The effects of UML modeling on the quality of software

Nugroho, A.

Chapter 4

The Effects of Using UML on Defect Density and Defect-fixing Effort

The contribution of modeling in software development has been a subject of debates. The proponents of model-driven development argue that big upfront modeling requires substantial investment, but it will payoff later in the implementation phase in terms of increased productivity and quality. Other software engineers perceive modeling activity as a waste of time and money without any real contribution to the final software product. In this chapter we explore the benefits of using UML in an industrial project. In particular, we report on an empirical investigation on the impact of UML modeling on the quality of the implementation (measured in defect density) and on the effort spent on fixing defects.

4.1 Introduction

The proponents of model-driven software development believe that the use of modeling will deliver benefits in software development [58, 87, 116]. The benefits, which can materialize in terms of improved quality and productivity, are based on some key assumptions. First, modeling provides techniques to design solutions to the problem domain that need to be addressed by software systems. Second, having modeled a system in a systematic manner assures that it has gone through a technical review process, thus ensuring the correctness of the implementation. Finally, the use of modeling ensures that design decisions captured in the software models are well documented. The availability of good documentation is

This chapter is an extended version of the paper entitled "Evaluating the Impact of UML Modeling on Software Quality: An Industrial Case Study", published in the proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS) 2009.

believed to minimize loss of information and misinterpretation in communicating decisions taken during development. Additionally, it might increase productivity during software maintenance [11].

On the other hand, many software engineers have started to question the role of modeling in software development. This phenomenon is marked by the emergence of a software development methodology that advocates spending more time and effort on creating and testing working software as early as possible rather than investing on upfront modeling [63]. Creating software models is seen as an expensive investment (both effort and tools) without a clear contribution to quality or productivity. The role of software models as a mean of communicating design decision is substituted by intensive communication and collaboration amongst team members and with the customers. This so-called agile methodology is believed to increase agility of projects in dealing with the-presumably-inevitable uncertainties in software development.

This chapter is not going to discuss evidence about the superiority of one methodology over the other. Rather, this chapter aims to answer whether there is empirical evidence that supports the assumed benefits of modeling in software development. Investigating the benefits of modeling is important because they are unclear or, at best, weakly supported by empirical evidence.

Therefore, in this chapter we evaluate the impact of modeling in a real software project. More specifically, we investigate the effect of UML modeling on defect density (defects per source lines of code) in software modules and on the effort spent on fixing defects. The results of this study show that the use of UML modeling potentially reduces defect density and defects-fixing effort.

The rest of this chapter is organized as follows. In section 2, we discuss some related works. In section 3, the design of this study will be discussed. Section 4 discusses the case study and the results of the analyses. In section 5 and 6, we discuss the hypothesis testing. Section 7 discusses the interpretation of the results, their implications, and limitations. Finally, in section 8 we outline some conclusions and future work.

4.2 Related Work

To the best of our knowledge, there has not been any research that investigates the use of UML modeling and its relation to the quality of the final implementation. Plenty of works, however, have been focused on investigating the impact of using certain styles, rigor, and UML diagram types on model comprehension and software maintenance.

Many studies that investigate the impact of modeling styles on model comprehension have been looking at the use of stereotypes. The work of Staron et al. for instance, suggests that UML stereotypes with graphical representation improve model comprehensibility [120]. Ricca et al. also found that stereotypes have a positive impact on diagram comprehension [111]. However, this finding was particularly true for inexperienced subjects—the impact was not statistically significant for experienced subjects. Genero et al. studied the influence of using stereotypes in UML sequence diagrams on comprehension [53]. While this study

revealed no significant impact, it suggested that the use of stereotypes in sequence diagrams was favored to facilitate comprehension. Another study was conducted by Cruz-Lemus et al. to evaluate the effect of composite states on the understandability of state-chart diagrams [40]. The authors stated that the use of composite states, which allows the grouping of related states, improves *understandability efficiency* when reading state-chart diagrams. Nevertheless, subjects' experience with state-chart diagrams was considered as a prerequisite to gain the improved understandability.

A previous study that looked into the formality of UML models and its relation with model quality and comprehensibility is from Briand et al. [27]. In their experimental study, Briand et al. investigated the impact of using OCL (object constraint language) in UML models on defect detection, comprehension, and impact analysis of changes. Although the overall benefits of using OCL on the aforementioned activities are significant, they have found that the benefits for the individual activities are modest.

Other studies investigated the effect of using different UML diagram types (e.g., sequence and collaboration diagrams) on model comprehension. The work of Otero and Dolado for instance, looked into three UML diagrams types, namely sequence, collaboration, and state diagrams, and evaluated the semantic comprehension of the diagrams when used for different application domains [106]. A similar study comes from the work of Glezer et al. They evaluated the comprehensibility of sequence and collaboration diagrams, and finally concluded that collaboration diagrams are easier to comprehend than sequence diagrams in real-time systems [56]. Another study conducted by Torchiano [124] investigated the effect of object diagrams on system comprehensibility. In two of the four systems used in the experiment, the use of object diagrams to complement class diagrams was found to have significant effects on the comprehensibility of the systems.

A previous work that is closely related to this study is conducted by Arisholm and Briand [11]. In the paper the authors evaluate the impact of UML documentation on software maintenance in a controlled experiment. The results show that for complex tasks and after certain learning process, the availability of UML documentation may result in significant improvements in terms of functional correctness of changes and the design quality of the changes.

Different from most of the aforementioned works, in this chapter we analyze whether the use of modeling, represented using UML, has any effect on the quality of the final implementation—measured in *defect density*. Furthermore, this work complements the experimental study of Arisholm et al. [11] in that we further investigate the effect of UML modeling on the effort spent on fixing defects in an industrial case study.

4.3 Design of Study

In this section, we discuss underlying aspects of this study, which include study objective, hypothesis formulations, and measured variables.

4.3.1 Objective

Using the GQM template [130], we can formulate the goal of this study as follows:

Analyze the use of UML modeling
for the purpose of investigating its effect
with respect to defect density of Java classes and defect-fixing effort
from the perspective of the researcher
in the context of an industrial Java system

As stated in the above template, this study essentially attempts to investigate the effects of UML modeling on downstream software development, and we focus on the following two questions:

1. What is the effect of modeling system structure on the quality of the implementation?
2. What is the effect of modeling system behaviors on the effort spent on fixing defects related to those behaviors?

4.3.2 Hypothesis Formulation

In the first research question, we aim to investigate the effect of UML modeling on the quality of software. We use defect density as a measure of software quality, and assess whether there is a significant difference in defect density between software modules that are modeled using UML and modules that are not modeled at all.

We have learnt in Chapter 3 that complex or critical components are more often included in a UML model, and thus they are expected to have higher defect density. However, regardless of their complexity or criticality modeled components presumably have the following characteristics. First, they are generally well thought of and better designed. Second, they are generally well described and well documented. Finally, all else being equal, modeled components should be easier to implement than the not modeled ones.

Having the above assumptions in mind, it is interesting to investigate whether such benefits materialize in terms of improved implementation correctness. We translate the first research question into the following hypothesis:

Hypothesis 1

- **Null hypothesis ($\mathbf{H1}_{null}$)**:
  There is no significant difference in defect density between implementation classes that are modeled using UML and those that are not modeled.
- **Alternative hypothesis ($\mathbf{H1}_{alt}$)**:
  The use of UML significantly reduces defect density of implementation classes.

We formulate the first hypothesis as a one-tailed hypothesis because we have a specific assumption about the direction of the cause-effect relationship between the independent and dependent variables: the use of UML reduces defect density.

In the second research question, we aim to evaluate whether the use of UML modeling significantly reduces the effort spent on fixing defects. This question is based on assumptions quite similar to those of the first hypothesis—that is, we expect that modeled behaviors are well thought of, better designed, and well documented. Using a UML model as a guide, software engineers can quickly identify the root cause of a problem in a system. Additionally, the UML model helps software engineers in identifying linked components that need to be fixed or changed relatively quickly. As a result of a faster defect assessment and impact analysis, fixing defects related to behaviors that are modeled is also expected to be much faster than in those that are not modeled.

