EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Exploring the effect of UML modeling on software quality

Flaton, Bas

Award date:
2008

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Exploring the Effect of UML Modeling on Software Quality

By
Bas Flaton

Supervisors:

dr. Michel R. V. Chaudron (TU/e)
ir. Franc Buve (Logica)

Eindhoven, June 2008

Abstract

Modeling ones design is generally considered good practice in the process of software development. Numerous payoffs are attributed to this practice, including increased software quality and easier software maintenance. However, empirical studies validating these payoffs are scarce. This validation should nevertheless be considered very important, since models do not represent any value by themselves. Therefore, only when payoffs have been shown to exist, should a company be willing to invest time and resources into creating these models.

The *de facto* standard for communicating a system's design is the Unified Modeling Language (UML). In this study we explore the relation between the level of detail of a system's UML diagrams – specifically class, sequence and state diagrams – and the defect density and average defect repair time of the resulting implementation. We did this by performing two case studies, both in an industrial setting, applying different approaches for each of them.

For both approaches definitions were formulated on how to record UML level of detail measures of the mentioned diagram types. While the first approach used qualitative rankings, the second applied design metrics to quantify this detail level.

Defect samples were manually inspected to try to relate each defect to parts of the UML models. During this process the defects were also typed, so that analyses could be performed using both the entire set of defects and specific defect types when required. To facilitate this defect typing task, defect taxonomies were compiled from the ones found in scientific literature and further tailored to the needs of this study.

The results from our case studies show evidence supporting the suggested payoffs: the availability of UML models lowers defect density and average repair time. However, some questions regarding validity of this evidence remain.

Preface

During my study of Computer Science at the Eindhoven University of Technology, I became interested in software architectures while following the Software Architecting course. Identifying (and ideally sidestepping) bottlenecks in software development by thinking about software structures at a high level of abstraction seemed a challenging, and therefore interesting task. A task that I would not mind performing after my graduation.

Unfortunately, there were not many follow-up courses available for the Software Architecting course. I decided that doing a master project in this area would provide an opportunity to learn more on this topic. During an initial talk with Dr. Michel Chaudron, who taught the Software Architecting course, we discussed options for doing a project within the EmpAnADa group. This group specializes in applying empirical analysis methods on software architecture related research questions.

Dr. Chaudron suggested researching the effectiveness of applying UML modeling techniques in order to lower rates of defect introduction and defect repair time in software development. This seemed a very relevant question and one that could provide useful information for further research and application of UML in industry alike —a feature I deemed very important. I accepted this topic and the rest, as they say, is history.

There are several people I wish to thank for their support during my graduation project. At the risk of accidently missing out somebody, I would like to express my gratitude to them here:

First of all I would like to thank Dr. Michel Chaudron for his willingness to supply me with both an interesting graduation project and to-the-point advice during the research process. His continued optimism on the feasibility of the project kept me going at the tougher stages of the process.

Second, many thanks go out to Ariadi Nugroho for the wonderful collaboration in analyzing the second case. I enjoyed our frequent discussions regarding details of the approach we chose for our analysis and wish him the best of luck in acquiring his doctorate.

Dr. Kees Huizing and Dr. Loek Cleophas I thank for their willingness to be part of my examination board. This despite the fact that this thesis would arrive at them on a rather short notice.

Other people I owe a word thanks are:

- Franc Buve, Felix Donkers and Lambert Mühlenberg, for the support they gave me during my internship at Logica. They offered great assistance in prioritizing my efforts and keeping the project on track planning-wise, as well as provided me with useful feedback.
- Tom Jansen, whose relentless support efforts during my internship at Philips Applied Technologies enabled me to perform the first case study.
- Christian Lange and René Ladan for their great company and for fulfilling the role of sparring partner during the time I spent within the EmpAnADa group.
- All developers who allowed me to take a look at their work, and providing help and feedback whenever needed.
- All co-graduate-students at the "Working Tomorrow" department of Logica Eindhoven for the wonderful time I had during my time there and the useful input they provided.

Last but not least I cannot think of a more appropriate place than in my master's thesis to express my deepest and sincerest gratitude to my parents, Cilly and Jos Flaton, for supporting me in every possible way, during my entire time at the university.

Bas Flaton

Eindhoven, June 9, 2008

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Research questions . . . 2
- 1.3 The approach in a nutshell . . . 2
- 1.4 Empirical research . . . 3
  - 1.4.1 Empirical approach . . . 3
  - 1.4.2 EmpAnADa . . . 4
- 1.5 Outline . . . 4

2 Nomenclature **7**
- 2.1 Defect . . . 7
- 2.2 Design . . . 7

3 Related research **9**

4 Case 1: Blu-ray Middleware **11**
- 4.1 BDMW description . . . 11
  - 4.1.1 RE/ROM HDMV component . . . 11
  - 4.1.2 BD-J application lifecycle management component . . . 12
- 4.2 BDMW definitions . . . 12
  - 4.2.1 Defect taxonomy . . . 12
  - 4.2.2 Design detail . . . 14
- 4.3 BDMW data gathering . . . 16
  - 4.3.1 Step one - gather code metrics . . . 16
  - 4.3.2 Step two - gather project characteristics . . . 18
  - 4.3.3 Step three - collecting defect data . . . 21
  - 4.3.4 Step four - typing defects . . . 21
  - 4.3.5 Step five - determining UML detail . . . 22
  - 4.3.6 Data accuracy . . . 22
  - 4.3.7 Storing the data: the database . . . 23
- 4.4 BDMW results . . . 23
  - 4.4.1 Additional analysis note . . . 23
  - 4.4.2 UML and defect prevention . . . 24

4.4.3 UML and maintenance 27
4.5 Conclusions first case 32
4.6 Case-specific threats to validity 32

5 Changing the process **35**
5.1 Problems with the BDMW approach 35
5.2 The new approach 36
5.2.1 Metrics 36
5.2.2 Eliminating the coverage measure 36
5.3 Drawbacks to the new approach 37

6 Case 2: PARTS **39**
6.1 PARTS description 39
6.2 PARTS definitions 39
6.2.1 Defect taxonomy 40
6.2.2 Design detail 41
6.3 PARTS data gathering 43
6.3.1 Step one - gather code metrics 43
6.3.2 Step two - gather project characteristics 43
6.3.3 Step three - collecting defect data 45
6.3.4 Step four - typing defects 46
6.3.5 Step five - determining UML detail 47
6.3.6 Storing the information: the database 49
6.4 Results 49
6.4.1 Additional analysis notes 49
6.4.2 Comparing defect density 50
6.4.3 Comparing average defect density per defect type 54
6.4.4 Correlating level of modeling detail to defect density 55
6.5 Conclusions second case 62
6.6 Case-specific threats to validity 64

7 Conclusions and evaluation **65**
7.1 Answers to the research questions 65
7.1.1 Research Question 1 65
7.1.2 Research Question 2 66
7.1.3 Research Question 3 66
7.1.4 Research Question 4 68
7.1.5 Research Question 5 68
7.2 Threats to validity 68
7.3 Recommendations 69
7.4 Future work 70
7.5 Guidelines for applying UML 71

A Existing defect taxonomies **73**

A.1 Taxonomy suggested in [CKC91] 73
A.2 Taxonomy suggested in [CBC+92] 73
A.3 Taxonomy suggested in [LTW+06] 73
A.4 Taxonomy suggested in [Bur03] 74
A.5 Taxonomy suggested in [IEE93] 75

B Database design 77
B.1 The BDMW analysis database 77
B.2 The PARTS analysis database 78

C Performed queries 81
C.1 PARTS: comparison of defect density modeled vs unmodeled system parts 81
C.2 PARTS: comparison of average defect density per defect type . 83
C.3 PARTS: correlation test LoD and defect density 83
C.3.1 All defect types, all sequence diagrams 83
C.3.2 All defect types, IOR sequence diagrams 89
C.3.3 Individual defect types 91

D Statistical tests 93
D.1 Kendall's τ 93
D.2 Mann-Whitney U-test 93
D.3 Kruskall-Wallis test 94
D.4 Shapiro-Wilk test 94

Bibliography 95

List of Figures

4.1 Defect density (defects/KSLoC) distribution. 26
4.2 Repair effort per defect type. 29
4.3 Per project modeling and repair effort ratios. 31
4.4 Per project modeling and repair effort (hrs per KSLoC). 31

5.1 Coverage approximation - graphical explanation. 37

6.1 PARTS: defect type distribution 46
6.2 PARTS: Defect density (per SLoC) of modeled and unmodeled system parts 51
6.3 PARTS: Defect density (per SLoC) of (split up) modeled and unmodeled system parts 53
6.4 PARTS: Average defect density distributions for modeled and unmodeled system parts. 54
6.5 PARTS: Sample defect density (per SLoC) (all SDs) 56
6.6 PARTS: scatterplot of correlation between LoD and defect density (per SLoC) (all LoD parts, all SDs) 57
6.7 PARTS: Sample defect density (per SLoC) (only IOR SDs) 59
6.8 PARTS: scatterplot of correlation between LoD (all parts) and defect density (per SLoC)(only IOR SDs) 60
6.9 PARTS: scatterplots of correlation between LoD (interesting parts) and defect density (per SLoC)(only IOR SDs) 60

7.1 Defect density (defects/KSLoC) distribution. 67
7.2 PARTS: Average defect density distributions for modeled and unmodeled system parts. 67

B.1 BDMW database schema. 77
B.2 PARTS database schema. 79

List of Tables

4.1 Class diagram detail attributes 15
4.2 Sequence diagram detail attributes 15
4.3 State diagram detail attributes 16
4.4 HDMV code metrics 17
4.5 ALM code metrics - Java 17
4.6 ALM code metrics - native 17
4.7 Project average model detail, completeness and defect density . . . 25
4.8 HDMV coverage of implementation by diagram types. 25
4.9 ALM coverage of implementation by diagram types. 26
4.10 Average defect repair effort per project 28
4.11 Repair effort per defect type versus created UML diagrams 30

6.1 PARTS code metrics 43
6.2 Modeled and unmodeled defect density: test for normality 51
6.3 Modeled and unmodeled defect density: Mann-Whitney test 52
6.4 Splitting up modeled defect density: Kruskal-Wallis test 52
6.5 Sequence diagram modeled and unmodeled defect density: Mann-Whitney test 53
6.6 LoD and defect density correlation: test for normality 56
6.7 LoD and defect density correlation: Kendall's τ 57
6.8 LoD and defect density correlation: test for normality 59
6.9 LoD (IOR based) and defect density correlation: Kendall's τ . . . 61
6.10 LoD and defect density correlation (individual defect types): test for normality 61
6.11 LoD (logic, all SDs) and defect density correlation: Kendall's τ . . 62
6.12 LoD (logic, IOR) and defect density correlation: Kendall's τ 63
6.13 LoD (user data i/o, all SDs) and defect density correlation: Kendall's τ 63
6.14 LoD (user data i/o, IOR) and defect density correlation: Kendall's τ 63

C.1 Result set of query from Listing C.1 82
C.2 Result set of query from Listing C.2 83
C.3 Result set of query from Listing C.3 88
C.4 Result set of query from Listing C.4 90

Chapter 1

Introduction

In this introductory chapter the motivation for performing this study is given. After this the research questions central to this study are postulated and explained. We then give a brief introduction to the empirical nature of this study. Finally an outline of the rest of this thesis is given. After reading this chapter the reader should have the basic understanding of the content of this thesis and the reasons for which it was written.

1.1 Motivation

Design modeling is generally considered good practice in almost any software development process. When studying modeling in scientific literature (see for instance [BCK03], Section 2.4), numerous pay-offs are attributed to it. These pay-offs include, among others, the following:

- Increased productivity (e.g. through easier component reuse and less rework due to misinterpretation of specifications)
- Better communication (e.g. through better domain understanding)
- Increased quality (e.g. less bugs introduced, non-functional requirements better met and allowing for better testing)
- Easier maintenance (e.g. through a better overview of the system)
- Better planning (e.g. better estimates, more effective personnel deployment)

While some of these pay-offs seem to be based on very credible reasoning, empirical investigation on the validity of these relations has up till now been scarce. Nevertheless this validation should be considered crucial by practitioners, since only then it would be justifiable to make any investments in software modeling techniques. This is because the models *by themselves* do

not represent value and in that way do not have a positive influence on a company's ROI[1].

Although it is very well possible that this usefulness of UML varies (between projects, people etc.), we believe that it does not do so arbitrarily. Instead of debating the overall applicability of a rigorous, upfront modeling approach versus the more agile approaches, we see an ideal solution in a decision tree that, given some project characteristics (like inherent complexity, team composition, time-to-market requirements, safety requirements etc.) can accurately predict the level of modeling that should be applied.

1.2 Research questions

For this study we focus on the influence of design modeling on software quality. Software quality is still a very broad subject. Specifically we are interested in creating bug-free software. In order to minimize the amount of defects in software, three activities can be used: defect prevention, defect detection and maintenance.

Using design documentation to improve defect detection tasks is already topic of various studies. Research ranges from the creation of highly generalized test cases, to automatic test generation using the available models and making predictions about which parts of a software product are most likely to contain defects. This study will focus on the other two activities – prevention of defects and maintenance activities – and whether they can be improved using models. Specifically, we will focus on the following research questions:

- RQ1: How does the level of detail in UML models influence a project's defect density?
- RQ2: How does the level of detail in UML models influence the defect density of individual types of defects found in a project?
- RQ3: How does the level of detail in UML models influence a project's average defect repair time?
- RQ4: How does the level of detail in UML models influence the average defect repair time of individual types of defects found in a project?
- RQ5: Does upfront modeling provide enough payoffs to justify its application?

1.3 The approach in a nutshell

We present the approach that was used to find answers to the above questions here, without going into details yet. The chapters that cover the individual

[1] Return On Investment —the ratio of money gained (or lost) on an investment.

cases will treat all these details in full. Nevertheless, it is useful to have a bird's-eye view of the general approach we had in mind, while performing this study. We will list the steps for an arbitrary case:

- Select a defect for further analysis from a bug tracking tool.
- Find out what changes were actually applied to fix the defect, by looking at the "before" and "after" images of the files that were altered during the repair task.
- Compact this information by assigning a type to the defect, according to a defect taxonomy.
- Having more detailed knowledge of the applied changes, look at the design documentation to find parts of the design that mention the changed part.
- Whenever related design parts are found, record their UML detail level.

After having performed the above steps for a decent amount of defects, we search the recorded information for answers to the questions from the previous section.

1.4 Empirical research

In this section some relevant information is provided regarding the nature of empirical research.

1.4.1 Empirical approach

There are many ways to conduct empirical research. One approach would be to do controlled experiments, which usually boils down to selecting a group of people for an assignment, which is specifically tuned to the study at hand, and either monitor the process of completing this assignment, or compare the results of individuals or groups. Advantage of this approach is that a lot of environment variables are under the control of the researchers (e.g. experience level of participants, available tooling, time constraints etc.). This can, however, pose threats as to how far the results are applicable to "real world" scenarios.

Another approach is performing case studies. Here the study is conducted in an actual industrial software development environment. This can add some confidence to the validity of the results in practical situations. Here, the hard part lies in the diverse nature of software development projects. Having practically no control over the variables mentioned in the previous paragraph makes it hard to interpret results, or, for that matter, to even try to predict which measurements are attainable and which are not.

For this study we chose the latter option of case study. This was partially done because the small amount of research on the topic was already mostly of the controlled experiment kind. Another motivation was that, since the study would be executed during an internship, this should ease the task of finding suitable cases – an opportunity that should not be cast aside lightly.

On a sidenote: the fact that both kinds of research should be considered invaluable is backed by other scientific fields. Looking at the fields of biological and medical science the same distinction in studies can be seen. In some areas both types of studies are even made mandatory, for example when developing new medication or new pesticides. Here the distinction is worded using the terms *in vitro* ("in a glass (test tube)" referring to analyses performed in laboratories) and *in vivo* ("in the natural environment").

1.4.2 EmpAnADa

Within the SAN expertise group a program was started to promote and execute **Emp**irical **An**alysis of **A**rchitecture and **D**esign Qu**a**lity (EmpAnADa). Its goal is to develop techniques to improve the quality of UML models. For the validation of proposed, quality improving techniques, both controlled experiments and case studies have frequently been conducted. EmpAnADa members therefore proved invaluable sparring partners in setting up this study.

1.5 Outline

After having devised a first approach to test the effect of UML level of detail on defect density and maintenance tasks, we tested it on an industrial case. With the experience and problems of this first case study in mind, we altered the approach before performing a second case study. The document is structured to reflect this division into two approaches and their respective case studies, thereby allowing for each case study to be read as a self contained part. The resulting outline is given below:

- In Chapter 2 some general terms are defined, so that there will be no confusion regarding their meaning, while reading the remainder of the document.
- Chapter 3 continues by describing some research topics related to this study.
- The first case will be handled in Chapter 4. Before listing and discussing the results acquired from the first analysis, it states all definitions used for this analysis and describes the applied approach.

- After analysis of the first case, some problems emerged that forced us to change our analysis process. Both the problems and the solutions found to deal with them are covered in Chapter 5.
- Having changed the approach according to the outcome of Chapter 5, we analyzed a second case. The results of this second case study are given in Chapter 6, which follows the same structure as Chapter 4.
- Chapter 7 answers the research questions based on findings from the two case studies. It also provides critique on the performed studies and discusses some questions regarding the validity of the entire study and its results. Finally some recommendations are given and directions for future research are suggested.

Chapter 2

Nomenclature

In this chapter the meanings of terms are given, that will frequently appear in the remainder of this thesis. Since these terms have numerous meanings in computer science literature it is important to explain *our* interpretation of them here.

2.1 Defect

In scientific literature the use of terms like defect, bug, fault and failure is rather erratic. In this thesis we use the definition as it is given in [Boa06], as we consider testers to have most experience in dealing with this terminology.

> **A defect is** a flaw in a component or system that can cause the component or system to fail to perform its required function, e.g. an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.

Where failure is then defined as:

> **Failure is** the deviation of a component or system from its expected delivery, service or result.

According to [Boa06] the term defect is synonymous to the terms bug and fault. Although we try to use the first term as much as possible, we occasionally use the others to comply with common practice (e.g. a *bug* tracking tool).

2.2 Design

In scientific literature one can find a lot of definitions of the word *design.* These differ mostly in levels of abstraction. Whereas some like to use design as the term for *architectural* layout of the software (high abstraction), others

use the word when describing “what the code does” (low abstraction). For this study we use the term for everything that is captured in the UML diagrams of a project. While this level may vary, at least *some* abstraction from the code level is assumed.

Chapter 3

Related research

This chapter will position the topic of this study in the surrounding landscape of software design research in general and related to software quality in particular.

The relation between presence of UML models and the speed and accuracy at which maintenance tasks can be performed were already empirically investigated using controlled experiments in [Hov06]. In these experiments students were divided into into small groups, after which half of those received detailed UML models of a system, whereas the other half did not. The groups were then asked to perform various maintenance tasks on the system, during which their times to complete these tasks were recorded. Intermediate measurements allowed for an analysis of the time it took to complete individual tasks, both with and without having to update the UML models to reflect the new implementation. The researchers concluded that UML had a generally positive impact on the quality of the solutions to maintenance tasks. However, the effort required to complete these tasks remained the same for both groups, when also taking into account the time to update the UML models.

Experiences with precise state modeling using UML state diagrams are reported in [HAA+06]. This paper is particularly interesting because of the similarities that the case in this paper shares with one of the projects studied in this thesis (i.e. the HDMV component, described in Section 4.1.1). Although we think the mentioned paper applies a somewhat less detailed approach, the researchers do report benefits from the precise state modeling, e.g. a more complete system specification and easier transition from specification to code. This last benefit could mainly be attributed to the decision of implementing a state pattern —a decision which presumably could have been made because of the insight provided by the state diagrams.

In [CBC+92] Orthogonal Defect Classification (ODC) is introduced as a technique to bridge the gap between statistical defect models and causal anal-

ysis. This paper already stressed the importance of identifying and validating cause-effect relationships between defects, the development process and the availability of in-process measurements to influence this relationship. ODC suggests leveraging the defect introduction curve and results from the classification of defects to suggest improvements to the development process.

Within the EmpAnADa program several related studies were performed. In [vOLC05] techniques for automatic code-to-UML correspondence were analyzed and implemented in a tool. It argues that models can only hope to influence software quality in general, when they capture relevant information of the system that is to be implemented. Correspondence measures can help quantify this relevance (viz. high correspondence means the models captured a large, thus relevant part of the system).

