An empirical study on the relation between the quality of UML models and the quality of the implementation

Master Thesis

by

Xuan Wang

19-12-2008

Supervisors:
Dr. Michel R.V. Chaudron
Dr. Walter A. Kosters
Ariadi Nugroho, M.Sc.

Leiden University
Leiden Institute of Advanced Computer Science

Abstract

Many researches have shown that modeling design at an early stage has a positive influence on software quality such as lower software maintenance cost. However, empirical studies on illustrating this positive influence are still insufficient. Although interviews with practitioners from the industrial world indicated the importance of good modeling design to software development, an innovative research approach with careful design is needed to give a sound confirmation using statistical analysis.

In this study, we mainly explore the relation between the level of detail in UML models and defect density in the implementation. We focus on the Unified Modeling Language (*UML*) because it is now considered as the de facto modeling language for software development. Two types of UML diagrams are discussed in this study, class and sequence diagram. For each diagram type, designed metrics are provided to measure the level of detail.

Our hypothesis is that higher level of detail in UML models is significantly correlated with lower defect density in the implementation. These two variables are connected by implemented classes which are modified to correct defects found in the implementation. The level of detail values of those implemented classes are represented by the level of detail of corresponding designed classes in the UML models.

Three case studies from the industrial world are performed to analyze this relation. However, the hypothesis mentioned above is only confirmed by one study case. After performing project comparison, other factors which are closely related to defect density are found to have a rather big influence on defect density in the other two projects. Among these factors, lacking of enough data points and having poor design-code correspondence are the main issues. It is quite difficult to analyze a relation if the other confounding factors of the dependent variable (defect density in our study) are likely to affect this variable. The analysis results can be very hard to interpret too.

Therefore, we still believe that this correlation discovered in one of the case studies is valid. Designing higher level of detail in UML models to a certain extend will help improve the quality of the implementation. Architects should spend more time creating UML models at a higher level of detail when UML models are used as the basis of the software implementation.

Preface

My interest in Software Engineering started while taking a course named An introduction to Software Engineering in the last year of my bachelor study. Since my knowledge and understanding of the whole software engineering field were quite shallow by then, I decided to pursue my master study in Leiden University and expected to continue with an in-depth study on software engineering. Unfortunately, quite few master courses are software engineering oriented. However, I still keep trying to work on as many practical software projects as possible. My first project study was finished in a company which gave me the first chance to work on a real industrial software project. It was a quite exciting experience which encouraged me to work on a master project in the software engineering field.

Coincidentally, a flier offering master projects on investigating the quality of software had drawn my attention while I was looking for a master thesis topic. In a short time, I got the chance to talk with Dr. Michel Chaudron and his PhD student Ariadi Nugrohoa, M.Sc. about the opportunity for me to find a nice topic in this field. Finally, we agreed on a research topic which explores the influence of using *UML modeling* on software quality. The usage of UML models is widely believed as a good way of improving software quality by many practitioners in the industrial world, but this is never confirmed in a scientific way. Many questions related to this topic are raised, among which the correlation between the *level of detail* in UML models and defect density in the implementation is mainly investigated in this thesis study. It would be nice if some of the research results or recommendations from this study are helpful for lowering defect density rates by designing UML models in a better way. This wish is also my biggest motivation while working on this study.

It is also a good chance for me to thank those many people who helped and supported me when I was working on this project. The first person I would like to thank is Dr. Michel Chaudron who trusted me and gave me this chance to "destroy" this project. His encouragement and wise advices were quite essential to me whenever I encountered the difficulties. Of course, I will not forget that he kept encouraging me to be more creative and having my own ideas during the research. I will keep that in mind even when I join the labor market.

I would very much love to give my great thanks to Ariadi Nugrohoa, M.Sc. I can not remember how many times he has helped me out. Also, thanks to his great ideas and suggestions during the project process and always being generous in sharing his knowledge. Being my mentor and seeing me almost every day must have been a "torture" to him. However, I enjoyed the nice collaborations a lot.

Many thanks to Dr. Walter Kosters who agreed to be my second supervisor and be part of my examination board. Also, special thanks to Drs. Jeroen F.J. Laros who was brave enough to read my draft report and provided tons of great suggestions on how to improve it.

Lots of thanks and love to my friends both in the Netherlands and China. Can't write all your names here but I am always very happy and full of energy while being with you. Finally, I would love to give my greatest thanks to my parents whom I have not seen for one and half years already. I am always amazed by their great kindness and tolerance to me. Hope I can be at home this year during the chinese spring festival.

Contents

Chapter 1

Introduction

In this introductory chapter the motivation for performing this research study is given. After this we present the main research questions of this study. Right after the research questions, the hypotheses used in this study are listed. Finally, an outline of the content of this thesis is given.

1.1 Motivation

For several decades, Software Engineering has been trying to become a true engineering discipline with firm and well-understood foundations. Researchers and practitioners from both the academic institutes and industrial fields always work hard to develop effective and successful software engineering processes. In general, a software development process consists of requirements specification, architecture design, implementation, testing, deployment and maintenance. This software development process is also known as *software life cycle*. The demand for productivity in software engineering is the main drive of changes in the ways that software development and maintenance are being processed. Basically, over 80 % of total cost is due to *software maintenance* (including corrective, adaptive, and perfective maintenance). Several primary means contributing most to improved productivity mentioned in [But97] are:

- Improvement of *software quality* is considered as a main mean to reduce the cost of maintenance.
- *Reuse* of code and other software components, such as requirements and design, is helpful for the purpose of reducing production cost and improving the quality of individual components.
- *Modeling* notations and tools which provide a model of the software are essential to understand software, thus reducing costs of development, maintenance, and evolution.

In this study, with the purpose of reducing maintenance cost by improving software quality as the main motivation, we focus on examining the influence of using diagrammatic models, particularly represented in *UML models* used during architecture design of software development, and on software quality represented in *defect density*. Defect density is chosen to represent software quality because

it is considered as a de facto standard measure of software quality. Meanwhile, we are interested in UML modeling, mainly because graphical design is better in communication and understanding concerning the software system than other means of modeling, such as textual and mathematical models. With these graphical notations, UML modeling is generally considered as a good practice in almost any software development process. In most projects, UML models are the first tools used for illustrating software architectures and believed to be helpful in increasing software quality. However, empirical investigation on validating the assumption that UML modeling is helpful in improving software quality is neither profound nor sufficient. Some practitioners from the industrial fields even question whether enough payoff can be obtained by using UML models. Using three significant industrial case studies, this study provides sufficient empirical evidence that the improvement of software quality benefits from using UML models.

1.2 Research questions

In this study, we would like to explore the relation between *level of detail in UML models* (from now on called *UML LoD* in short) and *defect density*. By doing three empirical case studies, we expect to find the significant relation between the two variables mentioned above. Further, we want to know which aspects of UML model details influence this relation strongest. Later, if possible, some recommendations on at which level of detail UML models should be developed are given. In order to achieve these goals, the following research questions are considered in each case study:

RQ1: Does the usage of UML models influence defect density in software systems?

RQ2: How does the level of detail in UML modeling influence a project's defect density?

RQ3: Which metrics have more contribution to predicting defect density?

RQ4: Which metrics have a stronger correlation with a certain defect type?

RQ5: Are there any recommendations on applying UML models to software development?

1.3 Hypotheses formulation

In this section, hypotheses of research questions used in this study are given.

RQ1: Does the usage of UML models influence defect density in software systems?

To explore this research question, three sub-questions are investigated.

- Is there a significant difference of defect density between modeled system parts using UML and system parts not modeled at all?

The hypothesis to this question is given based on the following observations and previous work [Fla]. First, modeled system parts are normally well designed in a good manner (i.e., visualization in UML models). Second, UML models enhance better communication among developers while implementing modeled system parts. Finally, a previous case study on the PARTS project in [Fla] indicated that modeled system parts had a significantly lower defect density than system parts that are not modeled at all. Therefore, the following hypothesis is tested in order to validate the above assumptions.

H_0: There is no significant difference of defect density between system parts that are modeled using UML and those which are not modeled at all.

H_{alt}: System parts that are modeled using UML have significantly lower defect density than those which are not modeled at all.

- Is defect density of system parts modeled using diverse types of UML diagrams significantly lower than that of system parts using single UML diagram type?
  This question is addressed if system parts modeled using UML are confirmed to have significantly lower defect density than those not modeled at all. The modeled system parts are divided into several sub-categories based on the diagram types they were modeled in; such as system parts that are only modeled in class diagrams, those only modeled in sequence diagrams, those modeled in both types of diagrams and those not modeled in neither of the diagram types. We believe the usage of UML diagrams helps the developers have a better understanding of the UML models and eases the communication between architects and developers. Further, modeling using diverse types of UML diagrams provides developers different views while describing the same design. Based on the observations mentioned above, the following hypothesis is given:

  H_1: There is no significant difference of defect density between system parts that are modeled using diverse types of UML diagrams and those which are only used in single UML diagram type.

  H_{alt}: Modeled system parts that are presented in diverse types of UML diagrams have significantly lower defect density than system parts which are only used in single UML diagram type.

- Is there a significant difference of defect density between modeled system parts presented in class diagrams and those only modeled as design classes in UML but not referenced in any class diagram?
  While doing case studies, some implemented classes were found modeled as design classes in UML models but not used in any of the class diagrams. We were curious to see whether there is a significant defect density difference between classes that are modeled only as design classes but not used in any class diagram and design classes which are also presented in class diagrams. The hypothesis is listed below:

  H_2: There is no significant difference of defect density between modeled system parts that are presented in class diagrams and those only modeled in the UML models.

H_{alt}: Modeled system parts that are presented in class diagrams have significantly lower defect density than those only modeled in the UML models but not presented in any class diagram.

RQ2: How does the level of detail in UML models influence a project's defect density?

In UML modeling, the level of detail can be measured by quantifying the amount of information that is used to represent a modeling element. We believe that the appropriate level of detail in UML models is important to assure that developers can easily understand the architecture design of the project and be able to implement the code in a clear and efficient way. This research question is investigated to see whether there is a significant correlation between the level of detail in UML models and defect density in the implementation. Two types of diagrams, class diagram and sequence diagram, are used to perform the tests. As is mentioned in previous work (see Chapter 3), a similar analysis was performed [NFC08] and an important conclusion was that there is a negative correlation between UML level of detail using sequence diagram metrics and defect density. This conclusion suggests that the level of detail in UML models can be considered as a predictor of a project's defect density. However, no significant correlation between UML level of detail using class diagram metrics and defect density was found. Although this correlation is not significant, a negative relationship is shown according to the statistical analysis. Therefore, in our study, the hypothesis of the relation between UML level of detail and defect density is:

H_3: There is no significant correlation between level of details of UML classes in UML models and the defect density of the implementation classes.

H_{alt}: UML classes that are modeled in a higher level of detail significantly correlate with a lower defect density in the implementation classes.

1.4 Organization of this report

- In Chapter 2, some important and frequently used terms are defined.
- In Chapter 3, previous research related to this study will be discussed.
- Since for each case, we used the same approach in exploring the influence of UML LoD on defect density, in Chapter 4, we list defect taxonomy and metrics using different diagrams that are shared by the three cases.
- In Chapter 5, we will explain the research approach in detail, each step taken to perform this study is discussed.
- We report the three case studies in different chapters. Chapter 6 first shows the analysis results of the case study mentioned in [NFC08, Fla]. The two new case studies are handled in Chapter 7 and 8, respectively. Each chapter can be read as a self contained part which consists of project definitions, data gathering, data analysis and analysis results.

- Chapter 9 provides a comparison of the three case studies in order to give readers a clear view of findings across projects.
- Chapter 10 tries to answer the research questions based on our findings. At the same time, the threats to the validity of the study are mentioned. Finally, some recommendations on future work are suggested.

Chapter 2

Nomenclature

In this chapter, we try to explain the meaning of several important and frequently used terms. Since sometimes these terms have different definitions in distinct situations, we would like to give our definitions here.

2.1 UML models

In [Fow03], *UML* is defined as "a family of graphical notations, backed by single meta-model, that help in describing and designing software systems, particularly software systems built using the object-oriented (OO) style".

In general, UML can be used in three modes: sketch, blueprint, and code generation. The first two means are more on an abstract level, while the last one is used for generating programming code. A *sketch* is used as a thinking tool which helps developers communicate some aspects of a system and alternatives about what are about to be done. The essence of sketching is selectivity. Only some important issues which will be written as code in the future are roughed out and visualized before the code implementation. Normally, UML models used as sketches are pretty informal and dynamic. The tools used for sketching are lightweight drawing tools, and the rules of using the UML models can be hardly kept. Meanwhile, they are also useful in documents, in which case the focus is selective communication rather than complete specification. In contrast, a *blueprint* is about completeness and designed in much more detail than a sketch with the aim of reducing programming to a simple and fairly mechanical activity. UML models used as blueprints are developed by an architect who builds a detailed design for a programmer to code. A common approach of using blueprints is for an architect to develop blueprint-level UML models as far as interfaces of subsystems but then let developers work out the details of implementing those details. Generally, the created blueprints should be able to show every detail about a class in a graphical form that is easier for developers to understand. Much more sophisticated tools are required in designing blueprints in order to handle the details required for the task. The difference between UML models used as sketches and blueprints is blurry. Sketches are deliberately incomplete and more explorative, while blueprints intend to be comprehensive and definitive. The third mode of UML is *code generation* tool. Developers draw UML diagrams that are compiled directly to executable code and the UML models

become the source code. Obviously, this usage of UML demands particularly sophisticated tools.

In this study, UML is used as a blueprint which is used for guiding the implementation. The idea behind this is that sketches are more for communication and have much less connection with the implementation, while code generation emphasizes too much on using UML as a modeling language rather than visualization of architecture designs.

2.2 Defect & Defect density

In [Fla], the definition of defect was given according to [Boa]. In order to make the results comparable, we tried to keep definitions as similar as possible. Here, we used the same definition of defect as follows:

> 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.

Failure is defined as:

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

The term defect is synonymous to the terms bug and fault according to [Boa]. Sometimes, we might use other terms to comply with common practice (e.g., a bug tracking tool) which refers to term defect in that case.

In this study, we intend to find the relation between the quality of UML models and the quality of the implementation. In particular, **defect density** is used as a measurement of the quality of the implementation. The definition of this term is given as follows:

> *Defect density* is the ratio of the number of defects appearing in each implementation class to the class' size measured in KSLoC (thousand source lines of code).

The last definition we would like to give is *defect-count*, which refers to the number of defects in each implementation class. This value is calculated using the information in the bug tracking and versioning tools.

Chapter 3

Related work

In this chapter previous research related to this topic is presented. We start with the early research related to this research area. Since two research studies are closely related to this study, we focus on introducing the results indicated in these studies. At the same time, the relationship between this study and the previous research is explained.

Nowadays, the quality of the software process is essential for productivity and competitiveness of a software product. Actually, it is stated in [But97] that quality of a software product is usually only achieved through a high-quality software development process. What's more, the increased acceptance of modeling notations and tools are considered as an essential step in understanding software, thus reducing costs of development. In the same research, design models were classified into three categories, diagrammatic models, textual models and mathematical models, with which one can have multiple views of a system. It also mentioned that Harel [Har92] is optimistic in the sense that developers can exercise greater intellectual control over complexity by using models throughout the development process. Further, Boehm and Bsili [BB01] emphasized the benefits of modeling at an early stage to help improve software quality. They stated that it is usually 100 times more cost effective fixing software problems in the analysis or design phase than fixing them after delivery. Hence, more effort is recommended to be put on requirement specification, early verification and validation, upfront prototyping and design. The importance of using models is again discussed in [Sel03]. Here, it is argued that engineering models can help reduce software maintenance cost by providing means to understand complex problems and develop solutions to them before the implementation phase. Hence, many unnecessary problems that are found in the later phases (such as code implementation, testing, deployment and maintenance) can be prevented well in advance to reduce fixing costs.

Talking about the Unified Modeling Language (*UML*), the advantages of using UML to software development have been widely discussed by researchers and practitioners. UML is now considered as the de facto modeling language for software development. It is designed as a visual multi-purpose language with thirteen diagram types to serve communication between developers, prediction of quality properties and test case generation. Some controlled experiments have been done to analyze the relation between the presence of UML models and some aspects of software quality, such as accuracy, speed, etc.

In [Hov06], it is found that UML had a generally positive impact on the quality of the solutions to maintenance tasks of the system. This conclusion is promising, however, it is still too abstract. On the other hand, Lange did several researches [LCM06, LC06, BLDC06] on revealing common problems with UML models and how these problems influence the software quality. Meanwhile, some techniques and practical recommendations on controlling these problems are discussed. In [LCM06], some common problems with UML models in practice were listed through a user survey and an industry case study. From the survey, it can be seen that around 66 % practitioners used UML models as an architecture design tool. *Completness* was chosen as the most prominent criterion on determining when modeling activities can end. It means that an UML model with complete information is likely to be well designed, thus having higher quality. Some major problems with UML models indicated by architects were *scattered information*, *incompleteness*, *disproportion* and *inconsistency*. Among these problems, problems with incomplete models were investigated and four problems were encountered as a result of incomplete models, such as bad quality of the implemented product, wrong product delivered in terms of not matching the stated requirements, high amount of testing effort and miscommunication between project members and stakeholders. Besides, some other problems with UML models were included such as diagram quality, informal use and lacking of modeling conventions. In [BLDC06], a controlled experiment was investigated to explore the effect of modeling conventions on defect density and modeling effort. The results indicated that decreased defect density in UML models is attainable at the cost of increased effort when using modeling conventions. In [LC06], effects of defects in UML models were explored through an experimental investigation. In this study, different defect types were defined based on a ranking of detection rate and risk of misinterpretation. One conclusion showed that defects in UML models are hardly detected and are potential risks that can cause misinterpretation and, therefore, reduce software quality.

The previous research mentioned above indicates that the quality of UML models has a positive influence on software quality. At the same time, the quality of UML models consists of several aspects, among which *completeness* is considered as a main criterion on creating UML models with high quality. In [NFC08, Fla], an empirical analysis was performed on finding the relation between *level of detail* in UML models and defect density found in the implementation phase. These two studies proposed a measurement for UML models called *level of detail* to investigate the relation between the quality of UML models and the quality of the implementation. We believe that *level of detail* in UML models is an important criterion when making complete UML models. A defect taxonomy (see Section 4.1) was introduced and metrics for measuring LoD in UML models were given. More information on creating defect taxonomy can be found in [CBC+92, CKC91]. The notion of using design metrics as quality indicators of fault-prone classes in the implementation phase was introduced by Basili [BBM96]. The suite of Object-Oriented (OO) design metrics introduced by [CK94] was experimentally investigated through eight medium-sized information management systems based on identical requirements. The results showed that several design metrics appeared to be useful to predict class fault-proneness during the early phases of the life-cycle. Besides, they are better predictors than "traditional" code metrics which can only be collected at a later phase of the software development processes. In [NFC08, Fla], several

design metrics (see Section 4.2) were used to measure the UML LoD. The case study performed in these two studies was based on a project named PARTS. The main conclusion of this study case is that implemented classes modeled in sequence diagrams with a higher level of detail are found to have a significant correlation with lower defect density in the implementation. However, whether this conclusion can be generalized is still a question. At the same time, lacking of enough data samples is another concern, for instance, the same conclusion was not obtained between LoD using class diagram metrics and defect density. This might be due to insufficient data points. As a result, more research is needed in the future.

In our study, we would like to analyze the relation between UML LoD and defect density through three different empirical case studies. A set of more fine-grained sequence diagram LoD metrics are needed in order to give a more precise measurement of UML LoD. Later, we try to see which metrics have the strongest influence on predicting defect density of implemented software systems. Finally, we would like to combine the conclusions from all these research and give some recommendations on at which level of detail UML models should be developed.

Chapter 4

Defect taxonomy and Level of Detail measurement

In this chapter, we list the same defect taxonomy and metrics used for UML level of detail measurement as mentioned in [NFC08, Fla]. For level of detail metrics used in sequence diagrams, a new set of metrics are introduced in this study which will be explained shortly.

4.1 Defect taxonomy

This defect taxonomy is a merger of two taxonomies created by Ariadi Nugroho and Bas Flaton respectively. It is based on both the previous experiences from Nugroho in analyzing a number of information systems and Flaton's own experience in analyzing the more technical and embedded BDMW system mentioned in his master thesis. It is also used in [NFC08] and in the second case study called PARTS in [Fla].

The detailed explanation of defect taxonomy is shown below:

1. **(static) User interface** - Any defect that only has something to do with the way the user interface looks (window or form sizing, font choices, positioning, missing labels or titles, textual improvement, etc.). This defect is mainly front-ended and has little to do with the back-ended source code.

2. **(navigation) User interface** - Defects regarding screen transitions (wrong destination, missing intermediate screen, etc.).

3. **Logic** - Defects caused by missing or wrong implementation of business or processing rules. Normally, it is a defect regarding conditional branching.

4. **Process flow** - Defect caused by missing or wrong process flows (e.g., incorrect order of operation execution). Different orders result in distinct outcomes.

5. **Race condition** - Unforeseen output as a result of unforeseen sequence or timing of events (locking problem which prevents accessing data at any time, etc.).

6. **Data handling** - Defects caused by missing or poor data handling.
   (a) **Data validation** - Input which is not, or incorrectly validated. It has two main aspects: input validation and exception handling caused by invalid input. It happens on the business/application layer which is responsible for checking the input.
   (b) **Data access** - Defects related to retrieving, storing, inserting, updating and deleting data from/to a data store (like a database). It happens during the interaction with a database.
   (c) **Session issues** - Defects related to session specific data. It happens, for example, when information is missing while using session to keep the data.
   (d) **Wrong variable used** - Wrong variable used in checking for, or assigning a value.
   (e) **Initialization** - Uninitialized or wrongly initialized variables/objects (or other data sources).
   (f) **Memory cleanup** - Missing or incorrect cleanup of data sources at the end of the process flow.
   (g) **Variable typing** - Incorrect type chosen or assumed for a variable (like considering a float type as an integer).
   (h) **Inconsistent operation arguments** - Wrong number or types of arguments when calling a function.
7. **User data I/O** - Defects related to missing or wrong data input and output from/to the user interface (a functionality/field is missing according to the design, exception handling not caused by wrong input, etc.). This type of defect is user interface related.
8. **Computational** - Erroneous calculation of values (like a wrong equation).
9. **Undetermined** - Defects that can not be classified in any of the classes mentioned above. Findings related to functional requirements changing problems are currently considered as undetermined.

4.2 Metrics for measuring UML level of detail

In this study, we still measure UML LoD by using metrics. In order to make the results comparable, we again use the metrics mentioned in the PARTS project [Fla]. Only two types of diagrams used in this study, class diagrams and sequence diagrams. As is mentioned in PARTS, the sequence diagram based UML LoD has an entire diagram as its unit of measure. This might cause a disproportion problem. In this study, a *disproportion* is a property that within one diagram a certain part is modeled at a higher level of detail than another part. The assumption that there is not much disproportion within sequence diagrams might be a threat to the validity of the results. In that case, we added a set of sequence diagram metrics which used UML LoD of the corresponding UML class of an implementation class as the LoD of that implementation class. In principle, this should reflect the nature of UML LoD better than using metrics based on diagram level.

Below, the used collections of class and sequence diagram metrics are described in two subsections.

4.2.1 Class diagram based level of detail metrics

- **NumAttrRatio** *(CD_{m1})*
  Measures the ratio of the total number of attributes of a class in a model to that in the implementation.

- **AttrSigRatio** *(CD_{m2})*
  Measures the ratio of attributes with a signature to the total number of attributes of a class.

- **NumOpsRatio** *(CD_{m3})*
  Measures the ratio of the total number of operations of a class in a model to that in the implementation.

- **OpsWithParamRatio** *(CD_{m4})*
  Measures the ratio of operations with parameters of a class in a model to that in the implementation.

- **OpsWithReturnRatio** *(CD_{m5})*
  Measures the ratio of operations which return values of a class in a model to that in the implementation.

- **AssocLabelRatio** *(CD_{m6})*
  Measures the ratio of associations with a label (e.g., association name) to the total number of associations of a class.

- **AssocRoleRatio** *(CD_{m7})*
  Measures the ratio of associations with a role name to the total number of associations attached to a class.

4.2.2 Sequence diagram based level of detail metrics

Sequence diagram metrics measured at both diagram level and class-instance level are the same. The only difference is the perspective of measuring the metrics. As mentioned before, metrics measured at diagram level consider the whole diagram as a unit of measuring, while class-instance based metrics regard the corresponding single design class in the diagram as a unit of measuring. We list the definitions as follows:

- **NonAnonymObjRatio** *(SD_{m1})*

  Diagram level: Measures the ratio of objects with a name to the total number of objects in a sequence diagram.

  Class-instance level: Measures the ratio of the implementation class modeled as an object with a name to the total number of this implementation class modeled as objects in a sequence diagram.

- **NonDummyObjRatio** *(SD_{m2})*

Diagram level: Measures the ratio of non-dummy objects (objects that correspond to classes) to the total number of objects in a sequence diagram.

Class-instance level: Measures the ratio of the implementation class modeled as a non-dummy object (object that corresponds to a class in the model) with a signature to the total number of the implementation class modeled as objects in a sequence diagram.

- **MsgWithLabelRatio** *(SD_{m3})*

  Diagram level: Measures the ratio of messages with a label (any text attached to the messages) to the total number of messages in a sequence diagram.

  Class-instance level: Measures the ratio of messages with a label attached to an object to the total number of messages attached to this object in a sequence diagram.

- **NonDummyMsgRatio** *(SD_{m4})*

  Diagram level: Measures the ratio of non-dummy messages (messages that correspond to class methods) to the total number of messages in a sequence diagram.

  Class-instance level: Measures the ratio of non-dummy messages (messages within the class under investigation which pop-up in the class diagram) attached to an object to the total number of messages attached to this object in a sequence diagram.

- **ReturnMsgWithLabelRatio** *(SD_{m5})*

  Diagram level: Measures the ratio of return messages with a label (any text attached to the return messages) to the total number of return messages in a sequence diagram.

  Class-instance level: Measures the ratio of return messages with a label attached to an object to the total number of return messages attached to this object in a sequence diagram.

