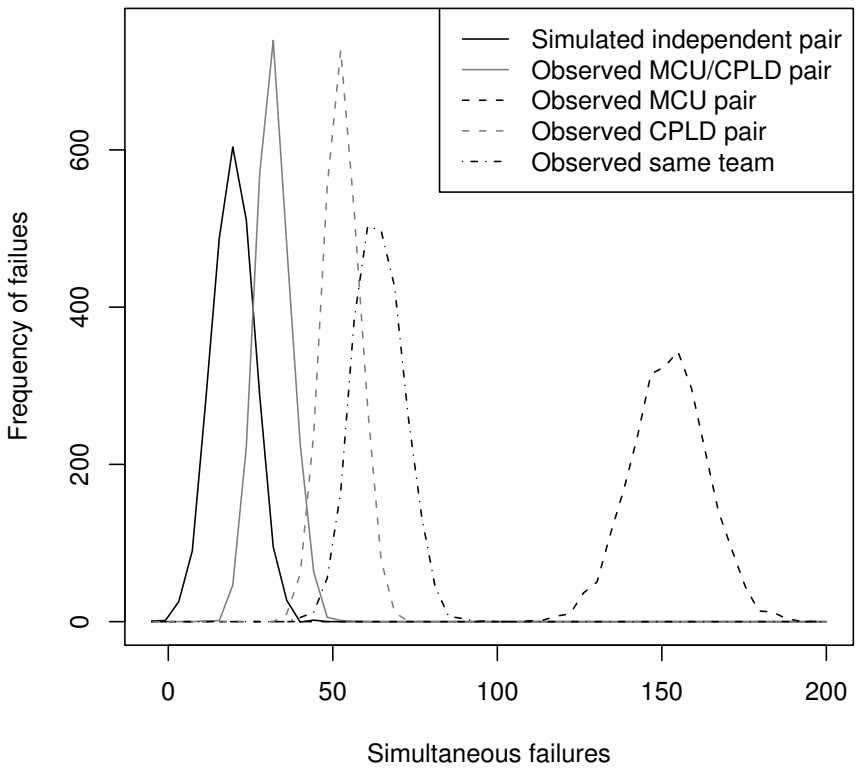

Figure 14.3: Strength assessment of different factors of influence

This simulation shows that the performance of taking one version from each group is slightly below the theoretic, independent model. As can be seen simultaneous failures for the diverse group occur less frequently than failures made in each homogeneous group. Thus a certain benefit of diversity can be expected, though independence may still not be given for it. Another

interesting result is that versions taken from the same team appear to increase independence of the results, too.

14.4 Experiment Results

Inducing independence into NVP appears as problematic and within this experiment, even the usage of different platforms, languages and development environments does not seem to assure independence. The main reason is that independence of failure events as given in

$$p(F_{MCU} \cap F_{CPLD}) = p(F_{MCU}) \cdot p(F_{CPLD})$$

is a very strong property. Independent events lower the probability of coincident events by the product of single properties which is difficult to assure as a property for NVP. Instead a factor based strength difference of simultaneous failures could be shown. Regarding the effect strength of diversity used within this experiment it appears to be superior to have development groups with different platforms compared to homogeneous groups. Other treatments like using certain numbers of groups or certain configurations of groups can be compared with the strength assessment resampling in order to find other viable factors.

Part V

Measurement Tools

15 Code Evolution Framework

During the execution of the experiments several tools were created for measurement and control purposes. The most important tool was the Code Evolution Framework. The aim was to gather data during the programming phase of an experiment. The main functionality was to copy the source code of a participant every time the compiler was used. The result was a huge data collection of different states of the created software. Consequently a second tool for data analyzing purposes was created. Based on this tool changes in metrics could be extracted. Using a test environment like the one presented in Chapter 16, an assessment of the increase of functionality as well as reliability over time was possible.

15.1 Example of an Analysis

In order to explain the principle of data collection and analysis, a data based example is depicted in Figure 15.1. The diagram presents an accumulated complexity metric over all functions and files of a participant's project. The underlying data is provided by the experiment of test driven development. Seven participants finished the task (1, 7 -12) and the successful persons with their respective number are shown in the diagram. As multiple versions were collected the course of the variable over time can be shown. In oder to make the results comparable the time had to be normalized. The reason is that the overall time needed by a participant and the frequency of individual compilation steps was not constant.

The main aspect introduced by this data collection is a very atomic measurement concerning points of time. General code evolution analysis takes place on source code repository commits. Thus intermediate steps are not present. The idea regarding experiments is that these intermediate steps include hints to problems and how these problems are solved. Other examples of data analysis with this tool can be found in the uncertainty related meta analysis described in Chapter 12.

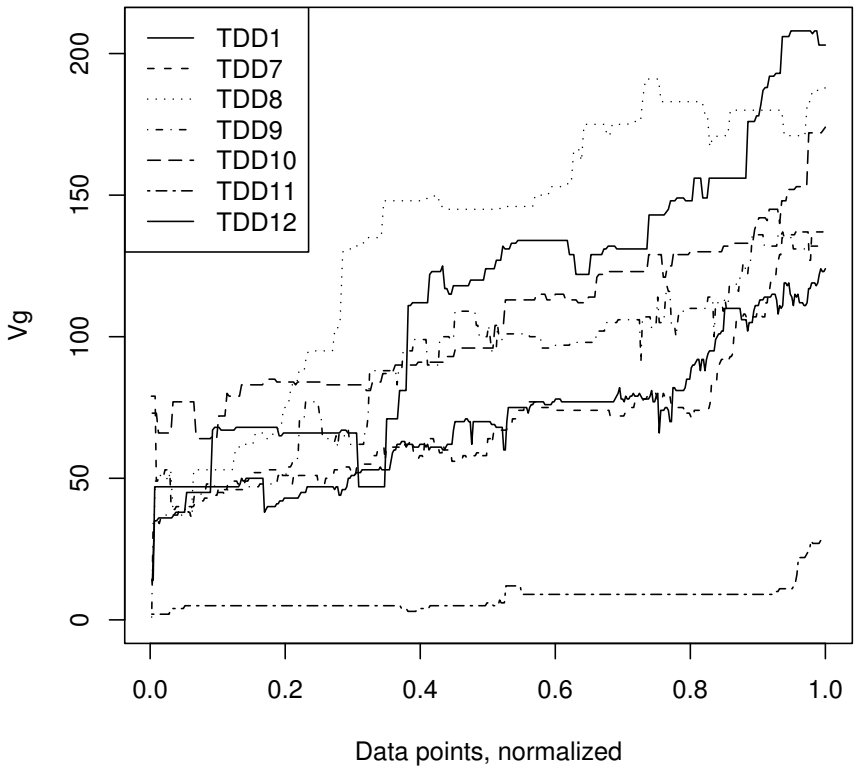

Figure 15.1: Accumulated complexity/McCabe metric over time taken from TDD experiment

15.2 Code Evolution Data Collector

The main program used for data collection was simply called "data collector" and works like a computer virus. It replaces the original compiler based on its name and every time this compiler is called, it executes a copy task before calling the real compiler. Using this design an integration into all development environments is possible, as nearly all of them are based on command line compilers.

One downside of using this design is that the path to the source code must be provided. The reason is that it is very difficult to extract the source code path from the command line for all compilers. In addition some programs implicitly used a compiler with the path of the source code given as a parameter making it difficult to determine the correct path. Therefore a user based configuration was provided allowing to specify the correct path. It was found out that reducing the number of files being copied was resourceful as the data repositories grew very large.

15.3 Code Evolution Data Analyzer

Regarding the data analyzing capabilities a graphical tool was developed which on one hand controlled data selection and on the other gathered and configured a variety of modules created for analysis purposes. The data was stored in a standard file system and had a tree like structure as shown in Figure 15.2.