Metrics have been defined on models in a variety of research topics before. One of these defined metrics across multiple UML diagram types (so called multi-view metrics) to analyze their performance as quality indicators [MCL04]. A paper analyzing the importance of modeling conventions (as opposed to coding conventions) defines several metrics aimed at measuring the level of adherence of various diagram types to these conventions [LDCD06].

In [LC06] the ramifications of defects regarding syntactical consistency between UML diagram types on the understandability of a system's design are explored. Different defect types were defined and through a controlled experiment amongst 111 students and 48 practitioners their influence on design comprehension were tested. The study has two main conclusions: defect detection rate depends highly on the defect's type, as does the level of agreement on how to interpret erroneous design parts.

Chapter 4

Case 1: Blu-ray Middleware

In this chapter the entire Blu-ray Middleware case (BDMW) will be covered. First the purpose of the system and the analyzed components will be described. Next, the definitions used during the analysis of the BDMW case are given. Then the analysis process is treated in-depth, after which we finally give the results and conclusions.

4.1 BDMW description

In this project Philips Applied Technologies has implemented a Blu-ray middleware stack. The stack facilitates playback of the different Blu-ray disc formats, as specified in the Blu-ray standard. From this middleware stack two components were selected for analysis. These components were selected because they exhibited large differences in both design detail and developer working style. At the same time they were both quite comparable in terms of responsibilities. More specific descriptions of component characteristics can be found in Section 4.3.2. Since both components were developed by nearly autonomous teams – the only thing these teams shared was the fact that they were part of the same project – we believe they can, for all ends and purposes, be considered two individual projects. Some brief explanations of the purpose of the two components are given below.

4.1.1 RE/ROM HDMV component

The Blu-ray standard defines two different authoring modes that can be used to author Blu-ray titles. One of these modes is called the HDMV mode. The options this mode provides are most comparable to the options available for DVD titles. The first component, conveniently also called HDMV, plays a crucial part in the playback of Blu-ray titles authored using this mode. The HDMV component is responsible for keeping track of the playback states a player can be in (i.e. started, stopped, suspended, changing a title, stopping

and starting playback, etcetera), as well as handling external input events from the user in all these states (like pressing the start or stop buttons). From now on, if we mention the name HDMV, we will be referring to the component rather than the authoring mode, unless explicitly stated otherwise.

4.1.2 BD-J application lifecycle management component

The application lifecycle management (ALM) component offers comparable functionality to the HDMV component described above, but in this case for Blu-ray titles compatible with the BD-J authoring mode. The main difference is that these titles use java Xlets [1] to control playback, although the BD-J mode offers some other, more advanced features. Despite the differences between the authoring modes, the ALM component receives about the same internal and external stimuli to process as the HDMV component. This is because the ALM component, like the HDMV component, is only charged with tracking playback states and handling user input events.

4.2 BDMW definitions

In this secion we give the definitions that were used during the analysis of the BDMW case. They were mostly developed upfront, but were adjusted and tuned during the analysis process as well.

4.2.1 Defect taxonomy

In literature various taxonomies are suggested to classify defects [CKC91] [CBC+92] [LTW+06] [Bur03] [IEE93]. For quick reference they are listed in Appendix A. From these taxonomies defect types were selected, taking into account their usefulness for this specific study. In other words, they had to be related somehow to information that can be captured in UML diagrams.

As a result we chose to fully discard the defect types related to software requirements documentation. We by no means wish to imply that requirements documentation quality has no effect on the number of defects occurring in software. We do, however, make the assumption that every project benefits from good requirements documentation in the same way.

Instead of hard evidence we only have common sense backing this assumption, however we would like to note that the Blu-ray Middleware project was implementing parts of the same standard. Although this standard was still in development during the course of the project, it was pretty much finished for the system's lifecycle part, which was implemented in the components we analyzed. We therefore believe the requirements documentation was comparable, both in terms of quality and stability.

[1] Xlets are basically Java applets, but without a required user interface representation.

Moreover, changes to a system due to changing system requirements can hardly be considered true defects in the sense that they could have been avoided by implementing the requirements correctly. This is because it were the *requirements* that were wrong in the first place. In terminology used in practice, these changes are actually submitted as *change requests* instead of *problem reports* for this very reason.

For a similar reason of being equally beneficial to all projects we also disregard problems related to configuration management. Defects found in the test harness are also not considered. Although creating test harnesses and test cases is subject to the same types of defects as other software products, testing environments differed greatly amongst the analyzed components.

The remaining defect types were judged on having enough distinctive power. Compromises were made between high enough detail level (in order to make a good distinction in the nature of defects), taxonomy coverage (it should be possible to type enough defects without using the "other" category) and a workable category size for the manual analysis, that would follow in a later stage of the study. The resulting taxonomy is given below:

1. **missing or incorrect conditional branching** – Conditional branching missing or condition not formulated correctly.
2. **missing or incorrect control flow** – Incorrect order of operations, or wrong or non-existent operation executed (but not as a result of *missing or incorrect conditional branching* problems mentioned above).
3. **race condition** – Unforseen output as a result of unforseen sequence or timing of events.
4. **undefined state behavior** – Undefined transitions between states or undefined behavior in a certain state.
5. **inconsistent operation arguments** – Wrong number or types of arguments when calling an operation.
6. **wrong variable used** – Wrong variable used in checking for, or assigning a value.
7. **initialization problem** – Forgotten or wrong initialization.
8. **typing problem** – Incorrect type chosen or returned.
9. **memory cleanup problems** – Memory leak, or double freeing of memory.
10. **algorithmic problems** – Erroneous calculation of values.
11. **other defects** – Defects that cannot be classified using any of the classes mentioned above.

4.2.2 Design detail

For each individual UML diagram type a matrix is given that defines the different *level of detail* attributes we take into account and the possible values they can assume. Filling in these matrices will give an idea of how much of the UML-provided syntax is used. This elaborate way of documenting the design detail level is due to the large amount of possible combinations of syntactical detail the UML allows. Moreover, possible non-standard additions to this syntax (e.g. plain text, tables, etcetera) have to be taken into account as well. An approach using a less detailed ordinal scale (e.g. low, medium or high detail) to describe the detail level for each attribute is insufficient, because we are really interested in the specifics of the diagram detail level, when relating this to defects. Only class, sequence and state diagrams will be considered during this study, since they represent the subset of most widely used diagram types in design documentation [DP05][LCM06].

Below follow per-diagram tables, listing the detail attributes for each type, along with their respective levels. Note that when defining each of these measures, the possible connection to prevention of certain defect types was taken into account. This was done in order to keep analysis doable: recording a much larger set of detail attributes (recall: by hand) would require an infeasible amount of effort.

Of course it makes no sense to try to record attributes that measure parts of the UML provided syntax that are hardly ever used. In our quest to determine which attributes to include, we did *not* use the documentation of the official UML standard as guidance, but primarily used [Fow03] instead (along with our own experiences in using UML). Many practitioners consider this book an authority on explaining how UML is used in practice. Since we would be analyzing industrial cases, it made sense to choose our detail attributes accordingly.

To give a motivation for the selection of attributes made, the last column of each of the tables holds references to the defect types we thought could be prevented by the respective attributes[2]. The numbers correspond to the defect types described in Section 4.2.1.
The class diagram (Table 4.1) is the only structural diagram type considered in this study. The attributes selected to define its level of detail were selected, taking into account their possible relation to structural defect types. We describe them here one by one.

The *attributes* attribute ranks how many variables are included in the diagram. When variables are included, the *attribute types* attribute tells if these variables were additionally given a type in the diagram. The *operations* attribute ranks how many methods and functions are included in the diagram, whereas the *operation arguments* and *operation return types* attributes tell

[2] These defect preventive properties were based on outcome of discussions only. They were in no way influenced by the outcome of earlier experiments.

Table 4.1: Class diagram detail attributes

Class diagram		
attributes	**levels**	**defect types**
attributes	none, key, all	6
attribute types	not used, used	8
operations	none, key, all	2(missing)
operation arguments	not used, used	5
operation return types	not used, used	8
associations	no types, different types	9, 7
labels	not used, used	6
cardinality	not used, used	8

whether these methods and functions had their arguments and return types defined in the diagram as well.

Associations tells whether or not distinction is made between aggregation, composition, navigability and dependency relations. This information can be used to decide which objects instantiate and cleanup other objects. The *labels* attribute states if associations are labeled or not in order to describe variable names or the reason for the relation to exist. *Cardinality* tells if association relations are assigned cardinalities. These can help in determining the type of variables implementing the association.

Table 4.2: Sequence diagram detail attributes

Sequence diagram		
attributes	**levels**	**defect types**
instance types	corresponding, not corresponding	2
control flow	just arrows, descriptive labels, corresponding methods, methods with attributes	2, 5
object creation	not used, used	9, 7
guards	not used, used	1

The sequence and state diagrams capture the behavioral aspect of a software system. Their detail attributes (tables 4.2 and 4.3 respectively) are therefore chosen with a possible link to behavioral defects in mind (although some structural defects may be prevented as well).

The sequence diagram attribute *instance types* shows whether or not object types correspond to classes in the class diagram. Corresponding names could allow a developer to check if a certain class implements all required functionality. The main reason for creating sequence diagrams – as the name suggests – is to make the sequence of actions performed by various collaborating objects explicit. In a sequence diagram this is visualized by arrows going from one lifeline to another. The *control flow* attribute tells how detailed

this flow of control is documented. Explicit instantiation of additional classes required in a scenario is registered in the *object creation* attribute.

The *guards* attribute tells whether or not syntactical options for modeling conditional branching have been used in the sequence diagrams. This may help reduce *missing or incorrect conditional branching* defects. An alternative to using these guards is to model alternative flows of execution in separate sequence diagrams.

Table 4.3: State diagram detail attributes

State diagram		
attributes	**levels**	**defect types**
transitions	just arrows, descriptive label, corresponding methods	4, 2
guards	not used, used	1

For state diagrams the *transitions* attribute tells what information is added to the arrow depicting a transition from one state to another. If transitions can only happen when certain conditions apply, this information can be captured in transition *guards*. Apart from the defects mentioned in the table, state diagrams may also help in eliminating *race conditions*. We expect that the overview of the state behavior of a system, resulting from sufficient state modeling, eases reasoning about the origin of this type of problem.

4.3 BDMW data gathering

In this section we describe the steps we took during the data acquisition phase of the BDMW analysis.

4.3.1 Step one - gather code metrics

In order to make a comparison between the two components possible, standard code metrics for their implementations were recorded. The metrics were collected using the SourceMonitor tool.

4.3.1.1 HDMV code metrics

The metrics in Table 4.4 were taken from the C++ source files, *excluding* the header files. These were not taken into account, because later on comparison with metrics from Java will be made and Java does not separate method definition and implementation. We assume that the header files by themselves have no effect on both the number of defects and their prevention, and can therefore safely be disregarded.

Table 4.4: HDMV code metrics

metric	**value**
SLoC	14000
#statements	7200
#Classes	188
#Methods/Class	3.95
Programming Language	C++
avg mcCabe Cyclomatic Complexity	2.33

4.3.1.2 ALM code metrics

ALM applied two programming languages in its implementation (more on that later) so we present two tables containing code metrics for each of them. The metrics in Table 4.5 are from the Java part. Table 4.6 shows metrics taken from the native part of the component. Since this was programmed in C, which is not object oriented, some metrics are slightly different.

Table 4.5: ALM code metrics - Java

metric	**value**
SLoC	4400
#statements	2700
#Classes	34
#Methods/Class	7.14
Programming Language	Java
avg mcCabe Cyclomatic Complexity	3.88

Table 4.6: ALM code metrics - native

metric	**value**
SLoC	2040
#statements	1200
#functions	61
Programming Language	C
avg mcCabe Cyclomatic Complexity	5.3

4.3.2 Step two - gather project characteristics

Only recording code metrics is not enough when attempting to compare projects and their UML detail level. Some other characteristics should be taken into account as well. Therefore we collected answers to the following questions as well:

project environment What languages were used to implement the system and what tools were used for development and modeling?

developer experience How much experience did the developers have in the technologies used in the project and UML in particular? How did they acquire this knowledge?

adopted process Was the development process organized in an iterative way, or in a more classical waterfall approach?

working style What did the working style of the developers look like and how were the UML models fitted into this?

Below these topics will be dealt with one by one. Where appropriate the topic is divided into an HDMV and ALM part and conclusions are drawn from their comparison.

4.3.2.1 Project environment

HDMV C++ was chosen as programming language for the HDMV component. The programming environments used were both Eclipse and Microsoft Visual C++. Microsoft Visio was used to create the UML models. The main reason for this was the freedom provided by Visio to vary the level of modeling detail as one sees fit. Other tools, especially those that allow for automatic code generation, often prescribe a specific level of detail, which is often considered to be too restrictive. Also, this freedom Visio provides can be used to add non-standard notations to the UML diagrams.

ALM This component is divided into two parts: a Java part and a native part, written in C. The native part implements the message passing functionality to parts of the system that lay outside of the Java scope (viz. closer to the hardware), but complements the classes defined in the Java part. They communicate through the Java Native Interface[3]. The programming environment used for the Java part was Eclipse. Just as with the HDMV component, the models were created using Visio.

[3] JNI – an interface that allows for Java code to call and be called by other applications written in other, platform dependent languages.

Furthermore, since both components were part of the same overall project, they shared the same infrastructure for defect reporting and code versioning. Both tools used for this belong to the Telelogic CM product family: CM/Change for bug tracking and CM/Synergy for version control. These tools were integrated with each other, thereby allowing for fast tracking of the files changed during repair.

It can be concluded that HDMV and ALM developers had similar tools to their disposal while creating their components. We therefore assumed that tooling would not have an important influence on any of the findings.

4.3.2.2 Developer education

HDMV Only one developer carried the responsibility for the HDMV component, although a second developer was available for direct feedback. This developer had the following background:

- Completed a Bachelor program in “Technische Computerkunde” (the part of electrical engineering closest to computer science).
- Acquired his knowledge of UML by himself, studying various books on this topic. No new UML concepts needed to be learned for this project.
- He had ten years of experience as software engineer, working with C++, Java, UML, Rational tools and more.

ALM The main developer – some other developers implemented parts of the component under his supervision – of the ALM component has the following background:

- Has a Master degree in Electrical Engineering.
- Acquired his knowledge of UML only from practice. Knows basic use of different UML diagram types.
- Has a thirteen year history of professional programming. Languages include C (but knowledge needed refreshing) and Java (some unknown concepts were encountered during the project).

What we conclude from this information is that both components were developed by experienced software engineers, who already had considerable knowledge of, and programming skill in the languages they worked with during the course of the project. Furthermore they had sufficient knowledge of UML to create different diagrams at considerable detail level, although the HDMV developer seemed to have some more UML skill, since he used it more frequently and more intensively.

4.3.2.3 Adopted process

Since both components were part of the same overall project (BDMW) they adopted the same overall process defined for this project. This process was incremental in nature, using nine increments from start to completion, spanning in total around two calendar years of development. Both components mostly committed to the increment deadlines, although HDMV skipped one increment delivery date at the start of the project, since the developer did not agree with the time that was given for thinking up and creating the first design of the component, which in his opinion was too short.

4.3.2.4 Working style

HDMV At every start of an implementation iteration the developer created or changed UML models to reflect the functionality that was to be added. These models were created in Microsoft Visio. The level of detail of the models was determined by the relevance of the information that needed to be communicated. Important features were intensively documented, while less important ones received lower modeling detail or were omitted completely. When the models achieved a sufficiently high level of detail (subjective judgement by the developer), they were almost mechanically translated to code. This translation was entirely done by hand. In short one could say that for each major project increment (see Section 4.3.2.3) a small waterfall process was applied.

When maintenance tasks were completed the models were checked for validity against the new code. In case deviations were spotted, the models would be adjusted accordingly. Ergo, round-trip engineering was applied for HDMV.

ALM The attitude of the developer towards more formal way of modeling was not very positive. He considered it to usually be the cause of too much overhead when compared to the benefits it delivers. Models should therefore primarily be used as a means of communication and should only be detailed enough to be able to reach consensus on major design decisions. As far as documentation goes, good comments in the code were considered superior to corresponding UML diagrams.

Within each major project increment the ALM component was extended and changed applying many small iterations. At the end of each iteration the component was first extensively unit tested before being committed to the main build branch of the versioning tree. A complete build of the system was performed multiple times per day to allow for continuous regression testing. An attempt was made at delivering working code as quickly as possible, keeping drag from documenting components to a minimum. Instead, in places where the code was not considered

self-explanatory, time was spent on documenting the code inline, using comments.

The design documents were hardly ever update after the first version, suggesting that no round-trip engineering was applied for ALM.

One can see that apart from the overall project iteration deadlines, the two components were developed using very different working styles. This should be mentioned as a threat to validity towards any of the conclusions we may draw later on.

4.3.3 Step three - collecting defect data

In order to select a number of defects for further analysis, first the complete set of defects related to each of the components had to be identified. Herein both components posed the same problem: the CM/Change tool did not contain detailed and reliable enough information to query for these related defects using only this tool. Therefore the set of defects for both components could not be narrowed down as far as needed.

This problem was overcome by creating a shell script that first narrowed down the set of defects as far as possible using the CM/Change tool. For all defects that passed the search criteria a list of changed files was generated. Each of the files in a list was then compared – by filename – to the files of the component we are creating the set for. If at least one of the changed files matched the defect was added to the set. During the manual analysis of the defects, false positives that were still in the set were removed.

From the generated set a selection of defects was made. Since the HDMV component only had twenty-five defects reported for it, we analyzed all of them. The ALM component had about fifty defects reported on it (revised, see above) and we chose the twenty most recent ones – this would hopefully result in better recollection of the developer when verifying our analysis – along with five randomly chosen older ones to see if the nature of these defects was fundamentally different from the more recent ones. No major differences in defect nature were found, leading us to conclude our sample reflects the total population well enough[4].

4.3.4 Step four - typing defects

Each defect selected for analysis had to be given a type according to the taxonomy mentioned in Section 4.2.1. In order to do this, changes made in the code to solve a defect were looked up and examined by the researcher. This was done by taking diffs from the files that were changed and comparing

[4] Also, we point out that our sample in fact covers half of the total set of defects, which we thought to be a very reasonable portion

the "before" and "after" snapshots. The changed files were found by using the links to old and new versions available through the bug tracking tool.

Instead of determining the defect types manually, one could suggest taking the defect types that are sometimes stated in the used bug tracking tools, but often filling in these types (and filling them in accurately at that) is considered by developers to be a waste of time and therefore these types are almost always left blank. Even *if* they are filled in, one could still question their accuracy, since, apart from the waste-of-time attitude, most developers also lack proper education on what the differences between defect types are.

4.3.5 Step five - determining UML detail

After a defect is properly typed, the analysis continues with judging the detail level of the design "near" it. Guidelines to choose what parts of the design need to be examined are needed here. These guidelines have to do with both the type of the defect that is analyzed and the options UML provides to capture different kinds of information: for clearly behavioral defect types we primarily looked at the (behavioral) sequence and state diagrams, with the goal of finding diagram that modeled the faulty functionality. In collecting this local design detail for each of the analyzed defects we hoped to be able to say something about the relation between level of UML detail and defect density.

Additionally, average detail levels for all diagram types used in a component's design documentation were recorded. This was necessary because we could not assume that the parts of the design of which we already measured the level of detail – due to their relation to defects as described above – would accurately reflect the level of detail of the entire design. Large portions with a completely different detail levels could have been left out.

Having these average detail levels allows us to say something about the defect density of a component, compared to its *entire* design. Obviously this statement would be meaningless if was based on just some parts of the component's design.

4.3.6 Data accuracy

Several actions have been undertaken to strengthen confidence in the accuracy of the data acquired in each of the analysis steps. They are listed here:

- For each of the components the set of defects was checked for accuracy with the developer. Both developers agreed that the resulting sets accurately reflected the total number of defects reported for the components.

- Intermediate results of the manual analysis of defects were discussed with both developers. During these discussions the developers had the

opportunity to make objections to the results and at the same time they answered questions regarding hard-to-analyze defects.

- Results of the entire BDMW case were presented to various project members and quality assurance employees, again giving them the opportunity to make objections.

4.3.7 Storing the data: the database

All information that was gathered during the BDMW case study was stored in a Microsoft Access database. It was considered to be the best match to our requirements: a simple-to-use database with the ability to create some data entry forms in a few button clicks. Using a database enabled us to reshape the way in which the analysis data was represented as we saw fit. For a quick overview of the database schema and a short explanation regarding the use of each of the tables, we refer to Appendix B.1.