Similar to the first hypothesis, we formulate the second hypothesis as a one-tailed hypothesis:

Hypothesis 2

- **Null hypothesis ($\mathbf{H2}_{null}$)**:
  There is no significant difference in fixing effort between defects related to modeled behaviors and defects related to unmodeled behaviors.
- **Alternative hypothesis ($\mathbf{H2}_{alt}$)**:
  Defects related to modeled behaviors require significantly less effort to fix than defects related to unmodeled behaviors.

Notice the difference in the unit of analysis between hypothesis 1 and 2. The unit of analysis of hypothesis 1 is *implementation classes*, and we assess the effect of using UML for modeling these classes. On the other hand, the unit of analysis of hypothesis 2 is *defects*, in which we examine the effect of using UML for modeling system behaviors or functionality related to the defects.

4.3.3 Measured Variables

In Table 4.1 we provide all variables measured in this study. In the following passages we discuss the measured variables in further details.

Measured Variables in Hypothesis 1

The independent variable of Hypothesis 1 is the use of UML modeling. The use of modeling is defined as the presence or availability of UML diagram(s) that describe a given implementation class (i.e., Java class). The use of UML modeling is measured in a nominal scale with two categories: *Modeled Classes(MC)* and *Not Modeled Classes (NMC)*. Two UML diagram types are considered, namely class diagram and sequence diagram. As such, the modeled classes can be of the following categories: 1) modeled in class diagrams; and 2) modeled

Table 4.1: Measured Variables

	Independent Variable	Dependent Variable	Covariates
Hypothesis I	The use of UML	Defect density	Code complexity Code coupling
Hypothesis II	The Use of UML	Defect-fixing Effort	Total modified files Total coupling Total complexity Total size (KSLoC) Defect priority Total person involved

in class- and sequence diagrams (as instances/objects of classes). We hereafter refer to the independent variable as *UMLforCLASS*.

The dependent variable of Hypothesis 1 is the quality of the final implementation, which is measured in defect density. Defect density of an implementation class is determined by the number of defects found in that class (defect-count) divided by the class size (in kilo SLoC). Defect-count, in this respect, is measured from the number of times a class is modified to solve distinct defects. Hence, if a class is modified five times to solve the same defect, it would be considered as having *only* one defect-count. We hereafter refer to defect density as *DDENS*.

In addition to the independent and dependent variables, we selected two significant factors that might confound the main factor of the first hypothesis, namely code complexity, which is measured using the McCabe's cyclomatic complexity metric (MCC) [84], and coupling between objects (CBO) [35]. MCC and CBO metrics are well-known for their significant relations to class fault-proneness [71, 121]. Considering their potential influence on defect density, we consider MCC and CBO as co-factors (covariates) and control their effects on defect density. Having controlled these confounding factors, we expect to see the true effect of using UML modeling on the defect density of system classes.

Measured Variables in Hypothesis 2

The independent variable in hypothesis 2 is, similar to the first hypothesis, the use of UML modeling. However, as mentioned earlier, in the second hypothesis we are concerned with the use of UML for modeling system behaviors or functionality that are found to be faulty (defective) during the verification or testing phase. We consider the relation of a defect to a sequence diagram(s) to determine whether it is modeled or not modeled. A defect is said to be modeled if it is caught within a functionality or behavior that has been modeled in a sequence diagram(s). We only consider sequence diagrams because sequence diagram is the UML diagram type that was used most frequently to specify system functionality in the software project that we used for a case study. Thus, the independent variable is measured in a nominal scale with two categories: *Modeled Defects (MD)* and *Not Modeled Defects (NMD)*. We hereafter refer to the independent variable as *UMLforDEFECT*.

The dependent variable in hypothesis 2 is effort to fix defects (hereafter referred to

as FixEffort). FixEffort is defined as the amount of time (hours) that is required to fix a particular defect. More specifically, we consider FixEffort as time duration from the moment a defect is assigned to a particular person until the moment it is declared as solved.

We also identify several confounding factors that we suspect might influence FixEffort:

- **Modified files (ModFile).** Represents the total number of Java files that are modified to solve a particular defect.
- **Total coupling (TCBO).** Represents the sum of coupling between objects (CBO) of Java files that are modified to solve a particular defect.
- **Total complexity (TMCC).** Represents the sum of complexity (McCabe's measure) of Java files that are modified to solve a particular defect.
- **Total size (TKSLOC).** Represents the total size (in KSLoC) of Java files that are modified to solve a particular defect.
- **Defect priority (DPRIO).** Represents the priority scale assigned to defects. Defect priority is determined by a responsible engineer in the project. Its scale is as follows: 1: Resolve Immediately; 2: Give High Attention; 3: Normal Queue; 4: Low Priority.
- **Total persons involved (TPERS).** Represents the total number of people involved in the course of fixing a particular defect.
- **Defect type (DTYPE).** Represents the type of defects. The taxonomy of defect types will be discussed shortly.

Considering the above confounding factors, it is logical to think that FixEffort can be significantly influenced by MoDFile, TCBO, TMCC, and TKSLOC in that the higher the values of these confounding factors, the higher is FixEffort. Additionally, DPRIO is also essential because it might correspond to the speed of fixing defects. TPERS might either increase or decrease the speed of fixing defects depending on the effectiveness of the collaboration amongst the persons involved. Finally, the types of defects might have significant influence on defect-fixing effort.

4.4 Case Study

The software project that we selected for the case study should meet two conditions. First, the projects must use UML modeling to certain extent. Further, the UML models should be used for guiding the implementation and are modeled in machine-readable forms (e.g., utilizing UML CASE tools). We also required the UML CASE tools to have an XMI export facility, which allowed us to export the models to the measurement tool. Second, the project must utilize a bug tracking system with which it is possible to trace back source files that are modified to solve defects. Having selected the projects to be studied, we performed data collection to obtain data of UML models, source code, defect registration, and change sets (source files modified to solve defects). The collected UML data and source code are the

Table 4.2: Project Summary

Projects	# staff	Duration	Off-shored	Status	Model Size	SLoC
IPS	25 people	2.3 years	India	finished	104 UCs 266 classes 34 CDs 341 SDs	152,017

latest version of project data that could be found in the CVS (Concurrent Versions System) repository.

In Chapter 6 and 7 the same case study will be used, thus the same case study descriptions apply. In those chapters, additional information about the case study will be given when necessary.

4.4.1 Project Context

The system under study is an integrated healthcare system for psychiatrists in the Netherlands, hereafter referred to as IPS (not a real name). The project was started in 2002, and it was built as a web service using Java technology. In the IPS project, the RUP (Rational Unified Process) methodology was used, and this project involved off shoring to India. The modeling of the system was done in the Netherlands, while approximately 60 percent of the implementation and testing activities were done in India. For creating the UML models, the project used Rational XDE, and for the version control system and bug tracking system, the project used Rational ClearCase and Rational ClearQuest respectively.

In the IPS project, the UML model was used as an implementation guide, thus the model was created before writing the implementation code. The UML model was created by the designers and was later implemented by the developers in India and the Netherlands. In addition to the UML models, textual specifications were also used to guide the implementation of the system. These specifications are textual and are mainly in the form of detailed use case descriptions. From our observation, most functional requirements generally have corresponding use case descriptions. Additionally, a software architecture document that provides a high level description of the system is available. Hence, regardless of whether certain parts were modeled or not modeled using UML, there exists some textual specifications that describe how the system should be implemented.

Testing of the system consisted of the following test types: unit test, system test, regression test, integration test, user acceptance test, and production acceptance test. While the user acceptance test and production test were done by the customer, the rest of the tests were the responsibility of the project team. The basis for the tests is derived from artifacts such as use case specifications, supplementary specifications, and user interface documentations (e.g., screen designs). Furthermore, testing was done iteratively. For each iteration, the parts of the system that were going to be tested were defined in advance.