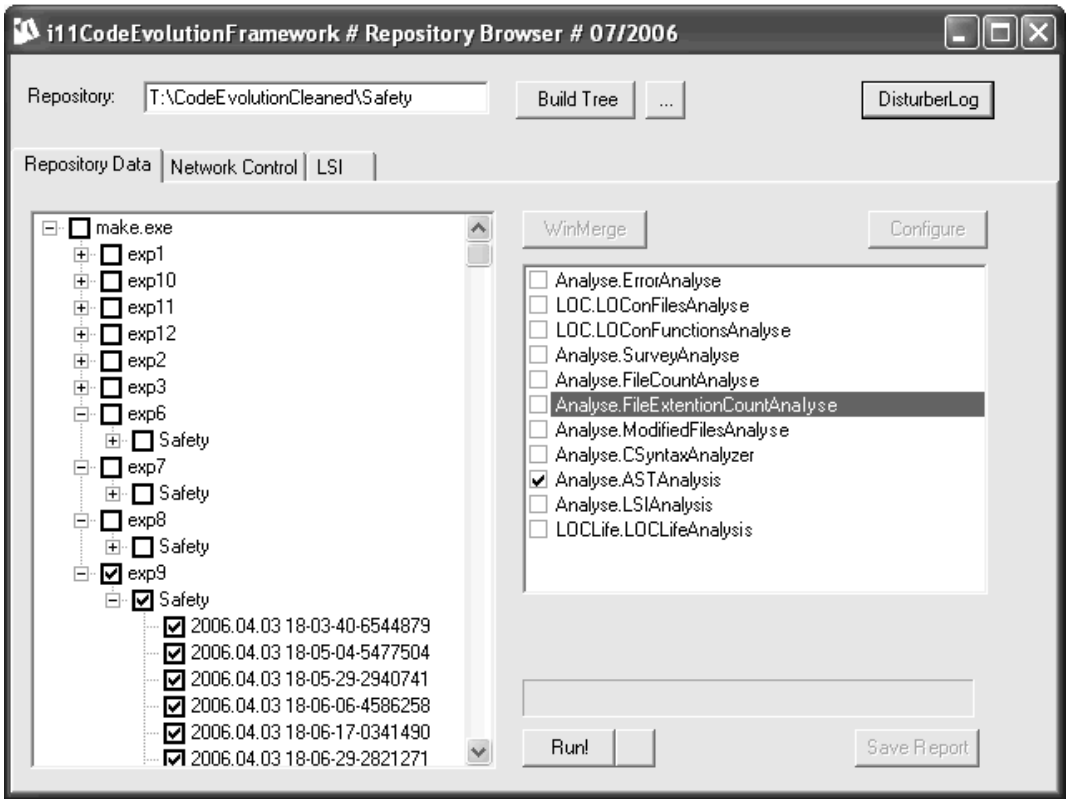

Figure 15.2: Structure of stored experiment data

As the change over time in source code may be analyzed in different ways a module based approach was taken. The interface for creating new modules is described in Section 15.5.

The following analysis modules were implemented:

- **Error Analysis:** This analysis detects whether a compilation step was successful or not. It allows to quantify the number of compilations which contained a syntactical or linker error.
- **LOC on Files:** This module performs a file based lines of code analysis. Results are subsumed for all project files.

- **LOC on Functions:** This analysis provides lines of code for single functions. File names are added to function names in order to preserve unambiguousness.
- **Survey Analysis:** As it is possible to link a survey dialog to the source code compilation, this module gathers the results from each executed survey. Due to compilations appearing in rapid succession, the validity of the surveys is regarded rather low.
- **File Count Analysis:** This module counts the files in the project. It allows a rough interpretation of the overall design when the overall lines of code of the project are used for comparison.
- **Modified Files Analysis:** The analysis marks files that have been changed between two successive steps. It is thought to describe the extend of change between steps.
- **C Syntax Analysis:** Within this module, syntactical information of a project was thought to be measured. As the embedded source contained numerous assembler instructions, the module could not be completed in a reasonable amount of time and was replaced by the AST Analysis module.
- **AST Analysis:** In this module, the external library *DevCodeMetrics* is used to compute metrics like lines of code and McCabe (or cyclomatic) complexity. Results are given function wise. The main reason to use the external module was its speed as well as the ability to store the results in xml files.
- **LSI Analysis:** A special form of analysis is the latent semantic indexing [57]. Its aim is to identify similar functions based on the text of these methods. Using the data of single steps it was thought to identify semantic changes like single refactorings over time. This module suffered from noise making it difficult to interpret changing cluster structure over time.
- **LOC Life Analysis:** One of the main views of the code evolution is its time information which can be used together with metrics calculations. The LOC Life Analysis focuses on single lines of code and calculates the number of changes, the fraction of removed lines with respect to the final version and so on. It allows to rate underlying behavior of the developer. This can by explained by the module's main benefit being the calculation of changed lines and removed lines which can be interpreted as unwanted,

time consuming aspects of a software development. Details of this analysis type are presented in the uncertainty experiment presented in Chapter 12.

- **Command Line Analysis:** This extension allows to execute batch files on the gathered data. This is regarded a simple interface compared to the more sophisticated programming interface of Section 15.5.

15.4 Source Stepper

In order to validate analysis steps in terms of correctness a calculation by hand had to be done. Visualizing changes between two consecutive program versions for result validation was an important need which was fulfilled by the Source Stepper application. It was developed as a web based tool developed using the programming language flash. It allowed to browse through the data structure. Comparison of two successive file versions was done with a diff algorithm and additions, deletions and changes were marked with different colors. The graphical user interface is shown in Figure 15.3. The upper line of the application allows to select the experiment, participant, date and file which should be shown. The main text area shows the actual selected version (on the right) and its predecessor. The lower part is made up of buttons to step through versions and individual changes. In addition lines of code as well as compilation success and the time in minutes between both versions are presented.

A special way of evaluating changes in the application is given by the fields of "type of change", "quality of change", and "semantics". These three fields represent a *qualitative assessment* of the change which needs a continuous review of each change. The type can be specified as "initial coding" for new code, "make working" for general changes, "test step" for adding debugging related output, "architecture outline" for creating a program skeleton, and refactoring. Although the expression "make working" seems to be very informal, this type of change was by far the most important one and no better explanation can be found to describe this type of change. The field quality of change comprises "worse", "unchanged", and "better" and aims at describing how the quality of code was altered by a change. As changes within steps usually are quite small, this information almost always contained "unchanged" and thus a description of this aspect was found to be difficult. Finally, the semantics field allowed to give an open answer and was used to describe the technical aspect that the group worked on. Results gained from this field are shown in Figure 15.4. The experimental data was gathered during the N-version programming experiment.

Figure 15.3: Graphical user interface of the source stepper applications

Apart from the overall development time which was different by a factor of two, single technical aspects show a proportional difference in time.

15.5 Module Interface for Data Analysis

The data collection mechanism of having a background application copying a source code folder can be regarded as a functional complete tool. The analysis on the other hand can be expanded by new modules. In order to encourage reuse the interface for writing new .Net modules is described in the following.

Two types of analysis modules can be created. The first module type "ITableAnalysis" creates table based data which can be directly used in statistics programs. It gathers data on basis of one row per version in the experiment. Accordingly, each column of the resulting table provides information of one variable that is calculated by the module. The basic interface is the abstract class "ITableAnalysis" which must be used as a super class for the new module. Two