4.4 BDMW results

This section will reveal the results from all analyses performed on the data collected in the BDMW study. The results from the respective measurements will be presented one after another, each accompanied by its own explanation and interpretation.

4.4.1 Additional analysis note

We have one additional analysis note to make before we present the results:

While typing the defects (as described in Section 4.3.4) we often had difficulties in either typing a defect as an *undefined state behavior* problem, or instead type it a *missing or incorrect conditional branching* or *control flow* problem. This was caused by the fact that undefined state behavior could usually be attributed to one of the following causes:

- Problems with handling different events arriving in a certain state, which usually are handled in different (conditional) branches.

- Problems with the wrong action (control flow) being selected after arrival of an event in a certain state.

This problem was even more prominent in the ALM component than in the HDMV component, since the ALM component did not implement a state pattern as HDMV did. This made identifying a problem as being state related much harder for ALM.

Due to the above observations we decided to divide the found *undefined state behavior* defects over the *missing or incorrect conditional branching* and *control flow* types. We kept our *undefined state behavior* defects in the analysis, however, but they will no longer add to total defect density, since we would then be counting certain defects twice. This is also the reason why the *undefined state behavior* defect type is put apart from the rest in the upcoming figures.

4.4.2 UML and defect prevention

In order to explore the defect preventive capabilities of UML we first measured the defect density for each of the projects and compared these projects based on their average UML detail level and coverage. Here we would expect the defect density to decrease, if indeed models have a defect preventive property, as either the detail level or coverage of the models increases. Getting into a bit more detail we then compared the defect density for each individual defect type found in the projects, to see if some defect types benefit more from the models than others, in terms of defect prevention.

4.4.2.1 Comparing component defect density

Table 4.7 shows the average levels of detail for the models created for each of the inspected components, as well as the components' defect density. We conclude from this table the following:

- The class diagram level of detail only differs in the use of different types of associations for the HDMV component and the use of labels to describe these associations.
- Sequence diagram level of detail seems the same. However, we note that the sequence diagrams found in HDMV contained non-standard references to states in which certain method calls were performed (i.e. states were added on the lifelines in the sequence diagrams). Regardless of the fact that it is a non-standard use of UML syntax, we believe it to add detail to the HDMV sequence diagrams that was worth mentioning.
- HDMV clearly has a higher level of detail when it comes to state diagrams.

Of course the level of modeling detail alone is not enough to make an accurate assessment of the amount of information that is captured by the models. This is because, although highly detailed, diagrams still hold little to no information if they only cover a very small part of the entire system. We therefore provide an indication on the coverage of each diagram type in tables 4.8 and 4.9 for HDMV and ALM respectively. From these indications we can see that,

Table 4.7: Project average model detail, completeness and defect density

DETAIL LEVEL	ALM	HDMV
class diagram – attributes	none	none
class diagram – attribute types	n/a	n/a
class diagram – operations	not used	not used
class diagram – operation arguments	n/a	n/a
class diagram – operation return types	n/a	n/a
class diagram – associations	no types	different types
class diagram – assoc labels	no	yes
class diagram – cardinality	used	used
sequence diagram – instance names	corresponding	corresponding
sequence diagram – control flow	corresponding	corresponding
sequence diagram – object creation	unknown	used
sequence diagram – guards	unknown	not used[a]
state diagram – transitions	just arrows	corresponding methods
state diagram – guards	not used	used
DEFECT DENSITY		
defects per KSLoC	7.4	1.7

[a]Although some sequence diagrams described alternative flows of the same case.

although the sequence diagram level of detail in ALM seemed comparable to HDMV's, this conclusion is actually nullified by the fact that there is only one implementation class covered (in only one diagram at that).

Table 4.8: HDMV coverage of implementation by diagram types.

diagram type	coverage indication
Class diagrams	One diagram covering 8 out of 12 classes not related to state implementations. Other class diagrams (3) covering all state-implementing classes.
Sequence diagrams	10 diagrams covering behavior of 7 distinct implementation classes. These were by far the most relevant classes of the component.
State diagrams	17 diagrams covering all states (82) implemented by the component and all state changes possible between states.

Finally looking at the defect density numbers in the first table we conclude that the component with both higher level of detail and higher coverage levels in its UML models shows drastically lower defect density.

Table 4.9: ALM coverage of implementation by diagram types.

diagram type	coverage indication
Class diagrams	One diagram covering 8 out of 16 classes.
Sequence diagrams	1 diagram covering behavior of 1 implementation class.
State diagrams	3 diagrams covering 18 states. Number of implemented states was far greater (guess: about 50) but exact amount is hard to know, since no state pattern was implemented.

4.4.2.2 Comparing average defect density per defect type

Figure 4.1 shows the defect density distributions for each of the analyzed cases. They are based on the distribution of the defects studied in the manual analysis phase. For ALM the resulting defect densities of the individual types were simply multiplied by two, to reflect the total number of defects found in the component, which was twice as many as the number of defects analyzed. Although the validity of this step may be questioned, no better guess could be made. The only countermeasure used to ensure validity was a quick analysis of the output of the defect selection script, to see if a different pattern could be observed in the part that was not analyzed. This was not the case.

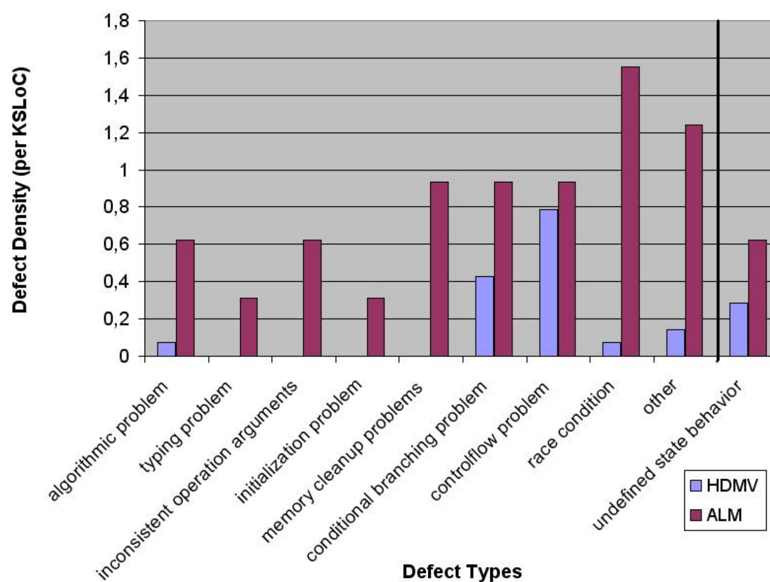

Figure 4.1: Defect density (defects/KSLoC) distribution.

When we again look at the tables in the previous section, we can see that the

component adopting a higher degree of modeling detail and coverage shows lower defect density numbers across the board. It is interesting to see however, that certain defect types seem to benefit more from increased modeling effort than others in terms of lower defect density:

- The *missing or incorrect control flow* type seems to receive almost no benefits from the availability of sequence and state diagrams in the HDMV design documentation at all. Recall that we stated in Section 4.2.2 to expect sequence and state diagrams to prevent certain defect types, including *missing or incorrect control flow*. Taking a better look at our database, we discovered that actually one third of the HDMV *missing or incorrect control flow* defects was not related to models. Another third of the defects was related to models that were defective themselves. This high defect rate for related models (regarding this defect type) may be a reason for the poor defect preventive capability we have measured.
- The *missing or incorrect conditional branching*, *race condition* and *undefined state behavior* defect types *do* seem to benefit from the large amount of detailed state diagrams created for HDMV.
- Another prominent difference, the one for *memory cleanup problems*, was found to be caused by one part of the code that needed to be corrected numerous times. The developer admitted that this was a particularly hard problem to solve and that this also had to do with some lacking knowledge of the programming language.
- We could come up with no explanation for the existence of *typing problem*, *inconsistent operation arguments* and *initialization problem* defect types in ALM but not in HDMV. These defect types were considered by us to be related to the level of detail of the class diagram, but this level of detail is quite comparable for both cases and at any rate the relevant syntax for preventing these defects was applied in neither of the two.
- The big difference for the *algorithmic* defect type could not be explained. We simply cannot see how UML can help prevent such computational errors.

4.4.3 UML and maintenance

In this section we used the available repair effort numbers from the BDMW project to further analyze the possible relation between these and the use of UML. Here we were interested in four aspects, each of which will be addressed in the subsequent sections:

- Compare average defect repair effort from the two components.

- Compare repair effort distribution.
- Compare defect repair effort against UML usage.
- Compare modeling effort to repair effort.

4.4.3.1 Per project average defect repair effort

First, we show the average defect repair effort calculated for the HDMV and ALM components (Table 4.10). Revisit Table 4.7 to check the detail level and completeness of the UML models for each of these components. Analogous to the defect density, we see that the average defect repair effort is lower for the component with higher modeling detail and coverage.

At this point we wish to note that this result was also found by the study described in [Hov06].

Table 4.10: Average defect repair effort per project

project	avg effort (hours)
HDMV	2,8
ALM	7,7

4.4.3.2 Average repair effort per defect type

Next, it would be interesting to see if any correlations can be found when looking at individual defect types. To this end we again split up the numbers according to their respective defect types. Figure 4.2 shows the resulting average effort distribution. We notice that HDMV scores lower than ALM for all defect types. However, the difference between them varies greatly from type to type.

Most interesting observation is the fact that the *missing or incorrect control flow* defect type seems to be hardly influenced by the availability of UML models. One reason for this may be that it already took only two hours to fix this type of defect and it may simply not be possible to perform this task any faster.

4.4.3.3 Repair effort versus UML detail level

Given the results from the above sections regarding repair effort for the two components, it would be nice to know if this difference can actually be attributed to differences in level of UML detail. To find out if this is the case, the entire set of defects (so we are using both HDMV and ALM defects for this analysis) is split up. As discriminators we use the different diagram types

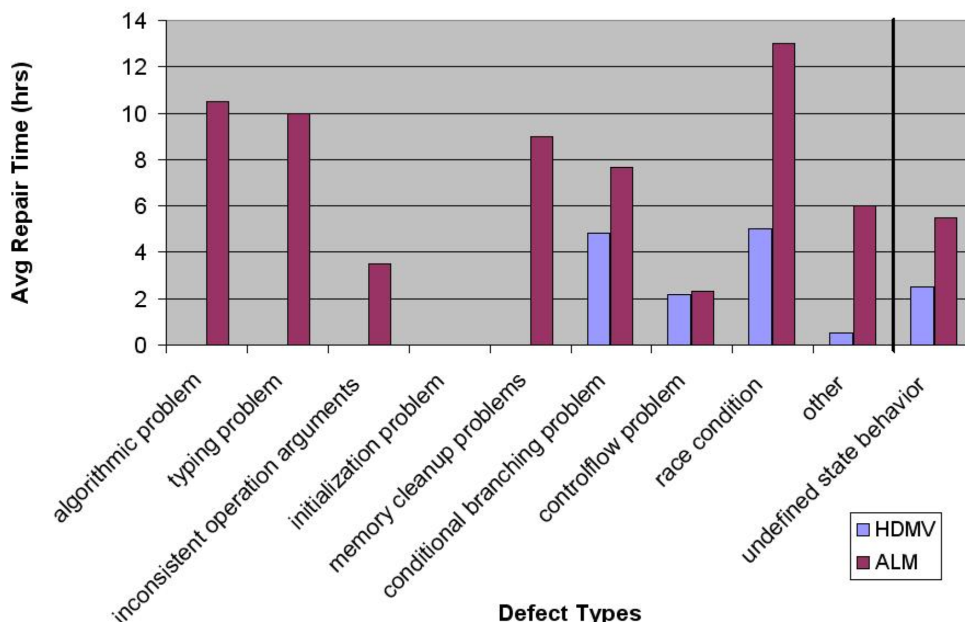

Figure 4.2: Repair effort per defect type.

to which a defect was related (because the functionality that was faulty was modeled in those diagrams). If the difference in repair effort between defects is in fact related to UML usage, we would expect a lower defect repair time for defects related to UML, compared to those not related to UML. Because we have further shown that the defect repair time is also related to a defect's *type*, we use this type as an additional discriminator.

Table 4.11 shows the results for some of the defect types. The other defect types were omitted because they were only represented in one category of UML usage. If we look at the table, we can see that the results are inconclusive. The *m/i conditional branching* and *race condition* defect types seem to show quite some difference in favor of defects related to UML models, while at the same time *undefined state behavior* defects show hardly any difference and the *m/i control flow* type even favors defects unrelated to UML models.

4.4.3.4 Modeling versus maintenance

In this section we compare the total effort spent on modeling with the total effort spent on maintenance tasks. Here we are looking for the answer to possibly the most relevant question of the entire study: does the upfront modeling approach pay off (enough)? To this end we compared the modeling and maintenance effort totals.
We acquired the effort numbers for modeling in two different ways. The total modeling effort for HDMV was extracted from the bug tracking tool. Each

Table 4.11: Repair effort per defect type versus created UML diagrams

type	state	sequence	avg effort
m/i conditional branching	yes	no	2,3
m/i conditional branching	no	yes	3
m/i conditional branching	no	no	7
m/i control flow	yes	yes	3
m/i control flow	yes	no	2,4
m/i control flow	no	no	1,9
race condition	yes	no	2
race condition	no	no	13,6
undefined state behavior	yes	no	3
undefined state behavior	no	no	4

time the design had to be changed, the HDMV developer added this task to the tool and recorded the hours spent on these tasks here as well. The same approach did not hold for ALM, since no hours spent on modeling were recorded in the bug tracking tool at all. Instead, we asked the ALM developer to make an informed guess about these numbers, confronting him with the document change record from the newest version of the design document. Both developers agreed with the used numbers below.

Figure 4.3 suggests the existence of an inverse relation between modeling and maintenance effort. Here we plotted the amount of modeling and maintenance effort as percentages of the total amount of time spent on these two tasks.

When we go one step further and actually normalize the hours spent towards the size of the projects (by dividing effort number by component size in KSLoC), we get the chart shown in Figure 4.4. For ALM and HDMV we see that, although the time spent on modeling HDMV (per KSLoC) is almost twice as much, this is easily compensated for by the lower repair effort.

We note that the results we find here are much stronger than the results found in [Hov06], which tries to find similar relations using controlled experiments. One reason for this may be the bigger size of the components we have analyzed, compared to the smaller sized component that was used for the controlled experiment. This could have increased the usefulness of models in understanding and keeping an overview of the component to be changed, resulting in bigger differences in repair time between the component with and without these models.
Although it would have given a more complete picture when we had also added numbers covering the initial implementation effort for each of the components, there was no way for us to get our hands on them. Both developers did not dare to give an estimate either. They did, however, agree that having ones design documented in models should speed up the implementation process,

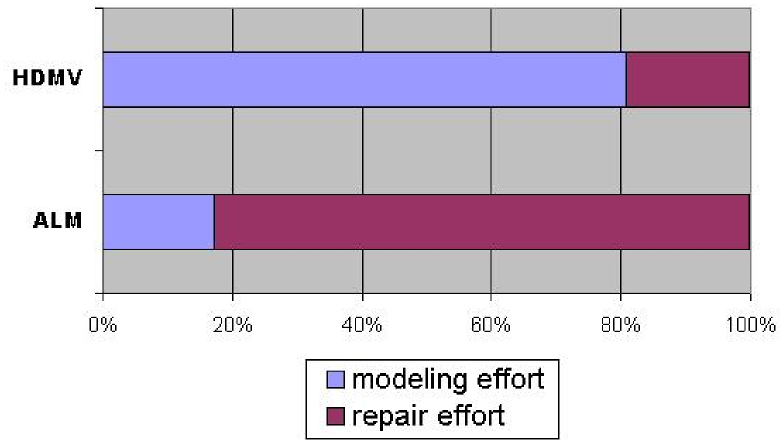

Figure 4.3: Per project modeling and repair effort ratios.

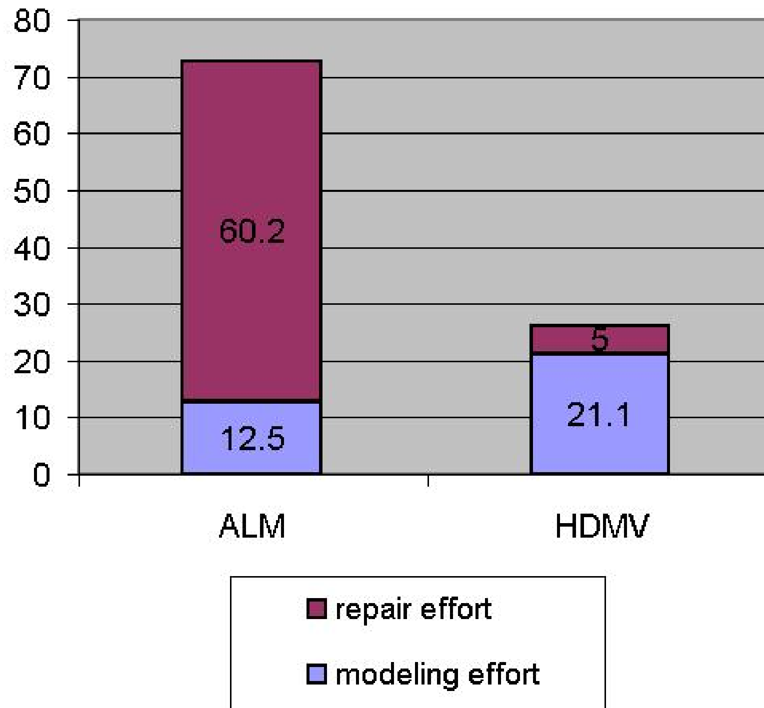

Figure 4.4: Per project modeling and repair effort (hrs per KSLoC).

although the ALM developer obviously did not think this would outweigh the amount of time put into creating the models in the first place.

4.5 Conclusions first case

The results from the different analyses performed within the BDMW case provides us some confidence towards the relevance of our research questions. Although the size of the data set was too small to perform any statistical tests, the defect density and repair time numbers seem to favor HDMV over ALM too much to simply be a coincidence. With the biggest difference between these components being the amount of UML modeling performed upfront, we have strong suspicion that this is (at least for an important part) the cause of the differences in these numbers.

4.6 Case-specific threats to validity

Some case-specific threats to validity are mentioned here:

- The programming languages used for implementing the components were different. This can pose problems, since different programming languages may be more or less susceptible to making programming mistakes than others, thereby giving another explanation for differences found in defect density. Although we could not find any statistics on defect density for programming languages in scientific literature, a field expert in testing made the informed guess that projects implemented in C++ normally displayed a slightly higher defect density than projects implemented in Java.

- A related problem to the one mentioned above has to do with possible differences in verboseness (amount of SLoC needed to describe certain behavior) between programming languages. This questions the validity of conclusions based on comparing defect density between HDMV and ALM, since the SLoC metric is used in calculating this density. The only mention we could find in literature regarding verboseness of Java and C++ was [Pre00], which concludes that the mean SLoC size of a Java, C and C++ programs is comparable.

- The state pattern implemented in HDMV may explain a large part of the difference in both defect density and repair times compared to that of ALM. This could be due to the more understandable and flexible structure this pattern brings. Of course one of the hardest things in applying design patterns is discovering *when* to apply them. Maybe having more elaborate models at ones disposal helps in this task, but this is just speculative and we will not go into further detail here.

- Putting the focus on the fast delivery of working code (which was the applied working style for ALM) may result in choosing the simplest solution to a problem at the time. Such “greedy” choices may in the end result in code becoming unclear, unstructured and inflexible and in this sense a different choice of working style may be responsible for the differences we have reported.

- The components were developed (mainly) by only one developer each. Although we believe that both developers qualify as being equally experienced (see 4.3.2.2), this assumption still poses a threat to the validity of our conclusions.

Chapter 5

Changing the process

Having gained experience from the first case study, we found some problems in the approach used. Before embarking on a second case study we thought it wise to revise our approach in order to see if we could make improvements on it, thereby increasing the effectiveness of the second case study in answering our research questions. This chapter lists the problems encountered and the proposed solutions to them. The second case study will then go on and apply these solutions, by providing new definitions and analysis steps.

5.1 Problems with the BDMW approach

We start by describing the main problems we perceived during analysis of the BDMW case.

1. The first and most prominent problem was found to be the time it took to perform the analysis for a sufficiently large number of defects. This was primarily due to the amount of manual labor that had to be performed in parts of the analysis.