When this study is conducted, the IPS project was already finished. The system was used by the client for sometime, but was later abandoned because of business economical

reasons. The characteristics of the project are provided in Table 4.2.

4.4.2 Data Collection and Preprocessing

To obtain data about UML classes and other metrics, the UML model first had to be exported from the UML CASE tools into an XMI format. Using a tool called SDMetrics [4], the XMI file was read and model information, such as classes and other structural diagrams, could then be easily extracted. However, due to a limitation of the UML CASE tool, sequence diagram information could not be exported to XMI. Therefore, we needed to manually inspect every sequence diagram to register the instances/objects of classes that were modeled in sequence diagrams.

The processing of source code was mainly aimed at calculating code metrics from the implementation classes. In this study we are mainly interested in the size, coupling, and complexity metrics. These code metrics were calculated using an open source tool called CCCC (C and C++ Code Counter), which in fact is also able to calculate metrics from Java files.

Processing defect data mainly involved two steps. The first step was to obtain registered defects from the ClearQuest repository and store them in the analysis database. The second step was to obtain change sets. This step was performed automatically using a Perl script that recovers change sets associated with every defect. Because change sets are registered in a ClearQuest textual format and they contain other information, text parsing was performed to mine data of the modified files (note that only Java files were taken into account). Further, defect-count of each Java file was determined based on the frequency it was corrected to solve distinct defects. Java files that were modified to solve defects are hereafter referred to as *faulty classes*.

We employ a relational database to store the above data. This database can be accessed via a web interface to enable remote collaboration for data collection and analysis. Once the data of defects, UML classes, implementation classes, and faulty classes were stored in the analysis database, we could query various information, which include: 1) implementation classes that are modeled, and the diagrams in which they are modeled; 2) implementation classes that are not modeled; 3) code metrics of the implementation classes; 4) defect density of the implementation classes—if they were found to be faulty during testing.

To test the second hypothesis, we needed to collect the change history of every defect. Change history essentially records the states or phases in solving defects. The states in fixing defects are defined in the defect solving procedure.

Figure 4.1 shows an activity diagram depicting the adopted procedure for solving defects. Activities are represented using rounded rectangles, and the arrows indicate transition between activities. The diamonds are decisions (conditional branch), and the arrows connected to them are marked with the conditions. Roles that perform the activities are depicted using *swimlanes* (represented by the rectangles). The initial state in an activity diagram is indicated by the black circle, while the final state is the encircled black circle.

In the simplest path of the procedure, issues are first submitted and examined (an *issue*

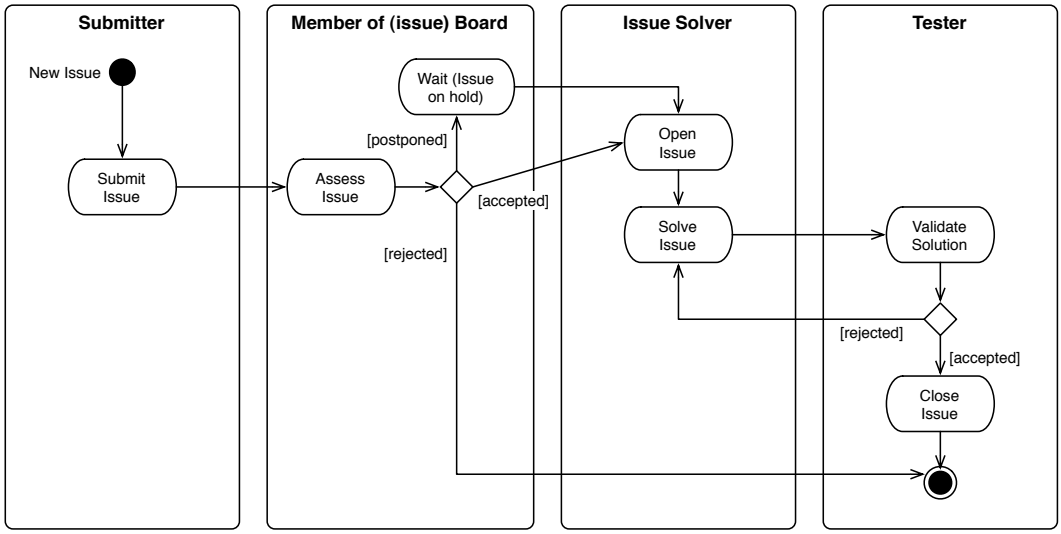


Figure 4.1: Defect solving procedure

refers to a general problem reported during testing). If a reported a issue is accepted as defect (i.e., they deserve fixes), then it is assigned to a developer(s) to be solved. The assigned developer will then open and solve the defect. Solved defects will be retested and closed if they pass the test—otherwise, defects will have to be again fixed. Any of these activities (i.e., whenever they are performed) are registered in the change history, and the information that was available includes the time a certain activity takes place and the person who performs it. Please bear in mind that FixEffort was calculated as time duration from the moment a defect was assigned to a particular person (i.e., activity *Open Issue* in Figure 4.1) until the moment the defect was closed. We exclude earlier activities because they are not related to the main effort in fixing defects.

To test the second hypothesis we also needed to determine whether behaviors that are found to be defective are modeled or not modeled using sequence diagrams. To this aim, for each defect we performed a careful analysis to check whether a defect was related to behavior(s) that are modeled using sequence diagrams. This analysis was done manually, by looking at the description of defects. Tracing defects to the sequence diagrams was sometimes a quite straightforward process if information about use case numbers related to defects was mentioned in the defect report. However, because such information was not always available, we frequently needed to carefully determine the context of a particular defect, and subsequently find the corresponding sequence diagrams in which the behavior was modeled. If we could find *at least* one sequence diagram that matched a defect, then the defect is said to be *modeled*, but otherwise it is said to be *not modeled*.

4.4.3 Analysis Methods

The formulated hypotheses essentially aim at comparing two groups of subjects, namely modeled and not modeled implementation classes, and defects. Therefore, in the analysis we used statistical techniques to compare mean difference between groups.

To test the hypotheses, we use the ANCOVA (Analysis of Covariance) test [115] because it would allow us to control the effects of covariates on the outcome variable. However, as we later found out the data set we use to test Hypothesis 1 violated the assumptions of normality. Hence, for testing the first hypothesis we finally decided to use the Mann-Whitney test [83] as the main statistical test. Nevertheless, the ANCOVA test would still be used for the sake of result validation. Furthermore, because we could not use ANCOVA as the main statistical test for testing the first hypothesis, we needed to perform a pair-wise sampling to account for the effect of the covariates (in Section 4.5.2 we discuss the pair-wise sampling in further detail). In testing Hypothesis 2, we use both the Mann-Whitney and ANCOVA test.

In this study we have a specific assumption about the direction of the hypothesis—that is, we hypothesize that the use of UML modeling will reduce both the defect density of classes in the implementation and the effort spent on fixing defects. Consequently, testing of the mean difference between groups will be performed as one-tailed test. Further, in the analyses and hypothesis testing we considered a significance level of 0.05 level ($p \leq 0.05$) to indicate true significance.

4.5 Testing Hypothesis 1: The Effect of UML Modeling on Defect Density

In this section, we discuss the main analysis to test the effect of UML modeling on the defect density. We start by describing some descriptive statistics.

4.5.1 Descriptive statistics

The core part of the IPS system (excluding framework classes) consisted of 812 Java classes. Table 4.3 shows the descriptive statistics of defect density, coupling, complexity, and size of all classes across groups. One notable trend that we can see in Table 4.3 is that MC classes generally have higher complexity, coupling, and size than NMC classes—this confirms the survey result reported in Chapter 3. However, we can also see in the table (the mean value) that there is only a slight difference in defect density between the two class groups. Statistical tests confirm that except for defect density, the differences in complexity, coupling, and size between MC and NMC classes are statistically significant.