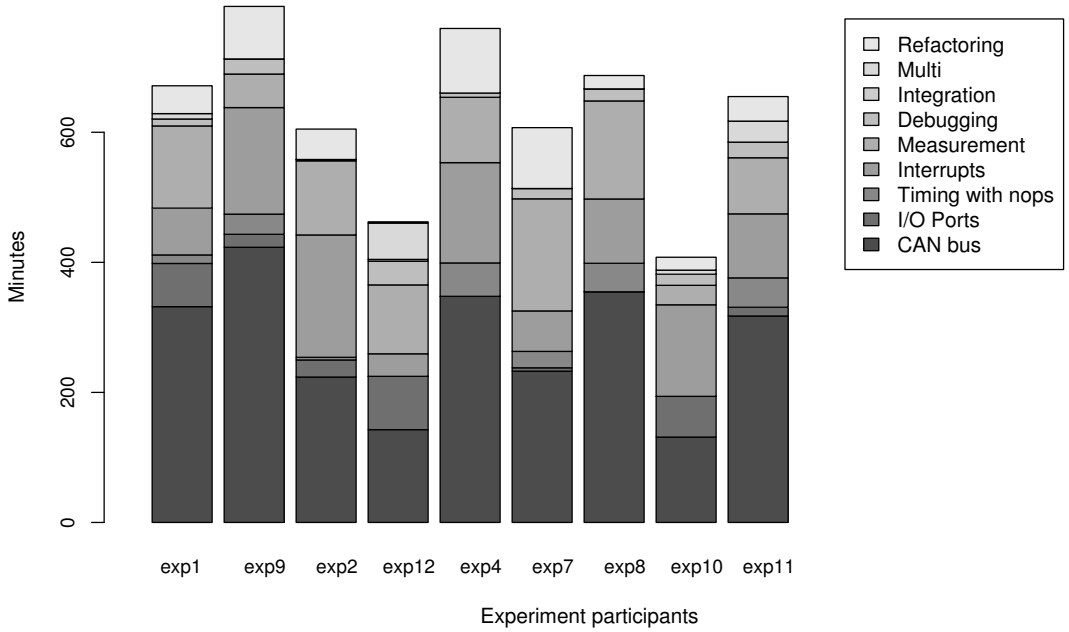

Figure 15.4: Effort spent on different aspects in N-version experiment

main methods must be implemented: "getReportHeader" which is supposed to return the comma separated header line of the analysis and "getReportResult" which returns the results for one single repository entry. The later function has a second signature with a flag indicating to use numeric ("0","1") of textual ("false", "true") boolean results. All three methods are given an "IRepositoryEntry" object which allows to list and access all files of this point of time as a stream.

```
public __gc __abstract class ITableAnalysis : public IAnalysis
{
public:
ITableAnalysis(void);
virtual ~ITableAnalysis(void);
/** This method returns the header for a comma separated value file.*/
virtual System::String * getReportHeader(IRepositoryEntry * entry) = 0;
/** This method returns the data for a comma separated value file.*/
virtual System::String * getReportResult(IRepositoryEntry * entry) = 0;
/** This method returns the data for a comma separated value file.*/
virtual System::String * getReportResult(IRepositoryEntry * entry,
System::Boolean useBoolean) = 0;
protected:
```

```
  TableReport* report;
};
```

The second interface is used for more sophisticated analysis modules which may contain complicated data structures. An example for this are abstract syntax trees which can be extracted from the project. As the data is a tree like in nature, its storage as a sequential list is rather cumbersome. Accordingly a tree like data structure leading to an output of an xml file was developed. The resulting interface is:

```
public __gc __abstract class IXMLAnalysis : public IAnalysis
{
public:
IXMLAnalysis(void);
virtual ~IXMLAnalysis(void);
virtual XmlElement * getReportResult(IRepositoryEntry * entry)=0;
protected:
XmlDataDocument * xmldoc;
};
```

The main method that must be overwritten is again the "getReportResult" which is given the current repository entry as a parameter. The main difference to the table based approach is the return value consisting of a system defined xml element. Its internal structure must be supervised by the analysis object.

16 Progress Measurement Environment

In order to control the resulting source code of experiments, a measurement environment was developed. Müller et al. [64] points out to either control the complexity of the task and to measure the time needed, or to do the opposite and control the time (making it constant) and measure the progress. During the experiments that were executed, it was found to be very difficult to have a variable time. The reason is that some experiment participants were very fast and thus spent time on details, while other hardly finished the main experiment task. Obviously a constant time with a variable degree of software functionality appears to be a superior experimental design.

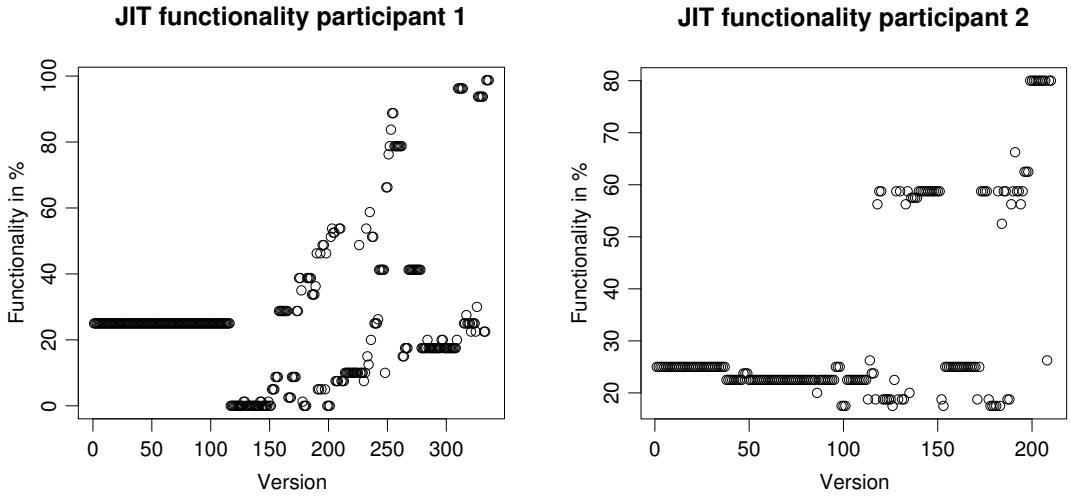

Figure 16.1: Change of functionality by version for two exemplary implementations

In order to measure progress in terms of functional additions to a software, the domain of Just-In-Time (JIT) compilers was selected. This kind of compiler receives a precompiled byte code and translates it during runtime to its own machine code. Subsequently the byte code program is executed by the machine. The resulting execution is considered to be as fast as machine code is used.

This is especially different to virtual machines or interpreters which execute an intermediate step to run a given program.

The underlying idea to work on this domain is the increasing amount of single byte code commands that are correctly translated by a JIT. Accordingly counting the number of correctly translated byte codes is used as a measure for project progress or functional completeness. Figure 16.1 depicts data taken from two exemplary implementations based on the JIT environment requirements. The left diagram is taken from a test participant with long planning horizon while the right participant followed a short planning principle. The abscissa shows the different versions collected while the mantissa displays the completed functionality in percent.

One of the early problems discovered by this tool is that progress is not a monotonic function at least in terms of translated byte code commands. One side effect of software development is that small changes may remove a major part of functionality which may occur intentional or unintentional as can be seen in Figure 16.1. For example functions that might work correctly may by removed by a comment in order to test a different function. The resulting overall progress might be blurred up to the point that the complete functionality only appears near completion of an experiment.

Using this environment influences software development on its own. As a subset of tests is used for acceptance testing, using it early during development leads to a different course of the variable as using it right before the end of the experiment. Accordingly availability of the environment is a design question of an experiment that must be answered in the beginning.

Additional information provided by the Progress Measurement Environment is the change in overall instructions and the change in branches which is shown in Figure 16.2 (sides resembling to the data above). As noted above monotonic increase in size (instructions) and complexity (branches) may not be given naturally all over the time, as severe shifts in these variables can be seen. This leads to the problem of assessing project completion, because measuring the variables presented here during an ongoing project might not indicate the maturation of a project. For planning software engineering processes measuring of the variable of progress might be a problem.

In addition to measuring the internal variable of progress, the environment allows to dynamically define an acceptance test. It is possible to define a minimum set of byte code commands that must be developed in order to successfully develop the compiler. The advantage in this case is that the number of features may by selected based on the average development speed. In addition to the dynamic definition of development success this design omits the problems when creating one complex functionality as an experiment task.