2. The qualitative nature of the measurement of level of detail attributes proved rather inconvenient. When trying to compare the influence of different attributes on defect prevention and maintenance tasks, it may be more useful to use a ratio scale of measurement. An option for which a ratio scale would be required is in combining individual attributes into one level of detail measure. Although it is at this point unknown if different attributes need different scalars while performing these kinds of calculations, at least an attempt at finding such scalars can be made.

   Moreover we also use this qualitative scale in capturing the average level of detail. However, something like an average cannot be calculated from an ordinal scale like the one we applied. The average we mention is based on subjective judgement. This makes it hard to compare findings from different cases analyzed using this approach.

3. Proven measures for design coverage of UML diagram types were not found in scientific literature. Had we had these measures, we would have been able to use them to calculate the sizes of modeled and unmodeled system parts for both ALM and HDMV. Since we record for each defect if it is related to one of the models, we could then have used this defect count together with the size information to find out if there were differences in defect density between modeled and unmodeled system parts. This would have further strengthened our case that *models* are the cause of the lower defect density of HDMV compared to ALM.

   The rather textual indications we gave for diagram coverage in tables 4.8 and 4.9 are not suited for this and defining accurate ratios measuring coverage from scratch requires a complete study of its own. Other options to deal with this problem should be investigated instead.

5.2 The new approach

Changes to the BDMW approach will be discussed in this section. Some of these changes were introduced to solve specific problems we pointed out in the previous section. They will be referred to whenever appropriate (using their numbers). The major changes will be treated in the subsections below.

5.2.1 Metrics

Instead of using ranking qualifications for the attributes describing the level of detail of each of the diagram types, we decided to change these into design metrics. This will help us in the following ways:

- Use of metrics will help prevent the objectivity problem mentioned in (2), in the sense that, when the calculations for each of these metrics are defined, their values will always be produced in the same way.
- Also, if level of detail attributes are assigned numerical values, this will allow for further experimentation in combining them, which, as described in (2), was hindered to certain extent by the qualitative ranks used before.
- Last but not least calculation of metrics can be automated, thereby lowering the amount of time required to produce different level of detail attributes by hand, solving part of (1).

5.2.2 Eliminating the coverage measure

As mentioned in (3) we could not find proven metrics for diagram coverage. With this coverage we would have liked to compare the number of defects

found in modeled system parts to the number found in unmodeled system parts, to see if the defect density differs significantly amongst the two.

In Figure 5.1a we capture this notion of coverage graphically. The parts of the implementation classes within the inner circle represent modeled system parts (modeled in a sequence diagram for instance). The coverage value would be defined as the ratio between the modeled and total system size.

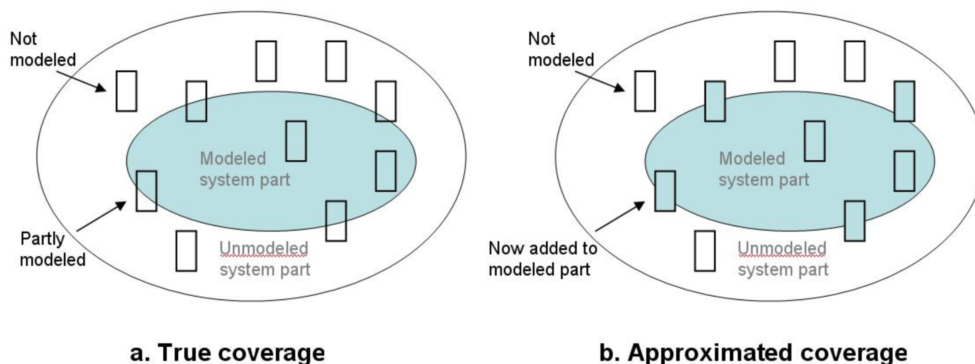

Figure 5.1: Coverage approximation - graphical explanation.

The new approach will circumvent the use of a coverage measure by including all classes, that are covered at least partially by diagrams, in full in the modeled system part (Figure 5.1b). The size of this part is then easily computed as the total number of SLoC of all shaded classes in the figure. Comparing defect density between modeled and unmodeled system parts will now be very straightforward: we can simply compare the average defect density of modeled and unmodeled classes.

Of course the unmodeled class parts that are added to the modeled part in this way, will influence the defect density of the modeled part. However we argue that this unmodeled part can not threaten any results that suggest a defect preventive quality of models. This is because we believe it is safe to assume that availability of models does not *increase* defect density. Unmodeled parts of the classes added to the modeled system part will then have a defect density value equal to the modeled parts at best, but definitely not lower.

The defect density that is computed for each of these added classes will therefore give an upper bound of the defect density for the part of the class that was modeled. From this it follows that, if we find a significantly lower defect density for modeled system classes, choosing the modeled system part as described above, this result will also hold for the real modeled system part.

5.3 Drawbacks to the new approach

The main drawback we identified with the new approach is that we loose specific information regarding the local level of detail of a diagram at the place

of the defects. Because diagrams sometimes have some degree of disproportion in their level of detail – meaning that some parts of the diagram are more detailed than others – there will always be some uncertainty when using a diagram's level of detail measure for each of the defects described in it. In other words: some accuracy is lost. However, a quick scan of the diagrams created for the second case showed little disproportion. Detail level only varied from diagram to diagram, but not within each individual one.

Chapter 6

Case 2: PARTS

This chapter describes the second case study, that was performed within the PARTS project. This chapter has the same basic structure as Chapter 4, the BDMW case. Again, we will start by describing the purpose of the analyzed system, immediately followed by the definitions that were used during this second case. Then an in-depth description of the analysis process is given. The results and conclusions will be treated last.

We note that part of the contents of this chapter have been described in a paper ([NFC08]) that was submitted as a research paper to the MODELS2008 conference. At the time of writing this paper is in the process of review.

6.1 PARTS description

PARTS was built to be an integrated healthcare system for psychiatrists in the Netherlands. It is an information system with which psychiatrists can manage patient information, treatments history, appointment planning, medication prescriptions and more.

It was chosen as subject for a second case because it matched the requirements we had put up for a candidate very nicely (something which we knew by this time to be hard enough to find indeed), namely: UML was used to model the system and the used version control system and bug tracking system were integrated with each other, allowing for fast tracing of source files that were modified to solve defects.

6.2 PARTS definitions

For this second case study we changed a lot of the definitions used during the analysis of the BDMW case. This was done because of, and according to, the reasons and suggestions treated in Chapter 5.

6.2.1 Defect taxonomy

Some adjustments were made to the defect taxonomy described in 4.2.1. Mainly this had to do with the different nature of the two systems: whereas BDMW was very technical in nature, a piece of middleware, PARTS had a far more direct link to the end user, having to deal with, amongst others, screen design, user interface navigation and direct user data input and output. A large part of the adjustments originated from discussions with Ariadi Nugroho[1], who had already created his own taxonomy for use on a number of case studies, including the PARTS case. Both the previous experiences from Nugroho on analyzing information systems and our own experience in analyzing the more technical and embedded BDMW system were taken into account. We ended up with the taxonomy given below, which has become a merger of Nugroho's and our own taxonomy.

1. **(static) user interface** – Any defect that only has something to do with the way the user interface looks (window sizing, form sizing, font choices, positioning of UI elements, etc.).
2. **user interface - navigation** – Defects regarding screen transitions (wrong destination, missing intermediate screen, etc.).
3. **logic** – Defects caused by missing or wrong implementation of business or processing rules.
4. **process flow** – Defect caused by missing or wrong process flows (e.g., incorrect order of operation execution)
5. **race condition** – Unforseen output as a result of unforseen sequence or timing of events.
6. **data handling** – Defects caused by missing or poor data handling.
   a) **data validation** – Input not, or incorrectly validated.
   b) **data access** – Defects related to retrieving/storing data from/to a data store (like a database).
   c) **session issues** – Defects related to session specific data.
   d) **wrong variable used** – Wrong variable used in checking for, or assigning a value.
   e) **initialization** – Uninitialized or wrongly initialized variables (or other data sources).

[1] Ariadi Nugroho MSc is a PhD student at the Leiden Institute of Advanced Computer Science. His research is in the field of software quality estimation using UML. He is co-author of the paper that was submitted to the MoDELS2008 conference covering some of the findings of the PARTS case study.

f) **memory cleanup** – Missing or incorrect cleanup of data sources.

g) **variable typing** – Incorrect type chosen or assumed for a variable.

h) **inconsistent operation arguments** – Wrong number or types of arguments when calling an operation.

7. **user data i/o** – Defects related to missing or wrong data input and output from/to the user interface.

8. **computational** – Erroneous calculation of values.

9. **undetermined** – Defects that cannot be classified in any of the classes mentioned above.

Compared to the previously used defect taxonomy from Section 4.2.1 the following concrete changes have been made: First, the *undefined state behavior* type has been removed. From the BDMW analysis we experienced it as a defect type that often brought up discussion. This is because this type is very hard to determine, unless a state pattern is implemented. If this is not the case it is very hard to make a distinction between this type and, for instance, *missing or incorrect control flow* or *missing or incorrect conditional branching*[2].

Second, a general defect type *data handling* was added from the defect taxonomy used by Nugroho. We found that a number of defect types used in the BDMW case could actually be seen as subtypes of this general defect type. Because we expected a large number of defects to be assigned the *data handling* type (based on previous experience), we decided to include these subtypes into the taxonomy. Furthermore, the user interface related defect types were added. These were not accounted for in the BDMW taxonomy, but since we expected their numbers to be quite high we thought it unwise to leave them in the *other defects* category.

6.2.2 Design detail

As described earlier, we measure UML level of detail (from now on abbreviated as LoD) in this second case by using metrics. We started by defining metrics for class and sequence diagrams only. For the moment this was sufficient, because both the PARTS case as well as other cases we had selected for research in the near future, use only these two diagram types to document their design. The used collections of class and sequence diagram metrics are described in the two subsections that follow.

[2]For the moment using their names from the old taxonomy, since that is where the *undefined state behavior* type was still defined.

6.2.2.1 Class diagram based level of detail metrics

For each class x in class diagram:

- *AttrSigRatio*(x)**:**
  The ratio of attributes with signature to the total number of attributes of a class x in the diagram.

- *OpsWithParamRatio*(x)**:**
  The ratio of operations with parameters to the total number of operations of class x in the diagram.

- *OpsWithReturnRatio*(x)**:**
  The ratio of operations with return (give return values) to the total number of operations of class x in the diagram.

- *AssocLabelRatio*(x)**:**
  The ratio of associations with label (e.g., association name) to the total number of associations attached to class x in the diagram.

- *AssocRoleRatio*(x)**:**
  The ratio of associations with role name (in the opposite end) to the total number of associations attached to a class in a model.

6.2.2.2 Sequence diagram based level of detail metrics

For each sequence diagram y:

- *NonAnonymObjRatio*(y)**:**
  The ratio of objects with name to the total number of objects in sequence diagram y.

- *NonDummyObjRatio*(y)**:**
  The ratio of non-dummy objects (objects that do not correspond to any class) to the total number of objects in sequence diagram y.

- *MsgWithLabelRatio*(y)**:**
  The ratio of messages with label (any text attached to the messages) to the total number of messages in sequence diagram y.

- *NonDummyMsgRatio*(y)**:**
  The ratio of non-dummy messages (messages that correspond to class methods) to the total number of messages in sequence diagram y.

- *ReturnMsgWithLabelRatio*(y)**:**
  The ratio of return messages with label (any text attached to the return messages) to the total number of return messages in sequence diagram y.

- *MsgWithGuardRatio*(y):
  The ratio of guarded messages (messages with conditional checks) to the total number of messages in sequence diagram y.

- *MsgWithParamRatio*(y):
  The ratio of messages with parameters to the total number of messages in a sequence diagram y.

Note that, contrary to the LoD based on class diagrams described in the previous subsection, the sequence diagram based LoD has an entire diagram as its unit of measure. The ramifications of this decision will be discussed later in this chapter (viz. Section 6.3.5).

As can be seen, both class and sequence diagram based metrics are expressed in ratios. We believed this to be favorable as opposed to using absolute numbers, because absolute numbers are also dependent on the size of a class, or of a sequence diagram. And having a larger diagram by no means implies that its level of detail is higher as well.

6.3 PARTS data gathering

Specifics on the gathering of data for the analyses that will follow are described in this chapter.

6.3.1 Step one - gather code metrics

Some code metrics for PARTS are given in Table 6.1

Table 6.1: PARTS code metrics

metric	value
SLoC	150k
#statements	55k
#Classes	1000
#Methods/Class	12
Programming Language	Java
avg mcCabe Cyclomatic Complexity	2.5

6.3.2 Step two - gather project characteristics

The same characteristics were recorded as the ones recorded for BDMW.

6.3.2.1 Project environment

The project was built as a web service using Java technology. The Apache Struts framework was applied to encourage developers to adopt a model-view-controller architecture. IBM Rational XDE was used to create the UML models. Other IBM Rational tools were adopted for code versioning and bug tracking as well. These were IBM Rational ClearCase and ClearQuest respectively. As stated earlier, these tools were integrated with each other, allowing fast tracking of the files changed during maintenance tasks.

No information regarding the used IDE was available to us, although it would seem logical to assume that this IDE came from the Rational product family as well, like Rational Application Developer.

6.3.2.2 Developer experience

Not much information could be obtained on this topic. We expect, however, that the system's architects had sufficient knowledge of UML and experience in creating design documentation with it. We believe we can assume this, because the way in which the development was organized (see working style below) was standardized in the developing company. It would make sense that having such a standardized development process suggests also having capable people at each step of way. This would then also lead us to assume that programmers at least had the required knowledge of UML to accurately *read* the designs.

As far as the programming experience goes we have even less information. From the analysis we had to do on the source files we did not get the feeling that we were dealing with extremely talented (or at least educated) people, although of course to understand the combination of all technologies used in this project, one would need considerable training all the same.

6.3.2.3 Adopted process

The project was developed in four major increments, each lasting several months. We concluded this from the defect descriptions in the bug tracking tool, which mentioned target releases and document change dates.

6.3.2.4 Working style

The requirements and upfront design of the system were created in the Netherlands. It was then off-shored to India where actual implementation was done. When big problems popped up regarding incorrect design, a part of the system would be sent back to the architects in the Netherlands to be corrected. When these parts were updated, they again followed the original implementation path in India.

Testing was performed in the Netherlands. Judging from the defect descriptions some of the defect repair work was performed in the Netherlands as well, but a large part was sent back to India.

6.3.3 Step three - collecting defect data

The defect data was based on the latest version of the PARTS project data. This version was chosen because most of development activities had taken place in this version. It is important to note that only defects found during testing were taken into account. To give an idea of the number of defects reported in each of the stages we give some numbers based of the latest version of PARTS recorded in ClearQuest:

- Test: 1546
- Review: 771
- Acceptance test: 212
- Integration test: 70

The preprocessing of the defect data further consisted of the following steps:

First, the source files that were corrected/modifed to solve each defect were extracted from the ClearCase repository. Here we have only taken into account the *.java* files (thereby excluding mostly configuration files (usually *.xml*) and *.jsp* files). The idea behind this was that these files could never be related to UML classes (and therefore would not be useful to our study) and defects changing only files of these types would probably fall into defect types that would not relate to UML anyway (.jsp files are strongly related to the UI, especially if no .java file is altered as well). A Perl script was employed to perform this activity automatically.

566 out of 1546 defects reported during testing had modified source files attached to them. We found the following reason why defects can exist that are not related to modified source files. For starters, a lot of defects were solved by making changes to the database or application server only. Also, it is possible that a defect was solved indirectly, i.e. by solving another one. Finally, defects were often rejected for a variety of reasons, for instance because they could not be reproduced, or because they were found to be duplicates.

Since traceability of defects to modified source files is a prerequisite for our analyses, we had to exclude defects without it. The size of the target population for further analysis therefore shrank to 566 defects. This number was still considered too big (since still a large part of the analysis had to be done by hand) and we decided to take a random sample from this remaining collection. The sample size was initially targeted at 100, but later increased to 125 to compensate for some non-defects found. The sampling was performed

by assigning a random number to each defect and then perform a sorting based on these random numbers. The first 125 defects from the sorted list were added to the sample.

When we were confronted with a somewhat low hit ratio while relating the defects from the sample to the models, we decided to once again enlarge the sample size by another 39 defects. These defects were selected semi-randomly: it was still from a list sorted by the random numbers, but the defects considered were only ones that we thought had a reasonable chance of relating to the models. We were forced to take this measure, because having a low number of defects that related to models would have jeopardized the possibility for statistical analysis.

6.3.4 Step four - typing defects

In order to assign defects types according to our taxonomy we again checked the changes made in the source files to fix the defects, as was also the case with BDMW (see 4.3.4). Of course we typed defects according to our new taxonomy. For later reference we present the defect type distribution of PARTS, using the first 125 random defects as source data, in Figure 6.1.

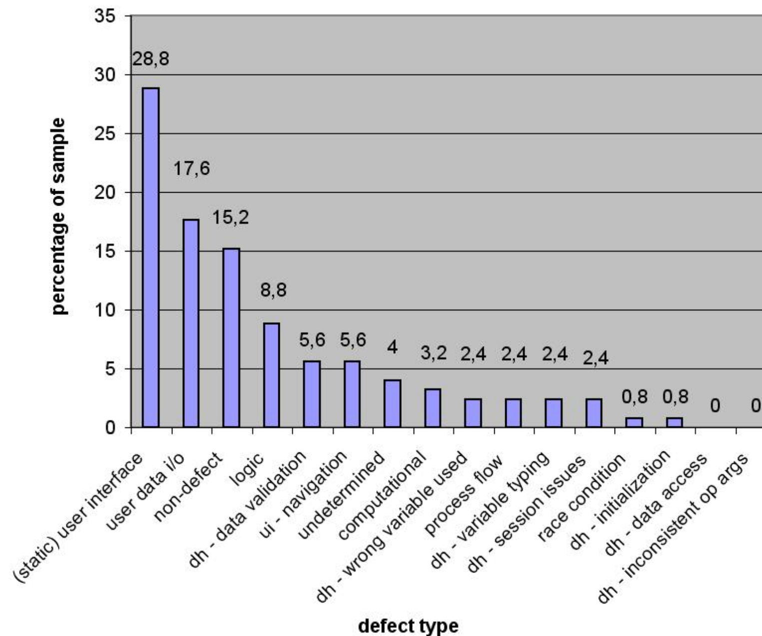

Figure 6.1: PARTS: defect type distribution

6.3.5 Step five - determining UML detail

Determining UML detail "near" a defect was done quite differently from our approach in the BDMW case. This had everything to do with the way in which we represented LoD attributes for this second case, viz. by using metrics calculated on class and sequence diagrams. When analyzing a defect, it would have to be related both to the class and sequence diagrams. The relation to the class diagram could automatically be filled in. For this we used the names of the source files that were changed and based on those names matched them to classes in the design. This was sufficient, because by using metrics to calculate the LoD measure, no subjective eye is needed to judge a diagram's LoD indepth, whereas in case of a qualitative measure it would.

Filling in the relation to sequence diagrams still required some human interaction. Using the information from the defect description and judging from the changed source files, the functionality that was altered was looked up manually within the entire set of sequence diagrams. If there were sequence diagrams that actually mentioned (parts of) the functionality, they were linked to the defect. As with the class diagram based LoD, we let the metrics do the rest.

Remember that linking defects to places in the UML design is just a part of the work. The other part is made up of calculating the metrics themselves. For class diagrams this task could be automated using the SDMetrics tool [tUdqmt], which could calculate the metrics using the *.xmi* files exported from Rational XDE. The original idea included automatic calculation of sequence diagram metrics from the *.xmi* files as well (an operation that is supported by the SDMetrics tool as well). Unfortunately, the exported *.xmi* files from XDE did not include sequence diagram information. In order to still be able to perform the analyses we decided to collect the metrics manually by inspecting all sequence diagrams in XDE itself. In order to be able to link implementation classes to sequence diagrams (we need to be able to do this when calculating LoD values, see below) we additionally recorded all implementation classes used in each sequence diagram, along with the number of messages connected to it.

Since in the end we want to have LoD measures for each implementation class, it remains to be explained how we calculate these from the individual LoD attributes we have acquired so far. We split this up into class and sequence diagram based LoD aggregates and will start by explaining the first.