It is interesting to note that MC classes, which are generally rank higher in terms of complexity, size, and coupling are in fact having a quite similar defect density as NMC classes. This is particularly true if we consider previous studies that report positive correlations between complexity, coupling, size and module fault-proneness (see for example

Table 4.3: Descriptive statistics of all Java classes

Measures	Not Modeled Classes (NMC)				Modeled Classes (MC)			
	N	Median	Mean	St.Dev	N	Median	Mean	St.Dev
DDENS	638	0.000	0.016	0.032	174	0.000	0.012	0.029
CBO	638	6.000	7.123	6.303	174	11.000	13.459	11.359
MCC	638	1.000	11.747	72.034	174	10.500	26.770	45.932
KSLOC	638	0.039	0.103	0.290	174	0.179	0.312	0.689

in [71, 74, 121]). The results in Table 4.3 raise a question whether modeled classes, which are notoriously more complex, have lower defect density because they are modeled using UML, or because defects in larger and complex classes are more difficult (hidden) to find [48]. This discussion shows that several factors might influence defect density, and thus it is important to identify them and control their effects in order to evaluate the true effect of UML modeling on defect density.

4.5.2 Controlling for the Confounding Factors

In testing the first hypothesis, we consider class coupling and complexity as the main confounding factor because both metrics have been considered influential to class fault-proneness. Ideally, we would use the ANCOVA test to analyze the main effect of a treatment when several confounding factors are accounted for. With this analysis we could control the variance of the confounding factors, hence providing us with a pure effect of the main treatment if there is one. However, because the defect density data set violated the assumption of normal data distribution and transforming the data did not fix the normality problem, we could not rely on ANCOVA for the main statistical test.

An alternative way to do the analysis is to perform a pair-wise sampling in which we selected classes of comparable complexity and coupling, and subsequently used a parametric test, i.e., Mann-Whitney, as the primary test to compare the defect density between groups. However, selecting classes that are comparable in terms of complexity and coupling left us with too few data points for a meaningful statistical test. Therefore, we decided to perform a pair-wise sampling based on coupling, and the effect of complexity would subsequently be assessed using the ANCOVA test.

To obtain classes of comparable coupling, we performed a pair-wise sampling by systematically selecting classes from both NMC and MC that have coupling values from 8 up to 10. This range of coupling values is selected mainly because 1) the range is reasonably small; and 2) within this coupling range we obtained the best proportion of NMC and MC groups (note that we aimed to obtain balanced groups when possible). In other range of coupling, the difference in size between NMC and MC group is too large. The pair-wise sampling has reduced the amount of classes from 812 to 113 Java classes, of which 68 and 45 belong to NMC and MC groups respectively. The CBO values of these 113 classes have a standard deviation value of 0.8, which means coupling values of these classes are very close to the mean value (+/- 0.8). A standard deviation value this small suggests that we have controlled the variance of class coupling to a minimum level.

Table 4.4: Descriptive statistics of the randomly sampled Java classes from the IPS project comparing NMC and MC

Variables	Not Modeled Classes (NMC)				Modeled Classes (MC)			
	N	Median	Mean	St.Dev	N	Median	Mean	St.Dev
DDENS	59	0.002	0.011	0.019	37	0.000	0.003	0.010
CBO	59	9.000	9.000	0.809	37	10.000	9.270	0.902
MCC	59	23.000	41.440	47.656	37	30.000	35.297	36.153
KSLOC	59	0.180	0.267	0.233	37	0.230	0.251	0.184

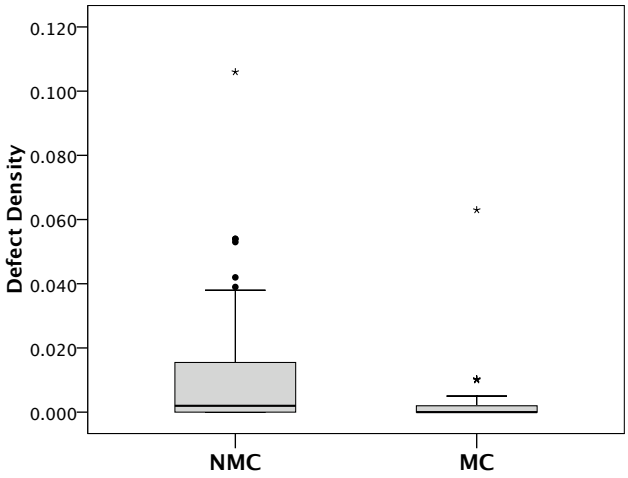


Figure 4.2: Box-plots of defect density in NMC and MC group

4.5.3 Assessing the Effect

The main question we wanted to answer is whether the use of UML helps reduce defect density of software modules in the implementation. In Section 4.5.2 we have discussed how we performed a pair-wise sampling based on class coupling to control its effect on defect density. Therefore, in this section we discuss the main hypothesis testing based on the sampled data set.

To mitigate bias during the pair-wise sampling, we further performed a random sampling on the sampled data set, in which we randomly selected 80 percent of the 113 Java classes for the analysis. Having done the random sampling we obtained 96 classes, of which 59 and 37 are NMC and MC classes respectively. Table 4.4 shows the descriptive statistics of these classes. If we look at the mean values in the table, we can see that after coupling is accounted for, NMC classes remained having a higher defect density than MC classes.

Figure 4.2 shows two box-plots that compare defect density between groups. The box-plots show a similar result presented in Table 4.4—that is, the defect density of the NMC group is higher than that of the MC group. We subsequently performed a statistical test

Table 4.5: Mann-Whitney test - Ranks of the measured variables of the NMC and MC groups

Variables	Groups	N	Mean Rank	Sum of Ranks
DDENS	NMC	59	53.95	3183.00
	MC	37	39.81	1473.00
CBO	NMC	59	45.11	2661.50
	MC	37	53.91	1994.50
MCC	NMC	59	48.49	2861.00
	MC	37	48.51	1795.00
KSLOC	NMC	59	47.65	2811.50
	MC	37	49.85	1844.50

Table 4.6: Mann-Whitney test - The significance of differences in the measured variables between the NMC and MC groups

	DDENS	CBO	MCC	KSLOC
Mann-Whitney U	770.000	891.500	1091.000	1041.500
Wilcoxon W	1473.000	2661.500	2861.000	2811.500
Z	-2.704	-1.607	-.004	-.376
Significance	**.003****	.108	.997	.707

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

to assess whether the difference in defect density between the NMC and the MC groups is statistically significant.

Table 4.5 and 4.6 provide the results of the Mann-Whitney test. We used this parametric test because the data set (i.e., defect density variable) violated the assumption of normal data distribution and data transformation could not solve the problem. For the sake of completeness we also provide the results for coupling, complexity, and size measures.

In Table 4.5, we can see that the mean rank of defect density for NMC is higher than that of MC. Because the Mann-Whitney test relies on ranking scores from lowest to highest, the group with the lowest mean rank (i.e., MC) is the one that contains the largest amount of lower defect density. Likewise, the group with the highest mean rank (i.e., NMC) is the group that contains the largest amount of higher defect density. Hence, the results show that classes that are not modeled tend to have higher defect density than the modeled classes.

Table 4.6 provides the actual Mann-Whitney tests. The most important part of the table is the significance value of the tests. We can see from the table that the difference in defect density is significant at 0.01 level ($p = 0.003$; 1-tailed). Note that none of the other measures are significantly different between the NMC and MC groups. Having obtained these results, we can conclude that, on average, classes that are modeled using UML have a significantly lower defect density than those that are not modeled. Therefore, we could reject the null hypothesis (H_{null}), and confirm the alternative hypothesis (H_{alt}): the use of UML modeling significantly reduces defect density of classes in the implementation.