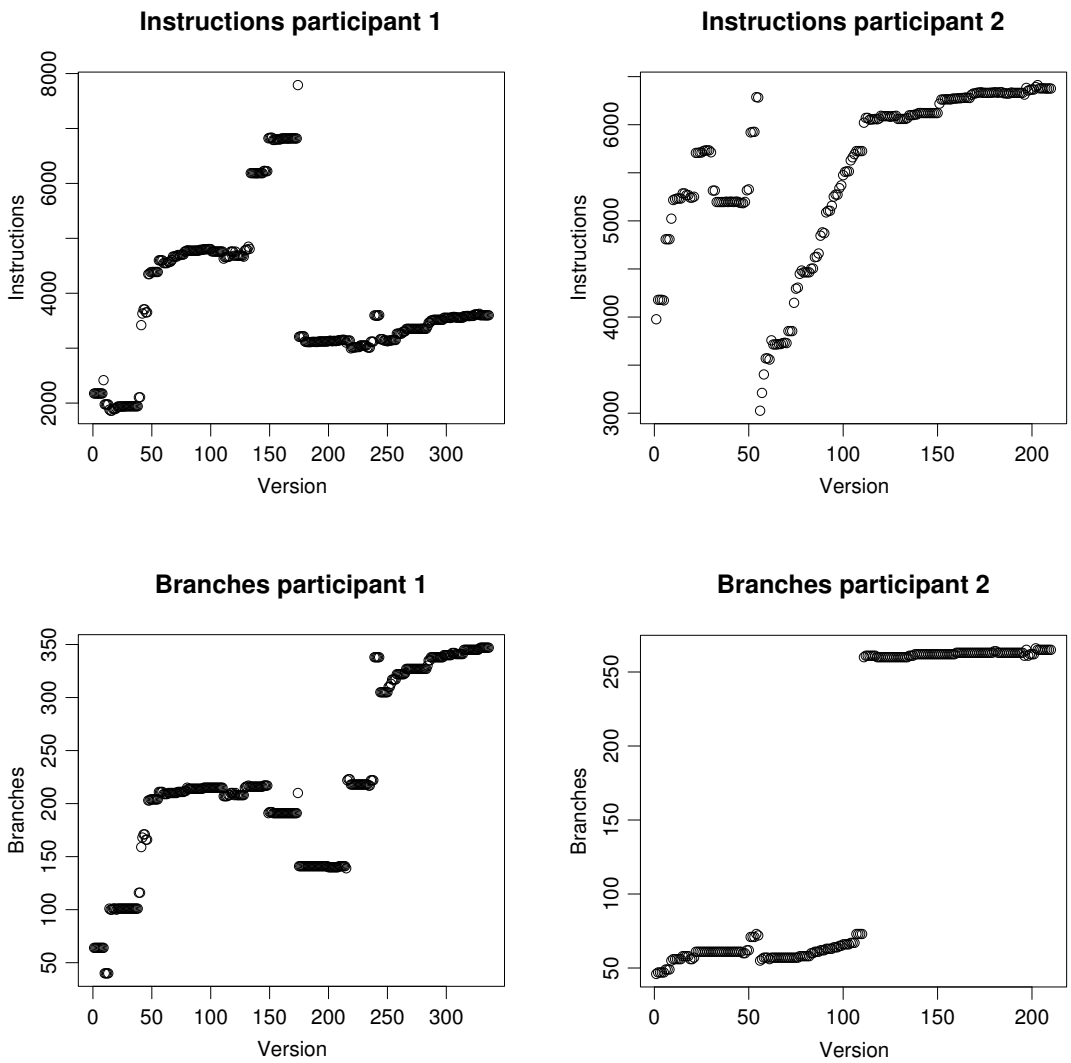

Figure 16.2: Changes of instructions and branches for two exemplary implementations

The later design might not allow to finish only fractions of a requirements specification. For an experiment with a fixed time frame this inability might lead to a high amount of missing data due to unfinished software versions. Summing up advantages of the Progress Measurement Environment comprise additional internal variables as well as benefits for general experiment design.

17 Disturber

The disturber is a .Net based tool that was developed for the purpose of treatment execution control. Its main application is that it opens a dialog after a certain time has elapsed. The dialog contains experiment specific information. This information may range from a survey used for data collection or a reminder to execute a certain treatment.

As an example the disturber was used within the refactoring experiment described in Chapter 5. It controlled the execution of a refactoring (and the control groups treatment) by showing a full screen dialog box asking to execute the according treatment. The dialog could not be hidden and it took five seconds to be allowed to close the window. This was made to assure that the current work was interrupted in order to ease switching to the treatment task.

Although the tool itself is very simple in nature the control enhancement of an experiment is enormous. Especially software engineering techniques with a recurrent type of work may be supervised by the tool. In terms of analysis the tool directly collects time based data from participants. The reason is that an interrupt leads to an application of a treatment which results in a difference in experiment data. By using a survey within the dialog box variable values may be gathered directly. Other measurement variables like executing refactoring or a documentation of the source code imposes differences in the data, too. Changes in the case of technique application were rather indirectly and had to be measured by comparing source code changes.

Another data source of this tool is the logging of user events. This data comprise events like login and logout tasks of a user as well as locking the screen which is regarded as a break. Thus development times can be gathered easily by calculating the time between login and logout.

One way to refine this simple tool would be to check whether a treatment task was executed or not. For example for refactoring, a check list of single refactoring steps can be provided which the user has to complete. Another way would be to check if a certain variable (lines of code) has changed. Thus using additional interaction and post processing of variables the tool might enhance control even more which is regarded to benefit the resulting data's validity.

Part VI

Conclusion

18 Lessons Learned

This chapter presents impressions and tacit knowledge regarding experiments' and the thesis' main work. Although this collection is regarded as informal in nature, the future direction of research as influenced by the thesis' main outcome is made up of the sections presented herein.

18.1 Lessons Learned for Empirical Work

Empirical work has a very negative nature. This appears in three different manifestations: first, in order to show an effect the absence of the opposite must be shown [11]. If that intended outcome of a result does not occur, this result is regarded frustrating and negative. Finally showing a result that may be unpleasant and thus is not published is regarded negative, too. These aspects are problematic because due to software engineering experimentation being uncommon and the execution being somewhat unrestricted design wise, criticism of certain experiment aspects is too simple. Consequently the "mechanical" execution of experiments is uninviting leading to exploratory studies. In fact simple collection of data without specifying a concrete hypothesis but with rigid control of variable quality is much more appealing. Although results cannot be considered as significant as they might occur due to mere chance, the results of such an approach may be used as a basis for a subsequent "real" experiment. Following the correct way ([12]) of having an assumption, creating variables and measuring for significance appears as an inferior approach with a low chance of success especially when variables have never been measured before.

Software engineering experiments must be done without the boundaries of technology. The reason is that most technical aspects change and are merely created due to a trend. Thus proving the effectiveness of a technique has a short time of relevance. As most technologies or process related techniques are based on implicit assumptions, it is considered more valuable to elaborate these implicit assumptions.

Funding of experiments is considered critical. The reason for this is that participant intention changes dramatically. One of the most important changes is the quality of the software itself, which is aimed to be complete and reliable

by paid participants. For lab course based experiments software quality is threatened as a student might only aim at a quality sufficient to receive a certificate. Further questions of reliability are dropped and testing in order to detect failures are omitted as these would lead to additional effort. Thus quality of experiments is considered to be higher when participants are paid.

One aspect of experiment design is the usage of a cross design. By using this design type each participant has to do both (or all) treatments that are part of the experiment procedure. The aim is to reduce effects of previous knowledge that might create unwanted differences in the independent variables being measured. While cross design allows to reduce effects imposed for example by the human factor, two disadvantages have to be mentioned. First a general history effect might seriously affect software engineering experiments. If a task has been developed successfully, the general approach to finish it might be memorized easily. Following this path to success with another treatment leads to shorter development time ignoring any treatment effects. The second problem using cross design is considered to be created by embedded systems. The underlying problem is the technical difference in the hardware platforms as well as the development environments being used. Productive (and realistic) software development requires a considerable amount of preparation regarding the hardware specific aspects of embedded systems. Summing up, the application of a cross design for embedded systems software engineering experimentation requires a well thought preparation.

18.2 Lessons Learned for Software Engineering

Interestingly the lessons learned for software engineering are based on observations of negative participant behavior during execution of experiments and lab courses. Consequently the first thing to be mentioned is that software engineering must be able to handle mediocre developers. The demand to have experts cannot be fulfilled for every project. Thus techniques and methods to support developers who tend to work rather unreflected or unambitious are considered important. Even when techniques are given, it must be made sure that these are applied. This is a matter of control for experimentation, but for an actual project sustainability of technique application becomes a problem. It depends on the individual developer if he is able to adopt a technique or if programming itself imposes such a difficulty that no other techniques can be executed by that person.

One of the most challenging aspects for embedded systems consists of programming peripheral hardware. Accordingly solving the basic handling of

sensors and actuators should have been the first step. This was not done by all participants and the focus of their work differed. Sometimes details were worked out like button and LED based user interaction, although this had to be considered secondary compared to the main functional task of the lab course. Another problematic focus appeared once as one participant used a complete object oriented design within an embedded, C language based programming task. Although the participant was successful, the resulting code size, code design and development time deviated considerably from the result of other participants. This cannot be simple considered as bad, but the impression remains that this capable programmer would have needed a fraction of time for the task if he used traditional procedural development. Another rather common behavior was a trial and error approach in which a complex code fragment or a peripheral aspect of the microcontroler was not programmed in a controlled, thoughtful way, but variable manipulation was rather random. Understanding of the problem or situation was not an aim of the participants resulting in an absence of testing. This behavior appeared as trial and is considered harmful for software engineering, too. Summing up a multitude of good and bad development "habits" were encountered. Controlling or rating them was not directly possible, but they seemed to have an impact on software engineering.

