Universiteit
Leiden
The Netherlands

An online corpus of UML Design Models : construction and empirical studies

Karasneh, B.H.A.

Citation
Karasneh, B. H. A. (2016, July 7). *An online corpus of UML Design Models : construction and empirical studies*. Retrieved from https://hdl.handle.net/1887/41339

Version:	Not Applicable (or Unknown)
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/41339

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/41339 holds various files of this Leiden University dissertation.

Author: Karasneh, B.H.A.
Title: An online corpus of UML Design Models : construction and empirical studies
Issue Date: 2016-07-07

Chapter 6

UML Repository As Benchmark for Quality Analysis

In this chapter, we illustrate some analysis that based on the UML Repository as a benchmark for quality analysis. First, we show common characteristics of class diagrams in the repository. In addition, we show the relation between models size and coupling metric. Then we describe our study of the relation between anti-patterns in design models and source code. We show the impact of the quality of the design models measured by anti-patterns on the quality of the source code measured by numbers of changes and faults.

This chapter is based on the following publications:

- Bilal Karasneh, Michel R. V. Chaudron, Foutse Khomh and Yann-Gaël Guéhéneuc. **Studying the Relation between Anti-patterns in Models and in Source Code.** *In Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, Osaka, Japan. 2016.*
- Bilal Karasneh, Michel R. V. Chaudron. **Online Img2UML Repository: An Online Repository for UML Models.** *In Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESS-MOD@MODELS 2013), pages 61-66, Miami, USA. 2013.*

In chapter two, we talked about software quality models, and showed how to measure software quality. In this chapter, we spot some characteristics of class diagrams in the repository based on their design metrics. We think these characteristics are important to assess the quality of class diagrams, because it helps to find common patterns in class diagrams, and measures these patterns in terms of quality. Furthermore, we present our study of the relation between anti-patterns in the design and quality of source code based on number of changes and faults.

6.1 Common Characteristics of Class Diagrams

Finding out common characteristics of the designs provide a way to measure their quality. The repository contains a big collection of class diagrams, which are related to different application domains and created by different designers. Thus, it can be considered as a good place to apply empirical studies. We analyze the size of class diagrams and the relation between size and maximum coupling. These metrics influence the complexity of class diagrams. Based on that analysis, we aim to find common patterns. We notice that coupling includes three relationships: association, composition and aggregation.

6.1.1 The Size of Class Diagrams

In this subsection, we describe the size of the diagrams in the repository. Figures 6.1 and 6.2 show a boxplot of class diagrams size and its distribution in the repository respectively. Table 6.1 provides descriptive details of figure 6.1. We see that the maximum number of classes is relatively small. We see that it is not common to find a class diagram with a big size - The median number of classes is nine.

Table 6.1: *Descriptive statistics of the size of class diagrams in the repository*

	No. Diagrams	Median	Quartile 25%	Quartile 75%
Diagrams size	810	9	5	13

6.1.2 Maximum Coupling

We believe that maximum coupling in class diagrams can be an important indicator of the complexity of class diagrams. So if there is at least one class in a class diagram that show a high coupling, this determines the complexity of that class diagram. Figure 6.3 shows the boxplot of the maximum coupling of the class diagrams in the repository. Table 6.2 shows the descriptive statistics of Figure 6.3.