A class diagram shows an obvious candidate to take metrics from when calculating a class's LoD aggregate: the designed class that corresponds to the implemented class. Although it was conceivable, this correspondence was never found to be a one-to-many relationship (i.e. an implementation class only corresponded to exactly one design class at most). If we *had* encountered a one to many relationship, we would have been forced to apply an approach similar to the LoD calculation based on sequence diagrams (see below). Hav-

ing pinpointed which metrics to use, we can do the actual calculation. As a first step we will simply add up all metrics and define the value of the LoD measure ($LoD_{cd-based}$) as the total. The equation for this LoD calculation is given in below.

Let c be an implementation class and c' be its class diagram counterpart.

The complete calculation of c's LoD aggregate will then look like this:

$$LoD_{cd-based}(c) = \sum ClassLoDMetrics(c') \tag{6.1}$$

It becomes slightly more complicated when computing a class's LoD aggregate using sequence diagram metrics. We observed that one implementation class may be instantiated in several sequence diagrams (one-to-many relation). This meant that we had to in some way take into account the level of detail of instances in *all* of these sequence diagrams. For each individual sequence diagram the assumption was made that within it there was not much disproportion (within-diagram differences in level of detail). That would justify using the level of detail metrics from entire sequence diagram as an approximation of the level of detail 'surrounding' the class's instance within it[3]. To calculate the LoD measure for a sequence diagram we again added up all metrics to get its LoD value. Returning to our problem concerning the one-to-many relation, we decided to remedy this by taking weighted averages of their calculated LoD values. The weights were assigned according to the number methods of the class that were modeled in the diagram. This is more accurately explained below.

Let $LoD(s) :: sequence\ diagram \rightarrow float$ be a function, taking for s a sequence diagram, and returning the LoD aggregate for s, calculated as follows:

$$LoD(s) = \sum SequenceDiagramLoDMetrics(s) \tag{6.2}$$

Let $MethIn(x, y) :: class \rightarrow sequence\ diagram \rightarrow int$ be a function, taking for x an implementation class and for y a sequence diagram in which x is instantiated, and returning the number the number of methods of x that is modeled in y.

Let c be an implementation class that is instantiated in both sequence diagrams $SeqA$ and $SeqB$.

The weight of $SeqA$ and $SeqB$'s LoD values then equals $MethIn(c, SeqA)$ and $MethIn(c, SeqB)$ respectively.

[3]We were forced to use the detail level of an entire sequence diagram, because our data set did not contain metrics at the class instance level.

The complete calculation of c's LoD aggregate ($LoD_{sd-based}$) will then look like this:

$$LoD_{sd-based}(c) = \frac{MethIn(c, SeqA) \times LoD(SeqA) + MethIn(c, SeqB) \times LoD(SeqB)}{MethIn(c, SeqA) + MethIn(c, SeqB)} \quad (6.3)$$

6.3.6 Storing the information: the database

A new database was set up to store al information required by the new approach. Because we required multiple people to be able to enter and use data at the same time and from different locations we switched to the MySQL DBMS and made data entry available through the internet, using some PHP web pages. We refer to Appendix B.2 for an overview of the database schema and an explanation on what information is stored in each of the tables.

6.4 Results

Unlike the BDMW case described earlier, the bug tracking tool of the PARTS project did not include any effort data regarding the time that was spent on fixing each individual defect. Therefore analyses of the influence of UML modeling on maintenance tasks, comparable to the ones performed on the BDMW case (Section 4.4.3), could not be repeated. Hence we will only cover the possible defect preventive properties of UML for this case. Again, the results from the respective measurements will be presented one after another, each accompanied by its own explanation and interpretation.

6.4.1 Additional analysis notes

We have two additional analysis notes to make before we present the results.

6.4.1.1 Filtering the defect sample further

For some parts of the analyses we refer to what we call (defects of) a *filtered sample*. With this filtering we mean that we have left out all defects of the following types:

- *(static) user interface*
- *user interface - navigation*
- *undetermined*, and
- defects we could not type

We left the first two out because we could not see how UML would ever be able to help avoid such types of defects[4]. The third and fourth were left out for obvious reasons. What we were left with was a total number of 83 defects that were considered to have "useful" types.

6.4.1.2 Faulty classes and defect density calculation

We calculated the number of defects in each class using the information in the bug tracking and versioning tools. For each defect a list of changed files is given. Looking at all these lists we could calculate the total number of times a file was changed to repair individual defects. We say "individual" here because a file may be changed multiple times while fixing the same defect (for instance when a fix was implemented in two stages). We left out these duplicates and only counted changes when they happened under distinct bug reports.

From this defect count we then calculated the defect density by dividing it with the class's source lines of code (SLoC) metric. Since all mentioned data required for defect density calculation was collected automatically, we could have calculated the defect count based on the entire defect data set (so using all 566 defects related to modified source files). However, the defect typing analysis step already showed us that about half of the defects in our target population of 566 defects was not useful for further analysis (as described in the previous section). Of course we assume here that the defect type distribution of our sample is representative for the entire population, but we have found no reasons to make us think otherwise. We decided to base our defect density calculation on the *filtered sample* defects only, thereby favoring the more accurate defect set over the higher number of data points.

6.4.2 Comparing defect density

This analysis compares the defect density for parts of the system that received different modeling attention. We split up the system into parts that were modeled using both class and sequence diagrams, using only class or sequence diagrams and using none of them. This allowed us to perform tests both for modeled and unmodeled system parts, as well as make distinction in the types of diagrams used. We acquired the data for this analysis from the database by executing the query in Listing C.1 in Appendix C. Since the entire result set is not that large (only 41 faulty classes came through all filters set in the query) we added it in Table C.1 in the same appendix.

[4] A non-standard use of UML activity diagrams as site navigation maps was later discovered within PARTS. It would be interesting to see if these maps can help avoid the *user interface - navigation* defect type; a task for which they seem to be particularly well suited. Mention of this use of activity diagrams was actually found in an internet publication (see [Lie04]), although it is probably no coincidence that the publisher is IBM (owner of the Rational tools used in PARTS).

Starting with the division of the system in a modeled and an unmodeled part, we first performed a test of normality[5] on the samples to see if we could apply Student's t test. Judging from Table 6.2 the hypotheses that the samples are normally distributed have to be rejected. The boxplots in Figure 6.2 show some outliers in the data. These were not caused by incorrect data entry and are therefore taken into account during further analysis.

We continue by performing a non-parametric Mann-Whitney test with the null hypothesis that the population distributions are the same[6]. Judging from the output of the test (Table 6.3) we reject the null hypothesis (at the 0,05 significance level) and conclude that the population medians are actually different: defect density is lower for modeled system parts.

Table 6.2: Modeled and unmodeled defect density: test for normality

	Shapiro-Wilk		
	statistic	**degrees of freedom**	**significance**
modeled	0,653	27	0,000
unmodeled	0,658	14	0,000

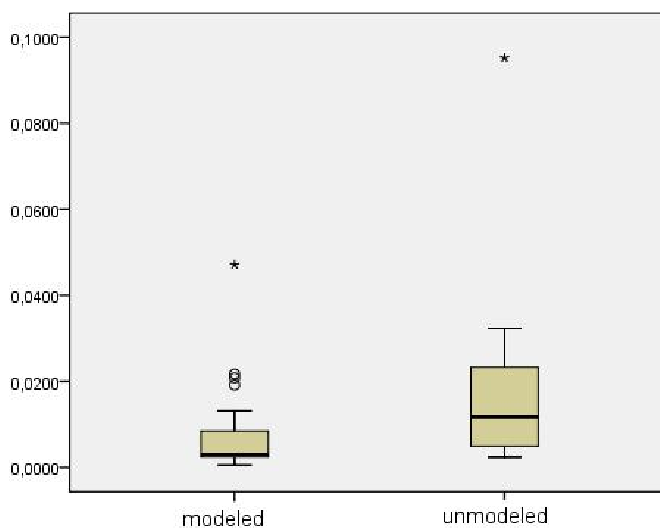

Figure 6.2: PARTS: Defect density (per SLoC) of modeled and unmodeled system parts

Next we split up the modeled faulty classes according to the diagram types they were modeled in. This is done disjunctively, so a class is put either in the

[5] Section D.4 explains the use of this test in further detail.
[6] Section D.2 explains the choice for this test in further detail.

Table 6.3: Modeled and unmodeled defect density: Mann-Whitney test

	Ranks		
	N	**mean rank**	**sum of ranks**
modeled	27	17,52	473,00
unmodeled	14	27,71	388,00

	Test statistics
Mann-Whitney U	95,00
Asymp. Sig. (2-tailed)	0,010

class, *sequence* or *both* collection, depending on whether it is only modeled in a class diagram or sequence diagram, or in both. Knowing that the unmodeled system part already did not pass the normality test, we immediately performed a Kruskal-Wallis test to check if any differences in sample means were worth analyzing[7]. The results of this test are listed in Table 6.4. Looking at the ranks in this table and the boxplots from Figure 6.3 one should not be too surprised that, although we did all analyses, we could only find significant difference between defect density means of the sequence diagram sample and the unmodeled sample. We again performed a Mann-Whitney test for this (results in Table 6.5).

Table 6.4: Splitting up modeled defect density: Kruskal-Wallis test

	Ranks	
	N	**mean rank**
modeled (class and sequence)	8	25,31
modeled (class only)	4	32,75
modeled (sequence only)	15	9,31
unmodeled	14	27,71

	Test statistics
Chi-Square	23,611
degrees of freedom	3
Asymp. Sig. (2-tailed)	0,000

From the boxplots we can see that partitioning according to used diagram types deals with a lot of the outliers from the previous analysis. Apparently it causes outliers to cluster together into groups (thereby not being outliers anymore). We can also see this in the boxplots: the new samples cover different parts of the original sample. It surprised us that classes modeled in both class

[7]Section D.3 explains the choice for this test in further detail.

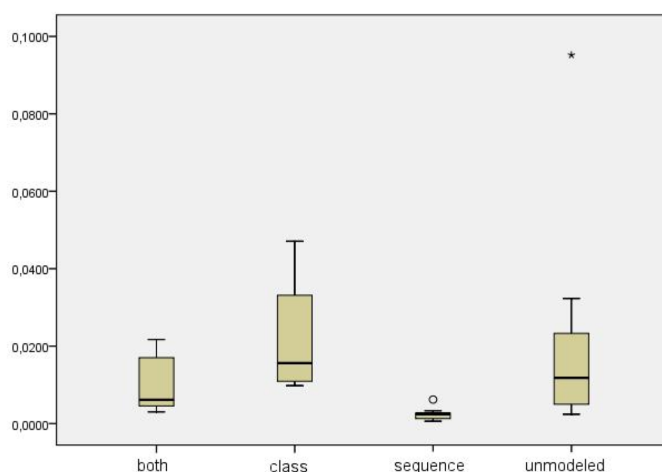

Figure 6.3: PARTS: Defect density (per SLoC) of (split up) modeled and unmodeled system parts

Table 6.5: Sequence diagram modeled and unmodeled defect density: Mann-Whitney test

	Ranks		
	N	**mean rank**	**sum of ranks**
modeled (sequence diagram only)	15	8,97	134,50
unmodeled	14	21,46	300,50

	Test statistics
Mann-Whitney U	14,500
Asymp. Sig. (2-tailed)	0,000

and sequence diagrams showed a higher defect density than classes modeled in sequence diagrams only.

More detailed inspection of the result set (C.1) showed a pattern in the classnames within samples. For instance, the part only modeled in sequence diagrams contained solely *EntityBeans* and *BusinessBeans*, while the part modeled in both sequence and class diagrams seems to favor *Delegates*. We conclude that this gives a second explanation for the difference in defect density between classes in different system parts: inherent magnitude of defect density caused by a class's role in the system. Actually this role is directly connected to the diagram types used to model a class, because the standardized approach, as was available for the PARTS developers, prescribed certain diagram types to be created for different types of classes.

Have these tests then all been in vain? We believe not. First, doing these

analyses has given us a better understanding of what our data is telling us: there *is* a difference in defect density between parts of the system, and we have two possible explanations for it. Second, there are several other projects lined up for future analysis that are set up as web services. If similar tests are performed on them we can compare results, based on both diagrams used and roles played, to find out which explanation will prevail.

6.4.3 Comparing average defect density per defect type

In the previous section we have analyzed the defect density of modeled and unmodeled system parts, making no distinction between the *types* of defects that contribute to it. That distinction will be made in this section. With this analysis we want to explore the effectiveness of UML in preventing different kinds of defects.

The data for this analysis is retrieved from the database, using the query in Appendix C.2 Listing C.2. For reference, the result set for the modeled system part is given in Table C.2. From this data we computed the defect density (number of defects per KSLoC) for each of the defect types.

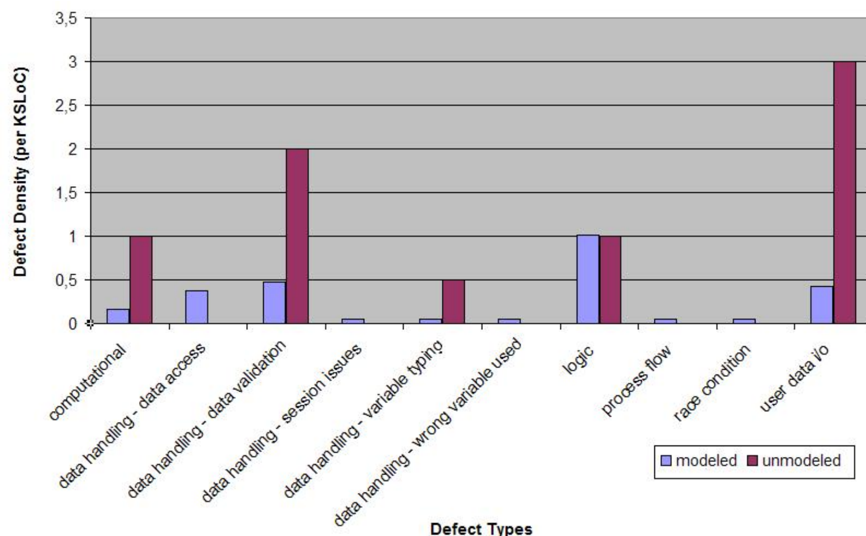

Figure 6.4: PARTS: Average defect density distributions for modeled and unmodeled system parts.

Figure 6.4 visualizes the outcomes of the calculations for modeled and unmodeled system parts. There are some interesting observations to be made from the picture. We list them here:

- The *user data i/o* and *data validation* defect types seem to benefit most from the presence of UML models. We do not mention the *variable*

typing type here, because of its low defect counts in the result sets.

- The *logic* defect type seems receive no benefits at all from the presence of UML models. To find the cause of this we took a harder look at our data and found that with two thirds of the modeled logical defects, the models themselves were *also* defective. Of course this could very well explain why these defects popped up in the code as well.

Unfortunately no further statistical tests could be applied on the data on which the graph is based, since a lot of defect types have a too low defect count to hope for any significant results.

6.4.4 Correlating level of modeling detail to defect density

In the previous section we have only partitioned the system into modeled and unmodeled parts. Next we will analyze if the level of modeling *detail* has any correlation with defect density. We now only take into account those classes of the systems that *are* modeled, assign each an LoD and do correlation tests.

We treat correlation based on sequence diagram LoD only. Judging from the defect type distribution of the random sample (Figure 6.1) the majority of the defects is behavioral in nature. Common sense suggests that a structural diagram, like the class diagram, cannot capture the information required to avoid these kinds of defects.

Furthermore, due to this skewness of the defect type distribution towards behavioral defects, we have a very small amount of faulty classes pointed out by our filtered defect sample that are modeled in a class diagram *only* (eight to be exact). We consider this too small a number to try any correlation tests. Moreover, if we, in an attempt to overcome this problem, tried to add faulty classes to this sample that are modeled in both class and sequence diagrams, we expect that any correlation that would be found then would in fact be caused by sequence diagrams instead.

Note that we do not state that the approach is not suitable to execute this analysis. The only thing we would need is a far greater sample size (so that enough faulty classes only related to class diagrams present themselves).

6.4.4.1 Additional confounding factor

We want to explicitly state here that the countermeasure suggested in Section 5.2.2, to avoid using a coverage metric, does not help us during the next analyses. This is because we will now descend to the level of individual classes when defining LoD and comparing system parts. It is clear that assuming a class to be fully modeled, while in fact only a part is modeled, will not help us at this level.

Since we have no alternate countermeasure or proven coverage metric for models available, we will assume that the coverage of all partially modeled

classes is comparable. This, however, is a direct threat to the validity of our results, because we believe that *if* model coverage varies between classes, this can also be a cause of variability in defect density.

6.4.4.2 First analysis: based on all sequence diagrams

The query we performed to get the data for the analysis is listed and described in Appendix C.3, Listing C.3. The result set is also given there (Table C.3).

From the result set we could select different LoD attributes to include into a faulty class's (sequence diagram based) LoD aggregate. We started out by just selecting all available LoD parts and adding them together. Having a defect density and LoD aggregate, we continued with a test for normality. From the results in Table 6.6 we can conclude that we cannot assume the defect density distribution to be normal. Boxplot 6.5 shows that there are outliers in the defect density data, however, after inspection it was concluded that these were not the result of incorrect data entry and therefore needed to be taken into account.

Table 6.6: LoD and defect density correlation: test for normality

	Shapiro-Wilk		
	statistic	**deg. of freedom**	**sign.**
defect density	0,574	30	0,000
LoD (all LoD parts)	0,945	30	0,126
LoD (interesting LoD parts)	0,972	30	0,591

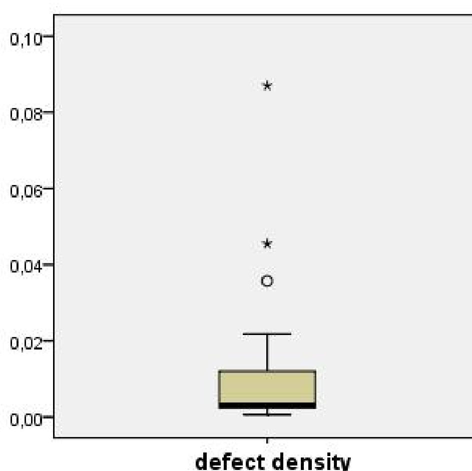

Figure 6.5: PARTS: Sample defect density (per SLoC) (all SDs)

For the above reasons we calculated Kendall's *tau* statistic in order to analyze the correlation[8]. The results of this test can be found in Table 6.7 and are visualized in the scatterplot in Figure 6.6. From these we can see that there exists a significant and negative correlation between the LoD aggregate and defect density values. Or, in other words, higher level of detail in sequence diagrams correlates to lower defect density in the faulty classes.

Table 6.7: LoD and defect density correlation: Kendall's τ

		defect density
LoD (all LoD parts)	**Correlation Coefficient**	-0,333[1]
	significance (2-tailed)	0,010
	sample size	30
LoD (interesting LoD parts)	**Correlation Coefficient**	-0,333[1]
	significance (2-tailed)	0,010
	sample size	30

[a] Correlation is significant at the 0,05 level (2-tailed)

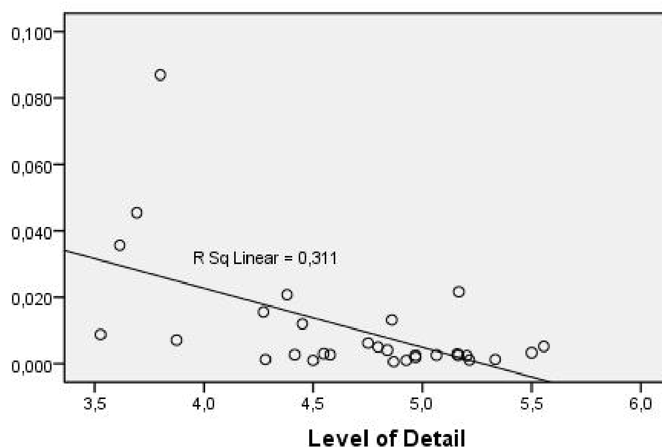

Figure 6.6: PARTS: scatterplot of correlation between LoD and defect density (per SLoC) (all LoD parts, all SDs)

Of course results will depend on the way in which we combine LoD parts in calculating the LoD aggregate. Taking a closer look at the result set (Table C.3), one can see that the variability profile differs greatly from one LoD part to another, viz. the *NonAnonymObjRatio*, *NonDummyObjRatio* and *MsgWithLabelRatio* parts are almost constant. We decided to do a second analysis, this time targeted at what we will call the "interesting LoD parts".

[8] Section D.1 explains the choice for this statistic in further detail.