18.3 The Black Matter of Software Engineering Experimentation

The measurement of development progress of a project revealed a serious problem. It occurred clearly during the pretest group of the second planning horizon experiment. The counting of successfully compiled byte code commands of the JIT compiler was used to assess the project's progress. To allow the participants to get feedback of their JIT compiler's status, a test environment was given to them. Both participants applied the test environment during a different point of time of their project. While one used it very early during the project, the other developer used it very late. The problem is that for the second developer, although he developed a usable and meaningful source code, the progress is not measurable externally for most of the time. After that most functionality was coming at once as depicted in Figure 16.1 on the right side. The work in the background with lines of code that are added or changed is regarded as black matter, because even without measurable increase in functionality or lines of code, progress might be made. It can be considered

as cognitive effort which is used to debug or understand a problem. Although time is consumed by this task quantification is very difficult. Especially the side effect of suddenly non working programs is unacceptable, as this does not reflect the effort or extend that resides in the source code, but is not measurable temporary.

The only hint on progress might be given by lines of code, which increase during that time. The quality of this variable is regarded at least uncertain, as code might not be used by the program or it might be removed during a later stage as presented in Chapter 12.

Being unable to measure a project's status that is having no hint on functionality, it is not only unacceptable from an empirical standpoint. Meaningful progress must be the base of a reliable project planning. Thus if functionality is removed, the duration of it missing, its permanence and probability of future occurrence must be accountable. One weak solution to assess the project status might be task based software planning as used in Extreme Programming. The basic principle is to estimate effort for tasks with a duration of only a few days. Tasks include development of functionality, tests, debugging tasks and so on. Accordingly a project status may be represented and measured in a better way using this kind of information. The only drawback is that the data source is human related and thus, validity is regarded not too high.

18.4 Effect Strength

A very important concept is termed effect strength. In computer science it appears for example as the term "silver bullet" which is used to denote strong factors. Regarding software engineering experimentation, effect strength is directly related to the number of participants needed within an experiment. The result of such an analysis is a probability value describing the likelihood of an experiment to show successfully a given effect strength for a given number of participants. This analysis which should be part of an experiment as first described by Miller et al. [62] and refined later by Miller [61], is called power analysis.

One source for estimating the required number is given by the works of Cohen [20] for statistical tests like t-tests, χ^2-tests, or correlation analysis (amongst others). Cohen presents power tables which for a given number of participants n and an effect strength d present the likelihood of an experiment to positively show a given effect strength. Effect strength is defined based on the type of

test, and for the t-test the effect strength d is given as

$$d = \frac{m_a - m_b}{\sigma}$$

with m_a and m_b being the population means (of the original measurement) and σ as the standard deviation. For example a medium effect strength of 0.5 would have a probability of 0.8 to be shown by an experiment when 80 participants took part. For a weak effect strength of 0.2 to have the same probability, 500 participants would be needed, while a strong effect of 1.0 would need only 21 participants. Another origin of power estimation is given by [12] in the form of a table for multiple tests which can be used as a rule of thumb, too.

Only for the refactoring experiment a power analysis was executed. As for this experiment bootstrap methods were used, the power analysis was executed using a resampling based approach. In order to calculate the probability experiments are simulated based on the original data. The resulting data is compared to the observed difference in means and the fraction of experiment with a same or higher difference is used to estimate the power of an experiment. By using more participants within the simulation, power estimates for higher values of n can be calculated. An example of this post analysis process is presented in Section 5.3.3.

The relationship of effect strength, number of participants and likelihood to show a difference within an experiment is important especially during the design phase of an experiment. Numbers of 30 participants or more, which are proposed for human sciences, appear as a stable value at least for medium to strong effects. Regarding the ability to estimate the power during execution, an experiment may be tested for power during its actual execution. Thus one rather uncommon way would be to start an experiment and execute it over time. In contrast most experiments are executed only for a certain, short period of time. Regarding this execution, apart from higher power values, funding may be a bit easier as it may be spread over a longer period of time.

19 Results

The last chapter presents the main contribution of this thesis to embedded software engineering and software engineering experimentation. The chapter is closed by intentions for future work consisting of a perfectly predictable software engineering experiment.

19.1 Agility and Embedded Systems

Although a direct positive effect of a single agile technique could not be found, some hints on the benefits of agility were found. Regarding refactoring a reduced memory consumption for the program text of the resulting program was found. Although not part of a major hypothesis, differences were found to be unlikely caused by randomness. The reason for this reduced memory assumption lies in the aggregation of code into single functions, or in a more general sense in the design principle of "once and only once". As memory consumption is important for embedded systems, application of refactoring might help to stay resourceful in this regard.

The analysis of types of work as presented in Chapter 4 revealed that most of the function related code is developed constantly throughout a project using the technique of short releases. In contrast initial planning appears to lead to architecture related development in the beginning and functional work in the end. A rather interesting result is that a higher fraction of work in short releases is spent on defect related development and general change of software. Two interpretations of this could be given: either short iterations lead to a higher amount of failures and change, or due to early functional development, problems are found earlier. Long planning projects appears to have long phases were no defect or change related work is done. Only at the very end the fraction of work for these issues increases.

The final indirect result concerning agility is that the basic work style of programming is rather based on randomness (cf. Chapter 12). The randomness can be observed in form of lines of code that are created but are later deleted and thus do not appear in the final version of the program. In addition the module that is worked on does not follow a straight sequence. Changes may

occur during later stages of a software development task. This observation leads to an indirect justification of using Agile Methods, as the ability to cope with change is said to be one major property of Agile Methods [7].

Negative aspects of Agile Methods can be measured for refactoring where an overhead for this technique can be shown. The reason for this clearly originates from the added changes enforced by this technique. Due to the absence of structural improvement the observation is considered problematic. Regarding the analysis of work types presented in Chapter 4, one unexpected observation was a peak in absolute development time for the short iteration based development right before the end of the project. As agility was assumed to circumvent such effects due to constant addition of functionality and defect reduction, results appear to support longer planning phases which provide a fixed development sequence. The last problem of Agile Methods is that their effect strength may not be sufficient to justify their application. Even though it might be possible that their effect strength is high and just by chance did not occur in the experiments carried out, their effect strength appears to be too low to be in the area of a "silver bullet". Especially when compared to the tool chain that was used, the subjective impression is that better tools (external test environments, extensive peripheral libraries, faster compile and debug cycle) would have improved the development more than an agile technique. The benefit of using superior tool chains might be caused by the software structure of embedded systems. The special aspect of the embedded software structure is made up of realtime requirements and hardware dependence which are technical problems. Solving these technical problems might only be possible for development methods directly designed for the need of embedded software engineering.

19.2 Variables

Another type of result developed within this thesis concerns the variables used in software engineering experiments. The reliability of variables describes the equality of repeated measurements using this variable. This can be simply tested using repeated measurements. It is influenced for example by noise from uncontrolled variables. Thus it is desirable to have dependent variables which are not influenced by the experiment setting itself. Validity of a variable describes if the intended concept really is measured by a variable. This can be checked by testing expected differences to occur within the variable measured. An example for this was made in the viscosity experiment, where CORBA based source code was correctly assumed to have a lower value than most other

parameters.

Other properties of variables are the scale types that are used. For example interpreting differences between two variables requires an interval scaled variable. Thus the allowed mathematical operations may be reduced up to the point that interpretation becomes meaningless. An example for this is the functional progress variable of the progress measurement environment. While a difference of implemented functionality in terms of compiled byte code commands may be calculated easily, interpreting this difference might be meaningless. The reason is that effort and complexity might be different for byte code commands. One very complex command may need a higher programming performance than two easy byte codes. The underlying assumption of that variable "who programmed more" accordingly might not be answered correctly.

Finally relevance of a variable must be considered, too. This is based on the usefulness for current theories and practicability.

In order to provide an overview of the variables used within the experiments, Table 19.1 provides an overview of variables along with a subjective assessment of usefulness (given as low, medium, high) that were used in the experiments of this thesis.

Variable	Reliability	Validity	Scale	Relevance
Time	Medium	High	Ratio	High
Progress	Medium	Low	Ordinal	Medium
Experience	Low	Low	Ordinal	High
C Knowledge	High	High	Interval	Low
Viscosity	High	High	Ratio	Medium
Lines of Code	Medium	Medium	Ordinal	High
Cyclomatic Complexity	Medium	Medium	Ordinal	High

Table 19.1: Variable assessment as used within the thesis

Variable reliability is considered high for the human based variables created in this thesis because the creation is based on a high number of questions and participants together with tests that assure linearity and Rasch conformance. When doing repeated measurements values are expected to be the same under the same conditions. This is not true for a survey based experience variable, as it was normally based on single survey entries like "How long have you programed in C?". Entries for this field are considered not to be precise and mood dependent. Validity is considered again high for viscosity and C knowledge, because perceived difficulties were met by the tests. Problems with validity occur for example for the variables progress and lines of code. Both

aim at measuring an increase in work that was done, but both fail to give an accountable value for cognitive work that was applied to the code. Scale indirectly gives an assessment of variable precision. For example using ordinal scaled variables, parameterized statistics test are not feasible. In addition ordinal variables only allow rough models to be created because of a lack of mathematical operations.