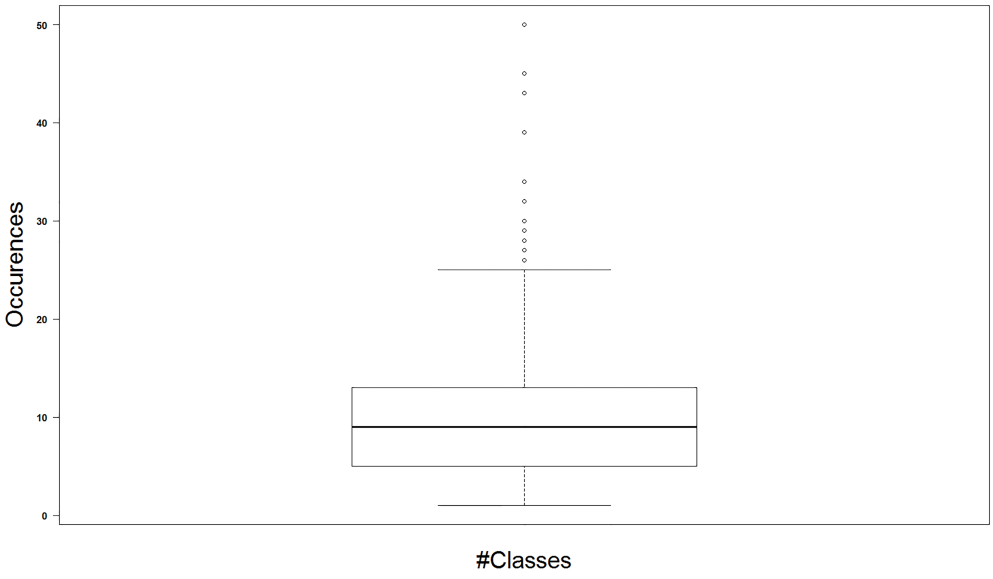

Figure 6.1: *Size of class diagrams in the repository*

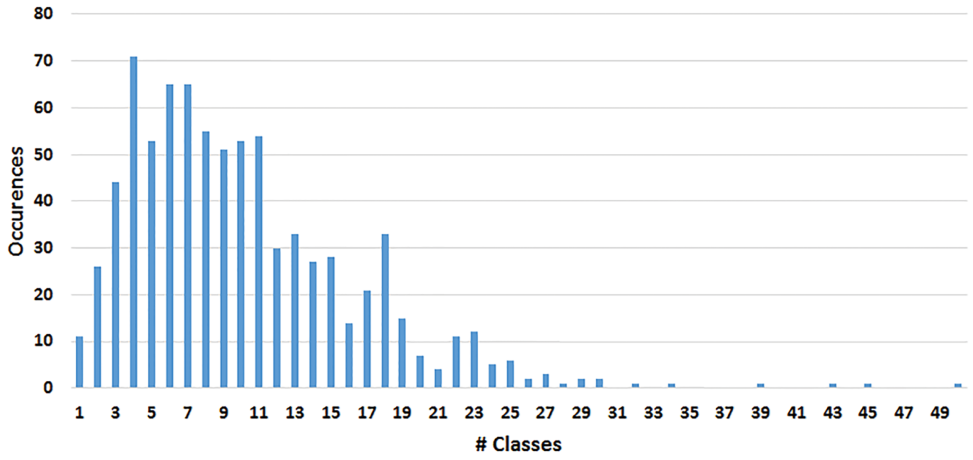

Figure 6.2: *Distribution of class diagrams size in the repository*

6.1.3 The Relation between Class Size and Max. Coupling

It is interesting to study the relation between the size of class diagrams and the maximum coupling that they indicate. Figure 6.4 shows a Bubble chart for diagrams size and their maximum coupling. Figure 6.4 shows that there is a positive correlation