To this end we left out the three parts we just mentioned and we also decided to drop the *NonDummyMsgRatio*. This last part was dropped since then we were left with what in our sense are the LoD parts that can hold the most detailed information on a system's behavioral design (viz. *MsgWithParamRatio*, *ReturnMsgWithLabelRatio* and *MsgWithGuardRatio*). We added them together to get a new LoD aggregate and performed the same tests we did with the LoD aggregate based on all LoD parts.

From Table 6.7 we can see that this new LoD aggregate scores exactly the same *tau*-value as the first one. The left-out LoD parts were not of much influence on the outcome of the correlation test, which, for all but the *NonDumM* part, is not very surprising, seeing that they are almost constant. No scatterplot is given, since it hardly differs from the previous one.

6.4.4.3 Second analysis: filter sequence diagrams

Still having the findings from Section 6.4.2 in the back of our heads – stating that differences in defect density may be explained by faulty classes originating from different system parts – we decided to again take a closer look at the result set of our query.

We noticed a few classes had very low values for their *SLoC* metric. Because only faulty classes are taken into account for this analysis (otherwise computing defect density would not be very useful) these classes contained at least one defect. However, most classes only had a very low defect count, which made the SLoC value the primary source of defect density variability. This resulted in these classes having a very high value for their *defect density* metric.

At the same time these classes had a below-average value for their LoD aggregate. Further investigation explained the origin of this commonality: they were all only modeled in sequence diagrams created to describe *use-case realizations* (from now on called UCRs). Sequence diagrams of this "sort" were found to be invariably lower in detail than their counterparts that described *interface operation realizations* (IORs). This is primarily due to the layer of the system for which they describe behavior: UCRs describe behavior at the user interface or presentation level, whereas IORs describe behavior one layer deeper in the system (where no user interaction is present anymore). Modeling user interaction is typically handled a bit more relaxed in terms of level of detail.

We believe the above observations ask that we split up our data set to exclude the faulty classes related only to UCRs, so that we can perform the test again and see if the results differ greatly from our previous ones. In total eight out of 30 faulty classes were only described by UCRs. We decided to redo the analyses of the previous section on the remaining 22 faulty classes only, since the eight UCR-related faulty classes were considered to be a too small group for statistical analysis.

The query was only slightly altered (see Listing C.4 and accompanying explanation) to get a new result set (Table C.4) to base our analyses on. Without further ado we list the performed statistical tests and there outcomes.

Starting with a normality test, Table 6.8 shows that once again the defect density distribution is not normal. The accompanying boxplot (Figure 6.7) shows some outliers, but these cannot be discarded.

Table 6.8: LoD and defect density correlation: test for normality

	Shapiro-Wilk		
	statistic	**deg. of freedom**	**sign.**
defect density	0,649	22	0,000
LoD (all LoD parts)	0,937	22	0,171
LoD (interesting LoD parts)	0,956	22	0,419

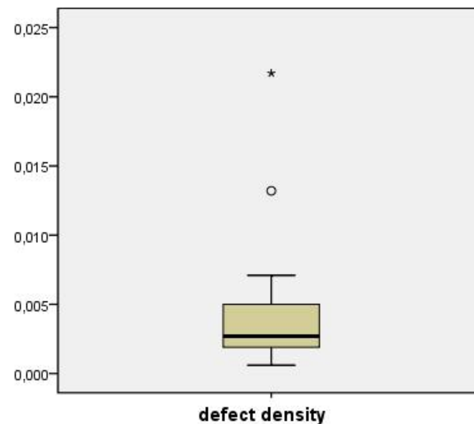

Figure 6.7: PARTS: Sample defect density (per SLoC) (only IOR SDs)

So we again move on to computing Kendall's τ in order to test for correlations. This test was performed on the same two LoD aggregates that were used in the previous section and we cover their results simultaneously. Table 6.9 gives the values for the τ statistic and the data sets are visualized in the scatterplots in figures 6.8 and 6.9.

Looking at the results we can see that all previously found, statistically significant correlations have vanished. It seems once again that other factors besides sequence diagram LoD play an important part in determining defect density values.

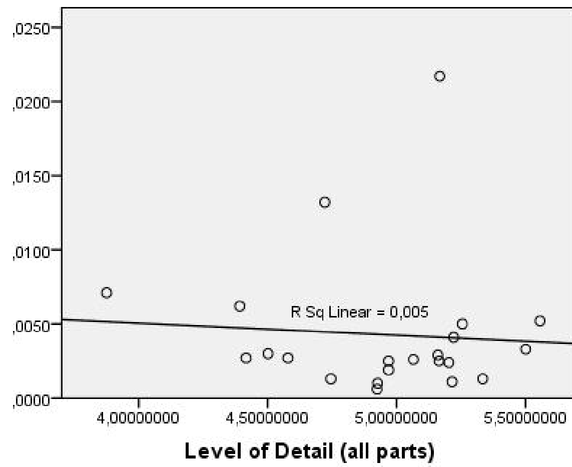

Figure 6.8: PARTS: scatterplot of correlation between LoD (all parts) and defect density (per SLoC)(only IOR SDs)

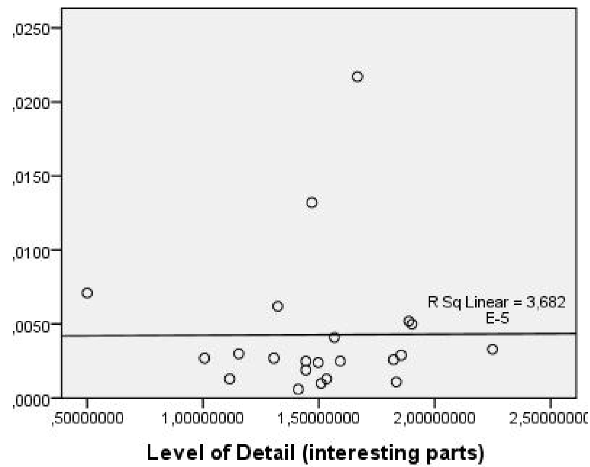

Figure 6.9: PARTS: scatterplots of correlation between LoD (interesting parts) and defect density (per SLoC)(only IOR SDs)

Table 6.9: LoD (IOR based) and defect density correlation: Kendall's τ

		defect density
LoD (all LoD parts)	**Correlation Coefficient**	-0,022
	significance (2-tailed)	0,888
	sample size	22
LoD (interesting LoD parts)	**Correlation Coefficient**	0,083
	significance (2-tailed)	0,592
	sample size	22

6.4.4.4 Digging even deeper: individual defect types

Since we have put a lot of time in typing a large sample of defects, we might as well try to leverage some of this information a bit more. Some defect types were found in quite large amounts. A quick scan showed that some types had enough faulty classes that were modeled in sequence diagrams related to them, so that correlation tests could be attempted using these specific types.

We selected two defect types for in-depth examination: *logic* and *user data i/o*. The same analyses were performed as described in the previous two sections, so we just give the relevant results here as compact as possible. Table 6.10 gives the outcomes of the normality tests performed. Here we see that in every data set but the last one the defect density sample does not seem to be normally distributed. Without giving the boxplots we simply state that no outliers (which were present in some of the samples) were discarded.

Table 6.10: LoD and defect density correlation (individual defect types): test for normality

	Shapiro-Wilk		
	statistic	**deg. of freedom**	**sign.**
defect density (logic, all SDs)	0,834	13	0,018
LoD (logic, all SDs, all LoD parts)	0,960	13	0,761
LoD (logic, all SDs, interesting LoD parts)	0,987	13	0,998
LoD (logic, all SDs, guarded message part)	0,925	13	0,290
LoD (logic, all SDs, parametered message part)	0,933	13	0,376
defect density (logic, IOR)	0,835	11	0,028
LoD (logic, IOR, all LoD parts)	0,918	11	0,301
LoD (logic, IOR, interesting LoD parts)	0,968	11	0,871
LoD (logic, IOR, guarded message part)	0,945	11	0,575
LoD (logic, IOR, parametered message part)	0,903	9	0,199
defect density (ud i/o, all SDs)	0,804	9	0,022
LoD (ud i/o, all SDs, all LoD parts)	0,978	9	0,955
LoD (ud i/o, all SDs, interesting LoD parts)	0,945	9	0,636
defect density (ud i/o, IOR)	0,873	6	0,238
LoD (ud i/o, IOR, parametered message part)	0,962	6	0,838
LoD (ud i/o, IOR, interesting LoD parts)	0,950	6	0,744

Although we could have done a Pearson test on the last data set, we chose to apply the same test to all sets. We will again calculate Kendall's τ coefficient. We present the results in table form only, leaving out the scatterplots. Tables 6.11 through 6.14 cover the outcomes of all tests.

Starting with the *logic* defect type, we see comparable results to the previous sections: taking into account all sequence diagrams we find some statistically significant correlations (and the ones that are not significant are close to the 0,05 significance level), but excluding the UCR related ones removes the correlations as well. We would especially like to point out the correlations between the *guarded message* LoD part and *logic* defect density, and *parametered message* LoD part and *logic* defect density. We wanted to give these parts special attention, since we thought these to be especially well suited to prevent defects of the *logic* type.

The *user data i/o* defect type did not show any significant correlation with the two LoD aggregates from the previous sections. We did not find any reason to correlate it to individual LoD parts, so we did not perform additional tests. We believe that without the existence of some sort of reasoning to back up a correlation test, this testing would just be “fishing for results”.

Table 6.11: LoD (logic, all SDs) and defect density correlation: Kendall's τ

		defect density
LoD (all LoD parts)	**Correlation Coefficient**	-0,359
	significance (2-tailed)	0,088
	sample size	13
LoD (interesting LoD parts)	**Correlation Coefficient**	-0,410
	significance (2-tailed)	0,051
	sample size	13
LoD (guarded message part)	**Correlation Coefficient**	-0,487[a]
	significance (2-tailed)	0,020
	sample size	13
LoD (parametered message part)	**Correlation Coefficient**	-0,615[b]
	significance (2-tailed)	0,003
	sample size	13

[a]Significant at the 0,05 level (2-tailed).
[b]Significant at the 0,01 level (2-tailed).

6.5 Conclusions second case

In applying statistical analyses on our data, we uncovered some statistically significant correlations between the level of detail at which a class is modeled in sequence diagrams and the defect density of that particular class. The

Table 6.12: LoD (logic, IOR) and defect density correlation: Kendall's τ

		defect density
LoD (all LoD parts)	**Correlation Coefficient**	0,127
	significance (2-tailed)	0,586
	sample size	11
LoD (interesting LoD parts)	**Correlation Coefficient**	0,091
	significance (2-tailed)	0,697
	sample size	11
LoD (guarded message part)	**Correlation Coefficient**	-0,200
	significance (2-tailed)	0,392
	sample size	11
LoD (parametered message part)	**Correlation Coefficient**	-0,200
	significance (2-tailed)	0,392
	sample size	11

Table 6.13: LoD (user data i/o, all SDs) and defect density correlation: Kendall's τ

		defect density
LoD (all LoD parts)	**Correlation Coefficient**	-0,222
	significance (2-tailed)	0,404
	sample size	9
LoD (interesting LoD parts)	**Correlation Coefficient**	-0,222
	significance (2-tailed)	0,404
	sample size	9

Table 6.14: LoD (user data i/o, IOR) and defect density correlation: Kendall's τ

		defect density
LoD (all LoD parts)	**Correlation Coefficient**	-0,200
	significance (2-tailed)	0,573
	sample size	6
LoD (interesting LoD parts)	**Correlation Coefficient**	-0,333
	significance (2-tailed)	0,348
	sample size	6

negative sign of this correlation suggests that the higher the level of detail is, the lower the defect density will be.

Most of the results, however, do not hold when certain parts of the system are kept out of scope. We refer to the start of Section 6.4.4.3 for more information. The question is whether this exclusion improves the quality of the sample or not.

6.6 Case-specific threats to validity

Some case-specific threats to validity are mentioned here:

- The threat mentioned and explained in Section 6.4.4.1.
- The choices for our LoD metrics were based on analysis of the UML provided syntax and discussions about which syntactic options had reasonable chance of preventing defects. However, these choices remain subjective and at the same time influence directly our LoD value calculation (and thereby our results).
- The assumption that not much disproportion exists within sequence diagrams (as stated in Section 6.3.5) may not always be true. When this assumption does not hold, the accuracy of a class's LoD aggregates that are calculated from sequence diagram metrics diminishes. This jeopardizes the validity of results found in the correlation test performed in Sections 6.4.4.2, 6.4.4.3 and 6.4.4.4. A suggestion to overcome this threat is given in Section 7.3.

Chapter 7

Conclusions and evaluation

In this final chapter we summarize our findings by answering the research questions defined in the introduction of this thesis. After doing this we cover some general threats to the validity of the conclusions, give some recommendations on future application of our used approach and suggest some directions for future work. We finally present some guidelines for the use of UML in software development.

7.1 Answers to the research questions

In this section we will answer the research questions using the results from the case studies. Each research question is answered in its own subsection. For answering the first two questions we could use the results of both case studies, whereas for the remaining three we could unfortunately only use the BDMW case. Reason for this was that the effort numbers were unavailable to us for the PARTS case (or any of the other cases available to us at the time).

7.1.1 RQ1: How does the level of detail in UML models influence a project's defect density?

Results from BDMW suggest that a higher level of UML detail lowers a project's defect density. When looking at PARTS we see hints of this relation in the outcome of Section 6.4.2, comparing modeled and unmodeled system parts. Section 6.4.4.2 furthermore shows statistically significant, negative correlation between UML LoD and defect density within the modeled part by itself.

However, concerns for the validity of the outcomes of the PARTS case have been expressed in Sections 6.4.2 and 6.4.4.3, suggesting some alternative explanations for our observations. Adding a third case, which is similar to PARTS in nature but differs in UML detail level, should provide the answers

needed to assess if these concerns are indeed valid (see also the future work section below).

7.1.2 RQ2: How does the level of detail in UML models influence the defect density of individual types of defects found in a project?

The first observation we make (judging from figures 7.1 and 7.2) is that the nature of a project has a large effect on the defect density distribution across defect types. This is exemplified best by looking at the *race condition* (both cases) and *control flow* (BDMW) and *process flow* (PARTS) defect types. This forces us to mostly base our conclusions on the results from individual cases.

Looking at the individual figures, we see that for both cases frequently occurring defect types all display lower defect density scores when UML models are available. However, as we have extensively described in the respective tests in Sections 4.4.2.2 and 6.4.3, we have to conclude that not all defect types benefit equally from applying UML modeling techniques.

The *conditional branching* (BDMW) or *logic* (PARTS) defect type deserves some special attention here, since it was the only defect type that was found frequently in both cases and at the same time showed differences in the effectiveness of UML on its prevention. When looking closer at our data sets, the PARTS case showed that out of the 15 *logic* defects related to sequence diagrams, 10 of these diagrams were faulty to begin with.

In the HDMV part of the BDMW case, guards were modeled in state diagrams instead of sequence diagrams. These state diagrams were also accompanied by large tables summarizing what events were allowed in what states (i.e. rules that should be translated into conditions in the implementation). Out of the five *conditional branching problems* in HDMV that were related to either state diagrams or the mentioned tables, the design was flawed three times.

These numbers suggest that the inability of UML to prevent *logic* defects, as perceived in PARTS, stems from the high defect rate (concerning this defect type) of the diagrams themselves. Moreover, this is not the first defect type for which we have noticed this. In Section 4.4.2.2 we already drew the same conclusion for the *missing or incorrect control flow* defect type.

7.1.3 RQ3: How does the level of detail in UML models influence a project's average defect repair time?

Looking at the results of Sections 4.4.3.1 and 4.4.3.2 we conclude that the component applying UML in a more rigorous fashion scores lower average defect repair times, both for the entire component as a whole and across all defect types within it. As was already noted, this result corresponds to the result found in [Hov06].

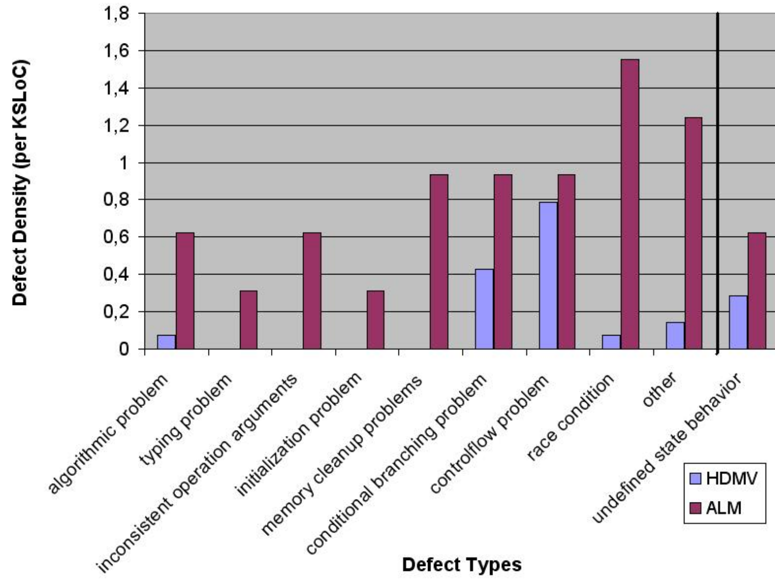

Figure 7.1: Defect density (defects/KSLoC) distribution.

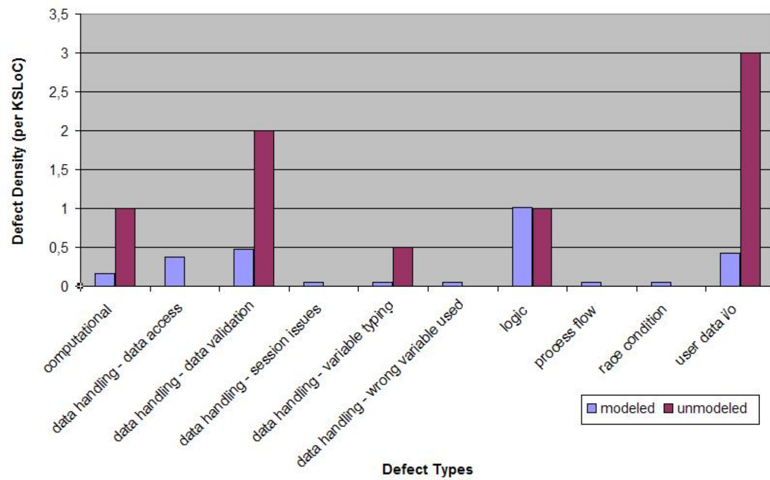

Figure 7.2: PARTS: Average defect density distributions for modeled and unmodeled system parts.

7.1.4 How does the level of detail in UML models influence the average defect repair time of individual types of defects found in a project?

Again looking at the results of Section 4.4.3.2, we see that Figure 4.2 suggests that, although all defect types seem to benefit from the availability of UML models, there is a difference in the improvement of repair time between defect types. The analysis in Section 4.4.3.3, aimed at identifying if UML modeling is the real source of the lower defect repair effort, confirms that only some defect types receive their lower defect repair time from UML modeling detail: the *conditional branching* and *race condition* defect types benefit considerably, while *control flow* and *undefined state behavior* defect repair times stay almost the same.

7.1.5 RQ5: Does upfront modeling provide enough payoffs to justify its application?

Providing enough payoffs means that the benefits attributed to modeling in some way weigh up to the costs made by applying it. In Section 4.4.3.4 we showed that although HDMV applied considerably more time on modeling, this investment was more than made up for by the total defect repair effort, which was much lower for HDMV than for ALM. Initial implementation effort has not been taken into account, but as argued earlier, we expect that *if* it influences this relation, it will only strengthen it. Our results contradict the findings in [Hov06], that suggest that payoffs in terms of lower repair time are canceled out by the extra effort that was put into modeling. However, an explanation for this was already suggested in the above-mentioned section.

We note that the term *enough* payoffs may also be interpreted differently, depending on your goals: in some situations – for instance when human life depends on correct functioning of a software system – payoffs may be considered high enough when the application of models leads to significantly lower defect density alone. In this case the anticipated cost of system failure is rated so high, that a certain amount of investment in acquiring lower defect density is considered to be justifiable. In this case the costs of modeling are subtracted by the potential cost of *not* modeling.

7.2 Threats to validity

Apart from the case specific threats to validity that were mentioned at the end of the respective cases, we additionally identified the following threats:

- Typing defects remained a subjective task. We believe, however, that this was a better option than just using whatever types were available from the bug tracking tools, since these are hardly ever filled in accu-

rately. In order to reduce different typing decisions being made by different people we regularly discussed problematic defects and randomly checked some defect types assigned by each other to see if the results were comparable.