In addition to the Mann-Whitney test, we performed an ANCOVA test to verify if the results are consistent. Performing an ANCOVA test regardless of the violation of normal-

Table 4.7: Results of assessing the impact of UMLforCLASS on DDENS using ANCOVA

Source	Sum of Squares	df	Mean Square	F	Significance
UMLforCLASS	1.869E-03	1	1.869E-03	6.825	**.010**
CBO	7.562E-08	1	7.562E-08	.000	.987
MCC	1.430E-03	1	1.430E-03	5.224	**.025**
Error	2.519E-02	92	2.738E-04		

ity assumption is justified because ANCOVA is quite robust to violation of the normality assumption [115]. In the ANCOVA test, we included class coupling and complexity as covariates. Class size is not included because it shares the same size factor as defect density. The results of the ANCOVA test are provided in Table 4.7.

The most important point to note from Table 4.7 is that the effect of using UML modeling remains significant ($p \leq 0.05$) even though coupling and complexity have been included as covariates in the analysis. This result basically means that the means of defect density between the groups, i.e., NMC and MC, are significantly different after controlling the effect of class coupling and complexity. Further, we see that complexity is a significant covariate, which is not surprising since we did not control its variance in the data set. Another thing to note is the value of sum of squares, which represents the amount of variation in defect density that is accounted for by the independent variable and the covariates. We can see in the table that the independent variable (i.e., the use of UML modeling) has the highest sum of squares value; hence, it explains the variability of defect density better than the covariates.

The above discussion shows that the results of the ANCOVA test are consistent with the results of the Mann-Whitney test—that is, the use of UML modeling significantly explains the variability of class defect density. Although the ANCOVA test is performed on a data set that violates the assumption of a normal data distribution, we should consider the results of the ANCOVA test as a complement to the results of the main statistical test. Overall, this result further strengthens the evidence about the effect of UML modeling on the defect density of software modules.

4.6 Testing Hypothesis 2: The Effect of Modeling on Defect-fixing Effort

This section describes the results of assessing the impact of UML modeling on the effort spent on fixing defects.

4.6.1 Descriptive Statistics

As discussed earlier, to test the second hypothesis we needed to determine whether behaviors that are found to be defective are modeled or not modeled using sequence diagrams. Thus, for each registered defect we identified the corresponding sequence diagrams—that is,

Table 4.8: Descriptive statistics of sampled defects from IPS comparing NMD and MD

Variables	Not Modeled Defects (NMD)				Modeled Defects (MD)			
	N	Median	Mean	St.Dev	N	Median	Mean	St.Dev
FixEffort (hour)	52	166.375	489.598	675.861	34	121.160	294.738	504.633
ModFile	52	2.000	3.730	4.542	34	2.000	2.647	2.072
TCBO	52	36.000	61.961	76.729	34	25.000	44.588	59.496
TMCC	52	72.500	174.000	224.168	34	74.500	150.647	347.948
TKSLOC	52	0.877	2.632	3.697	34	0.845	1.686	2.761
DPRIO	52	2.000	2.130	0.768	34	2.000	2.210	0.845
TPERS	52	4.000	4.115	1.308	34	4.000	4.529	1.673

sequence diagrams that describe the defective behavior.

There are 1546 defects that we considered in this study. These defects are those reported during testing (i.e., unit test, system test, regression test, and integration test) and represent 60 percent of the total number of defects. The rest of the defects are those reported during review (771) and acceptance test (212). We do not use these defects in the analysis because our focus is on pre-release defects found during internal tests.

Out of the 1546 defects, only 566 are traceable to the modified source files. The fact that there are a large number of defects that is not traceable to the modified source files is due to the following reasons. First, there are defects that were solved without modifying source files, which include changes in the database or application server. Second, it is possible that a finding was solved indirectly, i.e., by solving other defects. Finally, it is often the case that defects were rejected for some reasons, for instance because they could not be reproduced. In these cases, no source file was corrected to solve defects.

Since defect-source traceability is a prerequisite for the analyses, defects without this traceability had to be excluded; thus, the remaining population for the analysis is 566 defects. Out of these 566 defects a random sampling is performed. The sampling is performed by assigning a random number to each finding and then sorting the defects based on the random numbers. The sample is taken from the first 164 defects from the sorted list. The size of the sample is mainly constrained by the availability of resources to perform defect analysis.

After performing analysis of the 164 defects, it turns out that only 86 defects are suitable for our analysis. The rest of the defects could not be used because they were either solved by modifying non-Java classes (e.g., xml, Jsp files) or, if Java classes were indeed being modified, they could no longer be traced in the latest code snapshot. In the latter case, the reason being that the faulty Java classes might have been changed in names, deleted, merged, or moved to a different directory. While it might be possible to manually trace faulty classes that have changed in names or merged using certain features of the bug tracking system, the effort might have been substantial given the number of untraceable faulty classes (i.e., 77 out of 361 faulty classes associated with the 164 defects are not traceable to the implementation classes). Due to this reason, we decided to proceed with the 86 defects for conducting further analysis.

Table 4.8 provides the descriptive statistics of sampled defects in the NMD and MD groups. As we can see in the table, all of the measured variables are listed, and their

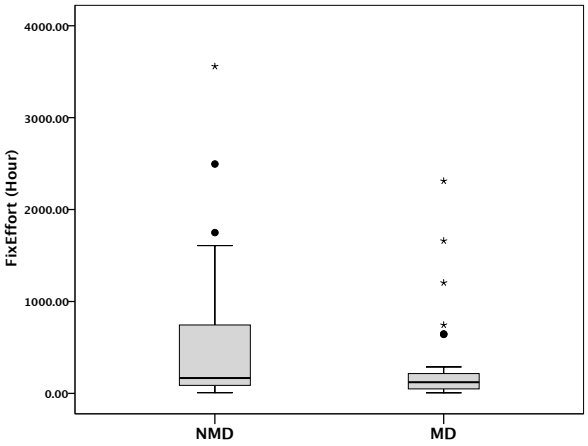


Figure 4.3: Box-plots of FixEffort in the NMD and MD groups

median, mean, and standard deviation are shown. If we compare the median values of the variables in both the NMD and MD groups, we can see that only FixEffort and TCBO seem to have a quite substantial difference. In the main analysis we will assess weather the differences in the measured variables between the NMD and MD groups are statistically significant.

Figure 4.3 provides an overview of FixEffort data using a box-plot. We can see that there are some outliers and extreme values in both the NMD and MD groups. After a careful check, these data points turn out to be valid, and thus we keep them in the data set. What is obvious from the figure is that the NMD group has a significantly larger data range (3550 hours), while the data range of the MD group is less than half that of the NMD group (1654 hours). In particular, we can see in the NMD group that the larger data range is caused by a sparse distribution of 50% of the data points with FixEffort above the median (the median is indicated by the horizontal line in the grey box).

As discussed earlier, we also consider the effect of defect type on defect-fixing effort. To this aim, we introduce a defect taxonomy based on surveys in the literature (e.g., [37], [36, 65]) and our own observations. The taxonomy has the following categories:

- **User Interface**. Defects related to static user interface layouts or caused by wrong or missing user interface navigation.
- **User Data Input/Output**. Defects related to missing or wrong data input/output from/to user interface.
- **Data Handling**. Defects caused by missing or wrong data handling, such as input data validation and session issues. Data access problems also belong to this category.
- **Computational**. Defects caused by missing or incorrect computation.