Other survey based variables were used in the experiments, but their quality was regarded too low to be mentioned here. The reason is that Lickert (cf. [12]) scales with for example values like "low", "medium", and "high" are imprecise and subjective and the mere selection of a question concept normally is considered to have a low relevance. Examples for questions are "Are you used to Agile Methods?". Although the experimental intention is clear, its interpretation is only of local interest and its resulting values are rough.

The final question generated by this variable assessment is how precise a software project can be measured. This comprises not only an individual view on each variable but a holistic approach to the problem – are all relevant variables available for measurement. The question of variable quality has an influence on the goodness of underlying models and their expressiveness.

19.3 The Perfect Experiment

The term perfect experiment originates from one idea: having a variable that plotted against the time needed to fulfill the programming task results in an obvious correlation. Regarding the execution of further experiments with small numbers, it is important to work with effects of an appropriate strength. Agile techniques like refactoring or short releases may not excel at their effect strength on a software engineering experiment. The human factor and even some basic working styles are assumed to have stronger effects. An example for this are participants which tend to write the complete source code without even one intermediate compiling step. Although this occurred rarely, the impact on the software being developed was catastrophic. Another misbehavior encountered was one of the worst assumptions of embedded software engineering consisting of the principle idea that the machine being programmed was indeterministic in nature. This lead to very slow progress for the effected groups. Finding variables with strong effects which are based on human ability and human behavior is supposed to yield stronger effect strengths than testing individual software engineering techniques.

Adopting non trend related experimentation might lead to more subtle side effects of indirect observations. For example when not using tests throughout

the development, it might be difficult to reach a certain level of software reliability which would be reached otherwise. Accordingly instead of creating experiments which plainly include test driven development, one might prohibit testing for one group and enforce it in a second group. From the point of view of experimentation the later design would not allow to draw the conclusion that test driven development would work. A generic effect of continuous testing might be shown. The same indirect observations might be made for refactoring. It could be accomplished by showing that initial design decisions might be unreflected, inadequate, or based on incorrect assumptions. One might argue than that refactoring of software is necessary. This appears a more viable approach than "mechanical" experiment design based on software engineering techniques. An experiment design like comparing similar methods (e. g. *test first* versus *test last* development [63]) is assumed to test for smaller effects which accordingly needs a higher number of participants and must be regarded impractical.

Finally the identification and quantification of habits and person abilities appears as a viable approach to determine relevant factors for software engineering experimentation. On basis of the strongest factors, it might be possible to successfully predict development times. The Rasch model and the other measurement procedures presented in this thesis will certainly help to discover new variable relations in this direction of computer science research.

A Basic Methodical Approach

A.1 Box Plots

Box plots are used to describe data in a generic way to give an overview of the it as depicted in Figure A.1. They describe the median value, indicates outliers, dispersion and skewness in the data. The box presents the centered half of the data starting with the first quartile (0.25), passing the median (0.5) to the third quartile (0.75). This range is referenced as the interquartile range (IQR). The whiskers indicate data below the first and above the third quartile.

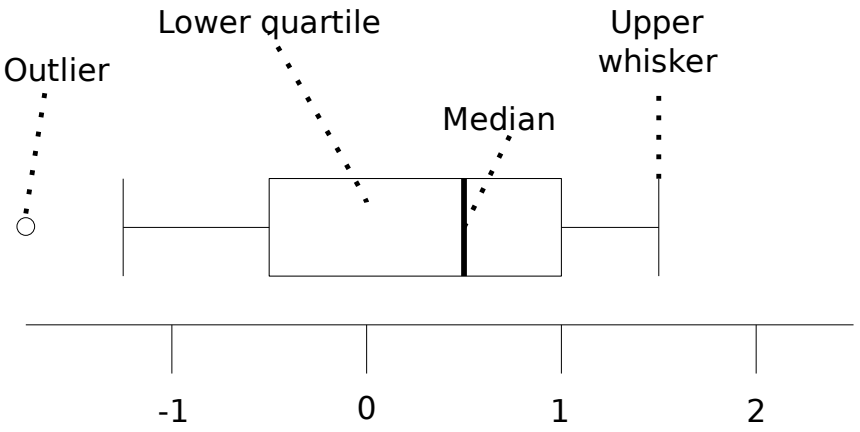

Figure A.1: Overview of box plot elements

Mild outliers are defined as values $1.5 \times IQR$ lower than the first or greater than the third quartile. Extreme outliers are considered to be more than three IQR away from their according quartile.

Skewness can be observed by the relative position of the quartiles to each other. Note that box plots normally are vertical.

A.2 Hypothesis Testing

In order to explain the principle of hypothesis testing, let us first have a look at the basic problem. Imagine two groups of four pigs with the measured weights:

- 6, 4, 7, 5, 3
- 8, 6, 5, 9, 7

The question that must be answered is if these two groups were created by chance. In other words, one must be able to judge how likely the group differences were created by a certain driving factor or just based on randomness. This is only necessary for groups were the difference is not that obvious. If the example above would contain lots of measurements and one group would have values being an order of magnitude greater than the other group, no special test would be needed. Accordingly only "close" groups must be tested.

To help answering the question of chance as the underlying factor three different approaches are presented in the following. First resampling is introduced together with an algorithmic presentation. After that the difference of a T-Test with its resulting values is depicted. Finally, a u-test which is used for ordinal scales, is shown.

A.3 Test Artifacts

Two important concepts of hypothesis testing must be explained here (although it must be admitted that there are more). First two sided (or two tailed) and one sided (or one tailed) tests describe the overall aim of testing non equality or directed (greater than) hypotheses. A hypothesis H_1 of a two sided test asks whether the two pig groups above have different mean weights:

$$H_1 : \mu_0 \neq \mu_1$$

with μ being the mean weight of a group. The only way to show that the hypothesis is true is to falsify the opposite hypothesis:

$$H_0 : \mu_0 = \mu_1$$

The reason for this approach of falsification is the "positive" way of showing the correctness of a hypothesis being unfeasible. To inductively show that every experiment with pigs always results in different mean values leads to an infinite high number if experiments and only one unsuccessful experiment would make the hypothesis illegal. As a consequence, falsification of the opposite is used.

The second important aspect of hypothesis testing is the p-value describing the probability of finding a result like the observed value. In the example above, a p-value of only a few percent for H_0 means that equal mean weights are observed rarely for the two groups. Thus, H_0 can be rejected leading to an acceptance of H_1. Four misinterpretations of the p-value must be mentioned:

- The p-value does not describe the probability of the observation, e. g. it is not $p(observation)$.
- It does not describe the probability of the null hypothesis $p(H_0)$.
- It is not the probability of the alternative hypothesis $1 - p(H_1)$.
- The p-value does not represent the probability of H_0 with the assumption that the observed results hold ($p(H_0|observation)$).

The p-value is the most important value that must be interpreted after running a test. If it is very small, it indicates a low likelihood for the H_0 hypothesis leading to an acceptance of the alternative (main) hypothesis. In general it is accepted to reject a hypothesis when it is below five percent. This acceptance criterion is called the level of significance or alpha error.

A.4 Resampling

The idea of resampling can be shown using an algorithmic description typical for resampling. The aim of the algorithm would be to recreate the observed situation using the assumption of randomness as driving factor. The pig weight example above would be:

1. Combine both groups into one data vector.
2. Repeat 1000 times:
   a) Sample with replacement five pig weights for the first group.
   b) Sample with replacement five pig weights for the second group.
   c) Calculate the mean for both groups.
   d) Record the difference between the two sample means.
3. Count how often the resampled differences exceed the observed difference.
4. Divide the result by 1000 to calculate the probability.

The observed mean values for the pig weight example are five for group one and seven four group two. In consequence it must be calculated how often the sampled mean values have a difference of two or more for a two sided (undirected) test. If the hypothesis was that group two had a greater mean, only the cases with group two having a difference of two or more would have to be counted.