- Results may depend strongly on the chosen defect types. The way in which these have been chosen remains subjective, although the taxonomy was based on earlier taxonomies found in literature.

- Only one case study could be used for answering research questions 3 to 5. This threatens the generalizability of these answers to other projects. Especially since we see some large differences between certain numbers (for instance defect type distribution) from both cases, we conclude that these results will probably not generalize to project of very different nature than BDMW.

- Not so much a threat, but rather an opportunity for validity would be the observation that complex system parts tend to receive more modeling attention. For them to have lower defect density would suggest a more powerful defect preventive quality in UML than the results of our tests can show here, since we would expect complex parts to contain inherently more defects. Using the SLoC code metric reduces the influence of code complexity a little, but some concepts, while not very verbose, are still harder than others.

7.3 Recommendations

Collect data on-the-fly It would be far less time consuming to get decent project data if we could convince developers to collect this data on-the-fly, meaning: during the task of solving defects. We found that it requires an enormous amount of effort to manually check each defect. When solving a defect, the developer knows exactly what changes he made to the system and therefore has all the knowledge needed to decently type the defect. The only investment it requires is that the defect taxonomy must be explained to the developers. Since we have gone through great lengths to keep the taxonomies from becoming too large, this should be a fairly small task. We further believe the data set to become more accurate, because a researcher could never hope to have the same understanding of a system under test as the system's developers themselves. This better understanding enables them to make better decisions when choosing amongst defect types.

Collect more fine-grained sequence diagram LoD metrics We would recommend to extend the approach used in the PARTS case to record LoD metrics of sequence diagrams on the class-instance level. This means

recording LoD measures in the database for each of the class instances inside a sequence diagram, instead of metrics for the diagram as a whole. We would then calculate the sequence diagram based LoD metrics for each class using these per-class metrics. This eliminates the need to make the assumption that sequence diagrams show no internal disproportion when it comes to their level of detail, because, if this disproportion does exist, our units of measurement (class instances) will be small enough to register this. In the approach used in the PARTS case, disproportion threatens the validity of conclusions

This should be done in order to deal with the assumptions we had to make regarding the absence of disproportion in LoD within sequence diagrams found in PARTS. While we believe that this assumption holds for PARTS, this may be different in future projects selected for analysis.

7.4 Future work

We see needs and opportunities for research on the following directions:

- As is always the case with empirical research, more is better! In selecting future cases for analysis we would advise to first select a case which differs from PARTS only in terms of modeling detail, keeping the rest as similar as possible. Comparing the outcomes will then help address the questions regarding validity of the conclusions drawn from PARTS. Specifically the doubts regarding system layers having a big influence on defect density can thereby be addressed further.

- We defined metrics for measuring LoD in class and sequence diagrams. Two improvements can be made in this respect. First, research can be done to determine what would be the best way to combine individual LoD values to one aggregate value (in other words: determine possible scalars for each LoD part). Second, an attempt can be made at defining LoD metrics for other diagram types.

- We noticed considerable advantages in terms of code clarity originating from the use of a state pattern in the HDMV component. It would be interesting to see if the application of design patterns in itself causes measurable benefits in terms of software quality. A quick search in literature shows not much empirical evidence exists as of yet.

- In this study we only compare LoD measures to defect density. It would also be interesting to see this study combined with the study described in [LC06] to account for both UML LoD and correctness when relating this to software quality improvements like lower defect density and repair time.

- Finally we arrive at coverage metrics. We believe future research in the same area as this study would be greatly helped by the availability of proven metrics that measure the amount of structure or behavior that is covered by created models. These measures should preferably be defined on individual diagram types. Any research outcomes on how to (automatically) calculate their values would be very relevant.

7.5 Guidelines for applying UML

When looking at the results from our research, we present the following guidelines in applying UML in upfront software design documentation:

- **Concentrate on creating behavioral diagram types.** In both case studies we see that most defects that are revealed during testing are of the behavioral type. In answering the second research question (see Section 7.1.2) we have shown that various behavioral defect types show lower defect density when diagrams are available that capture this behavior.

  From the answers to the first research question (Section 7.1.1) we also conclude that higher *level of detail* in behavioral diagrams leads to lower defect density. Of course it introduces extra costs to put this detail level into the diagrams, but the answer to research question 5 suggests that in the end these costs will be repayed.

  Meanwhile the structural defect types are lower in numbers, although the level of detail put into the class diagrams is not very high (see results from Section 4.4.2.2). We expect class diagrams will always be created (much as they have been in the past), but only using them to show overall system structure seems to suffice.

  A possible reason for this lower structural defect density may be found in the assistance that any serious IDE nowadays provides in uncovering structural defects. Variable type checking, code completion, method parameter suggestions; with the availability of all these features the necessity applying a high level of detail to ones class diagrams is considerably lowered.

- **Spend considerable time to check the correctness of the created models.** From the information in Section 7.1.2 we conclude that defects in models often translate to defects in implementation. In order to prevent too many defects from remaining undetected, sufficient effort should be spent to check if the models are consistent with the requirements.

- **Choose the set of used diagram types based on a project's nature.** The HDMV component, which displayed a lot of state behav-

ior, seemed to benefit considerably from the presence of state diagrams (again compare difference between ALM and HDMV in *undefined state behavior* and *race condition* defect density, Figure 7.1). The PARTS case did not apply state diagrams. However this would have probably been a waste of time anyway, since the race condition defect density was already negligible.

Appendix A

Existing defect taxonomies

This appendix contains synopses of defect taxonomies found in literature that were used to create the taxonomy used during the analysis of the BDMW case (as described in Section 4.2.1).

A.1 Taxonomy suggested in [CKC91]

- function
- checking
- assignment
- initialization
- documentation

All of the above categories are divided further into either *missing* or *incorrect*.

A.2 Taxonomy suggested in [CBC+92]

This taxonomy includes all categories from the previous one, but extends this by adding the following:

- timing
- build/package/merge

A.3 Taxonomy suggested in [LTW+06]

- Memory, including:

  - leak
  - uninitialized
  - dangling pointer
  - null pointer
  - overflow
  - double free

- Concurrency
- Semantic, including:
  - missing feature
  - missing case
  - corner case
  - wrong control flow
  - exception handling
  - processing
  - typo

A.4 Taxonomy suggested in [Bur03]

- Requirements and specification defects, including:
  - functional description defects
  - feature defects
  - feature interaction defects
  - interface description defects
- Design defects, including:
  - algorithmic and processing defects
  - control, logic and sequence defects
  - data defects
  - module interface description defects
  - functional description defects
  - external interface description defects
- Coding Defects, including:
  - algorithmic and processing defects

  - control, logic and sequence defects
  - typographical defects
  - initialization defects
  - data-flow defects
  - data defects
  - module interface defects
  - code documentation defects
  - external hardware, software interfaces defects

- Testing defects, including:
  - test harness defects
  - test case design and test procedure defects

A.5 Taxonomy suggested in [IEE93]

- Logic problem (8 subtypes)
- Computation problem (3 subtypes and 6 sub-subtypes)
- Interface/timing problem (3 subtypes and 5 sub-subtypes)
- Data handling problem (5 subtypes and 6 sub-subtypes)
- Data problem (6 subtypes)
- Documentation problem (10 subtypes)
- Document quality problem (5 subtypes)
- Enhancement (7 subtypes and 3 sub-subtypes)
- Failure caused by previous fix
- Performance problem
- Interoperability problem
- Standards conformance problem
- Other problem

Appendix B

Database design

The designs of the two databases are given in this chapter. This is primarily done to complement the performed queries, listed in the next appendix. Additionally some information regarding design decisions is given as well.

B.1 The BDMW analysis database

The graphical representation of the database schema can be seen in Figure B.1. The DBMS used is Microsoft Access. This DBMS was chosen because it offers easy form creation wizards and not much time was to be spent on setting up the database.

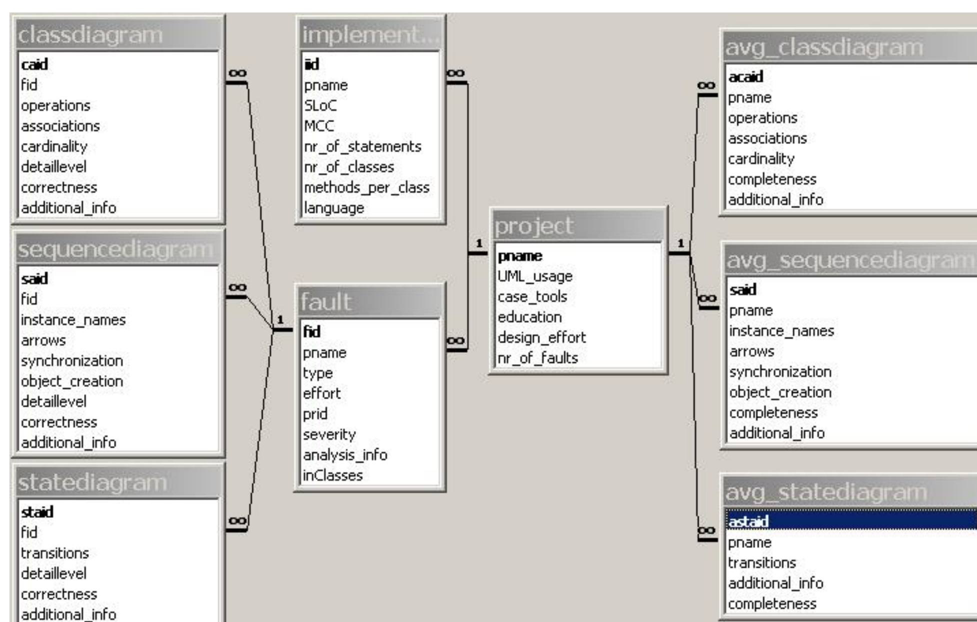

Figure B.1: BDMW database schema.

The central part of the schema is the 'project' table, which contains the project's name and some general project information. The 'nr of faults' field is needed in this table, because not all defects will be analyzed, yet we do want to know the total amount of defects at some point. This is stored here.

The one-to-many relation between 'project' and 'implementation' tables allows for multiple implementation parts (presumably based on difference in programming language) to be defined for a project. This table further contains some useful code metrics.

The fault table contains all information available for an analyzed defect. The 'analysis info' field is used to store notes regarding the analysis rationale, so that it can be looked up when needed. The 'inClasses' field was not used initially, but was later added in anticipation of the approach used for the PARTS case. It would have stored a list of all source files that were changed to fix a defect. In the end this database layout was abandoned before the field was put into use.

The three tables on the left and right side in the figure represent diagram detail tables for the three diagram types taken into account. The left ones are connected to the 'fault' table and thereby store diagram level of detail information whenever diagrams are found to be related to a defect. The right ones store average detail levels for each of the diagram types. They are needed because we only store local diagram detail levels when analyzing defects. Doing this could cause us to miss out on large parts of the design, that should be taken into account when judging a project's average level of UML detail. Also the tables on the right have one extra field, 'completeness', which is used to store project wide information regarding the amount of functionality or structure covered by the diagram types.

B.2 The PARTS analysis database

Figure B.2 shows the schema of the database used during analysis of the second PARTS. The DBMS used is MySQL. This DBMS was chosen because of our familiarity with setting up databases with it and making them available online, through a web interface.

We describe the purpose of each table shortly. A lot of the fields in the database are added for possible future use or because it was simple to "cheap" to acquire this data to pass up the chance. We will not discuss fields that are not used for this study, but rather focus on the ones that matter.
A 'projects' table contains all general project information. This is the table that has to be filled first before one can start adding additional project data into the other tables. The purpose of each of these other tables is briefly described below:

- The 'c_metrics' table stores all information regarding implementation classes. The only fields used in this study were 'classname', 'MCC'

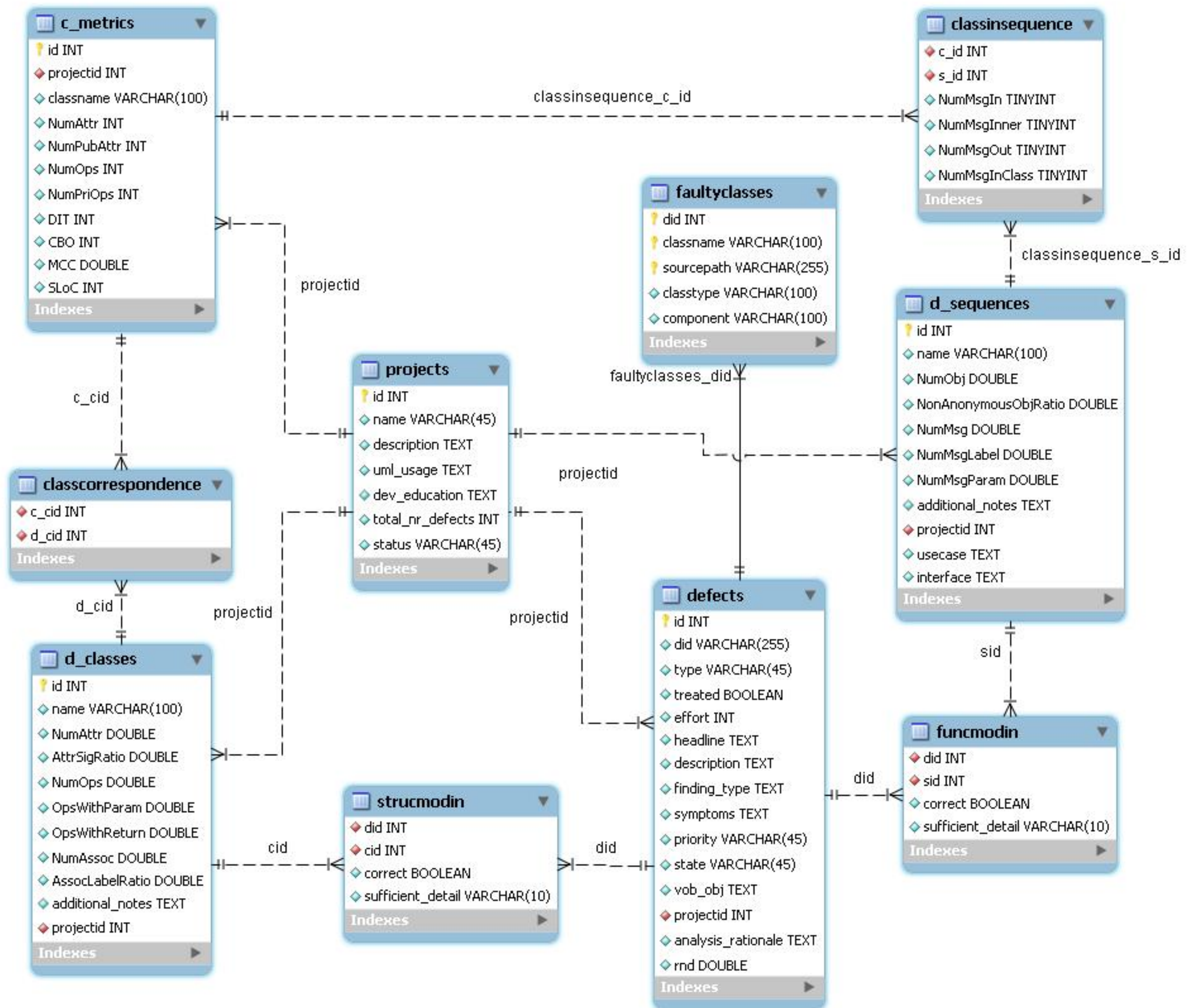

Figure B.2: PARTS database schema.

and ’SLoC’. ’classname’ is of special importance, because it is used to automatically put entries in the ’classcorrespondence’ table.

- The ’defects’ tables stores all of the defects reported in a project. Most of the content of this table is filled automatically by copying data from the bug tracking tool’s database, however, the ’type’, ’treated’ and ’analysis_rationale’ fields are filled during the manual defect analysis step, described in Section 4.3.3. The ’rnd’ field was used for random defect sampling, described in the same section.

- The ’d_classes’ and ’d_sequences’ tables record all information we need to calculate our design metrics. ’d_classes’ stores numbers for classes in the class diagram and ’d_sequences’ stores numbers for sequence diagrams as a whole. These are the numbers on which the ratios used in the analysis of the PARTS case are based, rather than the ratios themselves. This is because, when possible, data should be stored in the rawest form, so that it can later be combined in any way that seems appropriate.

- In order to relate defects to parts of the design the ’funcmodin’ and ’strucmodin’ tables were defined. Since the relation between defects and design parts is a many-to-many one, these tables were needed to store information efficiently. These tables, apart from the obvious foreign keys, store a correctness value (’correct’), which is used to record if the functionality or structure related to the defect was correctly designed or not.

- The ’classcorrespondence’ and ’classinsequence’ tables are used to link both diagram types to the implementation classes. The first is automatically filled in by performing automatic matching based on class names stored in the ’c_metrics’ and ’d_classes’ tables. In the future filling in the ’classinsequence’ table may also be automated (by parsing the *.xmi* files and performing name matching on class instances), but for the PARTS case study this information was collected manually. The extra fields in the ’classinsequence’ table are used to measure the amount of behavior of a class that is modeled in a specific sequence diagram.

- The last table to discuss is the ’faultyclasses’ table. Here the files that are changed in order to fix a defect are listed. The table also includes a ’c_cid’ field, which is used to match faulty classes to implementation classes (in ’c_metrics’), so that we can find the appropriate code metrics for a faulty class easily.

Appendix C

Performed queries

This appendix lists some of the queries performed on our data sets and gives samples of the result sets.

C.1 PARTS: comparison of defect density modeled vs unmodeled system parts

The query in Listing C.1 selects all faulty classes that are related to our filtered defect sample. Using additional filters (found in the *where* clauses) we discard the following faulty classes:

- Classes that do not correspond to implementation classes, since we want to know a class's SLoC metric to calculate defect density.
- Classes that reside in the presentation or struts layer
- Test classes

The resulting collection now holds on copy of a faulty class for each time this class is changed by one of the defects in our filtered sample. Finally we group by its classname and count this number of duplicates, the result of which will become a class's defect count.
We performed this query four times, while changing the last two *where* clauses to select faulty classes modeled in different diagram types. The results of the four queries are listed in table C.1. This table shows per part totals as well. These were calculated from the result sets in Microsoft Excel (so no reference to these fields will be found in the query mentioned above).

Querying for disjunct parts, regarding the diagrams in which faulty classes were modeled, allowed us to do both the modeled versus unmodeled defect density comparison as well as the defect density comparison of the individual parts using the same data set.

Table C.1: Result set of query from Listing C.1

BOTH modeled in CDs and SDs			
classname	**sloc**	**NrOfDefects**	**DefectDensity**
DiaryBusinessDelegate.java	202	1	0,005
GroupBusinessDelegate.java	152	2	0,0132
MedicalBusinessDelegate.java	1335	4	0,003
Patient.java	140	1	0,0071
Registration.java	92	2	0,0217
SearchPartyCriteria.java	144	3	0,0208
SeriesEntityData.java	193	1	0,0052
UserBusinessDelegate.java	245	1	0,0041
total sloc	2503		
total NrOfDefects		15	
total defect density			0,005992809
ONLY modeled in CDs			
classname	**sloc**	**NrOfDefects**	**DefectDensity**
DecursusGeneralReportEntityData.java	85	4	0,0471
GetAccessSearchCriteria.java	83	1	0,012
PartyValueObject.java	52	1	0,0192
Recurrence.java	102	1	0,0098
total sloc	322		
total NrOfDefects		7	
total defect density			0,02173913
ONLY modeled in SDs			
classname	**sloc**	**NrOfDefects**	**DefectDensity**
AssessmentProJustitiaEntityBean.java	786	1	0,0013
AssessmentWaoEntityBean.java	540	1	0,0019
ConsultationEntityBean.java	405	1	0,0025
DiaryBusinessBean.java	923	1	0,0011
GroupBusinessBean.java	647	4	0,0062
InvoiceEntityBean.java	377	1	0,0027
InvoicingBusinessBean.java	1777	1	0,0006
MedicalBusinessBean.java	5258	5	0,001
MedicationBusinessBean.java	604	2	0,0033
PartyEntityBean.java	786	1	0,0013
PatientBusinessBean.java	741	2	0,0027
RegistrationBusinessBean.java	1229	3	0,0024
SeriesEntityBean.java	397	1	0,0025
SessionEntityBean.java	386	1	0,0026
UserBusinessBean.java	1046	3	0,0029
total sloc	15902		
total NrOfDefects		28	
total defect density			0,001760785
NOT modeled in any SDs or CDs			
classname	**sloc**	**NrOfDefects**	**DefectDensity**
AWBZBudgetVO.java	239	1	0,0042
DecursusGeneralReportEntityBean.java	230	2	0,0087
DecursusGeneralReportEntityKey.java	43	1	0,0233
DiaryHelper.java	372	1	0,0027
EntrustLoginFilter.java	66	1	0,0152
GroupConstants.java	21	2	0,0952
MedicalConstants.java	31	1	0,0323
MedicationDataSource.java	202	1	0,005
PartyHelper.java	145	1	0,0069
ReferenceBusinessBean.java	128	1	0,0078
ReferenceBusinessDelegate.java	64	1	0,0156
RegistrationConstants.java	31	1	0,0323
SelectInvoiceAction.java	67	1	0,0149
TopicsDTO.java	423	1	0,0024
total sloc	2062		
total NrOfDefects		16	
total defect density			0,007759457

C.2 PARTS: comparison of average defect density per defect type

The query in Listing C.2 selects all faulty classes that are related to our filtered defect sample. Using additional filters (found in the *where* clauses) we discard the following faulty classes:

- Classes that do not correspond to implementation classes
- Classes that reside in the presentation or struts layer
- Test classes

The resulting collection now holds on copy of a faulty class for each time this class is changed by one of the defects in our filtered sample, along with the type of that defect. We now group by this defect type and count the number of rows (= defects) to get the defect count for each type.