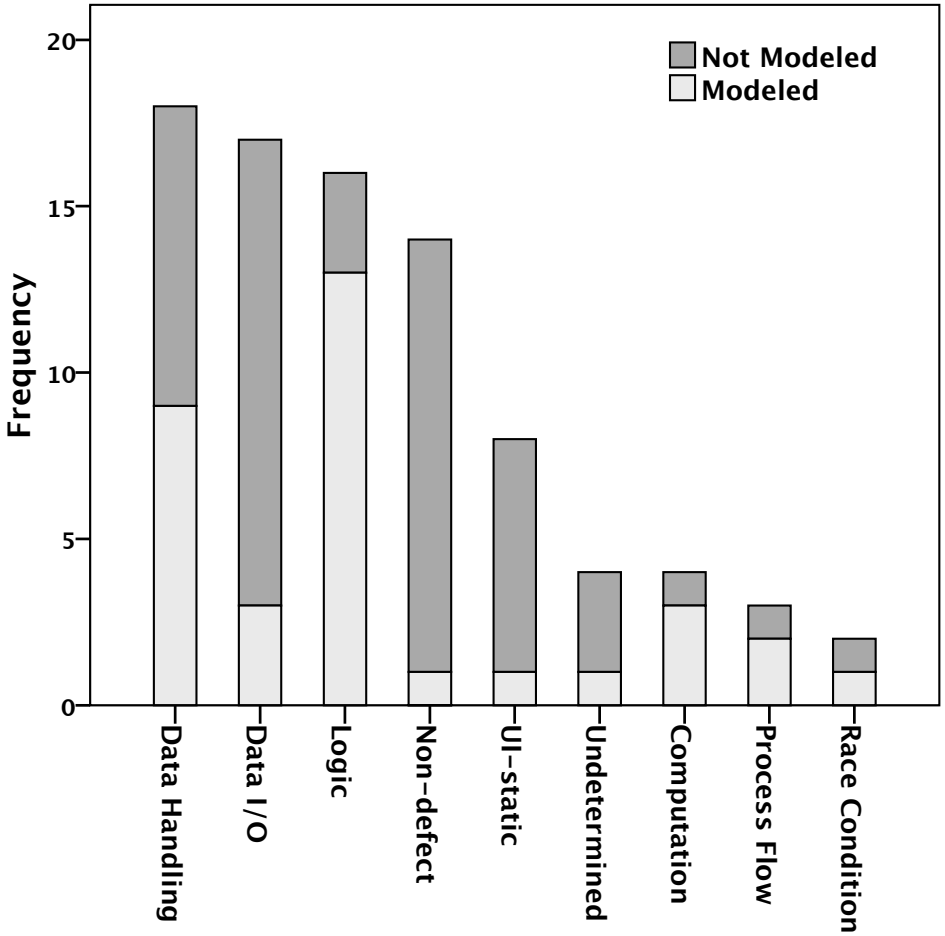


Figure 4.4: The distribution of defect types of the 86 sampled defects—also showing the frequency of modeled and not modeled defect per type

- **Logic/Algorithm**. Defects caused by missing or poor implementation of business rules or wrong formulation of conditions.
- **Process Flow**. Defects caused by missing or wrong process flows (e.g., incorrect order of operation execution).
- **Race Condition**. Defects caused by incorrect timing of events (e.g., unanticipated locking or synchronization).
- **Undetermined**. Defects do not belong to the above categories.

The above list outlines the general defect types. For example, two defect types, namely

user interface and data handling consist of several sub types. Detailed discussion of each defect type is beyond the scope of this paper and can be found in the aforementioned papers.

The type of each registered defect is determined primarily based on how it was solved, which is done by comparing the modified Java classes before and after correction (change analyses)—this can be done easily using the bug tracking system. In addition, the determination is based on defect information stored on the analysis database. However, it is possible that some defects were solved without modifying any files, thus their types are impossible to be determined.

Figure 4.4 shows the distribution of defect types of the 86 sampled defects that is coupled with information about the number of modeled or not modeled defect per type. As shown in the figure, the three defect types that occur most frequently are Data Handling, Data I/O, and Logic defects respectively. We also see that a large number of the 86 defects is categorized as non-defect. Defect classified in this category generally are those pertain to change requests such as change of or additional features requested by the customer. Additionally, the figure tells us that a large number of defects that are not modeled belong to Data I/O type. Similarly, we also see that a large number of defects that are modeled belong to Logic type. By accounting for the effect of defect type we will be able to adjust any discrepancies or bias of defect type (as shown in Figure 4.4) related to the main factor (i.e., UMLforDEFECT).

4.6.2 Assessing the Effect

To unveil the effect of UML modeling on the effort spent on fixing defect, we need to perform statistical analysis to compare the difference in defect-fixing effort between the NMD and MD groups. However, before performing such analysis we have to assure that the suspected confounding factors do not confound the relationship between the independent and dependent variables. It is even better, however, if we can systematically account for and quantify their influences. To these aims, in this section we perform two types of analysis, namely Mann-Whitney and ANCOVA tests, with which we can assess and validate the significance effect of UML modeling on defect-fixing effort after accounting for the effects of the confounding factors.

The first test is a non-parametric one using the Mann-Whitney test in which we assess two aspects: 1) the difference in defect-fixing effort between the modeled and not modeled defects; 2) the difference in terms of the identified confounding factors between the modeled and not modeled defects. The latter is required to assure that the confounding factors have no significant influence on the investigated causal relationship. Notice, however, that we do not take into account DTYPE in the Mann-Whitney test because DTYPE is a categorical variable. The effect of DTYPE will be accounted for in the ANCOVA test.

Table 4.9 and 4.10 provide the results of the Mann-Whitney test. Table 4.9 provides comparisons of mean ranks and sum of ranks of the measured variables, while Table 4.10 provides the results of the main statistical test. Table 4.9 shows that the mean ranks of the measured variables are generally quite similar. However, we can see that the difference in FixEffort between the NMD and MD groups is quite significant (i.e., nearly 11 points)—that

Table 4.9: Mann-Whitney test - Ranks of the measured variables of the NMD and MD groups

Variables	Groups	N	Mean Rank	Sum of Ranks
FixEffort	NMD	52	47.60	2475.00
	MD	34	37.24	1266.00
ModFile	NMD	52	45.10	2345.00
	MD	34	41.06	1396.00
TCBO	NMD	52	45.09	2344.50
	MD	34	41.07	1396.50
TMCC	NMD	52	43.09	2240.50
	MD	34	44.13	1500.50
TKSLOC	NMD	52	43.66	2270.50
	MD	34	43.25	1470.50
DPRIO	NMD	52	42.88	2229.50
	MD	34	44.46	1511.50
TPERS	NMD	52	41.65	2166.00
	MD	34	46.32	1575.00

Table 4.10: Mann-Whitney test - The significance of differences in the measured variables between the NMD and MD groups