- **MsgWithGuardRatio** *(SD_{m6})*

  Diagram level: Measures the ratio of guarded messages (messages with conditional checks) to the total number of messages in a sequence diagram.

  Class-instance level: Measures the ratio of guarded messages attached to an object to the total number of messages attached to this object in a sequence diagram.

- **MsgWithParamRatio** *(SD_{m7})*

  Diagram level: Measures the ratio of messages with parameters to the total number of messages in a sequence diagram.

  Class-instance level: Measures the ratio of messages with parameters attached to an object to the total number of messages attached to this object in a sequence diagram.

We still keep the sequence diagram metrics measured at diagram level because they are the metrics used in [NFC08]. We would like to compare the results of using sequence diagram metrics measured at both diagram level and class-instance level in order to see whether the conclusions drawn from [NFC08] are still valid.

Both class and sequence diagram based metrics are calculated in ratios instead of absolute numbers. The reason is the same as mentioned in [Fla]: we do not expect the UML LoD to be influenced by the size of a class or a sequence diagram.

Chapter 5

Research approach

In this chapter, the research approach performed in this study is explained in detail. First, we outline the steps that are involved in this approach. Later, each step is discussed separately.

The steps taken to perform this study can be summarized as follows: project selection, data collection, data preprocessing, class matching, data analyses, and reporting (see Figure 5.1).

5.1 Project selection

Two main criteria should be met while selecting software projects for case studies. First, the projects must use UML modeling to a certain extend. As mentioned in Chapter 2, UML models should be used as the basis of the implementation such as *blueprints* and modeled in machine-readable forms (e.g., utilizing UML CASE tools). The UML CASE tools, for example, are also required to have an `.xmi` export facility, which will allow us to export the models to the measurement tool. Second, the projects must utilize a bug tracking system which makes it possible to trace back source files that are modified to solve defects. This bug tracking system must provide sufficient information to determine the nature of the registered defects.

In this report, the three software projects chosen as case studies meet the two criteria mentioned above very well. The UML models were designed in the architecture design phase and were used as guidelines during the code implementation. IBM Rational XDE [Wikc] was used to create the UML models and export `.xmi` files for calculating the values of the metrics. Some other IBM Rational tools were adopted for code versioning and bug tracking, namely IBM Rational ClearCase [Wika] and IBM Rational ClearQuest [Wikb], respectively.

5.2 Data collection

After selecting the projects to be studied, the data collection step is performed to obtain data consisting of UML models, source code, and findings during defect registration. The collected UML models and source code were extracted from the latest version of the project data found in CVS (concurrent version system) repository. UML models were created and stored in IBM Rational XDE

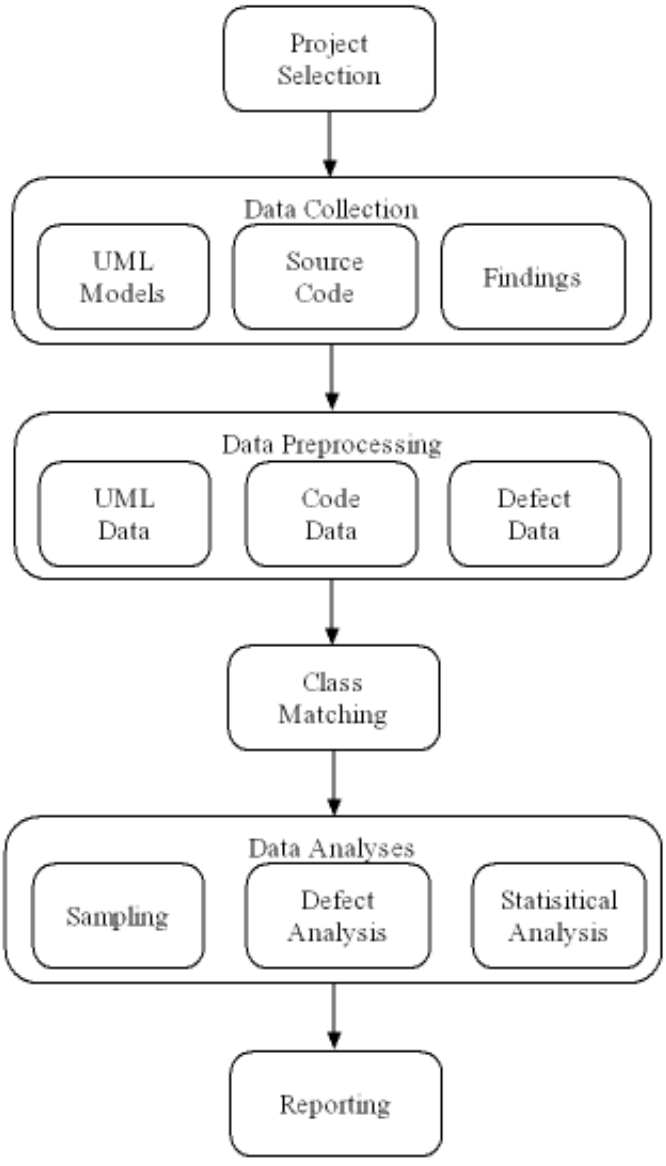

Figure 5.1: Visualization of The Research Approach

and could be used for LoD metrics calculation in the data preprocessing step. Source code could be obtained from IBM Rational ClearCase. The defect data being analyzed in this study refer to defects found during the development phase. However, not all registered data can be considered as defects. Before defect typing analysis, we refer to these data as *findings*. Findings recorded during defect registration were obtained from IBM Rational ClearQuest. The change sets attached to those findings can also be collected using this tool.

5.3 Data preprocessing

Data preprocessing is performed right after data collection. It consists of three steps, UML metrics calculation, code metrics calculation, and faulty classes identification. We will discuss each step in detail shortly.

5.3.1 UML related data

UML related data refers to data about components/classes and LoD metrics from UML models. To obtain this data, UML models first had to be exported from IBM Rational XDE into the `.xmi` format. A tool called SDMetrics [dqmt] was used to perform this task. The extracted data was stored in the analysis database. For class diagrams, the values of the UML LoD metrics from UML models were calculated automatically using SDMetrics which supports definition of customized metrics. However, the calculation of sequence diagram metrics could not be performed automatically because Rational XDE does not export the sequence diagram information in the `.xmi` exporting process. Hence, manual inspection of the sequence diagrams in Rational XDE was performed to calculate the values of the sequence diagram metrics measured at both diagram level and class-instance level.

5.3.2 Code related data

In this study, code metrics were used for calculating UML LoD using class diagrams. An open source tool named CCCC [sou] was used to perform this task and metrics were calculated from `.java` files. Additionally, the size of the implementation classes measured in *KSLoC* (Thousand Source Lines of Code) was obtained using the same tool. The same analysis database mentioned above was used to record this source code related information.

5.3.3 Defect related data

Data preprocessing for getting defect related data involved two steps. The first step was to obtain registered findings from the Rational ClearQuest repository and store them in the same analysis database. The textual information attached to each finding explains the nature of this finding, which was used to help determine whether a finding can be regarded as a defect and if so, of which type. Hence, this step is useful for defect typing which will be introduced shortly. The second step was to obtain change sets (source files modified to solve defects), which was performed automatically using a Perl script that recovers change sets associated with every finding. Here we only took into account the `.java`

files (thereby mostly excluding configuration files like `.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. Java files that were modified to solve findings are considered as *faulty classes* and were stored in the analysis database again.

One important note is that a faulty class may be changed several times while fixing the same defect. Hence, a faulty class was regarded as having only one defect-count in this case.

5.4 Class matching

After performing the three steps mentioned above, the data of UML classes, implementation classes, defect findings, and faulty classes can be obtained from the analysis database. The next step is to perform matchings between UML classes and the implementation classes, and between faulty classes and implementation classes. The two matching processes were done semi-automatically based on class/instance name and directory structure similarities. The connection between UML classes and faulty classes was made through implementation classes. This connection allowed us to identify which faulty classes were modeled as design classes and if so, modeled in which type of diagram.

5.5 Data analyses

In this study, data analysis consists of three parts: data sampling, finding analysis and statistical analysis. Before the main statistical analysis, an essential task is to perform defect typing. This is done to determine whether a finding can be regarded as a defect and if so, of which defect type according to the defect taxonomy defined in Chapter 4 (see Section 4.1). As noted before, the term *finding* was used to refer to the defects registered prior to defect typing. In fact, some findings registered during the defect registration are not defects according to our definition (see Section 2.2) or could not be identified. In the following paragraphs, both the finding analysis and statistical analysis are discussed.

5.5.1 Data sampling

Since defect typing is done manually, it is impossible to analyze all the findings if the defect population is huge. Some other concerns are limited time and resources. Hence, a sample data set is needed. The sample size differs from project to project, ranging from 100 to 200 data points. The sampling method will be explained in relevant chapter of each case study.

5.5.2 Finding analysis

In practice, a finding is regarded as a defect if it was registered due to explicit errors in the system or due to deviations from explicitly stated requirements. Hence, findings registered to incorporate additional functionality into the system were not regarded as a defect.

In addition to excluding non-defect findings, it is essential to determine the defect type of each defect based on the defined defect taxonomy. Defect typing

is done primarily based on how it is solved by comparing the modified Java files before and after correction. Additionally, the attached defect description is another important criterion of judgment. However, some defects might be solved without modifying any files or have unclear problem descriptions. Thus, their types might be hard to determine. In this case, they should be excluded from the further analysis.

Once all the findings had been analyzed and categorized, findings that did not meet the criteria of defects should be excluded. Furthermore, certain defect types were disregarded: *(static) user interface*, *(navigation)user interface*, and *undetermined* in particular. These defect types are excluded because they are not likely to be related to the use of UML models. Having excluded those irrelevant findings and defect types, we are certain that we do not overstate defect-count of faulty classes due to the use of irrelevant findings or defects.

5.5.3 Statistical analysis

After filtering the defects data, we proceeded with the statistical analysis to help answer the research questions listed in Chapter 1. Statistical tests either parametric or non-parametric were performed when there was a necessity. In principle, the parametric tests were considered first. For all parametric tests, two assumptions should be met: a normal data distribution; variances in the data are roughly equal (*homogeneity of variances*). To adhere to the first assumption, if the original data set is not normally distributed, a data transformation can be performed and the assumption is valid if the transformed data is normally distributed. However, if one of these two assumptions is still violated after performing data transformation, the correspondent non-parametric tests should be performed instead. Detailed explanation about these tests is discussed in relevant case studies. The statistical tool SPSS [Ana] (version 13) is used to perform the analysis.

5.6 The analysis database

An analysis database was built to store all the information needed according to this research approach. It is a MySQL database designed for performing queries to all data useful for statistical analyses. A front-end based on PHP was developed to enable entries in the database. An overview of the database schema and some explanations on several essential parts are given in Appendix A.

Chapter 6

Case study 1: PARTS

In this chapter, we describe the first case study performed within the PARTS project. This project was first analyzed in [NFC08] and some important conclusions were drawn based on the answers to the research questions in that study. In our study, this project is used again because we believe that the data sets are helpful for answering the research questions related in this study. The data sets used in this case study are not exactly the same as those used in [NFC08]. The values of sequence diagram metrics measured at diagram level were checked again and some changes had been made in the data according to the definition of those metrics. Besides, the values of sequence diagram metrics measured at class-instance level were calculated and added to the analysis database for further use. In our study, the second research question regarding analyzing how the level of detail in UML models influences a project's defect density is the same as the research question discussed in [NFC08]. Although the data sets have been modified, a comparable result of this research question mentioned in [NFC08] is still expected in our study. On the other hand, some new findings related to other research questions are also expected.

In this chapter, the description of the project itself is given first, immediately followed by discussions on data collection and preprocessing processes. Once the purified data is ready, the statistical analyses are performed. Finally, the results and conclusions are summarized and interpreted.

6.1 PARTS description

PARTS is an integrated healthcare system for psychiatrists in the Netherlands. With this information system, psychiatrists can manage patient information, treatments history, appointment planning, and medication prescriptions.

In the following sections, the project characteristics are discussed. Further detailed information can be found in [Fla]. The summary of the PARTS project is provided in Table 6.1.

6.1.1 Project environment

PARTS is a web service using java technology. The Apache Struts framework and a model-view-controller architecture [Dea] are used. The UML models were

Technology	Java
# staffs	25 people
Duration (in years)	2.3
Off-shored	India
Status	Finished
Model size	104 use cases 266 design classes 341 seq. diagrams 34 class diagrams
SLoC	152,017

Table 6.1: PARTS Project Summary

created using IBM Rational XDE. Furthermore, IBM Rational ClearCase and ClearQuest were adopted for code versioning and bug tracking respectively.

6.1.2 Developer experience

Not much information could be obtained on this topic. However, the system's architects were expected to have sufficient knowledge of UML and experience in creating design documentation with it due to the use of Rational Unified Process (RUP) [Kru]. This standardized development process also suggested that programmers at least had the required knowledge of UML to accurately read the designs.

6.1.3 Adopted development process

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

6.1.4 Working style

This project involved off shoring to India. The requirements and modeling of the system were created in the Netherlands, while approximately 60 percent of the implementation and testing activities were accomplished in India. When big problems emerged regarding incorrect design, a part of the system would be sent back to the architects in the Netherlands. When this part was updated, the original implementation path was again followed in India.

6.2 PARTS defect statistics

Defect data collection and preprocessing are mainly discussed in this section.

6.2.1 Data sampling

All findings registered as defects in ClearQuest were based on the latest version of the PARTS project repository. The statistics of the findings of the latest

version are as follows:

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

Only findings found during the **Test** phase were analyzed in this case study, which contains 1546 findings. Among these findings, 566 findings had modified source files attached to them. Some defects did not have attached modified source files because they were fixed by making changes to the database or application server only. Another reason for not having attached files is that some defects were solved indirectly, for instance, by solving another defect. Finally, defects were often rejected because they could not be reproduced. Since defect-source traceability is a prerequisite for the analyses, the 566 findings mentioned above were therefore chosen as the data set for further analyses. However, this number is still too large for defect typing analysis. Therefore, a random sampling was performed first. The sample size was initially set to 100, but was later increased to 164.

6.2.2 Descriptive statistics

After getting the 164 sample data points, each finding from the sample space was inspected and a defect type was assigned to it according to the defect taxonomy mentioned in Chapter 4. The sample's defect type distribution is shown in Figure 6.1. As shown in the figure, a relatively large amount of the findings belong to the category of UI-related defects (28 %), followed by user data I/O (17 %) and data handling (16 %). The rest of the defect types are equal to or lower than 10 percent of the sample size. Furthermore, a considerable number of findings fall into the non-defect category (27 %). Many of these non-defect findings are related to change requests. Finally, five percent of the analyzed findings are considered as undetermined.

Except for the UI-related, undetermined, and non-defect, findings assigned to the rest defect types were prepared for further analyses. Finally, 83 out of the 164 purified defect data were left for statistical analyses.

6.3 Determining UML LoD

In Chapter 4 (see Section 4.2), Metrics for measuring UML LoD are listed. However, LoD aggregates for calculating LoD are needed in the statistical analyses. Since two sets of metrics based on different types of UML diagrams were created, the functions for calculating UML LoD are introduced respectively.

Calculating the LoD of a class from class diagrams was performed by measuring the class diagram metrics at class level. Therefore, the LoD value of a class is based on information that is related to that particular class. The correspondence between an implementation class and its design class in the model

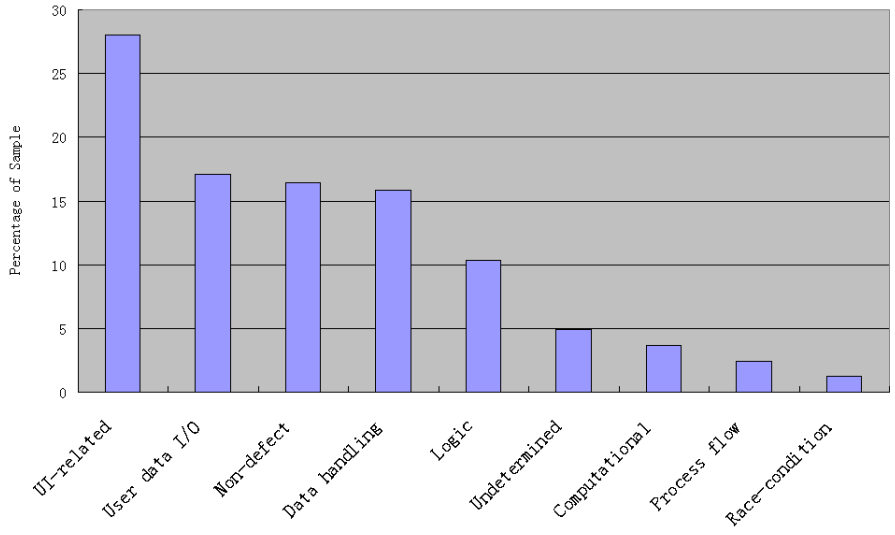

Figure 6.1: PARTS: Defect type distribution

is always a one-to-one relationship. For LoD aggregate based on class diagram metrics (LoD_{CD}), we simply sum up all metrics. The equation is shown below:

For implemetation class x, corresponding design class x':

$$LoD_{CD}(x) = CD_{m1}(x') + CD_{m2}(x') + ... + CD_{m7}(x') \tag{6.1}$$

Since an implementation class might appear as a class/instance in more than one sequence diagrams (one-to-many relationship), The LoD value of an implementation class based on sequence diagrams was calculated by taking into account all sequence diagrams where the correspondent design class of that particular class appears. Instead of calculating the accumulative value, the average LoD value of all sequence diagrams was used for the final LoD value of a given implementation class. Both LoD functions (LoD_{SD}) measured at diagram level and class-instance level used the same aggregate shown below:

For implementation class x, corresponding design class x' and n sequence diagrams related to x':

$$SequenceDiagramLoD = SD_{m1}(x') + SD_{m2}(x') + ... + SD_{m7}(x') \tag{6.2}$$

$$LoD_{SD}(x) = \frac{1}{n}\sum_{i=1}^{n} SequenceDiagramLoD_i(x') \tag{6.3}$$

6.4 Statistical analyses

After getting the purified data and the calculations of UML LoD, we are ready for the statistical analyses. In this section, the statistical analyses results are

Defect-count	# classes	Percentage
1	100	76.90
2	18	13.80
3	5	3.80
4	3	2.30
5	1	0.80
6	2	1.50
7	1	0.80
187	130	100.00

Table 6.2: Distribution of defects across faulty classes in the PARTS project

given. We start with comparing defect density between different modeled systems of PARTS to check whether the usage of UML models influences defect density. Later, the relationship between defect density and level of details in UML models is examined. Further, a closer look at the contribution of each individual metric to predicting defect density is performed. After this, which metric has a stronger correlation with a certain defect type is thoroughly discussed.

6.4.1 Descriptive statistics

Prior to the statistical analyses, a description of statistics is introduced first to present a statistical overview of this project. First of all, the distribution of defects across faulty classes is summarized in Table 6.2. As can be seen in the table, classes that were not modeled in UML models were also taken into account. A majority of the faulty classes only have one defect (76.90 percent), while the highest defect-count is seven.

As discussed previously, faulty classes are java classes that were corrected to solve defects. In total, 130 faulty classes were corrected to solve the 83 defects discussed before. The profile of these faulty classes with respect to their presence in the UML model is listed in Table 6.3. In general, faulty classes can be divided into two groups, classes that are modeled as design classes in UML models (*modeled*) and those not modeled at all (*unmodeled*). The modeled classes can be further divided into more detailed categories according to the way they are modeled in different types of diagrams. All this information was easily obtained from the analysis database.

6.4.2 Defect density comparison between different system parts

In this section, influence of the usage of UML models with respect to a project's defect density is analyzed. Faulty classes used in the analyses below are based on the information listed in Table 6.3.

First of all, defect density differences between system parts modeled as design classes in UML models and those not modeled at all were analyzed. Two boxplots that compare defect density of the modeled and unmodeled faulty classes are shown in Figure 6.2. As can be seen from the figure, faulty classes modeled

Faulty classes	# classes
modeled as design classes in UML models (*modeled*)	37
modeled in class diagrams (*modeledinCD*)	23
modeled but not referenced in any class diagram (*notmodeledinCD*)	14
modeled in sequence diagrams (*modeledinSD*)	30
modeled in class diagrams only (*modeledinCDonly*)	2
modeled in sequence diagrams only (*modeledinSDonly*)	9
modeled in both types of diagrams (*modeledinBoth*)	21
modeled but not referenced in any diagram (*modeledinNeither*)	5
not modeled in UML models at all (*unmodeled*)	93

Table 6.3: PARTS: The profile of faulty classes with respect to their presence in UML models

in UML models have a lower defect density than the unmodeled ones. This is illustrated by the horizontal bold lines in the grey boxes, which represent the median value of each group. The median values of the modeled and unmodeled classes are 4.9 and 15.15 respectively. After a careful check on the data, we were assured that the outliers and extreme values shown in the figure were not caused by errors in the data (e.g., caused by a wrong data entry), therefore they could not be excluded from the analysis.

Although a difference was found between defect density of modeled and unmodeled faulty classes, further analysis should be performed to see the significance of this difference. As was mentioned before about performing statistical analysis tests (see Section 5.5.3), a parametric test called the independent t-test was considered first. In order to perform this test, two conditions should be met: data is normally distributed; variances in the data are roughly equal (homogeneity of variances). If any of these two criterion is violated, we should then use the non-parametric test (i.e., Mann-Whitney test).

The test of normality was performed using **Shapiro-Wilk** for modeled faulty classes and **Kolmogorov-Smirnov** for unmodeded system parts (see Appendix C.1 for further information). However, neither defect density data of modeled system parts nor that of unmodeled parts was normally distributed; thus, the first criteria needed for an independent t-test was violated. In order to achieve a normal data distribution, an *area transformation* [KK77] was performed. By transforming the data, we expected to solve the data distribution problem and to reduce the effects of the outliers and extreme values. A normal data distribution was achieved after the transformation. The second criteria of homogeneity of variances was met by performing **Lavene's test** (see Ap-

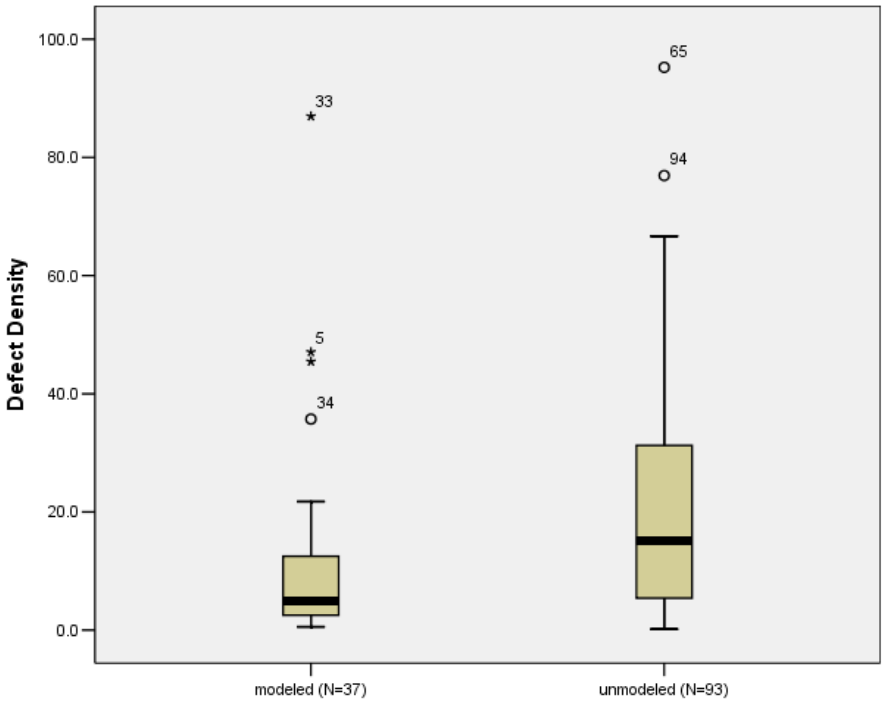

Figure 6.2: PARTS: Defect density (per KSLoC) of modeled and unmodeled system parts

	modeled/ unmodeled	N	Mean	Std. Deviation	Std. Error Mean
Normalized Defect Density	modeled	37	-0.456	0.996	0.164
	unmodeled	93	0.182	0.945	0.098

Table 6.4: PARTS: Group statistics of defect density between the modeled and unmodeled faulty classes

Project	t	df	Sig. (1-tailed)	mean Difference
PARTS *Equal variances assumed*	-3.419	128	0.001	-0.638

Table 6.5: PARTS: Independent t-test of defect density between the modeled and unmodeled faulty classes

pendix C.2). Since the two assumptions were now met, the independent t-test could be used. The results are shown in Table 6.5. The group statistics are given in Table 6.4 to present the mean values of the two normalized groups.

In Table 6.5, the most important point to note is in the significance column (Sig.). The test reveals a true significance when the significance value $p \leq 0.05$. For PARTS, this significant value is 0.001, which suggests that the mean difference of defect density between the modeled and unmodeled faulty classes is significant. Therefore, we can conclude that, on average, faulty classes that are modeled have a significantly lower defect density than those not modeled at all. the null (H_0) hypothesis mentioned before had to be rejected for this reason. This difference is significant at 0.01 level ($p \leq 0.01$), 1-tailed.

Another analysis was performed to compare the defect density of faulty classes based on the diagram types they were modeled in. As was shown before, faulty classes modeled as design classes in UML models could be splitted up into several groups: *modeledinCDonly*, *modeledinSDonly*, *modeledinBoth*, and *modeledinNeither*. In Figure 6.3, the defect density differences among all these data groups are illustrated. As can be seen from the graph, a big difference is found between faulty classes modeled in both types of diagrams (*modeledinBoth*) and the other three categories. Defect density is lower for *modeledinBoth* faulty classes. However, the differences among the other three groups are not obvious.

In order to see whether this difference is significant, the **One-way independent ANOVA** test was performed to compare mean values among multiple groups. The conditions under which ANOVA is reliable are the same as for the parametric test. The normalized data was obtained after an area transformation. However, Lavene's test could not be performed due to a too small amount of data points in the *modeledinCDonly* group. Since sample sizes between distinct groups were quite different and the population variances were not sure to be equal, the **Games-Howell** test of **Post hoc** procedures (see Appendix C.3) was used for exploring the data for any differences between means. The descriptive statistics are given in Table 6.6 and a big difference of mean values is between *modeledinBoth* and *modeledinNeither*, which are -0.395 and 0.688 respectively. Table 6.7 showed that the homogeneity of variance is met.