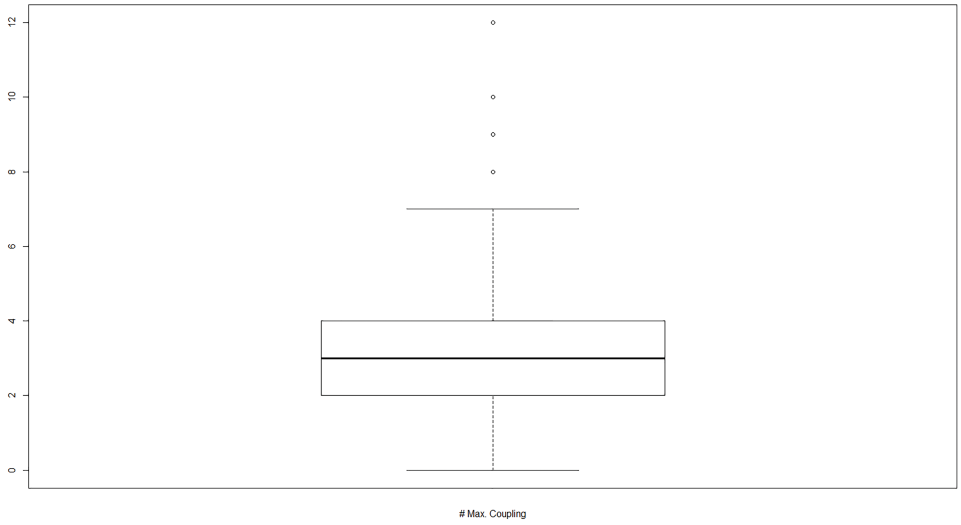

Figure 6.3: *Maximum coupling for diagrams in the UML Repository*

Table 6.2: *Descriptive statistics of Max. coupling in class diagrams in the repository*

	No. Diagrams	Median	Quartile 25%	Quartile 75%
Diagrams size	810	3	2	4

between diagrams size and maximum coupling. It shows that when the diagram size is high, the maximum coupling also will be high, which it makes diagrams more complex. In Figure 6.4, the trendline also shows the coefficient of determination (R2 = 0.07). A correlation of 0.10 is seen as a weak relationship, 0.30 as moderate, and 0.50 as strong relationship [80]. This interpretation is fit in our data because it is randomly collected, and the behavior, skill, and experience of the designer are different. The Pearson correlation that we calculated between the diagrams size, and the maximum coupling is significant ($p < 0.01$) and it is a moderate relationship ($r = 0.46$).

6.1.4 Discussion

Our study is a preliminary study of the size of class diagrams and maximum coupling. The size of the diagrams is relatively small, which it means that it is not common to have big diagram size. The maximum coupling is small, which could be considered as a hint of the complexity of class diagrams. The relation between class diagrams size and maximum coupling is moderate. However, this moderate relationship shows that the complexity of class diagrams increases when the size of class diagrams increases. We

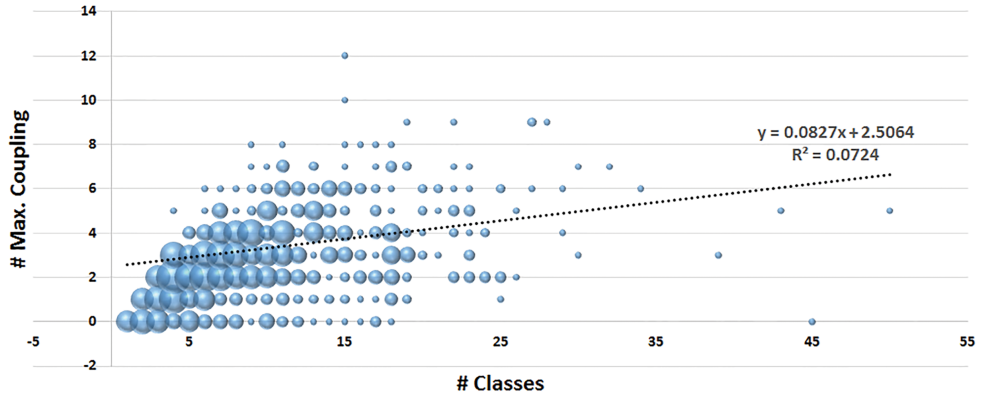

Figure 6.4: *Relation between Diagrams size and Max. coupling*

notice that 11% of class diagrams have a maximum coupling of zero. These diagrams do not contain any association, but they do inheritance and dependency relationships.

6.2 Studying the Relation between Design Quality and Source Code

Many software engineers focus on the quality of source code, and they do not give enough attention to the quality of the design. There are few studies about the contribution of software design on software development, because it is challenging to study.

In this section, we study the impact of software design on the source code. We investigated two open source software projects that have a design, different released versions of the source code, and we can access their changes and bugs among the different released versions. Then, we study the effect of anti-patterns in design on the quality of the source code measured by number of changes and number of bugs. For achieving this, we did two studies:

- First, we investigate seven open source software projects that have design and made a comparison between occurrences of anti-patterns in the design and the source code.