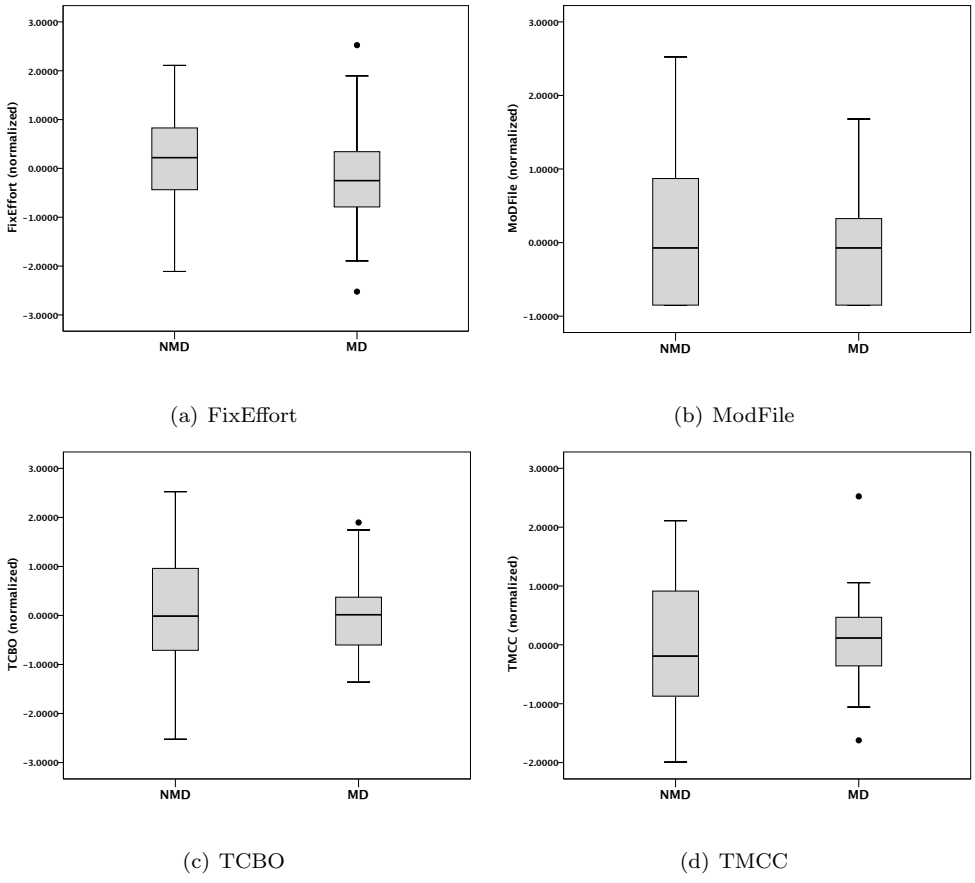
Variables	Mann-Whitney U	Wilcoxon W	Z	Significance
FixEffort	671.000	1266.000	-1.881	**.030***
ModFile	801.000	1396.000	-.760	.447
TCBO	801.500	1396.500	-.729	.466
TMCC	862.500	2240.500	-.190	.849
TKSLOC	875.500	1470.500	-.075	.940
DPRIO	851.500	2229.500	-.308	.758
TPERS	788.000	2166.000	-.872	.383

(*) indicates significance at 0.05 level (1-tailed)

is, NMD being the group with higher FixEffort.

Table 4.10 provides the main result of the Mann-Whitney test. It shows which variables are significantly different between the NMD and MD group. The most important part to look at is the significance value of the variables. We can see in the table that only FixEffort has a significant value below 0.05 ($p = 0.028$)—recall that we consider $p \leq 0.05$ as a threshold to indicate a true significance. The results in Table 4.10 also confirm that none of the identified confounding factors is significantly different between the two groups. Hence, we can conclude that the identified confounds have little if any effects on the cause-effect relationship between UML modeling and defect-fixing effort. Also note that the p-value is quoted as one-tailed because we formulate the hypotheses as a one-tailed hypotheses—that is, we have a specific assumption about the direction of the hypotheses.

Given the results of the Mann-Whitney test, we can reject the null hypothesis ($H2_{null}$), and accept the alternative hypothesis ($H2_{alt}$): the use of UML modeling reduces the effort spent on fixing defects. Furthermore, we have observed the significant effect of UML modeling on defect-fixing effort after assuring that some potential confounding factors have negligible effects on the result.



(a) FixEffort

(b) ModFile

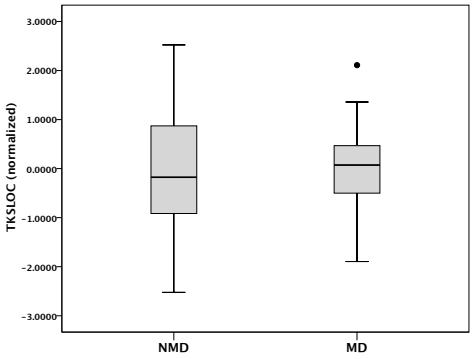
(c) TCBO

(d) TMCC

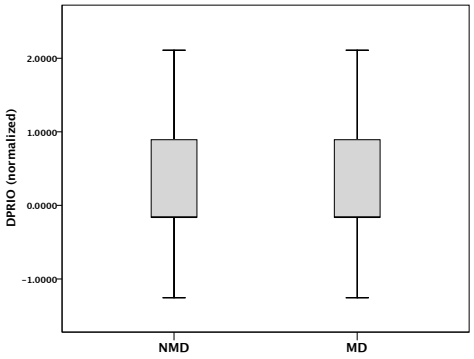
Figure 4.5: Box-plots of FixEffort, ModFile, TCBO, and TMCC in the NMD and MD groups (after data transformation)

Based on the results of the Mann-Whitney test we can see that the confounding factors in both the NMD and MD groups are not significantly different. This result has also led us to believe that the confounding factors have no substantial effect on the relation between the main variables. Nevertheless, it would be interesting if we can account for the effects of the confounding factors so that their influence can be quantified and totally controlled. Additionally, ANCOVA allows us to account for the effect of DTYPE as a co-factor. By incorporating DTYPE as a co-factor, we will also be able to assess whether there is an interaction between UMLforDEFECT and DTYPE.

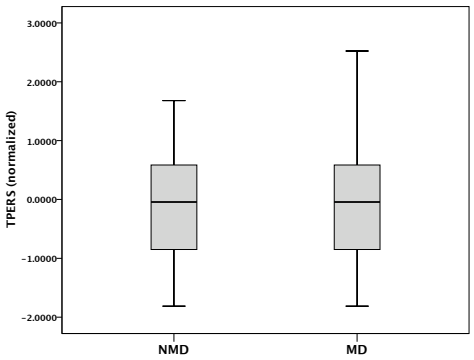
Before we can perform an ANCOVA test, we need to check whether the data set meets the assumptions of the ANCOVA test such as a normal data distribution and homogeneity of variance. As it is generally the case with software engineering data, our data set violates the normality assumption. To reduce the effects of non-normal data distribution,



(a) TKSLOC



(b) DPRIO



(c) TPERS

Figure 4.6: Box-plots of TKSLOC, DPRIO, and TPERS in the NMD and MD groups (after data transformation)

we perform data transformation using area transformation [75]. After performing the data transformation, it turns out that the procedure does not completely solve the non-normal data distribution. However, considering the robustness of the ANCOVA test to a violation of normality we are confident that a minor violation would not affect the validity of the results. Figure 4.5 and 4.6 show box-plots of the measured variables after transformation.

Table 4.11 shows the results of the ANCOVA test, in which the contribution of individual variables to the variability of defect-fixing effort is quantified. The most important point to note in the table is the significant values of the variables. The main factor (i.e., UMLforDEFECT) remains significant ($p = 0.030$) after controlling for the effects of the covariates. These results confirm the results of the Mann-Whitney test performed earlier—that is, the use of UML has a significant impact on defect-fixing effort. More specifically, the results show that the use of UML significantly reduces defect-fixing effort.

Table 4.11: Results of assessing the effect of UMLforDEFECT on FixEffort using ANCOVA—accounting for the effects of the confounding factors

Source	Sum of Squares	df	Mean Square	F	Sig.
UMLforDEFECT	3.239	1	3.239	4.915	**.030**
DTYPE	12.107	8	1.513	2.296	**.032**
UMLforDEFECT * DTYPE	6.498	8	.812	1.232	.296
ModFile	.089	1	.089	.135	.715
TCBO	.386	1	.386	.586	.447
TMCC	.485	1	.485	.736	.394
TKSLOC	1.048	1	1.048	1.589	.212
DPRIO	.523	1	.523	.794	.376
TPERS	17.926	1	17.926	27.198	**.000**
Error	40.863	62	.659		

Table 4.11 also shows that DTYPE is a significant factor that explains the variability in FixEffort. Nevertheless, we do not observe a significant interaction between DTYPE and UMLforDEFECT with respect to their effects on FixEffort. Additionally, the table shows that the number of persons involved in fixing defects (TPERS) is a significant variable that explains the variability of defect-fixing effort. If we consider the F value, which represents the ratio of variance explained and variance unexplained, the number of persons involved in fixing defects appears to have the highest influence on defect-fixing effort. The rest of the covariates, however, does not have a substantial influence on defect-fixing effort.