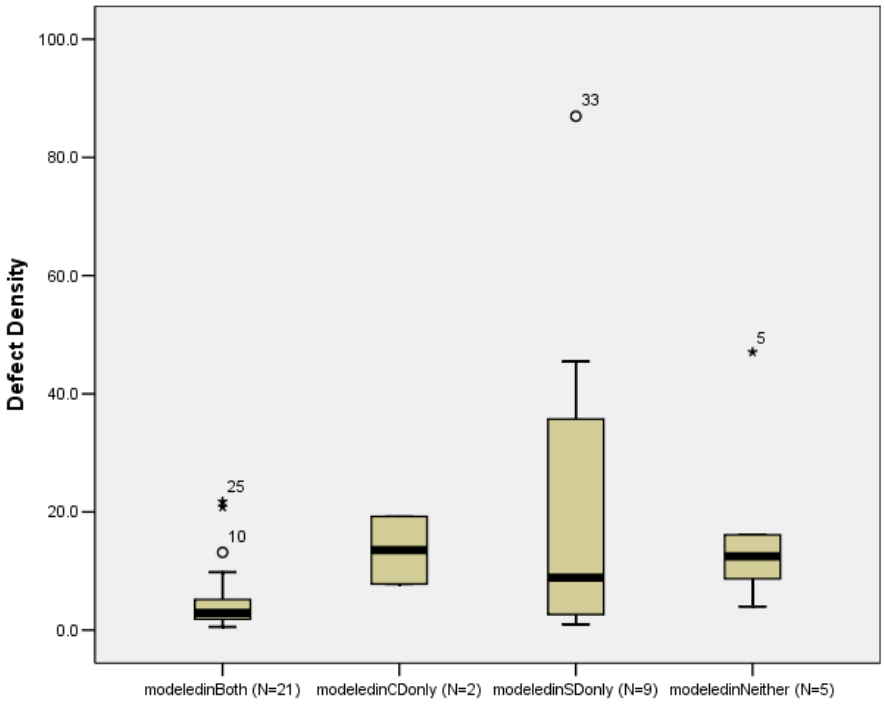

Figure 6.3: PARTS: Defect density (per KSLoC) of different modeled and unmodeled system parts

	N	Mean
modeledinBoth	21	-0.395
modeledinCDonly	2	0.603
modeledinSDonly	9	0.406
modeledinNeither	5	0.688

Table 6.6: PARTS: One-way ANOVA descriptive statistics

Levene Statistic	df1	df2	Sig.
0.915	3	33	0.445

Table 6.7: Test of Homogeneity of Variances among different modeled groups

The ANOVA test results can be found from Table 6.8 and there is a significant difference in defect density between groups ($p = 0.040$). However, no significant defect density difference is found between any pair of groups in Table 6.9. The biggest difference is between *modeledinBoth* and *modeledinNeither* groups.

As can be seen from the faulty classes' profile listed before (see Table 6.3), faulty classes that are modeled as design classes in UML models can be divided into two categories based on whether these classes are actually used in class diagrams or not. We were curious to see whether there was a defect density difference between faulty classes modeled in UML models but not referenced in any of the class diagrams (*notmodeledinCD*) and those which are modeled as design classes and were also used in class diagrams (*modeledinCD*).

From the boxplots shown in Figure 6.4, it can be seen that defect density of modeled faulty classes presented in class diagrams is lower than that of faulty classes modeled only as design classes but not referenced in any class diagrams. The median values of *modeledinCD* and *notmodeledinCD* groups are 2.996 and 10.449, respectively.

After checking the two conditions for parametric test, the data normality was met after data normalization and homogeneity of variances was also valid. Again, the independent t-test was performed and the results are shown in Table 6.10. From the significance value $p = 0.007$, we can conclude that defect density of modeled faulty classes that are presented in class diagrams is significantly lower than those only modeled in UML models but not used in class diagrams. The null (H_2) hypothesis mentioned in Section 1.3 was rejected.

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	7.853	3	2.618	3.098	0.040
Within Groups	27.882	33	0.845		

Table 6.8: One-way ANOVA test

(I)Group	(J)Group	Mean Difference (I-J)	Std.Error	Sig.
modeledinBoth	modeledinCDonly	-0.998	0.379	0.298
	modeledinSDonly	-0.801	0.429	0.292
	modeledinNeither	-1.083	0.364	0.075

Table 6.9: Post hoc test: Games-Howell procedure

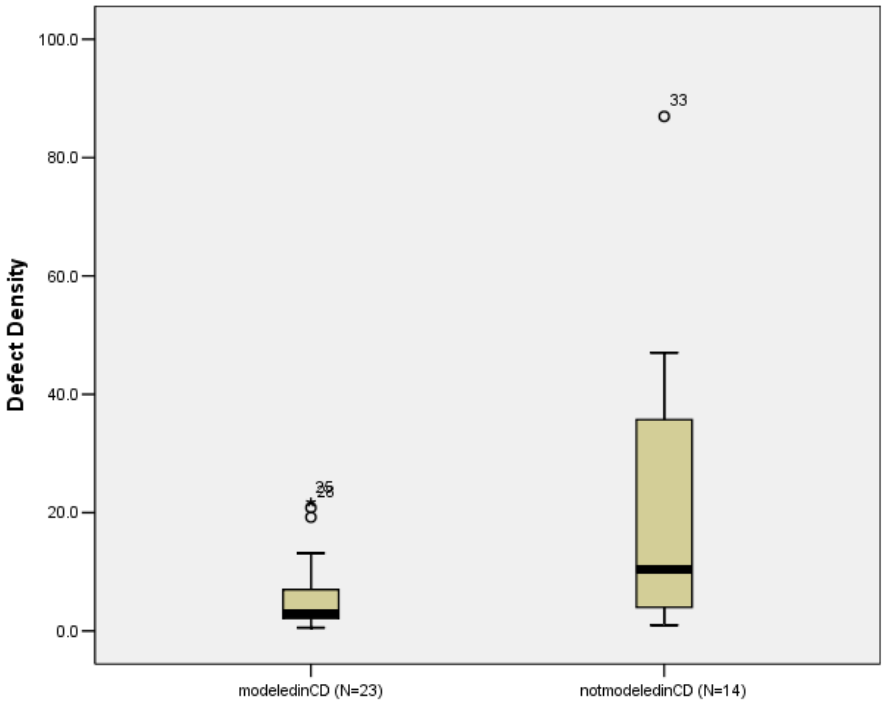

Figure 6.4: PARTS: Defect density (per KSLoC) of modeled faulty classes presented in CD or not

Project	t	df	Sig. (1-tailed)	mean Difference
PARTS *Equality of variances asumed*	-2.598	35	0.007	-0.815

Table 6.10: Independent t-test of Defect Density between Modeled Faulty Classes presented in CD or not

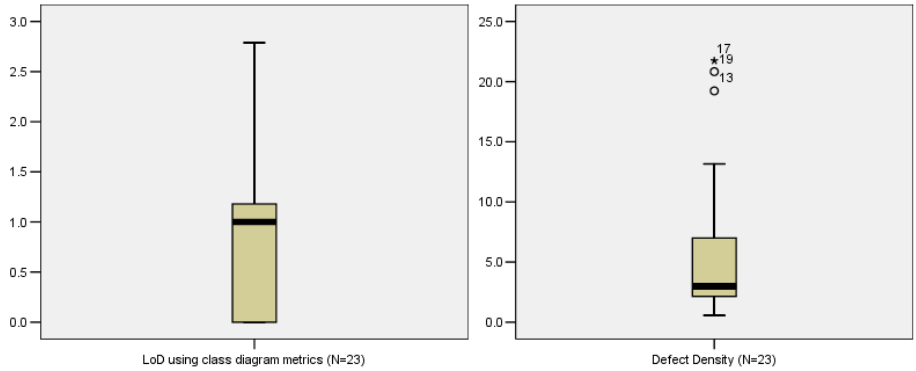

Figure 6.5: PARTS: Boxplots of class LoD (LoD_{CD}) and defect density

	Defect Density
Class LoD	0.173
Significance(2-tailed)	0.431

Table 6.11: Spearman's correlation coefficient between class LoD (LoD_{CD}) and defect density

6.4.3 Correlation Analyses between UML LoD and Defect Density

In this section, three correlation analyses are performed to investigate the relation between the level of detail in UML models of faulty classes and their defect density. As was mentioned before, a similar analysis was performed in [NFC08] and a significant negative correlation between UML LoD using sequence diagram metrics and defect density was found. Although no significant correlation was found in the same research, we still want to perform the correlation analysis between UML LoD using class diagram metrics and defect density. Later, LoD_{SD} measured at diagram level and class-instance level are examined respectively.

Using Class Diagram Metrics

This analysis tries to answer whether there is a significant correlation between LoD_{CD} and defect density. In total, LoD_{CD} values of 23 faulty classes were modeled in class diagrams and the descriptive statistics of the LoD_{CD} scores are shown in Figure 6.5. Since several outliers appeared in the defect density data, area transformation was performed to reduce the effects of the outliers and to achieve a normal data distribution. However, a normal distribution for the LoD_{CD} data still could not be achieved. Therefore, a non-parametric test was performed instead, called **Spearman's** correlation test (see Appendix C.4). The result of the analysis is shown in Table 6.11.

To our surprise, the correlation coefficient indicates a positive correlation between Class LoD using all the class diagrams metrics and defect density. However, this correlation is not statiscially significant. Therefore, there is no significant correlation between UML LoD using class diagram metrics and defect

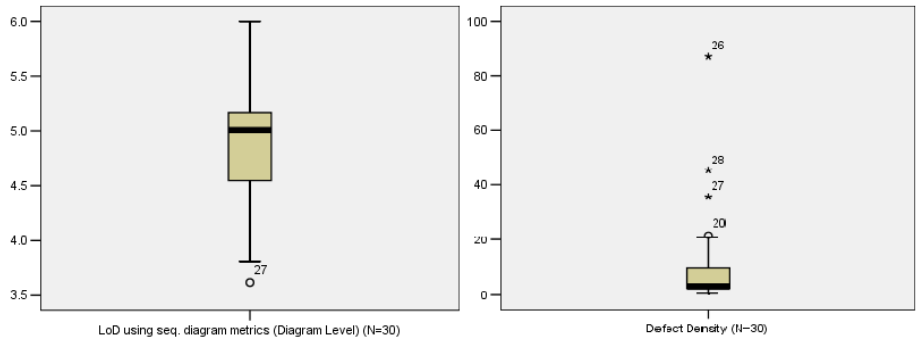

Figure 6.6: PARTS: Boxplots of class LoD (LoD_{SD} measured at diagram level) and defect density

	Defect Density
Class LoD	-0.459**
Significance (1-tailed)	0.005

**indicates siginificance at 0.01 level (1-tailed)

Table 6.12: Pearson's correlation coefficient of LoD_{SD} measured at diagram level and defect density

density.

Using Sequence Diagram Metrics

Two analyses were performed using sequence diagram metrics measured at diagram level and class-instance level, respectively. In total, 30 faulty classes were modeled in sequence diagrams. We first consider LoD_{SD} measured at diagram level and then LoD_{SD} measured at class-instance level.

LoD_{SD} measured at diagram level In this analysis, the level of details of the entire sequence diagram where a faulty class is modeled is considered as the LoD of that faulty class. The summary of the LoD scores and defect density of these 30 faulty classes is presented in Figure 6.6.

A parametric correlation test was performed on the normalized data, namely **Pearson's** correlation analysis (see Appendix C.4). Figure 6.7 illustrates the relation between the two variables after normalization. The result of the analysis can be found in Table 6.12.

From Table 6.12, it can be concluded that there is a negative (R = -0.459) and significant (p = 0.005) correlation between the two variables. Classes presented in sequence diagrams with a high LoD_{SD} tend to have a lower defect density than those used in sequence diagrams with a low LoD_{SD}. Furthermore, the *R Square* value, which measures the amount of variability of defect density that is accounted by LoD_{SD} measured at diagram level, equals 0.208. This means that LoD_{SD} measured at diagram level accounts for 20.8 percent of the variability of defect density in the implementation.

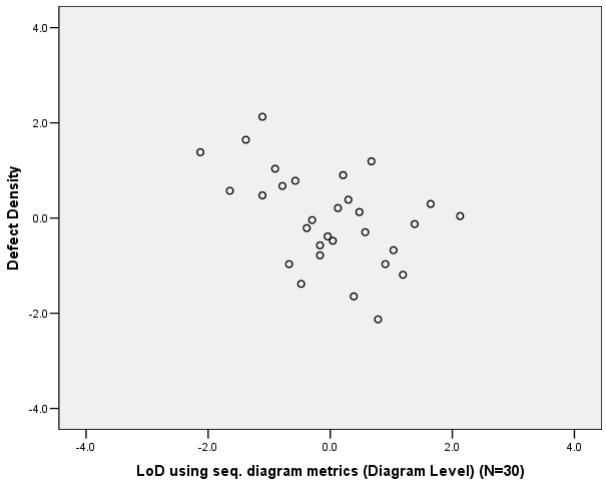

Figure 6.7: PARTS: Scatterplots of the relation between class LoD (LoD_{SD} measured at diagram level) and defect density

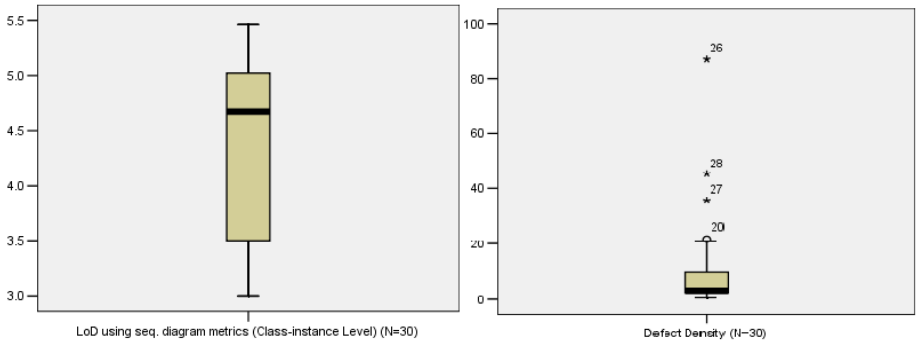

Figure 6.8: PARTS: Boxplots of class LoD (LoD_{SD} measured at class-instance level) and defect density

LoD_{SD} measured at class-instance level In this analysis, the LoD score of a faulty class is represented by the LoD of the correspondent design class in the sequence diagrams where that faulty class was modeled. Again, the summary of the LoD scores and defect density of the 30 faulty classes modeled in sequence diagrams is presented in Figure 6.8.

The results of performing the Pearson's correlation test on the normalized LoD_{SD} and defect density is shown in Table 6.13. Figure 6.9 illustrates the relation between the two variables after normalization.

Although the correlation is not as strong as that of using LoD_{SD} measured at diagram level, the correlation analysis again results in a negative and significant correlation between the two variables. The *R Square* value, which measures the amount of variability of defect density that is accounted by LoD_{SD} measured at class-instance level, equals 0.138. This means that LoD_{SD} measured at class-instance level accounts for 13.8 percent of the variability of defect density in the

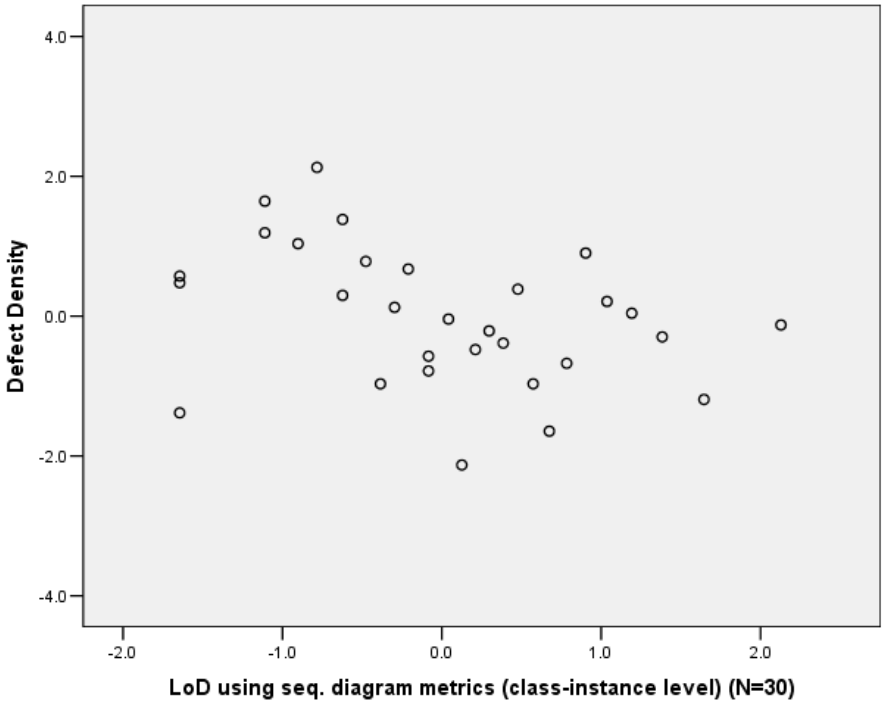

Figure 6.9: PARTS: Scatterplots of the relation between class LoD (LoD_{SD} measured at class-instance level) and defect density

implementation.

6.4.4 The contribution of individual metrics to predicting defect density

According to the correlation analyses mentioned above, a negative and significant correlation was found between the level of detail in sequence diagrams and defect density. The formulates for the LoD measures were generated by adding up all the metrics with equal weight factors. In this section, we focus on individual metrics' contribution to defect density prediction. Correlation analyses were performed to measure the correlation between each metric and defect density. At the same time, the *R Square* value, which measures the amount of variability

	Defect Density
Class LoD	-0.371*
Significance (1-tailed)	0.022

*indicates siginificance at 0.05 level (1-tailed)

Table 6.13: Pearson's correlation coefficient of LoD_{SD} measured at class-instance level and defect density

Test Method	Metric	Defect Density
Pearson's	MsgWithParamRatio	**-0.593****
	Significance (2-tailed)	**0.001**
	R Square	**0.352**
	NonDummyMsgRatio	-0.334
	Significance (2-talied)	0.072
	ReturnMsgWithLabelRatio	-0.332
	Significance (2-talied)	0.073
	MsgWithGuardRatio	-0.202
	Significance (2-talied)	0.284
Spearman's	NonAnoymObjRatio	0.230
	Significance (2-talied)	0.221
	NonDummyObjRatio	0.344
	Significance (2-talied)	0.063

**indicates siginificance at 0.01 level (2-tailed)

Table 6.14: Correlation analyses between individual metrics measured at diagram level and defect density

of defect density that is accounted by each metric, is examined. The idea behind this is to see which metrics are more suitable for predicting defect density. Since only metrics used for sequence diagram are worth further analysis, we started with analyzing sequence diagram metrics measured at diagram level, followed by metrics measured at class-instance level.

Metrics measured at diagram level After a closer look at individual metrics, the values of the *MsgWithLabelRatio* metric were always found constant. A couple of correlation analyses were performed between the rest of the metrics and defect density. Pearson's test was performed on normalized metrics, while Spearson's test was used on metrics which were not normally distributed even after normalization. The results are listed in Table 6.14.

An interesting finding is that the *MsgWithParamRatio* metric has a significant and negative correlation with defect density and this relation is even stronger than that of using LoD_{SD} aggregate (correlation coefficient equals -0.459). At the same time, the *R Square* value of this metric is larger than that of using LoD_{SD} aggregate (*R Square* equals 0.208). This result indicates that some metrics might have stronger predictive power for defect density than others. Therefore, these metrics should have a larger weight than the rest when making a LoD_{SD} aggregate, since the information offered by these metrics are more helpful in decreasing defect density.

Metrics measured at class-instance level In total, three metrics (*MsgWithLabelRatio*, *NonAnomyObjRatio* and *NonDummyObjRatio*) had constant values. Correlation tests were only performed on the remaining four metrics. According to the results shown in Table 6.15, *NonDummyMsgRatio* and *MsgWithGuardRatio* have a negative and significant correlation with defect density. Both of the relations are stronger than that of using LoD_{SD} aggregate (the correlation coefficient equals to -0.371). At the same time, the *R Square* values of

Test Method	Metric	Defect Density
Pearson's	MsgWithParamRatio	-0.303
	Significance (2-tailed)	0.103
Spearman's	NonDummyMsgRatio	**-0.466****
	Significance (2-tailed)	**0.010**
	R Square	**0.217**
	MsgWithGuardRatio	**-0.542****
	Significance (2-tailed)	**0.002**
	R Square	**0.294**
	ReturnMsgWithLabelRatio	-0.303
	Significance (2-tailed)	0.103

**indicates siginificance at 0.01 level (2-tailed)

Table 6.15: Correlation analyses of individual metrics measured at class-instance level and defect density

these two metrics are larger than that of using LoD_{SD} aggregate (which equals to 0.138).

6.4.5 The correlation between UML LoD metrics and defect density of individual defect type

In Chapter 4, a defect taxonomy with several typical defect types is listed. Since distinct defect types might have different characteristics, it is interesting to see which LoD metrics in UML modeling have stronger correlation with a certain defect type. In this section, defect types with a considerable amount of data points are analyzed. As the significant and negative correlation was found only between LoD_{SD} and defect density, sequence diagram metrics measured at diagram level and class-instance level are considered in the following analyses.

After checking the data points for each defect type, only defect types *logic* and *data handling* have enough data points for performing correlation analysis. The number of faulty classes related to these two defect types are 13 and 18, respectively.

The correlation analysis procedure is the same as discussed in the previous sections (see Section 6.4.3). The results are given in the following sections.

Logic Defect Type

In total, 13 faulty classes are related to logic defect type. The correlation analyses results are given in Table 6.16. As can be seen from the table, for sequence diagram metrics measured at diagram level, the two metrics *MsgWithParamRatio* and *MsgWithGuardRatio* have stronger negative correlation with defect density than the rest of the metrics. Additionally, *MsgWithParamRatio* even has a significant negative correlation. The four non-constant metrics measured at class-instance level are listed in the same table. Although none of them has a significant correlation with defect density, they all show a negative relationship.

Modeled level	Test Methods	Metrics	Defect Density
Diagram level	Pearson's	MsgWithParamRatio	**-0.840****
		Significance (2-tailed)	**0.000**
		MsgWithGuardRatio	-0.550
		Significance (2-tailed)	0.051
		ReturnMsgWithLabelRatio	-0.370
		Significance (2-tailed)	0.213
		NonDummyMsgRatio	0.308
		Significance (2-tailed)	0.306
	Spearman's	NonAnoymObjRatio	0.210
		Significance (2-tailed)	0.492
		NonDummyObjRatio	0.335
		Significance (2-tailed)	0.263
Class-instance level	Pearson's	ReturnMsgWithlabelRatio	-0.548
		Significance (2-tailed)	0.053
		MsgWithParamRatio	-0.459
		Significance (2-tailed)	0.115
		MsgWithGuardRatio	-0.430
		Significance (2-tailed)	0.142
		NonDummyMsgRatio	-0.287
		Significance (2-tailed)	0.342

**indicates siginificance at 0.01 level (2-tailed)

Table 6.16: Correlation analyses of LoD_{SD} metrics and defect density (*logic* defect type)

Modeled level	Test Methods	Metrics	Defect Density
Diagram level	Pearson's	NonDummyMsgRatio	**-0.529***
		Significance (2-tailed)	**0.024**
		MsgWithParamRatio	-0.439
		Significance (2-tailed)	0.068
		ReturnMsgWithLabelRatio	0.021
		Significance (2-tailed)	0.934
		MsgWithGuardRatio	0.157
		Significance (2-tailed)	0.533
	Spearman's	NonAnoymObjRatio	-0.194
		Significance (2-tailed)	0.440
		NonDummyObjRatio	-0.023
		Significance (2-tailed)	0.927
Class-instance level	Pearson's	ReturnMsgWithlabelRatio	-0.359
		Significance (2-tailed)	0.143
		NonDummyMsgRatio	-0.286
		Significance (2-tailed)	0.249
		MsgWithParamRatio	-0.273
		Significance (2-tailed)	0.273
		MsgWithGuardRatio	-0.215
		Significance (2-tailed)	0.393

*indicates siginificance at 0.05 level (2-tailed)

Table 6.17: Correlation coefficient of LoD_{SD} metrics and defect density (*Data handling* defect type)

Data Handling Defect Type

There are 18 faulty classes related to the *data handling* defect type. Table 6.17 shows the correlation analyses results for this defect type.

As can be seen from Table 6.17, for sequence diagram metrics measured at diagram level, metric *NonDummyMsgRatio* has a significant negative correlation with defect density. However, metrics *ReturnMsgWithLabel* and *MsgWithGuardRatio* based on diagram level are tested to have positive correlation with defect density. Since this positive correlation is not significant, it might be just a random errors or coincidence. The four non-constant metrics measured at class-instance level indicate a negative relationship, although none of them has a significant correlation with defect density.

6.5 Results and Conclusions

In this section, the main findings discovered in this case study are listed and possible interpretations are given. The implications of the results are also discussed.

The first finding is the influence of using UML models on defect density. As noted earlier, classes modeled as design classes in UML models were considered modeled, and otherwise not modeled. The result indicated that faulty

classes that were modeled, on average, had a lower defect density than those not modeled at all. This difference of defect density was statistically significant. This conclusion confirms that UML models have a positive influence on reducing defect density in the implementation. Well designed UML models can be good guidance in the implementation phase. Later, the modeled classes are splitted up into more detailed categories based on how they are modeled. Although a big defect density difference is found between classes modeled in both types of diagrams (*modeledinBoth*) and those modeled but not used in any diagrams (*modeledinNeither*), this differece is not statistically significant. However, we still believe that modeling classes in diverse UML diagrams offers developers clearer and better understandings of the design with different views. We also made a distinction between modeled classes presented in UML diagrams and those just modeled as design classes but not presented in any of the class diagrams. The defect density is significantly different and lower for modeled classes presented in UML diagrams. This finding also indicates the importance of UML diagrams in preventing defects in the implementation. Although some implemented classes were modeled as design classes, they are more helpful if presented in UML diagrams which actually illustrate how the classes should be implemented and how they relate to other classes.