A simulation based on the example pig weights results in 73 values having an (absolute) difference in means of two or more. Thus, the p-value is 0.073 which is more than the generally accepted level of significance of 0.05. Accordingly we cannot reject the null hypothesis that there is no difference in mean weight.

A.5 T-Test

The result of a T-Test is the same as above, namely a p-value describing the probability of observing an extreme result like the measured value. For the pig weight example the p-value for the T-Test is 0.0852. This is insignificant again.

The major difference between the T-Test and resampling is the technical approach to calculate the p-value. While resampling calculates the p-value reusing the observed values, the T-Test transforms the observed group mean values together with the variance of mean differences into t-values. For group sizes of $n_1+n_2 > 50$, the resulting t-values are normal distributed and for smaller groups the values are t-distributed. Consequently the values are interpreted based on the probability of occurrence for a "normalized" normal or t-distributed data. Thus the T-Test circumvents the computer based calculation of the distribution (coining the term bootstrap).

The special aspect of this approach is that certain demands must be fulfilled. First the observed data must be normal distributed in order to allow the transformation. Additionally the variances of both groups must be equal. Finally the statistics used in this test is the mean not being robust against extreme outliers.

A.6 U-Test

Ordinal variables can only be interpreted based on the order relation. This makes the mean statistic unusable for ordinal scaled variables. In order to test significant differences between groups the Mann-Whitney U-Test can be used. This test first converts the observed data into rank sums. Each value is replaced by its occurrence number for both groups. For group one of the pig weight

example above the ranks are 1, 2, 3, and 4 summing up to 10. The rank sum is 27 for group two. This calculation is a simplification as normally a special treatment is used for equal ranks (ties). The test now allows to interpret the observed differences based on precalculated probability tables for the observed rank sums leading to a final p-value.

The approach of the test can be best described by an extreme example. If one group is made up of values one to four and the second group has rank values five to eight it is very unlikely that this outcome is based on chance. The U-Test now helps to assess the case with ranks being mixed within groups and accordingly more equal rank sums for both groups. As this test is rather beautiful it must be pointed out that it uses the central tendency (or median) to assess group differences.

The p-value for the pig example above based on the U-Test is 0.11 being insignificant again.

Bibliography

[1] P. Abrahamsson and J. Koskela. Extreme programming: A survey of empirical data from a controlled case study. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE04)*, 2004.

[2] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. *Agile software development methods. Review and analysis.* VTT Publications, 2002.

[3] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen. New directions on agile methods: a comparative analysis. *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 244–254, 2003.

[4] T. Anderson and P. A. Lee. Fault tolerance terminology proposals. In *Proceedings of the 12th IEEE International Symposium on Fault Tolerant Computing*, 1982.

[5] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett. Understanding and predicting the process of software maintenance releases. In *Proceedings of the 18th International Conference on Software Engineering*, 1996.

[6] V. Basili, R. Tesoriero, P. Costa, M. Lindvall, I. Rus, F. Shull, and M. Zelkowitz. Building an experience base for software engineering: A report on the first CeBASE eWorkshop. In *Product Focused Software Process Improvement : Third International Conference, PROFES*, 2001.

[7] K. Beck. *Extreme Programming. Das Manifest.* Addison-Wesley, 2. auflage edition, 2000.

[8] D. Binkley. Source code analysis: A road map. In *FOSE '07: 2007 Future of Software Engineering*, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[9] B. Boehm. A view of 20th and 21st century software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 12–29, New York, NY, USA, 2006. ACM.

[10] T. G. Bond and C. M. Fox. *Applying the Rasch Model*. IEA, 2001.

[11] J. Bortz and N. Döring. *Forschungsmethoden und Evaluation*. Springer, 3. edition, 2003.

[12] J. Bortz and N. Döring. *Forschungsmethoden und Evaluation*. Springer, 4. edition, 2006.

[13] M. Broy and A. Rausch. Das neue V-Modell XT. *Informatik Spektrum*, 28 (3):220–229, June 2005.

[14] L. Burd and S. Rank. Using automated source code analysis for software evolution. In *SCAM*, pages 206–212, 2001.

[15] J.-M. Burkhardt, F. Deétienne, and S. Wiedenbeck. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7:115–156, 2002.

[16] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Using empirical studies during software courses. *ESERNET 2001-2003, LNCS 2765*, pages 81–103, 2003.

[17] T. Chau, F. Maurer, and G. Melnik. Knowledge sharing: Agile methods vs. tayloristic methods. In *WETICE Twelfth International Workshop on Enabling Technologies Infrastructure for Collaborative Enterprises*. IEEE, 2003.

[18] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transactions on Software Engineering*, 25(4):573–583, 1999.

[19] A. Cockburn. The end of software engineering and the start of economic-cooperative gaming. *Computer Science and Information Systems*, 1(1): 1–32, 2004.

[20] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[21] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the IJCAI-2003*, 2003.

[22] M. Davis. When is a volunteer not a volunteer? *Empirical Software Engineering*, 6:349–352, 2001.

[23] A. C. Davison. *Bootstrap Methods and their Application.* Cambridge University Press, 1997.

[24] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap.* Chapman & Hall / CRC, 1998.

[25] N. Fenton, P. Krause, and M. Neil. Probabilistic modelling for software quality control. In S. Benferhat and P. Besnard, editors, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, LNCS 2143, pages 444–453, 2001.

[26] N. E. Fenton and M. Neil. Software metrics: roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 357 – 370. ACM Press, 2000.

[27] G. H. Fischer and I. W. Molenaar, editors. *Rasch Models.* Springer, 1995.

[28] M. Fowler. *Refactoring. Improving the Design of Existing Code.* Addison Wesley, 1999.

[29] A. Garrido and R. Johnson. Challenges of refactoring c programs. *IWPSE: International Workshop on Principles of Software Evolution*, 2002.

[30] A. Garrido and R. Johnson. Refactoring c with conditional compilation. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, Montreal, Canada, 2003.

[31] B. Georgea and L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46:337–342, 2004.

[32] B. Geppert and F. Rosler. Effects of refactoring legacy protocol implementations: A case study. In *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium on (METRICS'04)*, pages 14–25, Washington, DC, USA, 2004. IEEE Computer Society.

[33] B. Geppert, A. Mockus, and F. Rößler. Refactoring for changeability: A way to go? In *11th IEEE International Software Metrics Symposium (METRICS 2005).*

[34] D. M. German. An empirical study of fine-grained software modifications. In *20th IEEE International Conference on Software Maintenance, 2004*, 2004.

[35] E. Grant and H. Sackman. An exploratory investigation of programmer performance under on-line and off-line conditions. *Human Factors in Electronics, IEEE Transactions on*, HFE-8(1):33–48, March 1967. ISSN 0096-249X.

[36] T. R. G. Green. Programming languages as information structures. In R. S. J. M. Hoc, T. R. G. Green and D. J. Gilmore, editors, *Psychology of Programming*, pages 117 – 137, San Diego, 1994. Academic Press.

[37] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, 2004.

[38] A. E. Hassan and R. C. Holt. Studying the evolution of software systems using evolutionary code extractors. *Proceedings of IWPSE 2004: International Workshop on Principles of Software Evolution*, 2004.

[39] R. Hatzinger and P. Mair. Extended rasch modeling: The erm package for the application of irt models in r. *Journal of Statistical Software*, 20(9): 1–20, 2007.

[40] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[41] W. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, Reading, MA, 1995.

[42] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects - a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5:201–214, 2000.

[43] M. John, F. Maurer, and B. Tessem. Human and social factors of software engineering: workshop summary. *SIGSOFT Softw. Eng. Notes*, 30(4):1–6, 2005.

[44] M. Jørgensen and K. J. Moløkken-Østvold. Eliminating over-confidence in software development effort estimates. In *Conference on Product Focused Software Process Improvement*, Lecture Notes in Computer Science, pages 174–184, Japan, 2004. Springer-Verlag.

[45] J. Karn and T. Cowling. A follow up study of the effect of personality on the performance of software engineering teams. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering*, pages 232–241, New York, NY, USA, 2006. ACM Press.

[46] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance (ICSM02)*, 2002.

[47] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[48] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, New York, NY, USA, 2005. ACM Press.

[49] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1), 1986.

[50] M. M. Lehman. Programs, life cycles, and laws of software evolution. In *Proc. IEEE 68*, number 9, pages 1060–1076, September 1980.