4.7 Interpretation of Results

In the last two sections, we have analyzed the effects of UML modeling on defect density and on the effort spent on fixing defects. Results from the case study show that modeled classes, on average, have a lower defect density that those that are not modeled. Statistical tests confirm that the difference in defect density is statistically significant. Furthermore, we have observed that the effort needed to fix defects related to modeled behaviors is significantly smaller compared to the effort spent on fixing defects related to unmodeled behaviors. This result essentially shows how the use of UML might improve productivity of performing changes or maintenance in software systems. It is also important to underline that we have accounted for potential confounding factors that could have influenced the results of the analyses.

To understand why classes that are not modeled have significantly higher defect density, we first need to consider the nature of these classes. Experience has shown that designers generally choose classes that are important and complex to be modeled. Hence, it is quite natural to assume that classes that are not modeled generally are trivial classes or pertain to straightforward concepts. Nevertheless, this assumption is not always true. In the context of this study for example, by simply looking at the complexity metric, we could easily observe that some classes that are not modeled actually have a very high complexity and coupling. In fact, the one class with the highest complexity is a class that is not modeled. Hence, it is very likely that some classes that are not modeled are in fact classes that are not trivial and

should have been modeled. Because these significant classes might be involved in complex operations in the system, the absence of specifications that describe their behaviors might have led to incorrect implementation.

Concerning the effect of using UML (i.e., sequence diagrams) on the effort spent on fixing defects, we particularly believe that the productivity gain is achieved because having specifications of defective functionality or behaviors in sequence diagrams helps software engineers locate the problem and determine which objects (class instances) are going to be affected by the corrective actions relatively quickly. In the case of defects without corresponding behaviors modeled using sequence diagrams, software engineers need to examine the code of the suspected-faulty classes in order to locate the problem. Moreover, finding connected classes that are going to be affected by the corrective actions might not be trivial when the problem is located in classes that are central to the system. Yet, the corrective actions tend to require extra time if performed by someone who is not familiar with parts of the system that need to be fixed.

The results discussed in this chapter are in line with the findings of a case study reported in a master's thesis [50]. The case study investigated the effects of the level of detail and completeness (model coverage) of a UML model on defect density and defect-repair effort in an embedded system. The UML designs of two software components were assessed. Similar to the results discussed here, the results of the case study show that the component with higher model coverage (uses more modeling) has lower defect density in the implementation and also lower defect-repair effort. Note, however, that the project used state chart diagram more than any other UML diagram types (in this respect class- and sequence diagrams).

The implication of the results of this study on research in the area of model-driven software development is two-fold. First, the result of this study should encourage more research on how to improve the quality of models, for example by investigating methods and techniques for a practical quality assurance of software models. More specifically, we need to investigate which attributes of software models are most influential to the quality of software systems (note that the attributes should also embrace a model's behavioral aspects because they might correlate better with defects in software). Additionally, the methods for maintaining and evaluating the model should also be investigated. Ideally, the methods should take into account their practicality and applicability in industry. The second implication is related to the trade-off of using modeling in software development. For instance, we need to investigate whether quality improvements and productivity gains achieved by introducing modeling lead to cost-saving that is higher than the costs invested in the modeling activities.

We also underline the implications of this study for software development in practice. First, the result of this study should encourage both project managers and software engineers to evaluate how UML is used in their projects. While we are aware of the fact that not all system parts needs be modeled, the decisions to model or not model system parts should be based on informed decisions. For example, components' complexity and criticality have been considered by developers as good candidates for more extensive modeling [96]. Second, based on the results of this study we also emphasize the needs for good quality models, which comprise syntactic and semantic aspects of models. To achieve this quality goal, practical model quality assurance activities such as design reviews and the use of

modeling conventions should be considered to be incorporated in the software development process (see the discussion in [95]). These quality assurance activities should help accentuate the impact of modeling on the final software quality. Finally, the evidence also reveals that modeling system behaviors using behavioral diagrams (e.g., sequence diagrams, state machine diagrams) can help software engineers identify and fix functional problems faster than without having UML diagrams at all. This result should encourage software designers to improve the coverage of their behavioral modeling. Ideally, there should be one-to-one mapping between use cases and the behavioral diagrams that describe them (e.g., sequence diagrams). However, if such mapping is not possible then software designers should focus on modeling critical or complex behaviors or functionality.

4.7.1 Threats to Validity

In this section, we discuss validity threats of this study. These threats to validity will be presented in their order of importance [130]: internal validity, external validity, construct validity, and conclusion validity.

The main threat to the internal validity of this study concerns our ability to control influences from other factors beyond what have been accounted for in this study. Therefore, more advanced research design is required to address other confounding factors, such as requirement quality, team composition, and team experience.

External validity threats concerns limitations to generalize the results of a study to a broader industrial practice. We can not make a strong claim that the results of this study are generalizable to other projects because every project is unique. Most importantly, the way UML models are used in a project is very influential to how they might affect the quality of the final implementation or the productivity in fixing defects. In this respect, a subset of the analyses discussed in this chapter (i.e., the first hypothesis) was replicated using two additional case studies, but we did not observe similar results (in Chapter 8, we discuss this issue in further details). Nevertheless, we believe that the results of this study is generalizable to projects in which the UML models are used to guide the implementation (hence, posses a sufficient level of quality), and the developers strictly conform to the models.

With respect to the threats of construct validity, we underline the effect of programming style on the defect density measure. For example, two supposedly similar classes (in terms of role and responsibility) might be programmed in different ways. Developers who have a verbose style of programming tend to produce more lines of code than those who are more effective in writing code. Thus, with the average defect-count being fairly equal, classes written by verbose developers will have lower defect density than those written by efficient developers. Nevertheless, careful analysis of class sizes between the modeled and not modeled classes shows no indication that verbose programming has distorted the defect density measure.

We are also aware that calculating defect-fixing time based on change history may not perfectly reflect the actual effort spent on fixing defect. Defect-fixing effort calculated from change history tends to over estimate the actual effort because it may contain effort of activities unrelated to defect fixing. This issue, however, is inevitable and can only be

avoided by introducing an effort registration tool through which engineers can register their effort whenever they perform defect-fixing activities.

Threats to conclusion validity relate to the ability to draw a correct conclusion from a study. In this study we have addressed factors that might threaten the conclusion validity of this study through a careful design of the approach and rigorous procedures in the data analyses.

4.8 Conclusions and Future Work

In this chapter we empirically investigate the impact of UML modeling on the defect density of software and on the effort spent on fixing defects. The main question this chapter aims to answer is whether the use of UML helps improve the quality of the final software product. Additionally, we seek to answer whether the use of UML significantly reduces the effort spent on fixing defects.

Using empirical data from an industrial Java system, we carefully evaluate the impact of using UML on the defect density of Java classes. After controlling for the effects of class coupling and complexity, we have found that the use of UML modeling remains a significant predictor that explains the variability of defect density in the implementation. More specifically, Java classes that are modeled using UML are found to have a significantly lower defect density than those that are not modeled. This result indicates the potential benefits of UML modeling for improving the quality of software.

The analyses in this chapter also show that defects related to behaviors modeled using sequence diagrams require significantly less time to fix than defects related to unmodeled behaviors. This result remains consistent after controlling for the effects of potential confounding factors, which include the number of modified files, the number of person involved in fixing defects, and defect priority. In essence, this result confirms that the use of UML might also help increase productivity of performing changes or maintenance in software systems.

This work is one of the first studies that attempts to investigate the benefits of modeling in an industrial software project. Hence, more research is needed to foster our understanding on the subject. We also encourage other researchers to perform similar studies in real software projects. Furthermore, we underline the importance of identifying and assessing other confounding factors, such as developers' experience. Assessing confounding factors will not only help us observe the pure effects of modeling, but also it might give us more insights about the circumstances under which modeling can deliver benefits in software development. Finally, we should also consider conducting this type of study in experimental settings, which will allow us to control the effects of confounding factors better.