The second finding is the correlation between UML LoD and defect density. The result in this case study is consistent with the one mentioned in [NFC08]. Classes that are modeled in a higher level of detail are inclined to have lower defect density. This finding indicates that software quality probably benefits from higher detailed UML modeling: more information in the UML models probably helps decrease misinterpretations of models among developers. Therefore, they are more instructive during the implementation. However, this conclusion was only confirmed by LoD_{SD} using sequence diagram metrics (see Section 6.4.3). The correlation is stronger between UML LoD using sequence diagram metrics measured at diagram level and defect density. One interpretation is that in this particular case three out of seven metrics measured at class-instance level have constant values for all modeled faulty classes. We believe this will decrease the predictive power of UML LoD_{SD}.

After finding the correlation between UML LoD_{SD} and defect density, we would like to know how much each LoD_{SD} metric accounts for the variability of defect density. Different metrics were listed based on at which level they were measured, either diagram level or class-instance level. The results indicated that some metrics (i.e., *MsgWithParamRatio*, *NonDummyMsgRatio*, and *MsgWithGuardRatio*) had stronger correlation with defect density and also had more contribution to predicting defect density. It means that the information provided by these metrics are more helpful in reducing defect density in the implementation phase.

Later, we wanted to have a look at the correlations between individual metrics and defect density of a certain defect type. However, due to the lack of enough data points, only *logic* and *data handling* defect types were considered in the analyses. For the *logic* defect type, metrics *MsgWithParamRatio* and *MsgWithGuardRatio* measured at diagram level had stronger correlation with defect density than the other metrics. This is not hard to understand because we believe these two metrics are important for demonstrating the logic performance inside a class. Hence, lacking of information of these two metrics will probably lead to logic problems. For the *data handling* defect type, the *Non-*

DummyMsgRatio metric was found to have a significant negative correlation with defect density. After checking the data set carefully, one data handling related defect type, called *data-access*, took a relatively larger percentage of the defects belonging to data handling defect types (38.9 percent). This defect type is mostly related to data handling from/to a data store, normally this part of the system is not designed into detail and dummy messages are used without being modeled. This might explain why the *NonDummyMsgRatio* metric has a significant correlation with the *data handling* defect type.

Chapter 7

Case study 2: RACE

In this chapter, a new case study is performed within the RACE project. This project is relatively much smaller than the PARTS project. However, we still believe it worth an investigation since it is still an empirical industrial product. The structure of this chapter is quite similar to that of the first case. The description of the project is given first, immediately followed by data collection and preprocessing. The statistical analyses are performed on the purified data after that. Finally, the results and conclusions are discussed.

7.1 RACE description

RACE is a risk management system developed for a financial organization in the Netherlands. In the following paragraphs, the project characteristics are discussed in detail. The summary of the RACE project can be found in Table 7.1.

7.1.1 Project environment

Race was built as a web server in Java language. The application frontend was done by the Apache Struts framework where a model-view-controller architecture was adopted. Swing was applied to build up stand alone application. The

Technology	Java
# staffs	10 people
Duration (in years)	1
Off-shored	India
Status	Finished
Model size	9 use cases 44 design classes 22 seq. diagrams 12 class diagrams
SLoC	125,168

Table 7.1: RACE Project Summary

oracle database was relatively easy and there was no coding inside the database itself. IBM Rational XDE was used to create the UML models. Other IBM Rational tools such as IBM Rational ClearCase and ClearQuest were used for code versioning and bug tracking.

7.1.2 Developer experience

Three architects were involved in this project and we believe that they had enough knowledge of UML and the ability of creating design documentation using UML. In particular, they had lots of experience using class diagram and sequence diagram which were mainly used in this project. Most developers had sufficient experience and knowledge in reading and understanding the UML models. Although some junior programmers might need some extra explanation of the details of UML models, they were able to understand the design correctly during the implementation.

The reporting part of the system was off sourced to India. Since no UML models were designed for this part of the system in advance and the code quality was horrible, we assume that developers there might have some difficulty in creating UML design models and implementing the code.

7.1.3 Adopted development process

The requirements document was organized in a waterfall process and was developed for about 3 years and full of great detail. On the other hand, the development process was developed iteratively using Rational Unified Process (RUP).

7.1.4 Working style

The requirements and the main UML design of the system were created in the Netherlands. At the same time, a majority of the system was implemented in the Netherlands. Only one part of the system, the reporting part, was completely off-sourced to India. This part was originally expected to be completed independently by the India department, including the UML design, code implementation, code review and testing. Actually, no UML design documentation was made for this part and code was implemented according to the related chapters in the requirements document.

Due to the culture difference, negotiations between the Netherlands and India were quite bad and few feedbacks were given to the Netherlands. Errors in the requirements were simply implemented in India even they were quite obvious. Lots of review and code rewriting were done back in the Netherlands.

7.2 RACE defect statistics

Defect data collection and preprocessing processes performed in this project are mainly discussed in this section. Since RACE is a rather small project, we decided to use all the findings registered in ClearQuest based on the latest version of the repository. In total, 109 findings were found in ClearQuest repository and 59 out of them were traceable back to the modified source files. Hence, these 59 target findings were left for the further analysis.

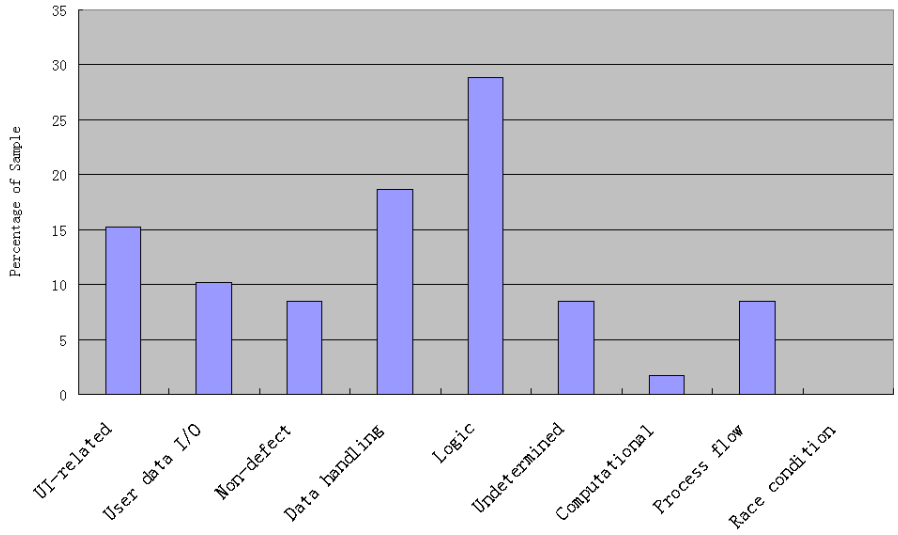

Figure 7.1: RACE: defect type distribution

7.2.1 Descriptive statistics

Prior to statistical analysis, the same defect typing procedure was performed on each target finding according to the defect taxonomy mentioned in Chapter 4. The defect type distribution is shown in Figure 7.1. As shown in the Figure, the *logic* defect type accounts for nearly 29 percent of the whole dataset, followed by the *data handling* (18 percent) and the *user data I/O* (merely 15 percent). The rest defect types are equal to or lower than 10 percent. A few defects fall into the *non-defect* and the *undetermined* defect type, both of which account for 8 percent of the dataset.

After excluding the irrelative defect types, which referred to the UI-related, undetermined, and non-defect, 40 defects were left for further analyses.

7.3 Determining UML LoD

Since the generalization of the conclusions drawn from PARTS is also an important motivation for performing this study, we currently still use the LoD aggregates created and used in the PARTS project (see Section 6.3). The idea behind this is to keep the other factors as similar as possible while comparing the analysis results between the two projects.

7.4 Statistical analyses

In this section, the statistical analyses results are given. The same analysis procedure as mentioned in the first case study is performed. However, some analyses are unable to perform due to not having enough data points.

Defect-count	# classes	Percentage
1	70	87.50
2	7	8.75
3	2	2.50
5	1	1.25
95	80	100.00

Table 7.2: Distribution of defects across faulty classes in the RACE project

Faulty classes	# classes
modeled as design classes in UML models (*modeled*)	11
modeled in class diagrams (*modeledinCD*)	2
modeled but not referenced in any class diagram (*notmodeledinCD*)	3
modeled in sequence diagrams (*modeledinSD*)	8
modeled in class diagrams only (*modeledinCDonly*)	1
modeled in sequence diagrams only (*modeledinSDonly*)	7
modeled in both types of diagrams (*modeledinBoth*)	1
modeled but not referenced in any diagram (*modeledinNeither*)	2
not modeled in UML models at all (*unmodeled*)	69

Table 7.3: RACE: The profile of faulty classes with respect to their presence in UML models

7.4.1 Descriptive statistics

A description of statistics is introduced to present a statistical overview of this project prior to statistical analyses. The distribution of defects across the faulty classes is shown in Table 7.2. As can be seen from the table, most faulty classes only have one defect (87.50 percent), while the highest defect-count is five.

In total, 80 faulty classes were used to correct the 40 target defects discussed before. The profile of these faulty classes with respect to their presence in UML models is listed in Table 7.3.

7.4.2 Defect density comparison between different system parts

In PARTS, we found out that the usage of UML models actually influenced the defect density of a project. Some important conclusions were drawn from the analyses (see Section 6.4.2). The same analyses were intended to perform on the

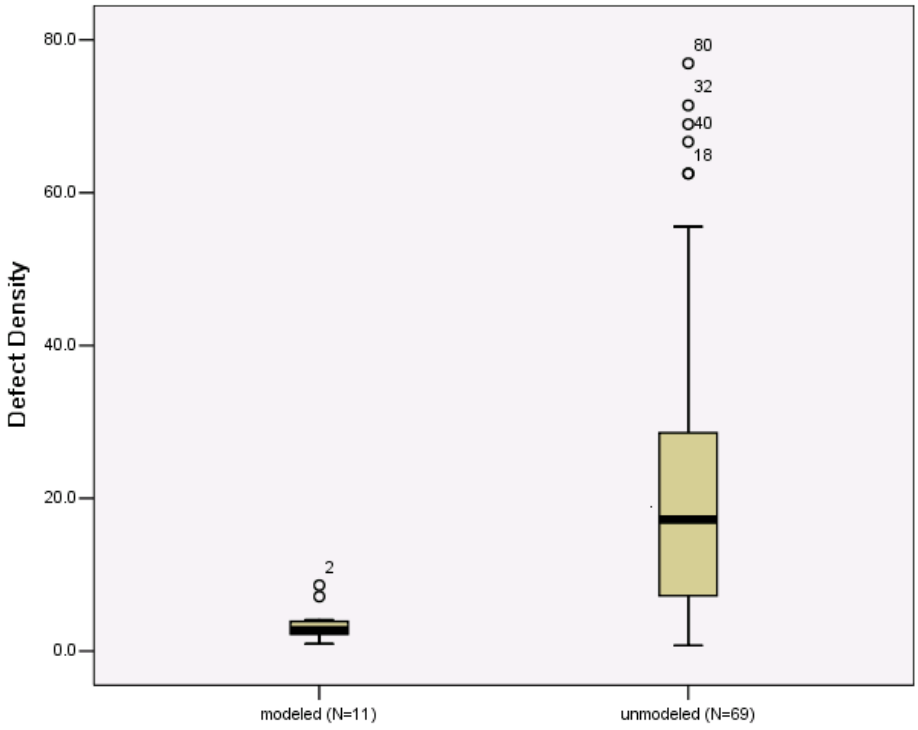

Figure 7.2: RACE: Defect density (per KSLoC) of modeled and unmodeled system parts

data set of this project.

The first conclusion drawn from the PARTS project is that faulty classes modeled using UML have a significantly lower defect density than those not modeled at all. In this analysis, the same hypothesis is tested. Figure 7.2 shows two boxplots that compare defect density of the modeled and unmodeled faulty classes. It can be seen that faulty classes modeled in UML models have a lower defect density than the unmodeled ones. The median values of the modeled and unmodeled classes are 2.71 and 17.24 respectively. The outliers and extreme values in the graph could not be excluded from the analysis because they are not caused by errors in the data.

Although there is a big difference between defect density average values of the two variables, the significance of this difference is not confirmed yet. Since defect density data of unmodeled system parts was not normally distributed, an area transformation was applied to both the modeled and unmodeled defect density data sets so that their relative differences were maintained. The independent t-test was performed on the normalized data and the results are shown in Table 7.5. However, Lavene's test (Sig. = 0.041) indicates that equality of variances could not be assumed, thus the results are given under the condition that equal variances are not assumed. The group statistics are first given in Table 7.4 to present the mean values of the two normalized groups.

As can be seen from Table 7.5, the significant value is $p = 0.000$, which suggests that the mean difference of defect density between the modeled and

	modeled/ unmodeled	N	Mean	Std. Deviation	Std. Error Mean
Normalized Defect Density	modeled	11	-1.097	0.469	0.141
	unmodeled	69	0.176	0.944	0.114

Table 7.4: RACE: Group statistics of defect density between the modeled and unmodeled faulty classes

Project	t	df	Sig. (1-tailed)	mean Difference
RACE *Equal variances not assumed*	-7.022	25.51	0.000	-1.274

Table 7.5: Independent t-test of defect density between the modeled and unmodeled faulty classes

unmodeled faulty classes is significant. Therefore, on average, faulty classes that were modeled have a significantly lower defect density than those not modeled at all. The consistent result is obtained in this case study, which supports our believe in achieving higher software quality by using UML models. This difference is significant at 0.01 level ($p \leq 0.01$), 1-tailed.

Unfortunately, the comparison of defect density of faulty classes based on the diagram types they were modeled in could not be performed due to too few data points (see Section 7.4.2). The same to analysis for testing defect density difference between faulty classes modeled in UML models but not referenced in any class diagram (*notmodeledinCD*) and those used in class diagrams (*modeledinCD*).

7.4.3 Correlation Analyses between UML LoD and Defect Density

In this section, we continue with the correlation analyses between the level of detail in UML models of faulty classes and their defect density. As was mentioned in the PARTS project, an important conclusion was that there is a negative and significant correlation between UML LoD using sequence diagram metrics and defect density. The same hypothesis was used in this study case (see Section 1.3). The correlation analysis between LoD_{CD} and defect density could not be performed due to a too small amount of data points. Therefore, only LoD_{SD} measured at diagram level and class-instance level were analyzed.

Using sequence diagram metrics

In total, 8 faulty classes were modeled in sequence diagrams. We first consider LoD_{SD} measured at diagram level and then LoD_{SD} measured at class-instance level.

LoD_{SD} measured at diagram level The summary of the LoD scores measured at diagram level and defect density of these 8 faulty classes is presented in Figure 7.3.

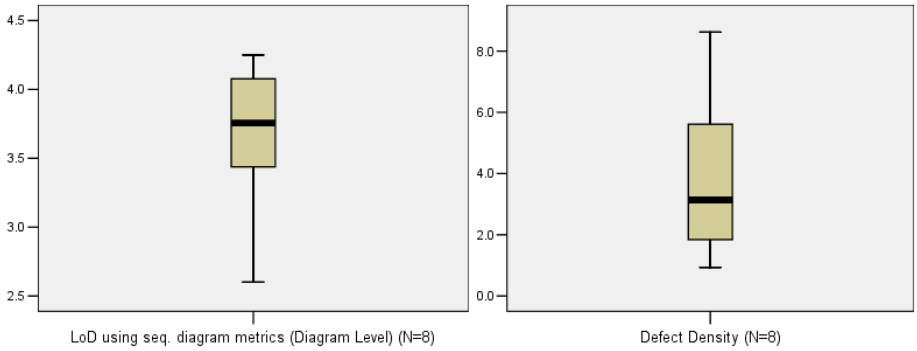

Figure 7.3: RACE: Boxplots of class LoD (LoD_{SD} measured at diagram level) and defect density

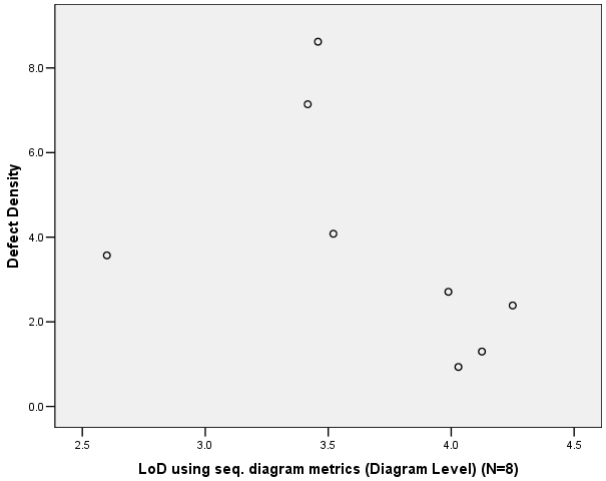

Figure 7.4: RACE: Scatterplots of the relation between class LoD (LoD_{SD} measured at diagram level) and defect density

Since both LoD_{SD} and defect density data points were normally distributed, Pearson's correlation analysis was applied. Figure 7.4 illustrates the relation between the two variables. The result of the analysis can be found in Table 7.6.

As can be seen from Table 7.6, there is no significant correlation between the two variables, however, a negative (R = -0.483) relation is confirmed. Later, Spearman's test was applied to the same data set and it revealed a significant correlation between LoD_{SD} and defect density (see Table 7.7). This correlation is even significant at 0.05 level ($p \leq 0.05$), 1-tailed. After checking the data samples and Figure 7.4, one data point (defect density equals 3.57 and LoD equals 2.6) is found to have a relatively lower LoD and defect density than the rest of the samples. We did the Pearson's test without this data point and a significant correlation between LoD_{SD} measured at diagram level and defect density is indicated (p = 0.012). Therefore, we assume that the appearance of this data point conflicts with the linear assumption of Pearson test which

	Defect Density
Class LoD	-0.483
Significance (1-tailed)	0.112

Table 7.6: Pearson's correlation coefficient of LoD_{SD} measured at diagram level and defect density

	Defect Density
Class LoD	-0.738*
Significance (1-tailed)	0.018

Table 7.7: Spearman's correlation coefficient of LoD_{SD} measured at diagram level and defect density

indicates the strength and direction of a linear relationship. At the same time, fewer data points might make the outliers' influence on the result much stronger than large amount of data samples. These two reasons can probably explain why Pearson's test could not reveal a significant correlation before. However, we could not find any reason to exclude this data point.

LoD_{SD} based on class-instance level The summary of the LoD scores measured at class-instance level and defect density of these faulty classes is presented in Figure 7.5.

Since both LoD_{SD} measured at class-instance level and defect density data sets were normally distributed, Pearson's correlation test was performed again. The result is shown in Table 7.8. Figure 7.6 illustrates the relation between the two variables.

Although the correlation is not statistically significant, it shows a negative correlation which is stronger than that of using LoD_{SD} measured at diagram level ($p = 0.112$). From Figure 7.6, we also suspected that the same data point mentioned before with relatively lower level of detail and defect density might influence the result of Pearson's test. After excluding that data point, a signifi-

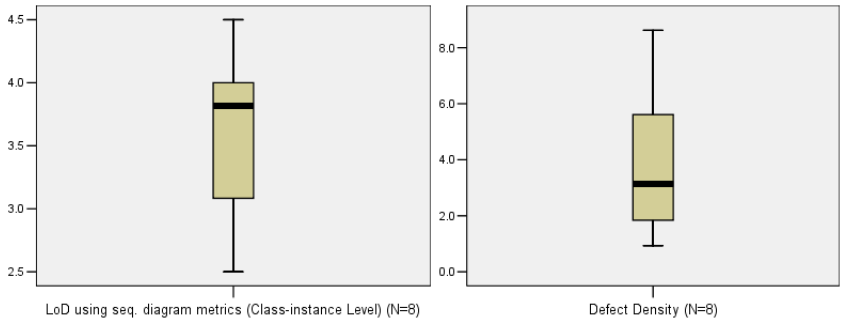

Figure 7.5: RACE: Boxplots of class LoD (LoD_{SD} measured at class-instance level) and defect density

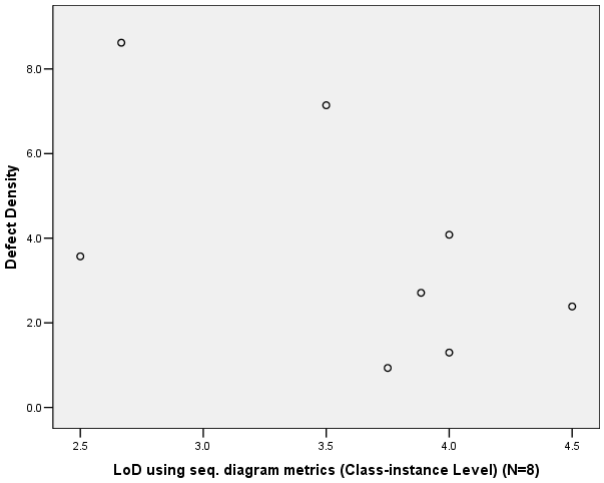

Figure 7.6: RACE: Scatterplots of the relation between class LoD (LoD_{SD} measured at class-instance level) and defect density

	Defect Density
Class LoD	-0.568
Significance (1-tailed)	0.071

Table 7.8: Pearson's correlation coefficient of LoD_{SD} measured at class-instance level and defect density

Test Method	Metric	Defect Density
Pearson's	NonDummyMsgRatio	-0.043
	Significance (2-tailed)	0.919
	MsgWithParamRatio	-0.134
	Significance (2-tailed)	0.751
	NonDummyObjRatio	-0.470
	Significance (2-tailed)	0.240
Spearman's	NonAnonymObjRatio	-0.055
	Significance (2-tailed)	0.898
	MsgWithGuardRatio	-0.546
	Significance (2-tailed)	0.162

Table 7.9: correlation analyses between individual metrics and defect density

cant correlation between the two variables is found ($p = 0.019$). This observation also indicates that we need far more data points to reduce the influence of the outliers.

7.4.4 The contribution of individual metrics to predicting defect density

Although a significant correlation between level of detail in UML models using sequence diagram metrics and defect density was not found, a negative relationship was indicated. Therefore, individual metrics' contribution to defect density prediction are worth an investigation in this section. Correlation analysis was performed to examine the correlation between each metric and defect density. The *R Square* value is calculated to measure the amount of variability of defect density that is accounted by metrics which indicate a significant correlation with defect density. Since only metrics used for sequence diagram worth a further analysis, we started with analyzing sequence diagram metrics measured at diagram level, followed by metrics measured at class-instance level.

LoD_{SD} metrics measured at diagram level

After having a closer look at each individual metric, metrics *MsgWithLabelRatio* and *ReturnMsgWithLabelRatio* were found to be constant. A couple of correlation analyses were performed between the remaining metrics and defect density. The results are listed in Table 7.9.

It can be seen from Table 7.9 that none of the rest metrics shows a significant correlation, however, they all indicate a negative correlation with defect density. *MsgWithGuardRatio* metric has a even stronger correlation with defect density than LoD_{SD} aggregate which has correlation coefficient as -0.483.

LoD_{SD} metrics measured at class-instance level

For LoD_{SD} metrics measured at class-instance level, three metrics *MsgWithLabelRatio*, *NonAnoymObjRatio* and *ReturnMsgWithLabelRatio* had constant values. The results are shown in Table 7.10 below.

Test Method	Metric	Defect Density
Pearson's	NonDummyMsgRatio	-0.572
	Significance (2-tailed)	0.193
	MsgWithParamRatio	0.464
	Significance (2-tailed)	0.247
SPearman's	MsgWithGuardRatio	-0.577
	Significance (2-tailed)	0.134
	NonDummyObjRatio	-0.454
	Significance (2-tailed)	0.259

Table 7.10: Correlation analyses of individual LoD_{SD} metrics measured at class-instance level and defect density

According to the results shown in Table 7.10, the *NonDummyMsgRatio* and *MsgWithGuardRatio* metrics have a stronger correlation with defect density than LoD_{SD} aggregate (correlation coefficient equals -0.568). However, to our surprise, the *MsgWithParamRatio* metric indicates a positive correlation with defect density. After checking the metric values of *MsgWithParamRatio*, one data point with the highest level of detail has the highest defect density. This influenced the relation between the two variables dramatically. Therefore, we believe lacking of enough data points is a big threat to the validation of the results.

7.4.5 The Correlation between UML LoD Metrics and Defect Density of Individual Defect Type

Since in total only 8 faulty classes are related to sequence diagrams, data points for each defect type are even smaller. Therefore, the correlation between UML LoD metrics and defect density of individual defect type can not be addressed in this case study.

7.5 Results and Conclusions

In this section, we list the findings discovered in this case study. However, some conclusions drawn from PARTS project are not confirmed in this case study, the possible interpretations are given in the following paragraphs.

First of all, the consistent conclusion is obtained from the analysis of UML models' influence on defect density. The result confirms that faulty classes that were modeled, on average, have a lower defect density than those not modeled at all. This difference of defect density is statistically significant. However, due to too few data points, it is impossible to perform the rest of the two analyses related to this research question.

Later, we examined the correlation between UML LoD and defect density. The correlation analysis between LoD_{CD} and defect density was not able to perform due to lacking of enough data samples. For LoD_{SD} measured at both diagram level and class-instance level, a negative relation between LoD_{SD} and defect density was obtained. However, neither of the two correlations was statistically significant. As mentioned earlier, two reasons might explain this situation.

Not having enough data points could be the first reason since we only have 8 data points which are far less than that used in the PARTS project. Another reason is that the result can be easily influenced by outliers (see Section 7.4.3). Therefore, we are still confident with the conclusion drawn from PARTS – there is a significant negative correlation between higher level of details in UML models and lower defect density.