- Second, we measured the relation between anti-patterns in design and the quality of the source code based on changes and bugs. We use the same two open source projects that we used in the first experiment.

Table 6.3: *Descriptive statistics of Max. coupling in class diagrams in the repository*

Project Name	Modeled classes	Not-Modeled classes	Total
ArgoUML	88	2731	2819
Wro4j	197	876	1073

6.2.1 Relation between Software Design and Source Code

We conducted an experiment with two big open-source software projects to measure the quality of classes that appear both in the design and the source code, and classes that appear only in the source code. We selected ArgoUML and Wro4j because we can access to the class changes and bugs reports for all versions of the software. More information about ArgoUML and Wro4j is in Table 6.8. For ArgoUML, we use nine different released versions and for Wro4j, we use six different released versions.

6.2.1.1 Experiment Design

We categorized classes in the source code into two categories:

1. Modeled classes, which are classes that appear in the design and source code.
2. Not-Modeled classes, which appear in source code only.

Then we used Mann-Whitney test to compare the means of the number of changes and the number of bugs between both categories. We used Mann-Whitney test because it matches the precondition in both cases.

The number of classes in the design is small as it is usual in open-source software projects. We collected the corresponding of the classes that appear in the design and code. Therefore, the Modeled classes are the classes that appear in the design and the code, and their corresponding in the code, and the Not-Modeled classes are the classes that are only in the code. Table 6.3 shows both categories for both the ArgoUML and Wro4j projects. We use IntelliJ IDEA[1] to find the corresponding classes in the code.

6.2.1.2 Results

Tables 6.4 and 6.5 show the mean of changes and bugs for both Modeled classes and Not Modeled classes in both ArguUML and Wro4j, respectively. In ArgoUML, the result of the Mann-Whitney test shows that there is a significant difference between Modeled classes and Not-Modeled classes in both terms of changes and bugs, where p-value = 0.000 and p-value = 0.007 respectively.

In Wro4j, the results are the same, where the result of the Mann-Whitney test shows that there is a significant difference between Modeled classes and Not-Modeled classes

[1] https://www.jetbrains.com/idea/

Table 6.4: *Means of classes Changes in ArgoUML and Wro4j*

	Project Name	Categories	Mean
Changes	ArgoUML	Modeled classes	22
		Not-Modeled classes	7.78
	Wro4j	Modeled classes	15.17
		Not-Modeled classes	6.56

Table 6.5: *Means of classes faults in ArgoUML and Wro4j*

	Project Name	Categories	Mean
Bugs	ArgoUML	Modeled classes	0.82
		Not-Modeled classes	0.42
	Wro4j	Modeled classes	2.91
		Not-Modeled classes	1.04

in both numbers of changes and bugs, where p-value = 0.000 and p-value = 0.000, respectively.

Next, we show the relation between the number of changes and two code metrics, Line of Code (LOC) and average Cyclomatic Complexity (AvgCyc). Table 6.6 shows the correlations between number of changes in ArgoUML and Wro4j with LOC and AvgCyc. The number of changes in Modeled classes has significantly higher correlations with LOC than Non-modeled classes in both ArgoUML and Wro4j: Modeled classes that have higher LOC have more changes. Table 6.7 shows the correlation between faults, LOC and AvgCyc: the number of faults in Modeled classes have significantly higher correlations with LOC than Non-modeled classes in both ArgoUML and Wro4j. More faults exist in Modeled classes that have higher LOC. AvgCyc does not have a correlation with Modeled classes or Non-modeled classes.