[51] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[52] M. M. Lehman and J. F. Ramil. Software uncertainty. In D. Bustard, W. Liu, and R. Sterritt, editors, *Soft-Ware 2002: Computing in an Imperfect World*, LNCS 2311, pages 174–190, 2002.

[53] R. Leitch and E. Stroulia. Understanding the economics of refactoring. In *The 7th International Workshop on Economics-Driven Software Engineering Research*, 2005.

[54] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 27–38, New York, NY, USA, 2003. ACM Press.

[55] M. Lippert, S. Roock, and H. Wolf. *Software entwickeln mit eXtreme Programming*. dpunkt, 2002.

[56] B. Littlewood and D. R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Trans. Softw. Eng.*, 15(12):1596–1614, 1989.

[57] J. I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *ICTAI '00: Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence*, page 46, Washington, DC, USA, 2000. IEEE Computer Society.

[58] M. Mäntylä, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance (ICSM03)*, 2003.

[59] B. McCloskey and E. Brewer. Astec: a new approach to refactoring c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 21–30, New York, NY, USA, 2005. ACM Press.

[60] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2), 2004.

[61] J. Miller. Statistical significance testing - a panacea for software technology experiments ? *The Journal of Systems and Software*, 73:183 – 192, 2004.

[62] J. Miller, J. Daly, M. Wood, M. Roper, and A. Brooks. Statistical power and its subcomponents - missing and misunderstood concepts in empirical software engineering research. *Journal of Information and Software Technology*, 1997.

[63] M. Müller and O. Hagner. Experiment about test-first programming. *IEE Proceedings Software*, 149(5):131–136, Oct. 2002.

[64] M. M. Müller, F. Padberg, and W. F. Tichy. Ist XP etwas für mich? empirische Studien zur Einschätzung von XP. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Software Engineering 05*, 2005.

[65] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[66] P. C. Pendharkar, G. H. Subramanian, and J. A. Rodger. A probabilistic model for predicting software development effort. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications - ICSSA 2003*, LNCS 2668, pages 581–588, 2003.

[67] M. Pizka. Straightening spaghetti code with refactoring? *Software Engineering Research and Practice*, 2004.

[68] L. Prechelt. The 28:1 grant/sackman legend is misleading, or: How large is interpersonal variation really? Internal Report 18, Universität Karlsruhe, Fakultät für Informatik, 1999.

[69] L. Prechelt. *Kontrollierte Experimente in der Softwaretechnik*. Springer, 2001.

[70] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, New York, NY, USA, 2005. ACM Press.

[71] M. B. Rosson. Human factors in programming and software development. *ACM Comput. Surv.*, 28:193–195, 1996.

[72] J. Rost. *Testtheorie - Testkonstruktion*. Verlag Hans Huber, 2004.

[73] F. Salewski, D. Wilking, and S. Kowalewski. Diverse hardware platforms in embedded systems lab courses: A way to teach the differences. In *Special Issue: The First Workshop on Embedded System Education (WESE)*, volume 2, pages 70–74. SIGBED Review, ACM, Oct. 2005.

[74] F. Salewski, D. Wilking, and S. Kowalewski. The effect of diverse hardware platforms on N-version programming in embedded systems - an empirical evaluation. In *Proc. of the 3rd. Workshop on Dependable Embedded Sytems (WDES'06)*, volume TR 105/2006, pages 61–66. Vienna University of Technology, Nov. 2006.

[75] M. Shepperd. Software project economics: a roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 304–315, Washington, DC, USA, 2007. IEEE Computer Society.

[76] J. E. Sieber. Protecting research subjects, employees and researchers: Implications for software engineering. *Empirical Software Engineering*, 6: 329–341, 2001.

[77] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, 2001.

[78] J. L. Simon. *Resampling: The New Statistics*. Resampling Stats, 1999.

[79] J. Singer, M.-A. D. Storey, and S. E. Sim. Beg, borrow, or steal (workshop session): using multidisciplinary approaches in empirical software engineering research. In *ICSE*, pages 799–800, 2000.

[80] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, and M. Vokáč. Challenges and recommendations when increasing the realism of controlled software engineering experiments. *ESERNET 2001-2003, LNCS 2765*, pages 24–38, 2003.

[81] I. Sommverville. *Software Engineering*. Addison-Wesly, 2001.

[82] W. F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, 1998.

[83] J. B. Todman and P. Dugard. *Single-Case and Small-N Experimental Designs: A Practical Guide to Randomization Tests*. Lawrence Erlbaum Associates, 2000.

[84] B. Walter and B. Pietrzak. Multi-criteria detection of bad smells in code with UTA method. In H. Baumeister, M. Marchesi, and M. Holcombe, editors, *Extreme Programming and Agile Processes in Software Engineering, XP 2005*, Sheffield, UK, 2005. Springer.

[85] Y. Wang. On the cognitive informatics foundations of software engineering. In *Proceedings of the Third IEEE International Cognitive Informatics, 2004*, 2004.

[86] Y. Wang. On cognitive properties of human factors in engineering. In *Fourth IEEE Conference on Cognitive Informatics, 2005. (ICCI 2005)*, 2005.

[87] D. Wilking and S. Kowalewski. Analyzing software engineering processes on source code level. In *In the Proceedings of the The 6th International Conference on Software Methodologies, Tools and Techniques, SOMET'2007 (in print)*, 2007.

[88] D. Wilking, U. F. Khan, and S. Kowalewski. An empirical evaluation of refactoring. *e-Informatica - Software Development Theory, Practice and Experimentation*, 1(1):28–44, February 2007.

[89] D. Wilking, D. Schili, and S. Kowalewski. Measuring the human factor with the rasch model. In *In the proceedings of the 2nd IFIP Central and East European Conference on Software Engineering Techniques, CEE-SET 2007 (in print)*, 2007.

[90] L. Williams, W. Krebs, L. Layman, A. Antón, and P. Abrahamsson. Toward a framework for evaluating extreme programming. *Empirical Assessment in Software Eng. (EASE)*, pages 11–20, 2004.

[91] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering. An Introduction.* Kluwer Academic Publishers, 2000.

[92] A. L. Wolf and D. S. Rosenblum. A study in software process data capture and analysis. In *Second International Conference on the Software Process (ICSP2)*, pages 115–124, 1993.

[93] B. D. Wright and G. N. Masters. *Rating Scale Analysis*. Mesa Press, 1982.

[94] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004. Member-Gail C. Murphy.

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from http://aib.informatik.rwth-aachen.de/. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: biblio@informatik.rwth-aachen.de

2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10 Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-01 * Fachgruppe Informatik: Jahresbericht 2005

2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation

2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09 Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking

2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed

2006-11 Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers

2006-12 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning

2006-13 Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities

2006-14 Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"

2006-15 Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices

2006-16 Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness

2006-17 Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines

2007-01 * Fachgruppe Informatik: Jahresbericht 2006

2007-02 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations

2007-03 Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase

2007-04 Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation

2007-05 Uwe Naumann: On Optimal DAG Reversal

2007-06 Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking

2007-07 Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08 Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches

2007-09 Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption

2007-10 Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes

2007-11 Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke

2007-12 Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13 Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs

2007-14 Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

2007-15 Volker Stolz: Temporal assertions for sequential and concurrent programs

2007-16 Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks

2007-17 René Thiemann: The DP Framework for Proving Termination of Term Rewriting

2007-18 Uwe Naumann: Call Tree Reversal is NP-Complete

2007-19 Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control

2007-20 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems

2007-21 Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains

2007-22 Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets

2008-01 * Fachgruppe Informatik: Jahresbericht 2007

2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing

2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René :Thiemann, Harald Zankl: Maximal Termination

2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler

2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations

2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs

2008-07 Alexander Nyßen, Horst Lichter:: The MeDUSA Reference Manual, Second Edition

2008-08 George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs

2008-09 George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs

2008-10 George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time

2008-11 George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows

2008-12 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages

2008-13 Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs

2008-14 Bastian Schlich: Model Checking of Software for Microcontrollers

2008-15 Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves

2008-16 Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study

2008-17 Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving

2008-18 Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.

Curriculum Vitae

Name	Dirk Wilking
Day of birth	30.03.1978
Place of birth	Osterholz-Scharmbeck

2004 –	Wissenschaftlicher Angestellter am Lehrstuhl Informatik 11 an der RWTH Aachen University
1998 – 2004	Studium der Informatik an der Universität Bremen
1997 – 1998	Grundwehrdienst
1984 – 1997	Grundschule, Orientierungsstufe und Gymnasium in Osterholz-Scharmbeck