Further, if possible, we would like to know how much each LoD_{SD} metric accounts for the variability of defect density. The correlation between individual metrics and defect density was analyzed. Although none of the metrics has a significant correlation with defect density, some information might still be useful in the later phase. For sequence diagram metrics measured at diagram level, the *MsgWithGuardRatio* metric has a stronger relation with defect density than LoD_{SD} using all the metrics. On the other hand, the *NonDummyMsgRatio* and *MsgWithGuardRatio* metrics from sequence diagram metrics measured at class-instance have a stronger correlation with defect density than LoD_{SD} aggregate. It is interesting to see that these two metrics, *NonDummyMsgRatio* and *MsgWithGuardRatio*, were also indicated to have stronger correlation with defect density in the PARTS project.

In conclusion, some results found in PARTS project are not able to perform or not statistically significant in this case study. We believe that lacking of enough data points is the main threat. Furthermore, the results can be easily influenced by one or two outliers could be another interpretation. Finally, the performance of the analysis methods used in this study case was not that robust because of the small amount of data samples. These are the reasons why a third case study is needed.

Chapter 8

Case study 3: BEHEERNET

The third case study is performed in the BEHEERNET project which is also an industrial project from the real world. The structure of this chapter is the same as the previous case studies. The project description and characteristics are introduced first, followed by data collection and preprocessing. Later, the statistical analyses are performed when the purified data set is ready. In the end, the results and conclusions about this project are summarized.

8.1 BEHEERNET description

BEHEERNET is developed as a Web service about pension marketing for insurance companies. Pension Fund organization can have an overview of the companies' and their employees' pension statuses. The employers are able to change the pension detail of their employees and store the pension document of their employees by using a document management system. The intention of this project is built to be accessible by many other systems. The project itself is pretty big and involves several development stages.

In the following paragraphs, the project characteristics are discussed into detail. Table 8.1 provides the summary of the BEHEERNET project.

Technology	Java
# staffs	18 people
Duration (in years)	2
Off-shored	India
Status	Finished
Model size	20-25 use cases 137 design classes 115 seq. diagrams 28 class diagrams
SLoC	135,454

Table 8.1: BEHEERNET Project Summary

8.1.1 Project environment

BEHEERNET is built as a web service using Java technology. The method of design used for the visualization of the blueprints of the system is based on the "4+1 View Model of Architecture", which is based on [Kru95] and standard within the Rational Unified Process. The UML models were created using IBM Rational XDE. IBM Rational ClearCase and ClearQuest were adopted for code versioning and bug tracking respectively.

8.1.2 Developer experience

Several architects have worked on this project. However, different architects were responsible for different development stages (i.e., different versions of the project). On average, one architect was involved in the project all the time. These architects have different opinions in UML modeling and their modeling styles are different too. The leading architect has the sufficient knowledge in designing UML models. Developers' experience in using UML models were quite little, especially the developers in India. They consistently had difficulties in understanding the UML models and asked for more detailed architectural designs.

8.1.3 Adopted development process

In this project, waterfall development approach was adopted. No iterative testing but only one big release was performed at the end of the project.

8.1.4 Working style

The whole development team was divided into three groups. Two were in the Netherlands but located in different cites. Another one big group was in India. The development and testing processes were mainly done in India. Due to the culture difference, the negotiation between the Netherlands and India was not good. Lots of time were spent on explaining the architectures by phone or online chatting. Misunderstanding about the UML models in India was a big problem during the development.

8.2 BEHEERNET defect statistics

Defect data collection and preprocessing processes are mainly discussed in this section.

8.2.1 Data sampling

All the findings registered as defects in ClearQuest were based on the latest version of BEHEERNET project repository. In total, 4061 findings were recorded in ClearQuest, among which 1784 findings had modified source files traceable back to the source code. Therefore, they were chosen as the data set for further analyses. However, this number is still too large for defect typing analysis. We first analyzed defects which have at least one modified implementation class

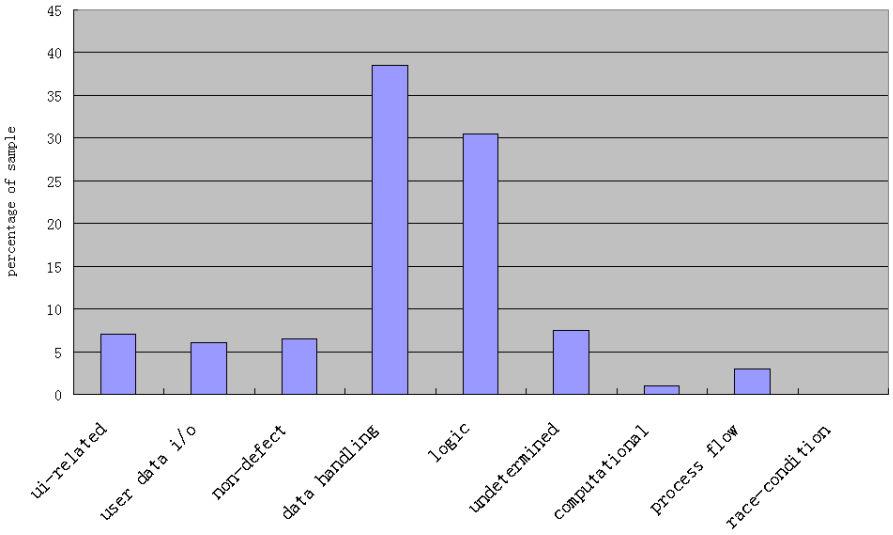

Figure 8.1: BEHEERNET: Defect type distribution

modeled in UML models. Later, a random sampling was performed to enrich the sample size to 200 data points.

8.2.2 Descriptive statistics

After getting the 200 sample data points, each finding from the sample space was inspected and assigned with a defect type according to the defect taxonomy mentioned in Chapter 4. The sample's defect type distribution is shown in Figure 8.1. As shown in the Figure, two defect types, *data handling* (38.5 %) and *logic* (30.5 %), have a relatively larger amount of the findings. The rest defect types are lower than 10 percent of the sample size. Besides, a few findings fall into *non-defect* category (6.5 %). Finally, around seven percent of the analyzed findings are considered as *undetermined*.

Except for the UI-related, undetermined, and non-defect, findings assigned to the rest defect types were prepared for further analyses. Finally, 158 out of 200 purified defect data were left for statistical analyses.

8.3 Determining UML LoD

In order to generate comparable analysis results, the same LoD aggregates (see Section 6.3) used in the previous two case studies were also adopted in this case.

8.4 Statistical analyses

After getting purified data and the calculations of UML LoD, we are ready for the statistical analyses. We start with comparing defect density between different modeled systems of BEHEERNET to check whether the usage of UML

Defect-count	# classes	Percentage
1	117	52.71
2	40	18.02
3	22	9.91
4	11	4.95
5	13	5.86
6	9	4.05
8	2	0.90
9	2	0.90
10	2	0.90
11	1	0.45
16	1	0.45
33	1	0.45
35	1	0.45
575	222	100.00

Table 8.2: Distribution of defects across faulty classes in the BEHEERNET project

models influences defect density. Later, relationship between defect density and level of details in UML modeling is examined. A closer look at the contribution of each individual metric to predicting defect density is performed. After this, which metric has a stronger correlation with a certain defect type is discussed.

8.4.1 Descriptive statistics

Prior to the statistical analyses, a statistical overview of this project is presented. First of all, the distribution of defects across faulty classes is summarized in Table 8.2.

As can be seen from Table 8.2, the number of modified faulty classes which are related to the chosen defects is 222. About half of the faulty classes only have one defect (52.71 percent). The range of the defect-count number which faulty classes are related to is quite wide and the highest defect-count is 35.

Table 8.3 listed the profile of these faulty classes with respect to their presence in UML models.

8.4.2 Defect density comparison between different system parts

In this section, whether the usage of UML models influences a project's defect density is analyzed. Faulty classes used in the analyses below are based on the information listed in Table 8.3.

We first analyzed defect density difference between system parts modeled as design classes in UML models and those not modeled at all. Figure 8.2 shows two boxplots which compare defect density of the modeled and unmodeled faulty classes. However, the defect density difference between the two groups is hard to illustrate due to several outliers found in unmodeled group. After a careful check on the data, a few classes of these outliers are abstract classes which have

Faulty classes	# classes
modeled as design classes in UML models (*modeled*)	43
modeled in class diagrams (*modeledinCD*)	13
modeled but not referenced in any class diagram (*notmodeledinCD*)	20
modeled in sequence diagrams (*modeledinSD*)	33
modeled in class diagrams only (*modeledinCDonly*)	6
modeled in sequence diagrams only (*modeledinSDonly*)	26
modeled in both types of diagrams (*modeledinBoth*)	7
modeled but not referenced in any diagram (*modeledinNeither*)	4
not modeled in UML models at all (*unmodeled*)	179

Table 8.3: BEHEERNET: The profile of faulty classes with respect to their presence in UML models

	modeled/ unmodeled	N	Mean	Std. Deviation	Std. Error Mean
Normalized Defect Density	modeled	43	-0.399	0.983	0.149
	unmodeled	179	0.096	0.982	0.073

Table 8.4: BEHEERNET: Group statistics of defect density between the modeled and unmodeled faulty classes

very low KSLoC, therefore their defect density values are much higher. However, we were assured that these outliers and extreme values shown in the figure were not caused by errors in the data (e.g., caused by wrong data entry), thus they could not be excluded from the analysis. Actually, faulty classes modeled in UML models have lower defect density than the unmodeled ones. This can be seen from the median value of each group. In BEHEERNET, the median values of the modeled and unmodeled classes are 12.14 and 17.34 respectively.

Later, independent t-test was performed on the normalized data set to analyze the significance of this difference and the results are shown in Table 8.5. The group statistics are given in Table 8.4 to present the mean values of the two normalized groups.

As shown in Table 8.5, the test reveals a true significance ($p = 0.003$). It suggests that the mean difference of defect density between the modeled and unmodeled faulty classes is significant. The consistent result mentioned in the previous two cases is obtained in this case study. The difference is significant at 0.01 level ($p \leq 0.01$), 1-tailed.

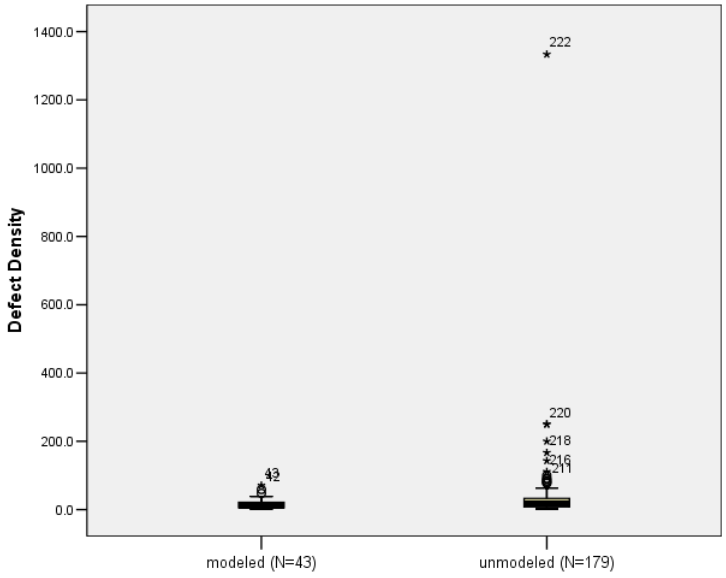

Figure 8.2: BEHEERNET: Defect density (per KSLoC) of modeled and unmodeled system parts

Project	t	df	Sig. (1-tailed)	mean Difference
BEHEERNET *Equal variances assumed*	-2.966	220	0.003	-0.495

Table 8.5: BEHEERNET: Independent t-test of defect density between the modeled and unmodeled faulty classes

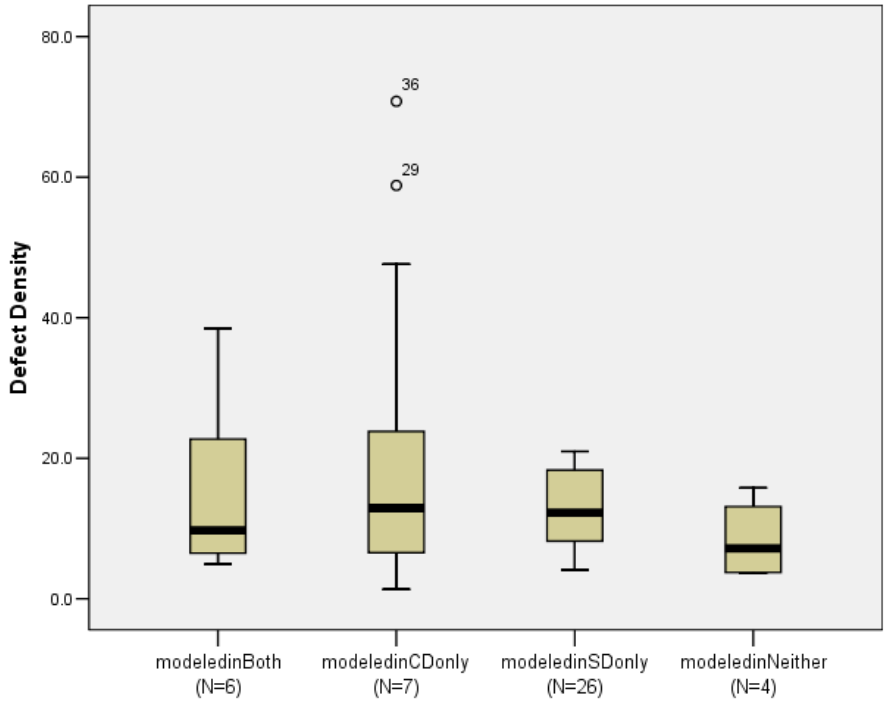

Figure 8.3: BEHEERNET: Defect density (per KSLoC) of different modeled and unmodeled system parts

Another analysis was performed to compare the defect density of faulty classes based on the diagram types they were modeled in. As is shown before, faulty classes modeled as designed classes in UML models could be splitted up into several groups, *modeledinCDonly*, *modeledinSDonly*, *modeledinBoth*, and *modeledinNeither*. After comparing the median values of defect density among all these data groups shown in Figure 8.3, there is no obvious difference between any pair of the groups. The biggest difference is between faulty classes modeled in neither types of diagrams (*modeledinNeither*) and faulty classes only modeled in class diagrams (*modeledinCDonly*). Defect density is lower for *modeledinNeither* faulty classes. The median values are 7.167 and 12.917, respectively.

In order to see whether this difference is significant, **One-way independent ANOVA** test was performed on the normalized data set to compare mean values among multiple groups. Again, the **Games-Howell** test of **Post hoc** procedures (see Appendix C.3) was used for exploring the data for any differences between means that exist. The descriptive statistics are given in Table 8.6 and the biggest difference of mean values is between *modeledinNeither* and *modeledinCDonly*, which are -0.477 and 0.067 respectively. Table 8.7 showed that the homogeneity of variance is met. The ANOVA test results can be found from Table 8.8. However, no significant difference in defect density between groups ($p = 0.798$) is found. Hence, there is no significant difference among defect density of faulty classes based on diagram types they are modeled in.

Normalized	N	Mean
modeledinBoth	6	0.065
modeledinCDonly	7	0.067
modeledinSDonly	26	-0.031
modeledinNeither	4	-0.477

Table 8.6: BEHEERNET: One-way ANOVA descriptive statistics

Levene Statistic	df1	df2	Sig.
1.582	3	39	.209

Table 8.7: Test of Homogeneity of Variances among different modeled groups

As can be seen from the faulty classes' profile listed before (see Table 6.3), faulty classes that are modeled as designed classes in UML models can be divided into two categories based on whether these classes are actually used in class diagrams or not. In PARTS, a significant defect density difference between faulty classes modeled in the UML models but not referenced in any of the class diagrams (*notmodeledinCD*) and those which are modeled as designed classes and also used in class diagrams (*modeledinCD*) was found. We were curious to see whether the consistent result found in PARTS can be obtained in this case.

As can be seen from the boxplots shown in Figure 8.4, there is not much difference in defect density of the two groups. The median values are 12.012 and 11.988, respectively. Since this difference is quite small, we can conclude that there is no significant difference between the two groups .

8.4.3 Correlation Analyses between UML LoD and Defect Density

In this section, three correlation analyses were performed to investigate the relation between the level of detail in UML models of faulty classes and their defect density. Although the significant negative correlation between UML LoD using sequence diagram metrics and defect density found in PARTS was not able to be confirmed in RACE due to too few data points, we still believe that higher LoD in UML models has a positive influence on reducing defect density in the implementation. The correlation analyses were performed first between UML LoD using class diagram metrics and defect density. Later, LoD_{SD} measured at diagram level and class-instance level are examined respectively.

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	1.058	3	0.353	0.338	0.798
Within Groups	40.692	39	1.043		

Table 8.8: One-way ANOVA test

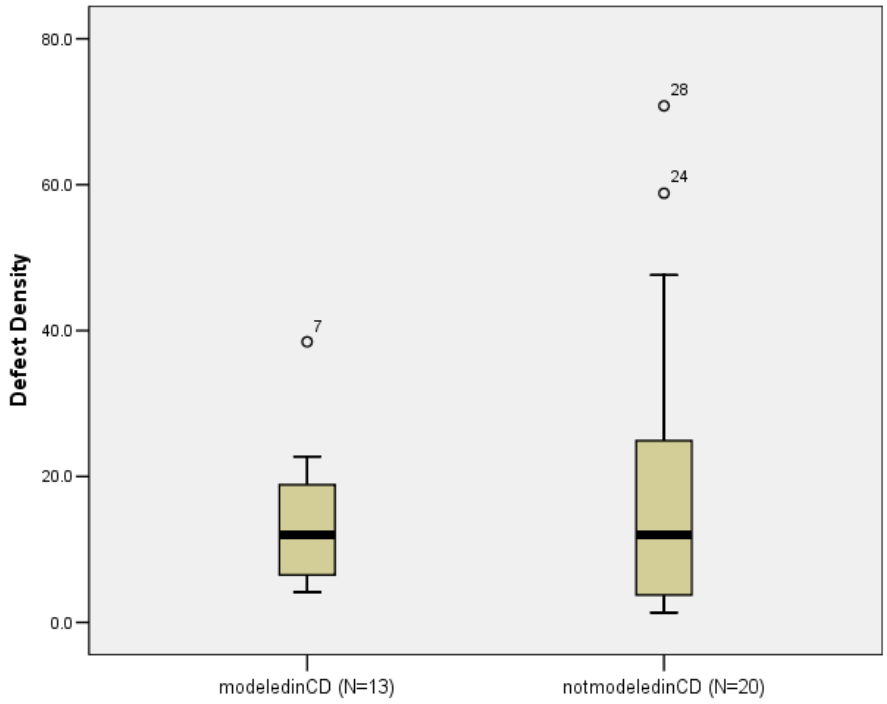

Figure 8.4: BEHEERNET: Defect density (per KSLoC) of modeled faulty classes presented in CD or not

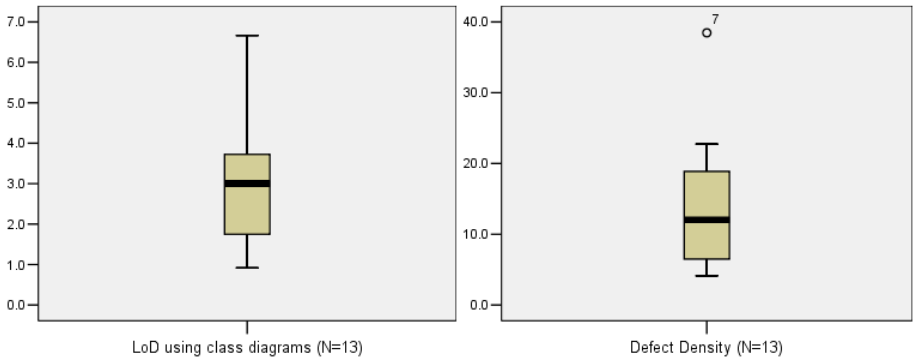

Figure 8.5: BEHEERNET: Boxplots of class LoD (LoD_{CD}) and defect density

	Defect Density
Class LoD	0.009
Significance (2-tailed)	0.977

Table 8.9: Pearson's correlation coefficient between class LoD (LoD_{CD}) and defect density

Using Class Diagram Metrics

This analysis intends to answer whether there is a significant correlation between LoD_{CD} and defect density. In total, 13 faulty classes were modeled in class diagrams and the descriptive statistics of the LoD_{CD} scores are shown in Figure 8.5. Since the original data set values of LoD_{CD} and defect density are normally distributed, **Pearson's** correlation test was performed. The result of the analysis is shown in Table 8.9. As was found in the previous case studies, no significant correlation between Class LoD using class diagram metrics and defect density is found in the BEHEERNET project.

Using Sequence Diagram Metrics

Two analyses were performed using sequence diagram metrics measured at diagram level and class-instance level, respectively. In total, 33 faulty classes were modeled in sequence diagrams. We first consider LoD_{SD} measured at diagram level and then LoD_{SD} measured at class-instance level.

LoD_{SD} measured at diagram level The summary of the LoD scores and defect density of the 33 faulty classes is presented in Figure 8.6. Figure 8.7 illustrated the relation between the two variables.

Pearson's correlation analysis was performed on the normalized data set. The result of the analysis can be found in Table 8.10. To our big surprise, as can be seen from the table, there is no significant correlation between the two variables. The scattor plot of the two variables after normalization even indicated a random distribution (see Figure 8.8). Immediately, we performed the correlation analysis between LoD_{SD} measured at class-instance level and

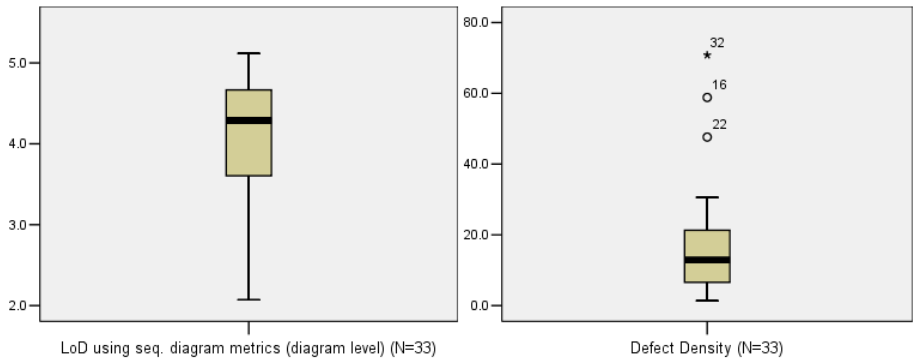

Figure 8.6: BEHEERNET: Boxplots of class LoD (LoD_{SD} measured at diagram level) and defect density

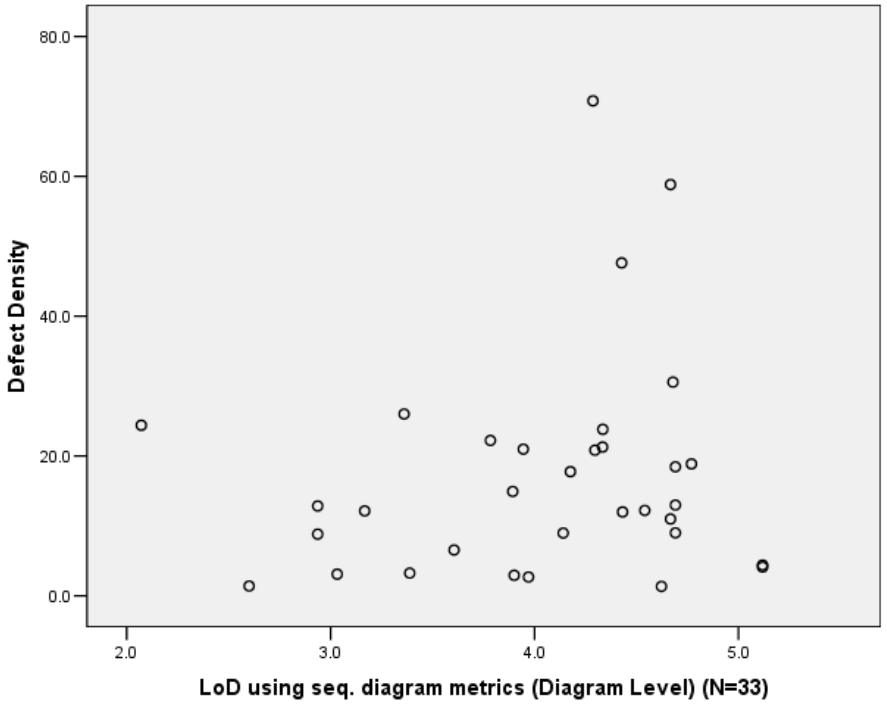

Figure 8.7: BEHEERNET: Scatterplots of the relation between class LoD (LoD_{SD} measured at diagram level) and defect density

	Defect Density
Class LoD	0.070
Significance (1-tailed)	0.349

Table 8.10: Pearson's correlation coefficient of LoD_{SD} measured at diagram level and defect density

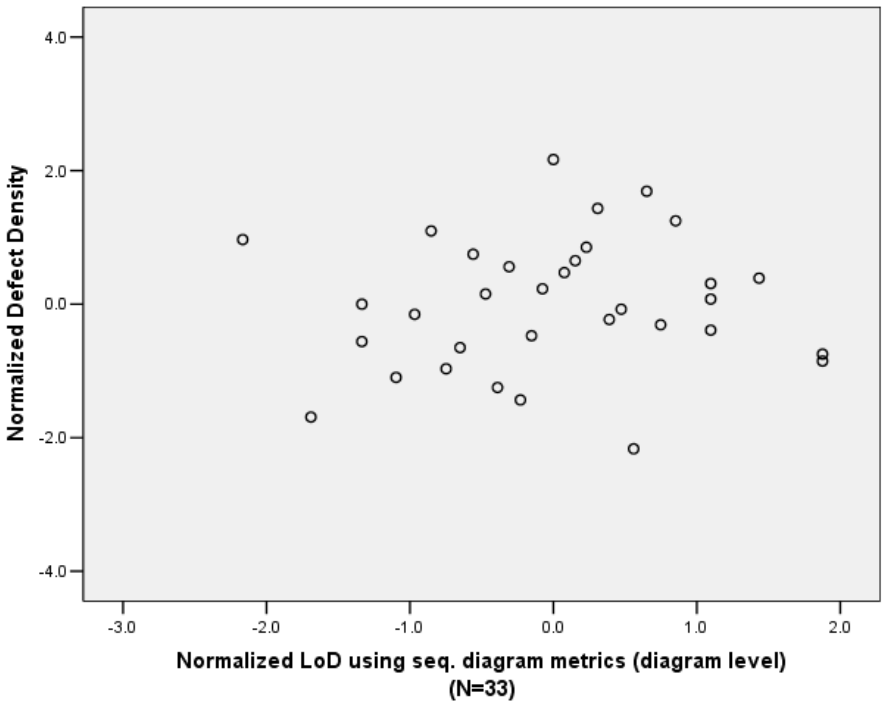

Figure 8.8: BEHEERNET: Scatterplots of the relation between class LoD (LoD_{SD} measured at diagram level) and defect density after normalization

defect density to see whether the similar result is found.