The additional query in the *select* part of the query sums all SLoC metrics for each *distinct* faulty class that is accumulated by the grouping on defect type. We say *distinct* here, because we do not want to add a faulty class's SLoC metric more than once for each defect type.
We performed this query two times, while changing the last two *where* clauses of the main query and the mentioned sub-query to select either the modeled or unmodeled system part.

Table C.2: Result set of query from Listing C.2

type	NrOfDefects	TotalKSLoC
computational	3	18,791
data handling - data access	7	18,791
data handling - data validation	9	18,791
data handling - session issues	1	18,791
data handling - variable typing	1	18,791
data handling - wrong variable used	1	18,791
logic	19	18,791
process flow	1	18,791
race condition	1	18,791
user data i/o	8	18,791

C.3 PARTS: correlation test LoD and defect density

C.3.1 All defect types, all sequence diagrams

The query in Listing C.3 selects all faulty classes related to our filtered sample of defects that are also related to sequence diagrams. For each of these faulty

classes LoD parts are calculated, using all sequence diagrams in which a faulty class appears. The defect density calculated by counting all defects from our filtered sample that changed the faulty class and dividing by the faulty class's *sloc* (implementation) metric.

The result set of this query is given in Table C.3. In order to fit this table nicely on one page, the names of the LoD parts have been abbreviated further. The reader should have no trouble, however, linking them to their original names from the query.

Calculating the different LoD parts independently allowed for testing different combinations of them when calculating the LoD aggregate for each faulty class. This action is more quickly performed in excel (by just adding extra columns to hold different summations) than can be done by altering the query to do different calculations.

Listing C.1: Query for comparing defect density of differently modeled system parts

```
select
  fc.classname,
  c.sloc,
  count(fc.classname) as NrOfDefects,
  coalesce(count(fc.classname)/c.sloc,0) as DefectDensity
from
  c_metrics c,
  faultyclasses fc,
  defects df
where
  c.id = fc.c_cid and
  fc.defectsid = df.id and
  df.treated = 1 and
  df.type not in
    (
      '(static) user interface',
      '',
      'user interface - navigation',
      'undetermined'
    ) and
  c.classname not like '%/presentation/%' and
  c.classname not like '%/struts/%' and
  c.classname not like '%Test%' and
  fc.c_cid not in
    (select distinct cis.c_id from classinsequence cis) and
  fc.c_cid not in
    (select distinct cor.c_cid from classcorrespondence cor)
group by fc.classname
```

Listing C.2: Query for comparing defect density distribution between modeled and unmodeled system parts

```
select
  df.type,
  count(df.type) as NrOfDefects,
  (
    select
      sum(sloc)
    from
      c_metrics c1
    where
      c1.id in
        (
          select distinct
            c2.id
          from
            c_metrics c2,
            defects df1,
            faultyclasses fc1
          where
            c2.id = fc1.c_cid and
            fc1.defectsid = df1.id and
            df1.treated = 1 and
            df1.type not in
              ('(static) user interface', '',
               'user interface - navigation', 'undetermined') and
            c2.classname not like '%/presentation/%' and
            c2.classname not like '%/struts/%' and
            c2.classname not like '%Test%' and
            (fc1.c_cid in (select distinct cis.c_id from classinsequence cis) or
            fc1.c_cid in (select distinct cor.c_cid from classcorrespondence cor))
        )
  )/1000 as TotalKSLoC
from
  c_metrics c,
  faultyclasses fc,
  defects df
where
  c.id = fc.c_cid and
  fc.defectsid = df.id and
  df.treated = 1 and
  df.type not in
    ('(static) user interface', '',
     'user interface - navigation', 'undetermined') and
  c.classname not like '%/presentation/%' and
  c.classname not like '%/struts/%' and
  c.classname not like '%Test%' and
  (fc.c_cid in (select distinct cis.c_id from classinsequence cis) or
  fc.c_cid in (select distinct cor.c_cid from classcorrespondence cor))
group by
  df.type
```

Listing C.3: Query for testing LoD to defect density correlation (all sequence diagrams)

```
select
  c.c_id, cm.sloc,
  coalesce((
    select
      count(fc1.classname)
    from
      faultyclasses fc1, defects df1
    where
      fc1.c_cid = c.c_id and fc1.defectsid = df1.id and
      df1.treated = 1 and
      df1.type not in
        ( '(static) user interface', '',
          'user interface - navigation', 'undetermined')
    group by
      fc1.classname
  ),0) as NrDefects,
  coalesce((
    select
      count(fc2.classname)
    from
      faultyclasses fc2, defects df2
    where
      fc2.c_cid = c.c_id and fc2.defectsid = df2.id and
      df2.treated = 1 and
      df2.type not in
        ( '(static) user interface', '',
          'user interface - navigation', 'undetermined')
    group by
      fc2.classname
  )/cm.sloc,0) as DefectDensity,
  sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)) as SumNrMsg,
  (sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)
    *coalesce(d.NumNonAnonymousObj/d.NumObj,0)))
    /sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)) as NonAnonymousObjPart,
  (sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)
    *coalesce((d.NumObj-d.NumDummyObj)/d.NumObj,0)))
    /sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)) as NonDummyObjPart,
  (sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)
    *coalesce(d.NumMsgLabel/d.NumMsg,0)))
    /sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)) as LabeledMsgPart,
  (sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)
    *coalesce(d.NumMsgParam/d.NumMsg,0)))
    /sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)) as ParamedMsgPart,
  (sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)
    *coalesce(d.NumMsgReturnWithLabel/d.NumMsgReturn,0)))
    /sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)) as LabeledReturnMsgPart,
  (sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)
    *coalesce((d.NumMsg-d.NumDummyMsg)/d.NumMsg,0)))
    /sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)) as NonDummyMsgPart,
  (sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)
    *coalesce(d.NumMsgWithGuard/d.NumMsg,0)))
    /sum(coalesce(c.NumMsgIn+c.NumMsgInner,0)) as GuardedMsgPart
from
  classinsequence c, d_sequences d, c_metrics cm
where
  c.c_id=cm.id and c.s_id=d.id and c.c_id in
    (
      select
        fc3.c_cid
      from
        faultyclasses fc3, defects df3
      where
        fc3.defectsid = df3.id and df3.treated = 1 and
        df3.type not in
          ( '(static) user interface', '',
            'user interface - navigation', 'undetermined')
    )
group by
  c.c_id
order by
  DefectDensity desc;
```

Table C.3: Result set of query from Listing C.3

c_id	sloc	NrDef	DDensity	SumNrM	NonAnO	NonDumO	LabM	ParM	LabRetM	NonDumM	GrdM
730	23	2	0,087	2	1,000	1,000	1,000	0,200	0,000	0,600	0,200
734	22	1	0,046	1	1,000	1,000	1,000	0,231	0,000	0,462	0,231
733	28	1	0,036	1	1,000	1,000	1,000	0,154	0,000	0,462	0,000
816	92	2	0,022	3	1,000	1,000	1,000	0,667	1,000	0,500	0,000
773	144	3	0,021	2	1,000	1,000	1,000	0,508	0,000	0,873	0,071
116	64	1	0,016	1	1,000	1,000	1,000	0,546	0,000	0,727	0,273
755	152	2	0,013	34	1,000	0,994	1,000	0,616	0,423	0,827	0,143
692	83	1	0,012	4	1,000	1,000	1,000	0,613	0,125	0,713	0,113
738	678	6	0,009	8	1,000	1,000	1,000	0,146	0,000	0,381	0,158
770	140	1	0,007	2	1,000	1,000	1,000	0,500	0,000	0,375	0,000
763	647	4	0,006	20	0,950	0,980	1,000	0,652	0,483	0,686	0,207
825	193	1	0,005	2	1,000	1,000	1,000	0,778	1,000	0,778	0,111
67	202	1	0,005	49	1,000	0,980	1,000	0,599	0,459	0,758	0,227
691	245	1	0,004	83	1,000	1,000	1,000	0,615	0,405	0,820	0,107
858	604	2	0,003	2	1,000	1,000	1,000	0,750	1,000	0,750	0,500
491	1335	4	0,003	96	0,997	1,000	1,000	0,659	0,036	0,855	0,170
698	1046	3	0,003	64	1,000	0,994	1,000	0,647	0,919	0,599	0,289
316	377	1	0,003	21	0,952	1,000	1,000	0,550	0,429	0,647	0,327
766	741	2	0,003	5	1,000	1,000	1,000	0,638	0,200	0,578	0,169
594	386	1	0,003	8	0,982	0,982	1,000	0,765	0,625	0,710	0,432
830	405	1	0,003	6	0,976	0,976	1,000	0,770	0,500	0,746	0,175
832	397	1	0,003	9	0,984	0,984	1,000	0,772	0,667	0,757	0,153
817	1229	3	0,002	15	1,000	1,000	1,000	0,846	0,533	0,823	0,118
828	540	1	0,002	6	0,976	0,976	1,000	0,770	0,500	0,746	0,175
826	786	1	0,001	5	1,000	1,000	1,000	0,867	0,600	0,867	0,067
786	786	1	0,001	16	1,000	1,000	1,000	0,503	0,219	0,560	0,052
77	923	1	0,001	58	1,000	0,965	1,000	0,672	0,905	0,673	0,257
499	5258	5	0,001	27	1,000	0,972	1,000	0,776	0,488	0,690	0,246
481	7008	7	0,001	2	1,000	1,000	1,000	0,714	0,000	0,786	0,071
295	1777	1	0,001	59	1,000	1,000	1,000	0,554	0,655	0,658	0,206

C.3.2 All defect types, IOR sequence diagrams

The query described here is almost identical to the one in the previous section. As can be seen in Listing C.4, the only difference is one extra line in the *where* clause that filters out the UCR sequence diagrams. The result set of this query is given in Table C.4. Of course LoD parts are now only calculated based on IOR sequence diagrams as well.

Listing C.4: Query to test LoD to defect density correlation (IORs)

```
.
.
.
where
  c.c_id=cm.id and c.s_id=d.id and c.c_id in
    (
      select
        fc3.c_cid
      from
        faultyclasses fc3, defects df3
      where
        fc3.defectsid = df3.id and df3.treated = 1 and
        df3.type not in
          ( '(static) user interface', '',
            'user interface - navigation', 'undetermined')
    ) and
  d.usecase = '' and d.interface != ''
group by
  c.c_id
order by
  DefectDensity desc;
```

Table C.4: Result set of query from Listing C.4

c_id	sloc	NrDef	DDensity	SumNrM	NonAnO	NonDumO	LabM	ParM	LabRetM	NonDumM	GrdM
816	92	2	0,022	3	1,000	1,000	1,000	0,667	1,000	0,500	0,000
755	152	2	0,013	16	0,938	0,938	0,938	0,630	0,635	0,643	0,204
770	140	1	0,007	2	1,000	1,000	1,000	0,500	0,000	0,375	0,000
763	647	4	0,006	20	0,850	0,900	0,900	0,632	0,483	0,626	0,207
825	193	1	0,005	2	1,000	1,000	1,000	0,778	1,000	0,778	0,111
67	202	1	0,005	21	1,000	0,983	1,000	0,706	0,905	0,661	0,291
691	245	1	0,004	40	1,000	1,000	1,000	0,656	0,792	0,773	0,120
858	604	2	0,003	2	1,000	1,000	1,000	0,750	1,000	0,750	0,500
491	1335	4	0,003	5	0,950	1,000	1,000	0,595	0,400	0,556	0,159
698	1046	3	0,003	64	1,000	0,994	1,000	0,647	0,919	0,599	0,289
316	377	1	0,003	21	0,952	1,000	1,000	0,550	0,429	0,647	0,327
766	741	2	0,003	5	1,000	1,000	1,000	0,638	0,200	0,578	0,169
594	386	1	0,003	8	0,982	0,982	1,000	0,765	0,625	0,710	0,432
830	405	1	0,003	6	0,976	0,976	1,000	0,770	0,500	0,746	0,175
832	397	1	0,003	9	0,984	0,984	1,000	0,772	0,667	0,757	0,153
817	1229	3	0,002	15	1,000	1,000	1,000	0,846	0,533	0,823	0,118
828	540	1	0,002	6	0,976	0,976	1,000	0,770	0,500	0,746	0,175
826	786	1	0,001	10	1,000	1,000	1,000	0,867	0,600	0,867	0,067
786	786	1	0,001	5	1,000	1,000	1,000	0,723	0,350	0,672	0,042
77	923	1	0,001	58	1,000	0,965	1,000	0,672	0,905	0,673	0,257
499	5258	5	0,001	27	1,000	0,972	1,000	0,776	0,488	0,690	0,246
295	1777	1	0,001	44	1,000	1,000	1,000	0,575	0,682	0,667	0,154

C.3.3 Individual defect types

The query for analyses of correlation between sequence diagram based LoD aggregates and defect density for individual defect types is simply another variation on the queries described in the two previous sections. This time the query can be altered by changing all defect type selection lines in all *where* clauses to only include a specific defect type (e.g. *logic*). By doing this only faulty classes (related to our filtered defect sample) are selected that are modeled in at least one sequence diagram. LoD aggregate parts are calculated, based on the sequence diagrams that are selected (all or just IOR related). Defect density is calculated counting defect from the chosen type only.

Appendix D

Statistical tests

In this appendix we list the statistical tests that were chosen during the analysis of the PARTS case and the reasons for which they were chosen (Chapter 6). Each test is described in its own section. [SS01] was used to familiarize ourselves with these tests. We performed all tests using version 16 of the SPSS tool [Ana].

D.1 Kendall's τ

Kendall's τ (or Kendall tau rank correlation coefficient) is a non-parametric statistic that can be used as an alternative to Pearson's product-moment correlation coefficient. The latter assumes, amongst others, that frequency distributions for sample populations are approximately normal. When this assumptions cannot be met, Kendall's τ coefficient can still be calculated. We used Kendall's τ instead of the more common Spearman's ρ, because it allows a more intuitive interpretation of the coefficient. We will not go into more detail here, but rather refer to [Noe86] for a nice and short explanation.

D.2 Mann-Whitney U-test

The Mann-Whitney U-test (MWU) is a non-parametric test that checks if two samples come from the same distribution. It is often applied when prerequisites for applying Student's t test are not met (e.g. no normal distribution, no equal variances). Although it is actually not specifically designed for it, the MWU test is used frequently to show statistically significant differences in population medians[1]. Rejecting a null hypotheses that two samples are taken from populations with the same distribution actually means that their distributions lay shifted to the left or the right from each other. Using the

[1] The test works with ordinal measurements, so no statements can be made about sample means. Medians are used instead. However these are both central points of measurement in samples, so conclusions are usually interpreted the same.

MWU test to test for difference in medians implicitly assumes that medians shift along with the distribution. This assumption, however, is common in practice.

D.3 Kruskall-Wallis test

The Kruskall-Wallis test (KW) can be seen as the non-parametric version of the one-way ANOVA test and is used to test equality of population medians across more than two samples. Since it is non-parametric it can be applied in situations where the normality assumption that the parametric ANOVA test does can not be guaranteed.

D.4 Shapiro-Wilk test

This test is one of the available test used to check if normality assumptions can hold for sample data. We chose to apply *this* normality test in a number of situations, since it is the standard normality test performed by SPSS. In general, testing for normality of samples is done when one wants to apply parametric statistical analyses that assume this normality to apply.

Bibliography

[Ana] SPSS Statistical Analysis. Tool website. `http://www.spss.com/spss/index.htm`.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, 2nd Edition.* Addison-Wesley Professional, 2003.

[Boa06] International Software Testing Qualification Board. Standard glossary of terms used in software testing. `http://www.istqb.org/downloads/glossary-1.2.pdf`, 2006.

[Bur03] Ilene Burnstein. *Practical Software Testing: A Process-oriented Approach.* Springer Inc., New York, NY, USA, 2003.

[CBC+92] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Trans. Softw. Eng.*, 18(11):943–956, 1992.

[CKC91] Ram Chillarege, Wei-Lun Kao, and Richard G. Condit. Defect type and its impact on the growth curve. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 246–255, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[DP05] Brian Dobing and Jeffrey Parsons. Current practices in the use of uml. In *Proceedings of the 1st Workshop on the Best Practices of UML*, Lecture Notes in Computer Science, pages 2–11. Springer Berlin / Heidelberg, 2005.

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[HAA+06] Nina Elisabeth Holt, Bente Cecilie Dahlum Anda, Knut Asskildt, Lionel Claude L Briand, Jan Endresen, and Sverre Frøystein. Experiences with precise state modeling in an industrial safety critical system. In Robert France Dorina C. Petriu Siv Hilde Houmb,

Geri Georg and Jan Jürjens, editors, *Critical Systems Development Using Modeling Lanuguages, CSDUML'06*, pages 68–77. Springer, 2006.

[Hov06] Siw Elisabeth Hove. The impact of uml documentation on software maintenance: An experimental evaluation. *IEEE Trans. Softw. Eng.*, 32(6):365–381, 2006. Member-Erik Arisholm and Senior Member-Lionel C. Briand and Member-Yvan Labiche.

[IEE93] IEEE. *IEEE Standard Classification for Software Anomalies, IEEE Std 1044-1993.* IEEE, 1993.

[LC06] Christian F. J. Lange and Michel R. V. Chaudron. Effects of defects in uml models: an experimental investigation. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 401–411, New York, NY, USA, 2006. ACM.

[LCM06] Christian Lange, Michel Chaudron, and Johan Muskens. In practice: Uml software architecture and design description. *IEEE Software*, 23(2):40–46, 2006.

[LDCD06] Christian Lange, Bart DuBois, Michel Chaudron, and Serge Demeyer. An experimental investigation of uml modeling conventions. In *9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings*, Lecture Notes in Computer Science, pages 27–41. Springer Berlin / Heidelberg, 2006.

[Lie04] Benjamin Lieberman. Uml activity diagrams: Detailing user interface navigation. `http://www.ibm.com/developerworks/rational/library/4697.html`, 2004.

[LTW+06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, New York, NY, USA, 2006. ACM.

[MCL04] Johan Muskens, Michel Chaudron, and Christian Lange. Investigations in applying metrics to multi-view architecture models. In *Euromicro Conference, 2004. Proceedings. 30th*, pages 372–379, 2004.

[NFC08] Ariadi Nugroho, Bas Flaton, and Michel R.V. Chaudron. An empirical analysis of the relation between level of detail in uml models and defect density. Paper submitted to MODELS2008, 2008.

[Noe86] G. E. Noether. Why kendall tau? `http://www.rsscse.org.uk/ts/bts/noether/text.html`, 1986.

[Pre00] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[SS01] Peter Sprent and N.C. Smeeton. *Applied nonparametric statistical methods*. Chapman and Hall/CRC, London, 3rd ed. edition, 2001.

[tUdqmt] SDMetrics the UML design quality metrics tool. Tool website. `http://www.sdmetrics.com/`,.

[vOLC05] Dennis J.A. van Opzeeland, Christian F.J. Lange, and Michel R.V. Chaudron. Quantitative techniques for the assessment of correspondence between UML designs and implementations. 2005.