LoD_{SD} measured at class-instance level Again, the summary of the LoD scores and defect density of the 33 faulty classes modeled in sequence diagrams is presented in Figure 8.9. The relation between the two variables is shown in Figure 8.10.

After performing the Pearson's correlation test on the normalized LoD_{SD} and defect density, the result is shown in Table 8.11.

As can be seen from Figure 8.10, the correlation analysis result again resulted in a non-significant correlation between the two variables. After examining the data set measured at both levels, several common outliers were found and their characteristics are listed in Table 8.12.

As can be seen from Table 8.12, "Delegate" classes refer to faulty classes which are designed as business delegate classes. Normally, a business delegate

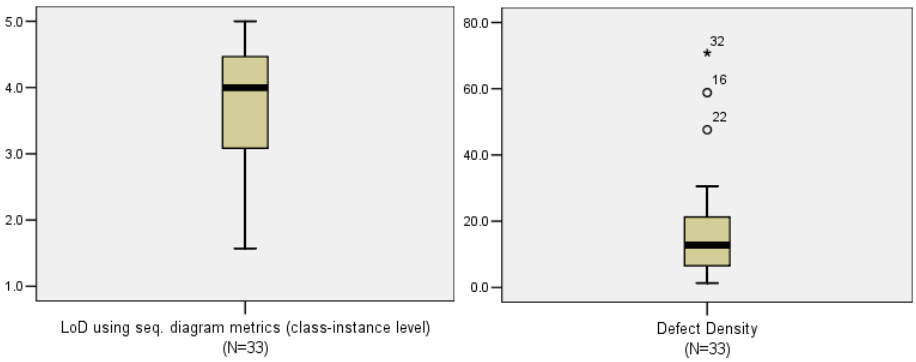

Figure 8.9: BEHEERNET: Boxplots of class LoD (LoD_{SD} measured at class-instance level) and defect density

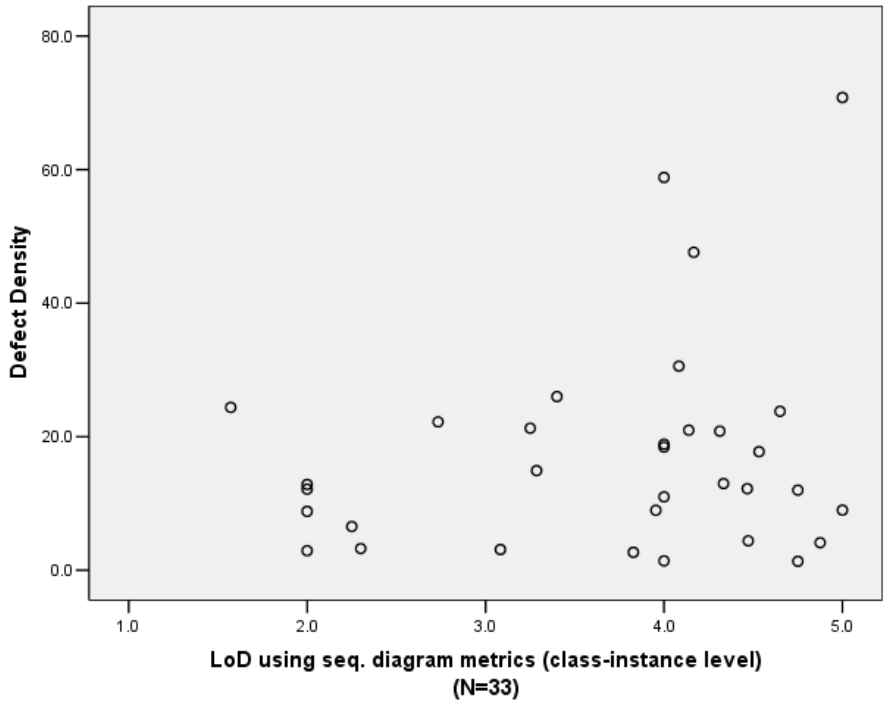

Figure 8.10: BEHEERNET: Scatterplots of the relation between class LoD (LoD_{SD} measured at class-instance level) and defect density

	Defect Density
Class LoD	0.091
Significance (1-tailed)	0.308

Table 8.11: Pearson's correlation coefficient of LoD_{SD} measured at class-instance level and defect density

Outliers	Characteristics	Interpretations
D̈elegatec̈lasses	low defect density low LoD	modeled as dummy objects with dummy messages attached to them
Salarylist	high defect density high LoD	low KSLoC, high defect count only modeled in one sequence diagram
SalarylistForm	low defect density low LoD	modeled in eight alternative flow sequence diagrams
TargetGroup	high defect density high LoD	very low KSLoC

Table 8.12: The outliers in the correlation analyses between LoD_{SD} and defect density

class acts as a client-side business abstraction which reduces the coupling between presentation-tier clients and the system's business services. In total, five faulty classes are designed as "Delegate" classes, all of which have low defect density and low LoD. One interpretation might be that these classes do not have high complexity and it is not necessary to design them into detail. Another outlier named "Salarylist" is found to have very high defect density and high LoD. Besides, it has low KSLoC (equals 0.113) but quite high defect count (equals 8). Being modeled in only one sequence diagram might be one reason, because the LoD value in this sequence diagram might not represent the true LoD of this class. Meanwhile, the high defect count might due to the low correctness of the class itself, since this class is only modeled in one sequence diagram and has no interaction with other objects. It might be also necessary to distinguish the basic flow and alternative flow sequence diagrams. As can be seen from "SalarylistForm" class, it is modeled in nine sequence diagrams among which eight diagrams are alternative flows with very low LoD. According to the LoDSD aggregate (see Section 6.3), the LoD of this class becomes quite low due to these alternative flows. However, we could not find any sound reason to exclude these outliers from the data set, therefore, we still could not find a negative and significant correlation between LoD using sequence diagram metrics and defect density.

8.4.4 The contribution of individual metrics to predicting defect density

Although we could not find any correlation between LoD using sequence diagram metrics and defect density, individual metrics' predictive power to defect density might be still worthy of investigation. Correlation analyses were performed to measure the correlation between each metric and defect density. we started with analyzing sequence diagram metrics measured at diagram level, followed by metrics measured at class-instance level.

Metrics measured at diagram level After a closer look at individual metrics. *MsgWithLabelRatio* metric values were found always constant. A couple of correlation analyses were performed between the rest of the metrics and defect density. The results are listed in Table 8.13.

Test Method	Metric	Defect Density
Pearson's	NonAnonymObjRatio	**-0.239**
	Significance (2-tailed)	**0.180**
	NonDummyMsgRatio	0.067
	Significance (2-talied)	0.712
	MsgWithGuardRatio	0.006
	Significance(2-talied)	0.972
	MsgWithParamRatio	**0.374***
	Significance (2-tailed)	**0.032**
Spearman's	NonDummyObjRatio	-0.042
	Significance (2-talied)	0.818
	ReturnMsgWithLabelRatio	0.051
	Significance (2-talied)	0.776

*indicates siginificance at 0.05 level (2-tailed)

Table 8.13: Correlation analyses between individual metrics measured at diagram level and defect density

Only one metric, *NonAnonymObjRatio*, indicates a negative correlation with defect density. Among the remaining metrics, *MsgWithParamRatio* metric has a significant positive correlation with defect density (*Sig.* = 0.032).

Metrics measured at class-instance level The correlation analyses results between sequence diagram metrics measured at class-instance level and defect density are shown in Table 8.14. Only two metrics, *returnMsgWithLabelRatio* and *NonAnonymObjRatio* indicate negative correlations with defect density.

8.4.5 The correlation between UML LoD metrics and defect density of individual defect type

As can be seen from Figure 8.1, a majority of defects belong to two defect types - *datahandling* and *logic*. The number of faulty classes related to these two defect types are 24 and 19, respectively. In PARTS, correlation analyses using sequence diagram metrics were performed on these two defect types. The same correlation analyses procedure was used in this project and the results are given in the following passages.

Data Handling Defect Type

In total, 24 faulty classes are related to data handling defect type. Table 8.15 shows the correlation analyses results for *data handling* defect type.

As can be seen from Table 8.15, for sequence diagram metrics measured at diagram level, *NonAnonymObjRatio*, *MsgWithGuardRatio* and *NonDummyObjRatio* indicate negative correlation with defect density. However, none of them is statistically significant. For metrics measured at class-instance level, *ReturnMsgWithLabelRatio*, *NonAnonymObjRatio* and *MsgWithGuardRatio* show a negative correlation with defect density.

Test Method	Metric	Defect Density
Spearman's	ReturnMsgWithLabelRatio	**-0.286**
	Significance (2-tailed)	0.106
	NonAnonymObjRatio	**-0.278**
	Significance (2-tailed)	0.117
	MsgWithParamRatio	0.223
	Significance (2-tailed)	0.212
	MsgWithGuardRatio	0.216
	Significance (2-tailed)	0.228
	NonDummyMsgRatio	0.034
	Significance (2-tailed)	0.850
	NonDummyRatio	0.188
	Significance (2-tailed)	0.295

Table 8.14: Correlation analyses of individual metrics measured at class-instance level and defect density

Modeled level	Test Methods	Metrics	Defect Density
Diagram level	Pearson's	NonDummyMsgRatio	-0.080
		Significance (2-tailed)	0.710
		MsgWithParamRatio	0.282
		Significance (2-tailed)	0.182
		NonAnonymObjRatio	-0.312
		Significance (2-tailed)	0.137
		MsgWithGuardRatio	-0.254
		Significance (2-tailed)	0.231
	Spearman's	ReturnMsgWithLabelRatio	0.047
		Significance (2-tailed)	0.826
		NonDummyObjRatio	-0.191
		Significance (2-tailed)	0.370
Class-instance level	Spearman's	ReturnMsgWithLabelRatio	-0.227
		Significance (2-tailed)	0.286
		NonAnonymObjRatio	-0.288
		Significance (2-tailed)	0.172
		MsgWithGuardRatio	-0.082
		Significance (2-tailed)	0.702
		NonDummyMsgRatio	0.136
		Significance (2-tailed)	0.527
		MsgWithParamRatio	0.161
		Significance (2-tailed)	0.451
		NonDummyObjRatio	0.116
		Significance (2-tailed)	0.589

Table 8.15: Correlation coefficient of LoD_{SD} metrics and defect density (*Data handling* defect type)

Modeled level	Test Methods	Metrics	Defect Density
Diagram level	Pearson's	NonAnonymObjRatio	**-0.344**
		Significance (2-tailed)	**0.149**
		MsgWithGuardRatio	-0.043
		Significance (2-tailed)	0.861
		NonDummyObjRatio	0.037
		Significance (2-tailed)	0.881
		NonDummyMsgRatio	0.137
		Significance (2-tailed)	0.576
	Spearman's	ReturnMsgWithLabelRatio	0.360
		Significance (2-tailed)	0.131
Class-instance level	Spearman	NonAnonymObjRatio	-0.287
		Significance (2-tailed)	0.234
		MsgWithParamRatio	0.316
		Significance (2-tailed)	0.187
		MsgWithGuardRatio	0.201
		Significance (2-tailed)	0.409
		NonDummyMsgRatio	0.099
		Significance (2-tailed)	0.687

Table 8.16: Correlation analyses of LoD_{SD} metrics and defect density (*logic* defect type)

Logic Defect Type

In total, 19 faulty classes are related to logic defect type. The correlation analyses results for *logic* defect type are given in Table 8.16. As can be seen from Table 8.16, for sequence diagram metrics measured at both diagram level and class-instance level, only *NonAnoymObjRatio* indicates negative correlations with defect density. However, neither is statistically significant.

8.5 Results and Conclusions

In this section, we list the findings discovered in this case study. The possible interpretations and the implications of the results are discussed.

First of all, the consistent result about the influence of using UML models on defect density is confirmed in this project. Faulty classes that were modeled, on average, had a lower defect density than those not modeled at all. This difference of defect density was statistically significant. This result implicates again that well designed UML models can be good guidance in the implementation phase. Later, the modeled classes were splitted up into more detailed categories based on how they are modeled. However, no significant difference of defect density was found. Further, we made a distinction between modeled classes presented in the UML diagrams and those just modeled as design classes but not presented in any of the class diagrams. There is no significant difference between the two variables either.

Prior to the correlation analyses between UML LoD using sequence diagram

metrics and defect density, the consistent results mentioned in the previous case studies were expected to be confirmed in this project. In the PARTS project, classes that are modeled at a higher level of detail are inclined to having lower defect density. This correlation is statistically significant. Although this correlation is not significant in the RACE project, a negative correlation was indicated. In the BEHEERNET project, to our surprise, no correlation was found between UML LoD using sequence diagram metrics and defect density. We also performed the correlation analyses between each UML LoD_{SD} metric and defect density, no significant correlation was found either. Only one or two metrics indicated negative correlations with defect density. We believe that it is necessary to make a comparison among these three projects to find the reasons why we could not get the same result in the BEHEERNET project.

Later, we wanted to have a look at the correlations between individual metrics and defect density of a certain defect type. *logic* and *data handling* defect types were considered in the analyses. For both *data handling* and *logic* defect types, none of the LoD using sequence diagram metrics had significant correlation with defect density. Only *NonAnonymObjRatio* metric indicated a relatively stronger negative correlation with defect density measured at both diagram level and class-instance level.

Chapter 9

Projects Comparison between PARTS and BEHEERNET

In this chapter, a comparison of the two case studies (namely, PARTS and BEHEERNET) we have performed before is given. We did not include the RACE project due to the too small sample size compared to the other two cases. This can be seen from the project summary listed below. Another reason for performing this comparison between PARTS and BEHEERNET is that a different conclusion about whether there is a correlation between UML LoD using sequence diagram metrics and defect density was found in the two projects (see Section 6.4.3 and 8.4.3). After the comparison, some observations are discussed to explain this difference.

9.1 Project characteristics comparison

In total, we performed three empirical case studies from the real world, PARTS, RACE, and BEHEERNET. These three projects were all developed as web service which can be divided into two parts, front–end and back–end. The front–end part is designed as an application which is responsible for interactions with clients, while the back-end is where the logic and services are implemented. UML models designed in these projects were created using IBM Rational XDE. Some other Rational tools such as IBM Rational ClearCase and ClearQuest were adopted for code versioning and bug tracking respectively. The summaries of the three projects are given in Table 9.1.

As can be seen from Table 9.1, the three projects use the same technology and they all have some tasks offshored to India. PARTS and BEHEERNET projects are considered to be comparable in terms of the project size (i.e., # staff, project duration, sloc). Although PARTS and BEHEERNET are similar in terms of project size, the UML model size is much larger for PARTS which might indicate that UML modeling in this project is more sufficient and better prepared. This idea is also indicated in the comparison of project characteristics between the two projects in Table 9.2.

As shown in Table 9.2, the two projects are quite similar in project environment and working style. However, In BEHEERNET, the developers did not have sufficient knowlege of using UML models. Furthmore, the UML models

Project	PARTS	RACE	BEHEERNET
Technology	Java	Java	Java
# staff	25 people	10 people	18 people
Duration (in years)	2.3 years	1 year	2 years
Off–shored	India	India	India
Status	Finished	Finished	Finished
Model size	104 use cases 266 designed classes 341 seq. diagrams 34 class diagrams	9 use cases 44 designed classes 22 seq. diagrams 12 class diagrams	20–25 use cases 137 designed classes 115 Seq. diagrams 28 Class diagrams
SLoC	152,017	125,168	135,454

Table 9.1: Projects summary comparison

Project	PARTS	BEHEERNET
Project environment	web service MVC architecture Apache struts & JSP	web service 4+1 view model architecture Apache struts & JSP
Developer experience	not sufficient ask for detailed UML models	not sufficient simply implement
Adopted Process	Iterative Four major increments	Waterfall One big release
Working style	60 % Implementation and testing offshored RD and UML models in NL	A majority part of implementation and testing offshored RD and UML models in NL

Table 9.2: Projects characteristics comparison

Project	PARTS	BEHEERNET
Population	1546	4061
# traceable findins	566	1784
Sample size	164	200
Sampling method	Random: the first 100 Targeted: the next 64	Targeted: the first 135 Random: the next 65

Table 9.3: Data sampling comparison

were designed at a high abstract level with the intention of giving developers more freedom while creating the code. The gap between developers' experience and the level of details in UML models results in a very low correspondence between UML models and the implementation. Developers' insufficient experience has become a big problem for BEHEERNET. Most of the time, developers just implemented the code without fully understanding the UML models. Sometimes, they even wrote the code without looking at the UML models. Another difference is the adopted development process. In BEHEERNET, waterfall approach was taken during the development and only one big release was given at the end of the project. Many defects could have been avoided if testing can be done iteratively during different development stages and lots of bug-fixing time can be saved.

9.2 Defect statistics comparison

In this section, we mainly compare defect related statstics of the two projects during defect data collection and preprocessing processes.

9.2.1 Data sampling method

The sampling methods used in PARTS and BEHEERNET are quite different which can be seen in Table 9.3. The third column in the table, " # traceable findings", refers to the number of findings which have modified source files attached to them. The defect sample is actually generated from these findings. Although the number of findings traceable back to the source code is much higher for BEHEERNET, there is not much difference in terms of sample size between the two projects. Two sampling processes were adopted for each project. The random sampling chooses defects randomly without any criteria. The targeted sampling filters findings which have modified source files and among which at least one source file is related to designed class in the UML models. The idea of using this sampling is to ensure that we have got enough data points for the statistical analyses, especially the correlation analysis. As can be seen from the table, the sampling methods of the two projects are quite different and the order of performing sampling processes is opposite too. This difference in choosing sample size might raise several problems, such as the defect type distribution comparison which will be shown shortly.

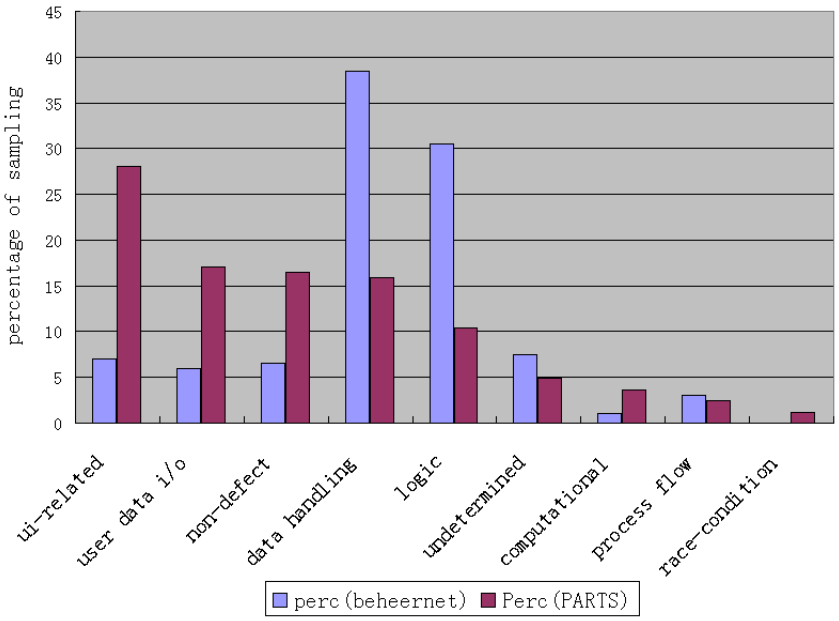

Figure 9.1: Defect type distribution comparison

9.2.2 Defect type distribution comparison

After choosing the sample size, defect type distribution comparison is shown in Figure 9.1. The listed defect types can be found in Section 4.1. Since a majority of the defects in the sample data set of BEHEERNET are related to the designed classes, many defects are closely connected to the back–end of the system. Therefore, it is not surprising to see that ui-related defects are much fewer than that in the PARTS project. Besides, defect types (i.e., data handling and logic) which mostly related to defects happen in the back–end have much higher percentages than those recorded in PARTS.

9.2.3 Purified defects statistics comparison

The purified data set for statistical analyses were chosen by removing defects related to ui-related, non-defect, and undetermined defect types. Table 9.4 shows the comparison of purified defects statistics. It is surprising to see that there is not much difference in the number of faulty classes modeled as design classes between the two projects. However, the number of relevant defects and the number of faulty classes related to those defects for BEHEERNET are almost twice as large as those for PARTS. The ratios of modeled faulty classes to faulty classes related to relevant defects are 28.5 % and 19.4 %, respectively. This suggests that the correspondence between UML models and the implementation is higher for PARTS. Although this difference might be again influenced by the sampling method, this ratio for BEHEERNET is expected to be the same as or even lower than the current value if the same sampling method used in PARTS is performed in BEHEERNET. This finding confirms the conclusion drawn from

Project	PARTS	BEHEERNET
Sample size	164	200
Relevant defects	83	158
# Faulty classes	130	222
# Modeled faulty classes	37	43
# Unmodeled faulty classes	93	179

Table 9.4: Purified defects statistics comparison

project characteristics comparison before (see Section 9.1).

9.3 Statistical analyses comparison

In this section, we mainly discuss the comparison of correlation analysis between UML LoD using sequence diagram metrics and defect density performed in the two projects.

9.3.1 Descriptive statistics comparison

Since the main research question in this study is about the correlation between the level of detail in UML models and defect density in the implementation, variables used to measure this relation are investigated in this section. In other words, the descriptive statistics comparison of faulty classes that are modeled in UML models and their defect density are given.

First of all, the defect counts distribution across faulty classes of the two projects is illustrated in Figure 9.2. This distribution again might be influenced by the sampling methods used in the two projects. The range of defect counts which a faulty class is related to is much wider for BEHEERNET. Around five percentage of the faulty classes are invovled in at least eight defects. On the other hand, the percentage of faulty classes related to only one defect is much higher for PARTS. This might indicate that the quality of the implementation in PARTS is higher than that of BEHEERNET.

For each case study we performed in the previous chapters, a profile of the faulty classes with respect to their presence in the UML models is given. A comparison of the profiles of the PARTS and BEHEERNET projects is listed in Table 9.5. As shown in the table, the percentage values of *modeled* and *unmodeled* faulty classes are calculated using the ratio of faulty classes belonging to these groups to the number of faulty classes related to defects in defect sample. The other percentage values are calculated using the ratio of faulty classes belonging to those groups to the total number of faulty classes modeled as design classes.

One interesting found from Table 9.5 is the modeling style difference. BEHEERNET has higher *notmodeledinCD* percentage which suggests that a higher percentage of designed classes created in the UML models are used in the diagrams of PARTS. The *modeledinCDonly*, *modeledinSDonly* percentages are higher for BEHEERNET, while the *modeledinBoth* percentage is much higher for PARTS (56.8 percent). These observations indicate that a variety of views

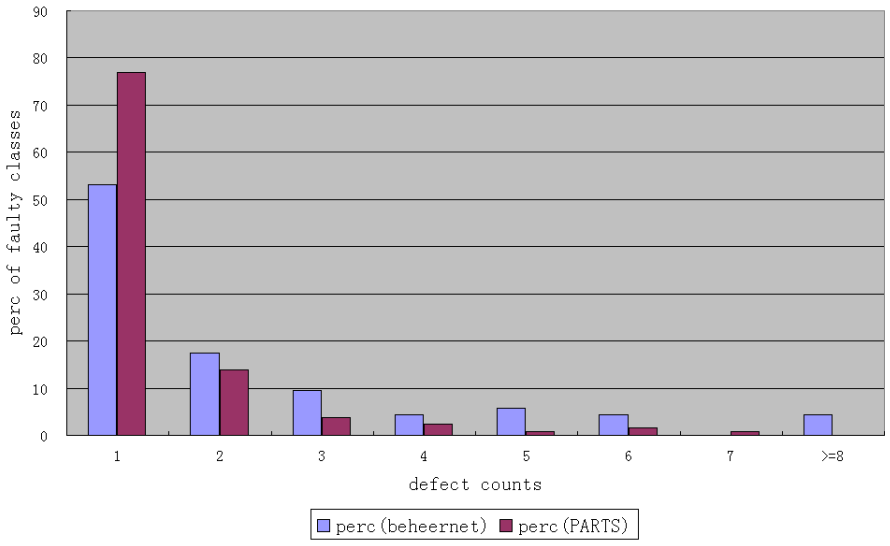

Figure 9.2: Defect counts distribution across faulty classes comparison

Faulty classes	PARTS	BEHEERNET
modeled as design classes in UML models (*modeled*)	28.5 %	19.4 %
modeled in class diagrams (*modeledinCD*)	62.2 %	30.2 %
modeled but not referenced in any class diagram (*notmodeledinCD*)	37.8 %	46.5 %
modeled in sequence diagrams (*modeledinSD*)	81.1 %	76.7 %
modeled in class diagrams only (*modeledinCDonly*)	5.4 %	13.9 %
modeled in sequence diagrams only (*modeledinSDonly*)	24.3 %	60.5 %
modeled in both types of diagrams (*modeledinBoth*)	56.8 %	16.3 %
modeled but not referenced in any diagram (*modeledinNeither*)	13.5 %	9.3 %
not modeled in UML models at all (*unmodeled*)	71.5 %	80.6 %

Table 9.5: The profile of faulty classes with respect to their presence in UML models comparison

Project	PARTS	BEHEERNET
# Faulty classes	30	33
LoD Seq. diagram Diagram level	R value: -0.459 Sig. (1-tailed): **0.005**	R value: 0.070 Sig. (1-tailed):0.699
LoD Seq. diagram class-instance level	R value: -0.371 Sig. (1-tailed): **0.022**	R value: 0.091 Sig. (1-tailed): 0.616

Table 9.6: The correlation analyses results using LoD_{SD} comparison

of the design classes are provided to the developers in PARTS. And, it is believed that it is easier for developers to understand the architectures of the whole project. Therefore, we believe that PARTS has a better UML modeling style than BEHEERNET which might make a difference in defect density in the implementation.

9.3.2 The correlation analyses between UML LoD and defect density comparison

The main research question in our study is about the impact of the level of detail in UML models on defect density in the implementation. Two sets of UML metrics are created to measure the UML LoD based on the diagram types (see Section 4.2). Besides, sequence diagram based metrics have different meanings based on whether they are measured at diagram level or class-instance level (see Section 4.2.2). In order to get the comparable correlation results, the same calculation method of UML LoD mentioned in Section 6.3 is used in the three case studies. For each case study, three correlation analyses were performed between UML LoD and defect density. Since none of the three projects has indicated a correlation between LoD_{CD} using class diagram metrics and defect density, only the comparison results of LoD_{SD} using sequence diagram metrics are shown in Table 9.6. The main difference is that LoD_{SD} using sequence diagram metrics has a significant negative correlation with defect density in the PARTS project but not in the BEHEERNET project. In the following paragraphs, some statistical descriptions about the two projects are compared in order to have an insightful understanding about the differences between the two projects.

First of all, defect density and UML LoD of the faulty classes modeled in sequence diagrams are compared. These two variables are used to perform the correlation analyses. Figure 9.3 compares the defect density difference between PARTS and BEHEERNET. The comparison of UML LoD is illustrated in Figure 9.4. It can be seen that BEHEERNET has higher defect density while UML LOD values using sequence diagram metrics measured at both levels are lower than that of PARTS (these results are proved statistically significant). As is mentioned before, the difference in defect density might be influenced by the sampling method. However, the difference in level of detail in UML models indicates that UML modeling in BEHEERNET was done in lower detailed and this conclusion can be confirmed by the project characteristics analysis discussed before.

Since defect density is the ratio of defect count of a faulty class to KSLoC of that faulty class, the distributions of defect count and KSLoC across faulty

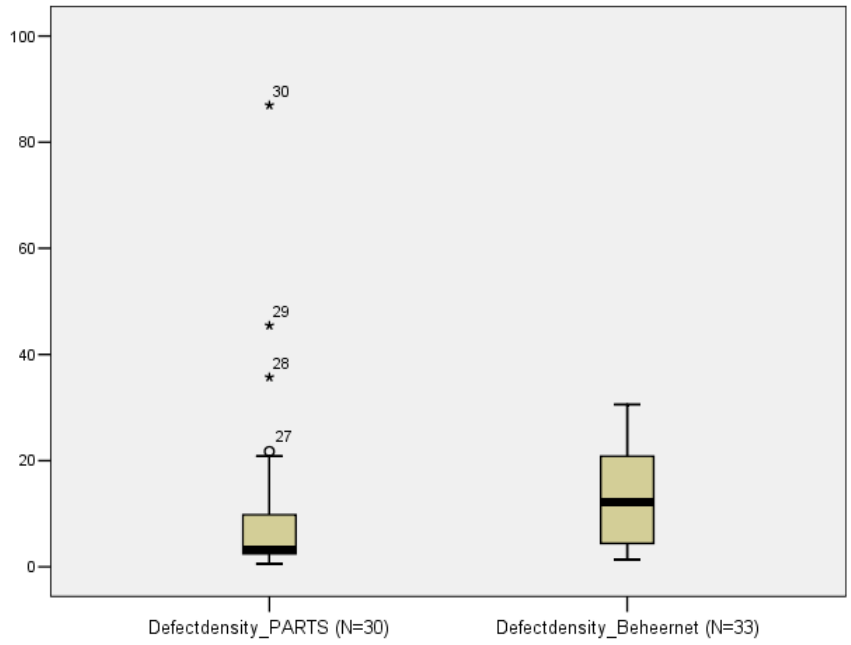

Figure 9.3: Defect density of faulty classes modeled in sequence diagrams comparison

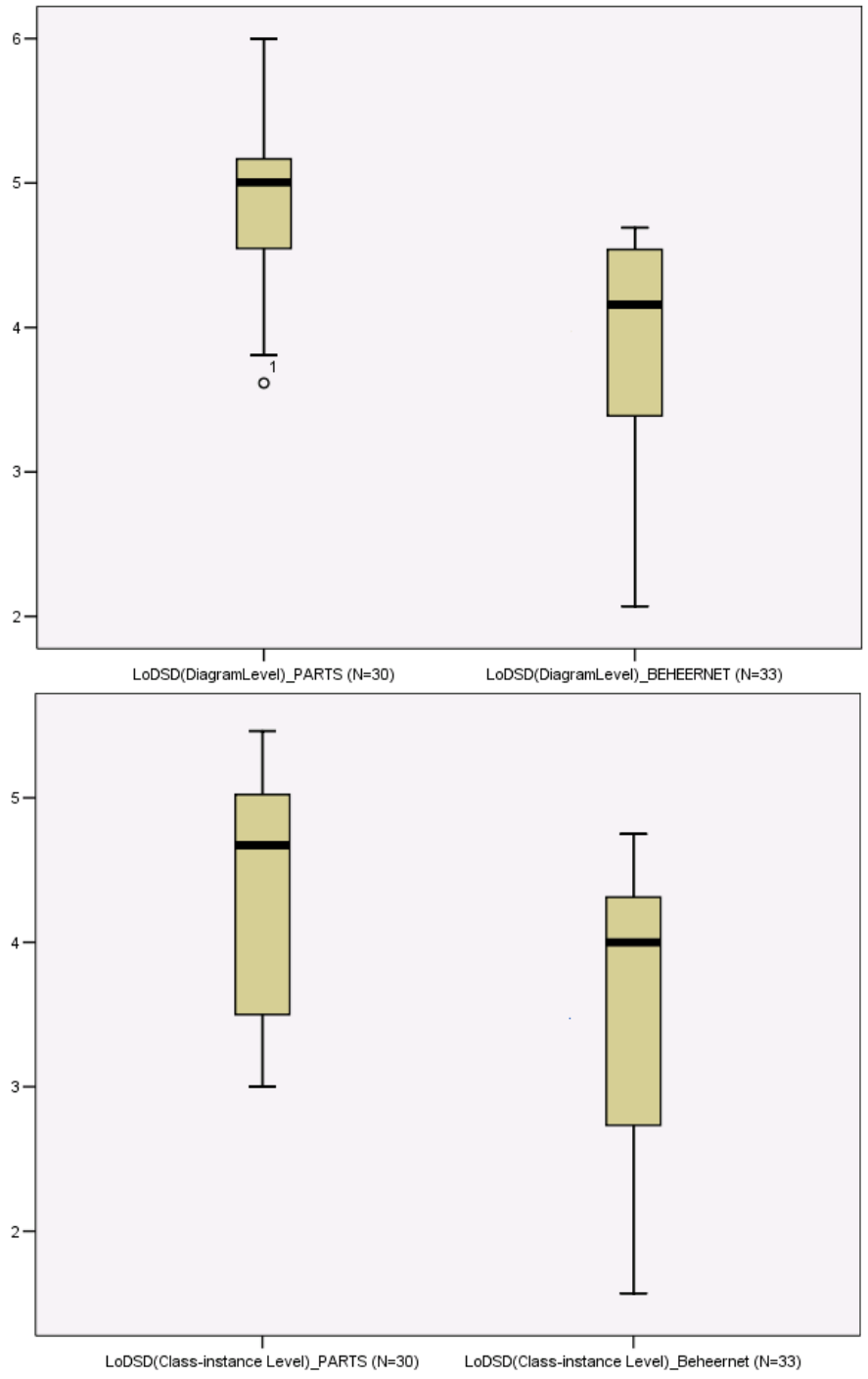

Figure 9.4: LoD_{SD} of faulty classes modeled in sequence diagrams comparison

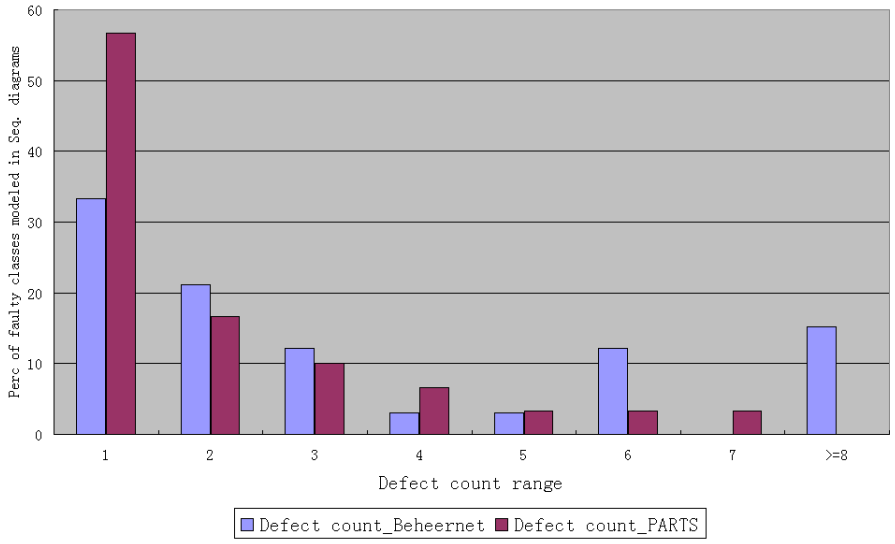

Figure 9.5: Defect count distribution of faulty classes modeled in sequence diagrams comparison

classes modeled in sequence diagrams are illustrated in Figure 9.5 and Figure 9.6, respectively. As can be seen from the two figures, the distribution of KSLoC of the two projects are quite similar and it can be proved that the difference in KSLoC is not statistically significant. However, about 15 percent of the faulty classes modeled in the sequence diagrams in BEHEERNET are related to at least eight defects. This strongly indicated that the implementation quality of BEHEERNET is lower than that of PARTS.

Meanwhile, defect density distribution of the two projects are also compared in Figure 9.7. Half of the faulty classes from PARTS only have defect density values between 1.0 and 5.0, while a big amount of the faulty classes from BEHEERNET have defect density in the range of 10.0 to 20.0.

After comparing defect density distribution, level of detail in UML models using sequence diagram metrics distribution is examined. Figure 9.8 and Figure 9.9 compare the LoD_{SD} distribution measured at diagram level and class-instance level. Figure 9.8 indicates a pattern of modeling designed classes in terms of level of detail (see Table 9.7). In general, LoD_{SD} measured at diagram level can be divided into four levels (from Level 1 to Level 4) and the LoD_{SD} values are from low to high. For Both projects, the percentages of LoD_{SD} measured at Level 1 and Level 4 are quite low, around 10 percent and five percent, respectively. A majority of the designed classes are modeled into Level 2 and Level 3, which have LoD_{SD} values in the middle. However, we could not tell more about the characteristics of each level due to the too few data points for Level 1 and Level 4. It is still interesting to investigate on it when enough data points are available.

At last, the number of sequence diagrams where a faulty class is modeled of the PARTS and BEHEERNET projects is compared. The comparison result is shown in Figure 9.10. One big difference between the two projects is that

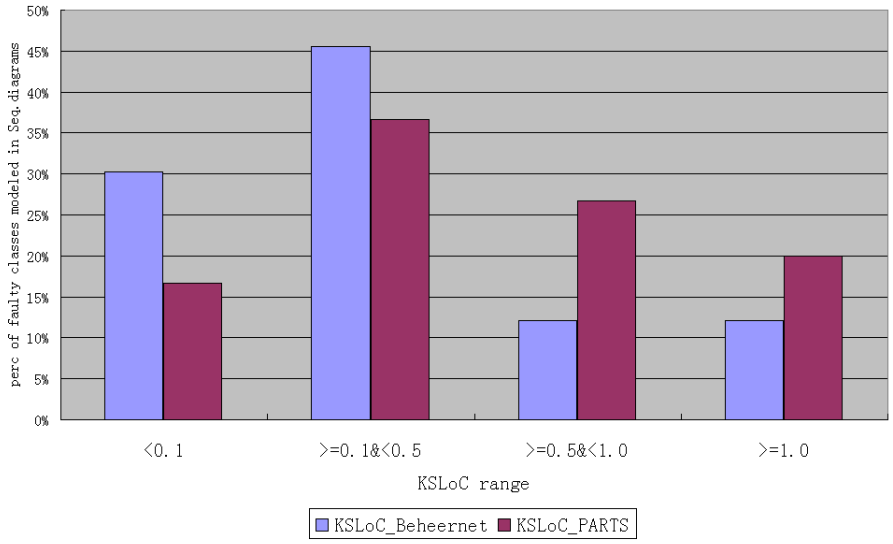

Figure 9.6: KSLoC distribution of faulty classes modeled in sequence diagrams comparison

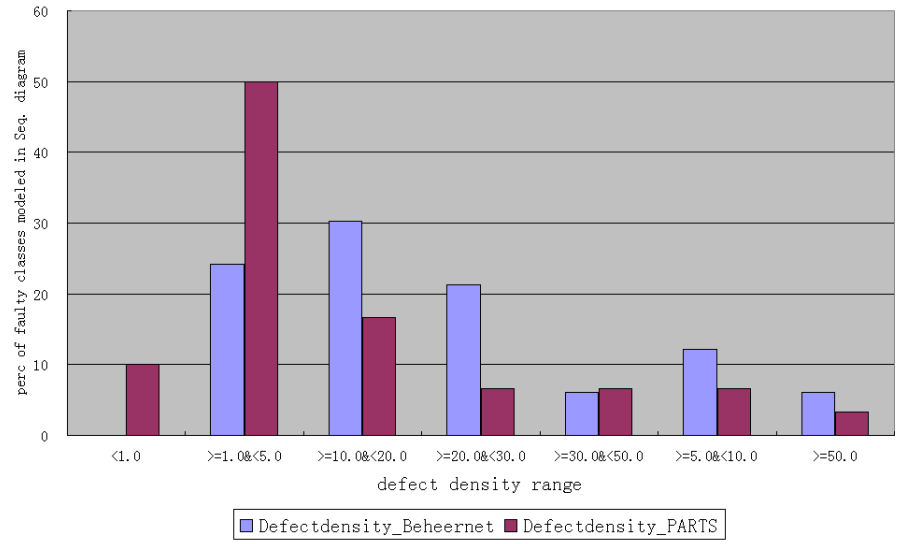

Figure 9.7: Defect density distribution of faulty classes modeled in sequence diagrams comparison

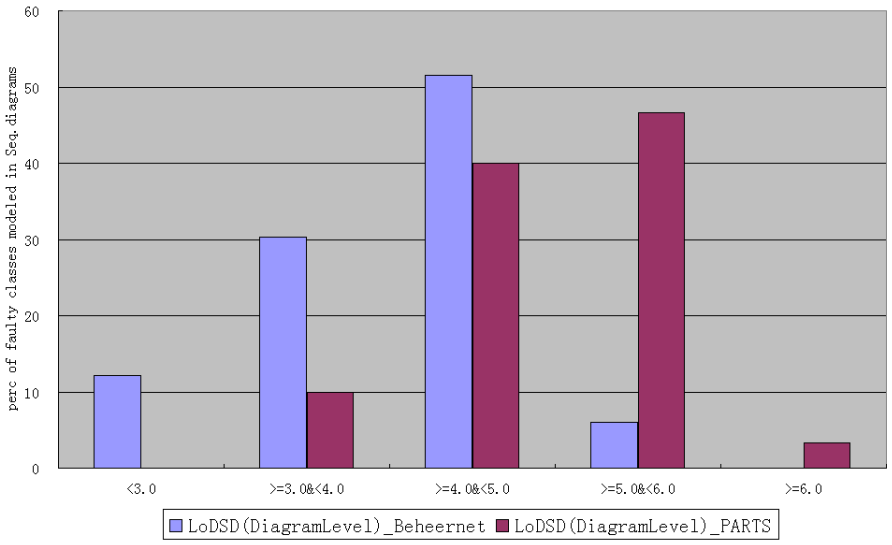

Figure 9.8: LoD_{SD} measured at diagram level distribution comparison

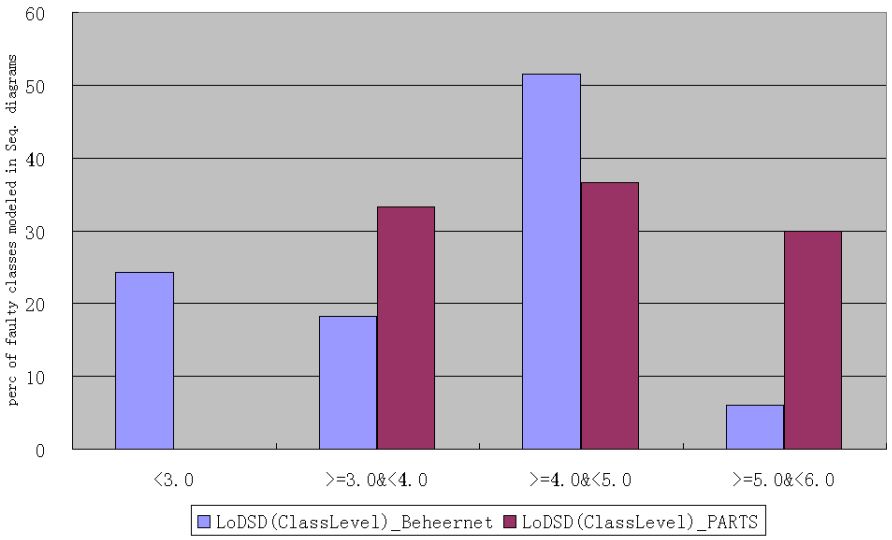

Figure 9.9: LoD_{SD} measured at class-intance level distribution comparison

Project	PARTS	BEHEERNET
LoD_{SD} Level 1	$LoD_{SD} < 4.0$ 10 %	$LoD_{SD} < 3.0$ 12.12 %
LoD_{SD} Level 2	$4.0 \leq LoD_{SD} < 5.0$ 40 %	$3.0 \leq LoD_{SD} < 4.0$ 30.30 %
LoD_{SD} Level 3	$5.0 \leq LoD_{SD} < 6.0$ 46.67 %	$4.0 \leq LoD_{SD} < 5.0$ 51.52 %
LoD_{SD} Level 4	$LoD_{SD} \geq 6.0$ 3.33 %	$LoD_{SD} \geq 5.0$ 6.06 %

Table 9.7: LoD_{SD} measured at diagram level pattern comparison

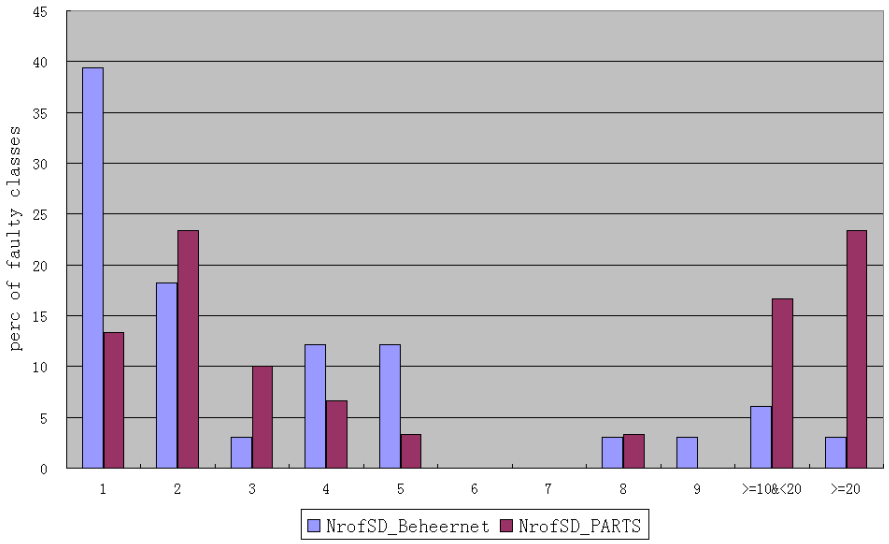

Figure 9.10: The number of Seq. diagrams distribution comparison

a majority of the faulty classes modeled as designed classes in BEHEERNET are only related to one or two sequence diagrams. On the other hand, around half of the modeled faulty classes in PARTS are related to at least 10 sequence diagrams. According to the calculation of LoD_{SD} (see Section 6.2), it is much easier to have outliers if a faulty class is only modeled in a few sequence diagrams. The influence of some extreme values can be reduced if a faulty class is modeled in more sequence diagrams. this might be another reason why there are relatively more outliers in BEHEERNET.

9.4 Conclusions

In this chapter, we mainly compare the differences between two case studies, PARTS and BEHEERNET. The biggest difference in terms of the research question between the two projects is the correlation analysis between UML LoD using sequence diagram metrics and defect density. Table 9.8 shows the difference.

Project	PARTS	BEHEERNET
# Faulty classes	30	33
Correlation analysis	negative significant	random distribution
LoD using Seq. diagram metrics	higher LoD	lower LoD
Defect density	lower defect density	higher defect density

Table 9.8: correlation analysis difference between PARTS and BEHEERNET

It is quite interesting to see that defect density is higher while UML LoD is lower for BEHEERNET even though no correlation is found between the two variables. This suggests having a look at how the UML models are used in the implementation. For BEHEERNET, a very poor design-code correspondence (see Section 9.1) indicates that the UML models are not well used in the implementaion. It is not hard to imagine that finding the correlation between UML LoD and defect density can be very difficult if the UML models do not have enough impact on the implementation. Besides, for BEHEERNET, defect density in the implemetation is likely to be affected by other confounding factors, such as developers' experience, adopted development process, and the quality of UML models like correctness. Developers for BEHEERNET did not have enough knowledge in understanding the UML models and many times they just implement the code without knowing whether it is the correct way. The waterfall approach was taken as the development process which makes it difficult to find the defects at an early stage, and some defects were introduced because of the errors made at the very beginning. The quality of the UML models was not good either for BEHEERNET. This can be seen from the purpose of designing the UML models and the modeling style. The purpose of designing the UML models was to give a general idea about some important part of the system and developers can have more freedom during the implementation, therefore they were designed at a very high abstract level. In BEHEERNET, not many designed classes are actually used in class diagrams or sequence diagrams, which makes it difficult to illustrate the connection of a designed class with the others and how a designed class should be implemented in the implementation. The correctness and completeness of a designed class can be low too. Table 9.9 lists all the causes of the difference in correlation analysis mentioned above.

As can be seen from Table 9.9, PARTS has a high design-code correspondence which means that the UML models have been well used in the implementation. Besides, defect density in PARTS is not affected by developers' experience, development process or the correctness and completeness of the UML models that much. Therefore, it is relatively much easier to find the correlation between UML LoD and defect density if it exists. Although the same correlation is not confirmed by BEHEERNET, we still believe that there is a significant and negative correlation between UML LoD using sequence diagram metrics and defect density.

Project	PARTS	BEHEERNET
developer experience	insufficient ask for more detailed models	insufficient simply implemented
adopted process	iteratively four major increments	waterfall one big release
UML models quality	much bigger model size higher quality	smaller model size lower quality
design-code correspondence	higher	lower

Table 9.9: summary of differences between PARTS and BEHEERNET

Chapter 10

Conclusions and evaluations

In this final chapter, research questions introduced in the beginning of this report are answered and the findings in this study are summarized. Then, the threats to the validation of the conclusions are covered. After that, the future work after this study is discussed. Finally, we present some guidelines in creating UML models.

10.1 Answers to the research questions

In this section, the answers to the research questions are given based on the findings in the three case studies: PARTS, RACE and BEHEERNET. However, some questions could not be answered in one or two case studies.

10.1.1 Does the usage of UML models influence defect density in software systems?

The first research question is to see whether the usage of UML models influences defect density in the implementation. We compared the defect density difference between modeled system parts and unmodeled system parts. This analysis was performed in all the three case studies. The consistent result that defect density of modeled system parts is significantly lower than that of unmodeled system parts has been found. This result confirms our believe that UML modeling at an early stage generally helps improve software quality, the reduction of defect density in particular.

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

We expected to find the correlation between the level of detail in UML models and defect density in the implementation. Two types of diagrams, class and sequence diagrams, were used in the case studies. Three correlation analyses were performed including UML level of detail using class diagram metrics and UML LoD using sequence diagram metrics (measured at diagram level and class-instance level). Our hypothesis is that UML classes that are modeled in

a higher level of detail significantly correlate with a lower defect density in the implementation classes.

The correlation analysis between UML LoD using class diagram metrics and defect density could not be performed in the RACE project due to having too few data points. A positive correlation was found in PARTS and BEHEERNET. However, none of the correlations is statistically significant. Therefore, there is no correlation found between UML LoD using class diagram metrics and defect density.

The main finding is that a significant and negative correlation between UML LoD using sequence diagram metrics and defect density is found in the PARTS project. It indicates that modeling UML models into higher level of detail helps achieve lower defect density in the implementation. However, the consistent result could not be obtained in the other two projects. An outlier in the RACE project makes the negative correlation insignificant. For BEHEERNET, a random distribution between the two variables is found. As can be seen from the project comparison between PARTS and BEHEERNET (see Section 9.4), we still believe that the correlation found in PARTS is significant and reasonable.

10.1.3 The contribution of individual metrics to predicting defect density comparison

The idea about analyzing the contribution of individual metrics to predicting defect density is that some measuring metrics probably have stronger correlation with defect density in the implementation and are more powerful in defect density prediction. If the same metrics can be found to be more important than the others in the three projects, more weight can be assigned to them while calculating the level of detail in UML models. However, after comparing the individual metrics' contribution to the three projects, it is not possible to find the common metrics which indicate stronger correlation with defect density or have more predictive power in defect density. This difference might due to different project nature and the UML modeling styles.

Even though no common metrics can be proved to be more predictive in defect density among all the three projects, more attention should be paid to several metrics while designing the UML models. In the PARTS project, the *MsgwithParamRatio* sequence diagram metric measured at diagram level has significant correlation with defect density. Another two sequence diagram metrics, *NonDummyMsgRatio* and *MsgWithGuardRatio*, also have significant correlation with defect density.

10.1.4 The correlation between UML LoD metrics and defect density of individual defect type

We are also interested in whether any metrics suggest stronger correlation with defect density for individual defect type. In total, only two defect types (*data handling* and *logic*) were examined due to having too few data points for the rest of the defect types. After comparing the relevant correlation analyses of the three projects, it is impossible to find the common metrics which have strong correlation with defect density among the projects. However, designing several metrics into more detail might be helpful in reducing certain defect types. For example, the *NonDummyMsgRatio* sequence diagram metric measured at

diagram level has a significant and negative correlation with defect density of the *data handling* defect. For the *logic* defect type, the *MsgWithParamRatio* and *MsgwithGuardRatio* indicate strong correlations with defect density while this correlation is even significant for *MsgWithParamRatio*. All these findings are only confirmed in the PARTS project and the generalization problem might be a threat to the validation of the conclusion.

10.2 Threats to validity

In this section, validity threats of this study are discussed. These threats to validity are presented in their order of importance [WRH+00]: internal validity, external validity, construct validity, and conclusion validity.

10.2.1 Internal validity

The main threat to the internal validity of this study involves our ability to control influences from other factors on the dependent variables (i.e., defect density). Since we are analyzing empirical projects from the real world, it is quite tricky to completely control all confounding factors that might affect defect density like employers' experience, adopted development process, the quality and complexity of the UML models. For example, it is impossible to get a correlation between UML LoD using sequence diagram metrics and defect density in BEHEERNET. After performing a comparison between PARTS and BEHEERNET, factors mentioned above are believed to have a big influence on the defect density which makes it very difficult to analyze the correlation between UML LoD using sequence diagram metrics and defect density.

Another threat is the sampling methods performed in the three projects are different. The whole defects population have been analyzed in RACE due to having small population of recorded defects. The main difference is between PARTS and BEHEERNET. In PARTS, we first did random sampling and got 100 defects, later the sample size increased to 164 by performing the targeted sampling which only focuses on defects having at least one faulty class designed in the UML models. In BEHEERNET, the targeted sampling was first performed and 135 defects were chosen. Another 65 defects were added using random sampling. This sampling method difference makes it hard to interpret some comparison results, such as defect type distribution, defect density comparison. The biggest problem is that whether any of these sampling methods can really represent the whole population of the project. If not, the conclusions we got from this study might be violated.

10.2.2 External validity

External validity threats concern limitations to generalize the results of a study to a broader industrial practice. Although we performed three case studies, except the first research question, the other three questions were only answered by one project: PARTS. We might need more case studies which differ from PARTS only in terms of modeling detail while keeping the rest as similar as possible, such as project characteristics, the UML model size, the quality of UML models, and design-code correspondence.

10.2.3 Construct validity

The threat to construct validity is that typing defects remained a subjective task. Sometimes, it is even hard to decide which defect type a defect should belong to. In order to reduce the uncertainty of defect typing decisions, we regularly discussed problematic defects and randomly checked some defect types to see the accuracy of the decisions. Since the last research question depends strongly on the chosen defect types, the analysis result might be threatened by this subjective defect typing process.

10.2.4 Conclusion validity

Conclusion validity is the degree to which conclusions we reach about relationships in our data are reasonable. This validity threat includes statistical power, assumption of statistical test, and reliability measures. In this study, one threat is whether the calculation of level of detail in UML modeling actually represents the level of detail in UML modeling. Improving the ability of the UML LoD aggregate to reflect the nature of the level of detail in UML modeling can be a future work.

10.3 Future work

In the future, more research can be done in the following directions:

- More case studies are needed. Although we already performed three case studies in this study, too many differences in terms of project characteristics, UML model size, the quality of the UML models, and design-code correspondence makes it difficult to generate comparable analysis results. Therefore, in selecting future cases for analysis, cases which only differ with PARTS in terms of modeling detail are preferred. Factors which might influence the dependent variable (i.e., defect density) should be kept as similar as possible.

- Measuring LoD in a better way. This is also a threat to the validity of the conclusion mentioned before. Two improvements can be investigated. Research can still be done to determine what would be the best way to combine individual LoD values to one aggregate value by assigning weight to different metrics. Although this is also a research question in our study, no consistent results in individual metrics' influence on defect density can be obtained from the three projects. This is probably caused by differences in the nature of the three projects. Therefore, a new case study is needed and the comparable results might be obtained to help determine how individual metrics should be combined in creating one UML LoD aggregate. Research can also be done to find other factors which can help calculate the UML LoD in a better way. for instance, normally there are two types of diagram flow used in sequence diagrams, basic flow and alternative flow. Basic flow is used to illustrate the whole behavioral process in general, while alternative flow only highlights a certain part of the basic flow. Sometimes, a faulty class can be designed in several sequence diagram with different flow types. which flow type represent the LoD of that

faulty class better and how to add this information to the LoD aggregate calculation might be interesting questions.

- Further research can be done at which level of detail UML models should be developed. A pattern of designing level of detail in UML models is found while comparing PARTS and BEHEERNET (see Table 9.7). However, the characteristics for each level were not be able to analyze due to having too few data points. More research can be done to analyze this pattern and distinguish the characteristics of each level. Further, at which level of detail UML models should be developed can be very helpful to guide UML modeling in the software development. Is it the case that the more detailed a UML model is, the lower defect density in the implementation? Or, the level of detail of UML models should be developed at a certain level, otherwise the defect density in the implementation will increase again?

- Investigating on the influence of UML modeling style to defect density. While comparing the PARTS and BEHEERNET projects, UML modeling style was indicated to have a influence on defect density in the implementation (see Section 9.3.1). Many research can be done to see whether there is a correlation between UML modeling style and the quality of the implementation. Furthermore, questions like which kind of modeling style is better for improving the software quality can be considered.

10.4 Guidelines for applying UML

After looking at the results from our research, some guidelines for applying UML in software development process are given:

- The usage of UML modeling has positive influence on improving the quality of the implementation is discussed in this study. System parts which are modeled have lower defect density than those which are not modeled. This suggests the developers and the architects of a project should pay enough time in designing UML models. Therefore, software maintenance cost can be reduced by providing means to understand complex problems and develop solutions to them before the implementation phase. Many unnecessary problems that are found in the later phases (such as code implementation, testing, deployment and maintenance) can be prevented well in advance to reduce fixing costs.

- Designing UML models into a certain level of detail which matches the developers' experience. If UML models are used as the basis of the implementation, UML models are useless if the level of modeling detail is too high or abstract for the developers. This is actually the case for one of the case studies in our research. Most developers of that project do not have sufficient knowledge in understanding the UML models created by the architects. Mostly, they only implement the code without understanding the models, or sometimes they even do not look at the UML models. The benefits of using UML models could not be paid off at the end of that project and lots of time were spent on bug fixing. Therefore, we strongly recommend that before determining the level of detail at which UML models

will be created, architects should have a better idea about the developers' experience in using UML models.

- Spend considerable time on the quality of the UML models such as correctness and completeness. It is useless for improving level of detail in UML models if the quality of the UML models is very poor. More defects can be introduced if the UML models themselves are poorly designed or incomplete. Besides, the UML models should allow to be updated based on the new requirements or changed requirements during the development. Therefore, developers do not have to follow the old UML models which are not valid any more.

Bibliography

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

[BB01] B. Boehm and V.R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[BBM96] V.R. Basili, L. Bri, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22:751–761, 1996.

[BLDC06] B.D. Bois, C.F.J. Lange, S. Demeyer, and M.R.V. Chaudron. A qualitative investigation of uml modeling conventions. In *MoDELS Workshops*, pages 91–100, 2006.

[Boa] International Software Testing Qualification Board. Standard glossary of terms used in software testing. available at *http://www.istqb.org/downloads/glossary-1.2.pdf*.

[But97] G. Butler. Quality and reuse in industrial software engineering. In *In Proceedings of Asia-Pacific Software Engineering Conference and International Computer Science Conference*, pages 3–12. IEEE Computer Society Press, 1997.

[CBC+92] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.Y. Wong. Orthogonal defect classification-a concept for in-process measurements, 1992.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CKC91] R. Chillarege, W.L. Kao, and R.G. Condit. Defect type and its impact on the growth curve, 1991.

[Dea] J. Deacon. Model-view-controller (mvc) architecture. available at *http://www.jdl.co.uk/briefings/MVC.pdf*.

[dqmt] SDMetrics: The UML design quality metrics tool. Tool website. available at *http://www.sdmetrics.com*.

[DW51] J. Durbin and G.S. Watson. Testing for serial correlation in least squares regression. *Biometrika*, 38:159–178, 1951.

[Fla] B. Flaton. Exploring the effect of uml modeling on software quality. Master Thesis supervised by Dr. M.R.V. Chaudron and Ir. F. Buve, Department of Mathematics and Computer Science – Eindhoven University of Technology, 2008.

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

[Har92] D. Harel. Biting the silver bullet: Toward a brighter future for system development, 1992.

[Hov06] S.E. Hove. The impact of uml documentation on software maintenance: An experimental evaluation. *IEEE Trans. Softw. Eng.*, 32(6):365–381, 2006.

[KK77] D.J. Krus and P.H. Krus. Lost: Mccall's t scores: Why? *Educational and Psychological Measurement*, 37(1):257–261, 1977.

[Kru] P. Kruchten. What is the rational unified process ? available at *http://www.ibm.com/developerworks/rational/library /content/RationalEdge/jan01/WhatIstheRationalUnifiedProcessJan01.pdf*.

[Kru95] P. Kruchten. Architecture blueprints - the "4+1" view model of software architecture. In *TRI-Ada Tutorials*, pages 540–555, 1995.

[LC06] C.F.J. Lange and M.R.V. Chaudron. Effects of defects in uml models: an experimental investigation. *IEEE Trans. Softw. Eng.*, 32(6):365–381, 2006.

[LCM06] C.F.J. Lang, M.R.V. Chaudron, and J. Muskens. In practice: Uml software architecture and design description, 2006.

[NFC08] A. Nugroho, B. Flaton, and M.R.V. Chaudron. An empirical analysis of the relation between level of detail in uml models and defect density. *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 5301 of LNCS:600–614, Springer-Verlag, 2008.

[Sel03] B. Selic. The pragmatics of model-driven development. *IEEE softw.*, 20(5):19–25, 2003.

[sou] sourceforge. c and c++ code counter. available at *http://sourceforge.net/projects/cccc*.

[Wika] Wikipedia. Ibm rational clearcase. available at *http://en.wikipedia.org/wiki/Rational_ClearCase*.

[Wikb] Wikipedia. Ibm rational clearquest. available at *http://en.wikipedia.org/wiki/Rational_ClearQuest*.

[Wikc] Wikipedia. Ibm rational rose xde. available at *http://en.wikipedia.org/wiki/Rational_Rose_XDE*.

[WRH+00] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

List of Figures

List of Tables

Appendix A

Database design

In this chapter, the design of the database used in this study is described. This database is mainly used for storing all information useful for statistical analysis. The database schema is listed in Figure A.1 and explanations on several important tables are given.

The DBMS used in this study is MySQL. The main reason for this is due to our familiarity with setting up databases with it and making them available online, through a web interface. According to the schema, some important tables used in the queries are described below:

1. **c_metrics**: all information regarding implementation classes in stored in this table. '*id*' is frequently used for connecting design classes in the UML models to faulty classes found in the implementation. Some other attributes are used for determining the class diagram metrics' values based on the definitions, such as *NumAttr*, *NumOps*, *OpsWithParam* and *OpsWithReturn*. Meanwhile, '*sloc*' provides the source lines of code for each implemented class especially the faulty classes.

2. **faultyclasses**: all classes related to defects are listed in this table. These are the classes examined in this study for which the correlation between UML models and defect density is analyzed. '*c_cid*' is used to set up the connections with other tables like **c_metrics**, **classcorrespondence**, **classincd** and **classinsequence**.

3. **defects**: all information about defects reported in a project is listed here. Most of the content is filled automatically by copying data from the bug tracking tool's database, while the '*type*', '*treated*' and '*analysis_dationale*' fields are filled during the manual defect typing step (see Section 5.5.2).

4. **d_classes** and **d_sequences**: all information needed for calculating design metrics in the UML models are stored in these two tables. **d_classes** stores information of class diagrams while **d_sequences** stores sequence diagrams' information. All these are useful for calculating metrics' values. Note that the raw data is stored here rather than the ratios used in the analysis.

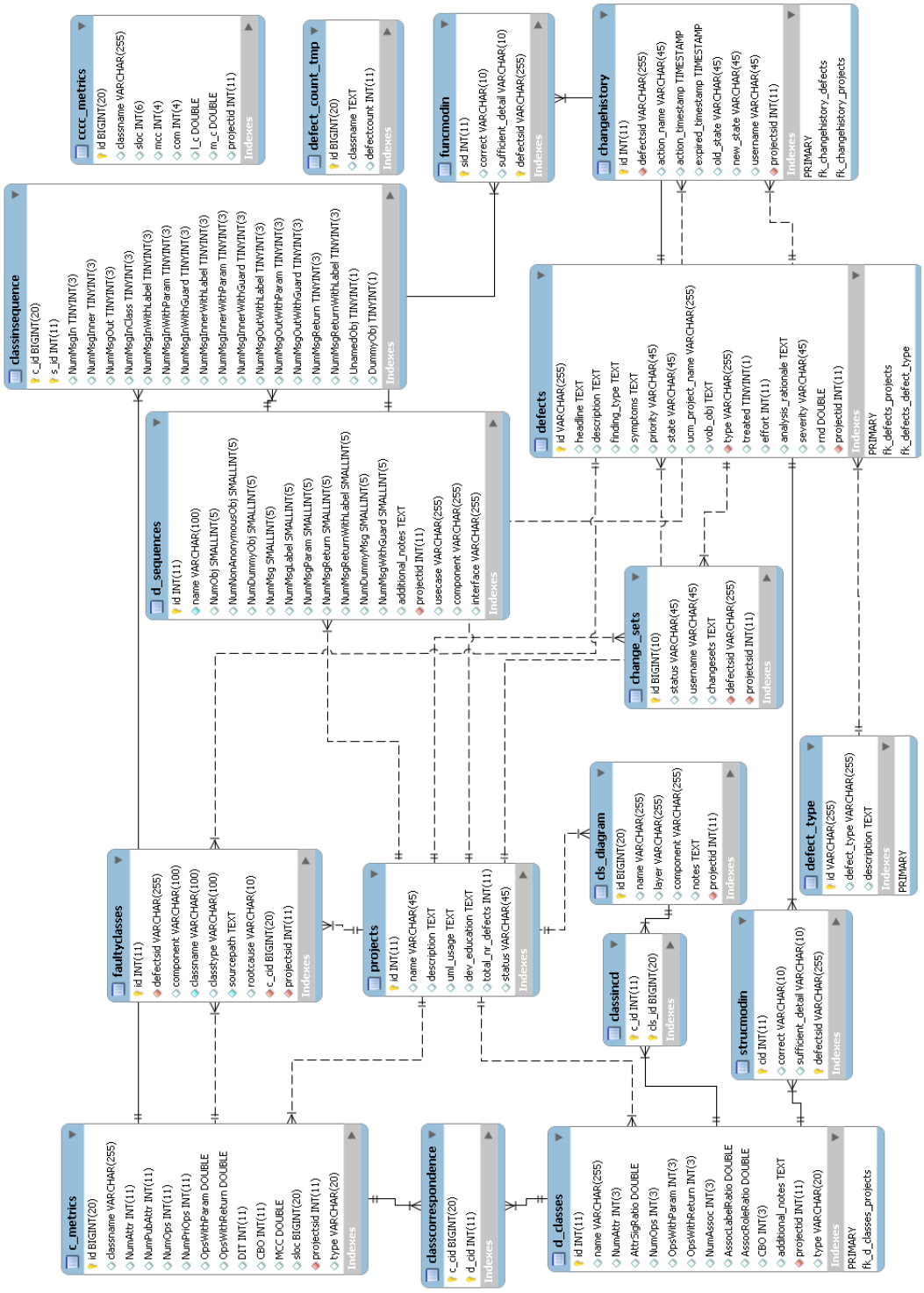

Figure A.1: Database schema

5. **classcorrespondence** and **classinsequence**: these two tables are used to link both diagram types to the implementation classes. The first table is automatically filled by performing automatic matching based on class names stored in the **c_metrics** and **d_classes** tables. Instead, filling in the **classinsequence** table was done manually and most fields contain the metrics' values based on class-instance level.

6. **funcmodin** and **strucmodin**: these two tables are for relating defects to parts of the design efficiently. A '*correct*' field contains a correctness value which indicates whether the functionality or structure related to the defect was correctly designed or not.

Appendix B

Performed queries

In this appendix, several important queries are listed to illustrate how we got the data for the statistical analysis on the analysis database listed in AppendixA.

B.1 Faulty classes modeled in different ways

Several queries were used to model faulty classes in different ways, such as classes modeled as design classes in UML models, faulty classes modeled in sequence diagrams, and classes not modeled at all. The detailed profile can be found in Table 6.3 as an example. At the same time, queries listed here are based on the PARTS project.

As can be seen from listing B.1, information such as the faulty classes' names, KSLoC, defects count and defect density are generated. Defect types like user interface related, non-defect and undetermined are disregarded. This query was performed several times while only changing the last two *where* clauses to select faulty classes modeled in different ways.

```
SELECT substring_index(fc.classname,'.',1) as ModeledFaultyClasses,
      c.sloc/1000 as KSLoC,
      count(fc.classname) as NrOfDefects,
      coalesce( count(fc.classname)/ (c.sloc/1000), 0 ) as
          Defectdensity
FROM c_metrics c, faultyclasses fc ,defects df
WHERE c.id = fc.c_cid AND
    fc.defectsid = df.id AND
    df.projectid = 4 AND
    df.treated = 1 AND
    df.type NOT IN(
      'ui-static' ,
      ' ' ,
      'ui-nav' ,
      'undetermined') AND
    ( fc.c_cid IN( SELECT cor.c_cid FROM classcorrespondence cor) OR
      fc.c_cid IN( SELECT distinct cis.c_id
            FROM classinsequence cis,d_sequences
            WHERE d_sequences.projectid=4 AND
                cis.s_id = d_sequences.id
           )
    )
```

```
GROUP BY fc.c_cid
ORDER BY substring_index(fc.classname,'.',1)
```

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

B.2 UML Level of detail

In total, three queries were used to generate the UML LoD based on class diagram and sequence diagram metrics. Instead of calculating the UML LoD aggregate directly from the queries, we first obtain the values of all the metrics since it is easier for the interpretation which might be necessary in the later phase. The queries are listed as follows:

```
SELECT fc.classname as classesModeledinCD,
 coalesce(dc.NumAttr/c.NumAttr,0) as NumAttrRatio,
 dc.AttrSigRatio as AttrSigRatio,
 coalesce(dc.NumOps/c.NumOps,0) as NumOpsRatio,
 coalesce(dc.OpsWithParam/c.OpsWithParam,0) as OpsWithParamRatio,
 coalesce(dc.OpsWithReturn/c.OpsWithReturn,0) as OpsWithReturnRatio
    ,
 dc.AssocLabelRatio as AssocLabelRatio,
 dc.AssocRoleRatio as AssocRoleRatio
FROM c_metrics c, faultyclasses fc,
 defects df, d_classes dc, classcorrespondence cor
WHERE c.id = fc.c_cid AND
 fc.defectsid = df.id AND
 df.projectid = 4 AND
 df.treated = 1 AND
 df.type NOT IN(
 'ui-static',
 '',
 'ui-nav',
 'undetermined') AND
 fc.c_cid IN ( SELECT cor.c_cid
        FROM classcorrespondence cor, classincd inc
        WHERE cor.c_cid = inc.c_id ) AND
 c.id = cor.c_cid AND
 cor.d_cid = dc.id
GROUP BY fc.c_cid
ORDER BY fc.classname asc
```

Listing B.2: Query for correlation analysis between LoD_{CD} and defect density

```
SELECT substring_index(cm.classname,'/',-1) as classesModeledinSD,
 sum(coalesce(NumNonAnonymousObj/NumObj,0)) as NumNonAnonymObj,
 sum(coalesce((NumObj-NumDummyObj)/NumObj,0)) as NonDummyObj,
 sum(coalesce(NumMsgLabel/NumMsg,0)) as MsgWithLabel,
 sum(coalesce((NumMsg-NumDummyMsg)/NumMsg,0)) as NonDummyMsg,
 sum(coalesce(ds.NumMsgReturnWithLabel/ds.NumMsgReturn,0)) as
     ReturnMsgWithLabel,
 sum(coalesce(NumMsgWithGuard/NumMsg,0)) as MsgWithGuard,
 sum(coalesce(NumMsgParam/NumMsg,0)) as MsgWithParam,
 count(cm.id) as NumberofSD
FROM classinsequence cs LEFT JOIN d_sequences ds
 ON (cs.s_id = ds.id) LEFT JOIN c_metrics cm
 ON (cs.c_id = cm.id)
WHERE cm.id IN (
```

```
 SELECT fc.c_cid
 FROM faultyclasses fc, defects df
 WHERE fc.defectsid = df.id AND
  df.projectid = 4 AND
  df.treated = 1 AND
  df.type NOT IN(
   'ui-static',
   '',
   'ui-nav',
   'undetermined')
 )
GROUP BY cm.id
ORDER BY substring_index(cm.classname,'/',-1) asc
```

Listing B.3: Query for correlation analysis between LoD_{SD} based on diagram level and defect density

```
SELECT substring_index(cm.classname,'/',-1) classesModeledinSD,
 sum(coalesce((cis.NumMsgInWithLabel+cis.NumMsgInnerWithLabel+
      NumMsgOutWithLabel)/
 (NumMsgIn+NumMsgInner+NumMsgOut),0))as SMsgWithLabelRatio,
 sum(coalesce((cis.NumMsgInWithParam+NumMsgInnerWithParam+
      NumMsgOutWithParam)/
 (NumMsgIn+NumMsgInner+NumMsgOut),0)) as SMsgWithParamRatio,
 sum(coalesce((cis.NumMsgInWithGuard+cis.NumMsgInnerWithGuard+cis.
      NumMsgOutWithGuard)/
 (NumMsgIn+NumMsgInner+NumMsgOut),0)) as SMsgWithGuardRatio,
 sum(coalesce(cis.NumMsgReturnWithLabel/cis.NumMsgReturn,0)) as
      SReturnMsgWithLabelRatio,
 sum(coalesce(cis.NumMsgInClass/(NumMsgIn+NumMsgInner),0)) as
      SNonDummyMsgRatio,
 sum(coalesce(1- cis.UnamedObj,0))as SNonAnonymObj,
 sum(coalesce(1- cis.DummyObj,0))as SNonDummyObj,
 count(cis.c_id) as NumberofSD
FROM c_metrics cm, classinsequence cis
WHERE cis.c_id = cm.id AND
 cm.id in (
  SELECT distinct c_cid
  FROM faultyclasses a, defects b
  WHERE a.defectsid=b.id AND
   b.projectid=4 AND
   b.type not in(
    'ui-static',
    '',
    'ui-nav',
    'undetermined') AND
   cm.projectsid=4
GROUP BY cis.c_id
ORDER BY substring_index(cm.classname,'/',-1)
```

Listing B.4: Query for correlation analysis between LoD_{SD} based on class-instance level and defect density

In class diagrams the LoD value is calculated by directly adding up the values of all the metrics generated from the query, since the correspondence between a design class in UML and an implementation class is a one-to-one relationship. However, one implementation class can appear in more than one sequence diagram (one-to-many relationship). Therefore, for each faulty class, we add up the values of all the sequence diagrams metrics related to this faulty class and the number of sequence diagrams related to this faulty class can also be obtained. Later, the calculation of UML LoD of this faulty class can be easily done in

Excel by dividing the summation of the values of all the metrics by the number of sequence diagrams related to this faulty class.

Appendix C

Statistical tests

Statistical tests used in this study are listed in this chapter. all tests were performed using version 13 of the SPSS tool [Ana]. Since some tests were already explained during the analyses in the case studies discussed in the previous chapters (i.e., **Pearson's correlation test**, **Spearman's correlation test**), they are not explained in detail.

C.1 Kolmogorov-Smirnov & Shapiro-Wilk tests

These two tests are used for testing whether a distribution is normal. They compare the scores in the sample to a normally distributed set of scores with the same mean and standard deviation. If the test is non-significant (significance $p \geq 0.05$) it tells us that the distribution of the sample is not significantly different from a normal distribution (i.e., it is probably normal). If, however, the test is significant ($p \leq 0.05$) then the distribution in question is significantly different from a normal distribution (i.e. it is non-normal). The difference between the two tests is that **K-S** test is used when the number of samples is larger than 50, otherwise, **Shapiro-Wilk** test should be performed.

C.2 The independent t-test

The independent t-test is used in situations in which there are two experimental conditions and different participants have been used in each condition, for instance, defect density difference between modeled and unmodeled system parts. The test compares the two means to see whether the difference is significant or not. Besides, the data samples of the two groups should be independent, other two conditions should be met. First of all, both groups should be normally distributed since it is a parametric test. The second condition is *Homogeneity of Variance* which is checked by **Levene's test** listed in the analysis results of the independent t-test. If the *Sig.* value for this test is less than .05 then the condition has been violated and *Sig.* value for comparing the means of the two groups should be checked under the condition labeled *Equal variances not assumed*. Otherwise, the *Sig.* value under the condition labeled *Equal variances assumed* should be checked. If the significance is less than .05 then the means of the two groups are significantly different.

C.3 One-way ANOVA

ANOVA is a parametric test used for analyzing situations in which there are several independent variables. In these situations, ANOVA explains how these independent variables interact with each other and what effects these interactions have on the dependent variable. The reason why ANOVA is used instead of performing several t-tests to compare all combinations of groups that have been tested is explained in [Ana]. The conditions under which ANOVA is reliable are the same as for all parametric tests based on the normal distribution (see Section C.2). The ANOVA test is rather complex and we will only explain the procedures used in our study. *Post hoc* tests designed for comparing all different combinations of the treatment groups were performed. The choice of the comparison procedure depends on the exact situation and several general guidelines are provided in [Ana].

C.4 Pearson's and Spearman's correlation coefficient

These two tests are used for correlation analysis. **Pearson's correlation** is a parametric test which requires an accurate measure of the linear relationship between two variables. On the other hand, **Spearman's correlation coefficient** is a non-parametric statistic test and can be used when the data has violated parametric assumptions such as non-normally distributed data. For both tests, a significant correlation is indicated when the *Sig.* value is less than .05.