6.2.1.3 Results Discussion

The result shows that there is a significant difference between Modeled classes and Non-Modeled classes categories in both cases changes and bugs. The means of changes and bugs of Modeled classes are higher than Non-Modeled classes, which mean the Modeled classes have more changes and bugs in both ArgoUML and Wro4j among different released versions. We explain this result as Modeled classes important and could have the main functionality of the system, so it has more changes during different released versions. We explain the faults by the positive relation between the number of changes and number of faults [81]. Therefore, classes in the design have more changes, this tend to have more faults. Indeed, the implementation should follow the design, which results in developers transferring problems from the design to the source code. We notice that, problems in the design may cause and propagate more problems in the

Table 6.6: *Correlation between Changes, LOC and AvgCyc*

	Project Name	Categories	Correlation		R^2	Formulas
			AvgCyc	LOC		
Changes	ArgoUML	Modeled classes	0.30	0.74	0.54	Y=0.727+0.018(LOC)
		Not-Modeled classes	0.21	0.43	0.19	Y=1.352+0.004(LOC) +0.092(AvgCyc)
	Wro4j	Modeled classes	0.06	0.62	0.39	Y=-1.937+0.315(L0C)
		Not-Modeled classes	0.00	0.38	0.17	Y=6.191+0.104(LOC)(AvgCyc)

Table 6.7: *Correlation between Changes, LOC and AvgCyc*

	Project Name	Categories	Correlation		R^2	Formulas
			AvgCyc	LOC		
Bugs	ArgoUML	Modeled classes	0.35	0.53	0.32	Y=-0.54+0.001(LOC) +0.041(AvgCyc)
		Not-Modeled classes	0.13	0.26	0.07	Y=0.07+0.000(LOC)
	Wro4j	Modeled classes	0.02	0.6	0.35	Y=-0.266+0.060(LOC)
		Not-Modeled classes	-0.01	0.40	0.18	Y=0.953+0.026(LOC) -0.417(AvgCyc)

source code because in the code more information and functions should be added.

6.2.2 Effects of Anti-patterns on Software Quality

There are very few studies on the origins of the occurrences of anti-patterns in the source code. Knowing where and when anti-patterns are introduced could help software designers and developers improve the quality of the software systems. In this section, we explore the origins of anti-patterns by tracing them back to design models. We conduct our study using the models selected randomly from the UML Repository, which is a unique repository containing pairs of architects and designers' models (UML class diagrams) linked to the corresponding source code, when available. For each system, we analyze both its UML design models and its source code.

Since their inception in software engineering, design patterns (i.e., reusable solutions to recurring design problems) [82] and anti-patterns (i.e., poor solutions to design and implementation problems) [83] have been the subject of many research works. This research works focused on design patterns specification [84], detection [85], and on the analyzes of their impact and life-cycle. Tufano et al. [86], Vaucher et al. [87], and Chatzigeorgiou and Manakos [88], investigated the evolution of anti-patterns in software systems, and observed that anti-patterns are not necessarily only introduced in the source code during maintenance and evolution activities. They reported that many classes are "born" anti-patterns. Following on this observation, we set out to investigate whether design models produced by architects and designers before the implementation contains anti-patterns. In addition, we see if theses anti-patterns translate in the source code, i.e., if these anti-patterns concern the same classes in design models and the source code implementing these models.

On the one hand, as long as the code follows the decisions embedded in the models, the same patterns/anti-patterns from the models should appear in the source code. On

the other hand, if we can use the right patterns and follow the right design decisions early on in the development cycle, we could prevent the occurrence of anti-patterns in the code.

6.2.2.1 Related Work

There are many research works on the definition, specification, detection, correction, and analysis of the life-cycles of design patterns and anti-patterns. Because we focus on anti-patterns, we describe here three works that (1) showed that anti-patterns do impact negatively class change- and fault-proneness, (2) studied the introduction and removal of some anti-patterns qualitatively, and (3) reported four lessons on their life-cycles. We describe the specification and detection of anti-patterns in Sections 6.2.2.3 and 6.2.2.4.

Khomh et al. [89] investigated the impact of anti-patterns on classes in object-oriented systems by studying the relation between anti-patterns and class change- and fault-proneness. They showed that in 50 out of 54 releases of the four analyzed systems, classes participating in anti-patterns are more change and fault-prone than others.

Vaucher et al. [87] studied the "God class" anti-patterns, which describes large classes that "know too much or do too much". The literature postulated that God classes are created by accident as functionalities are incrementally added by developers to central classes over the course of their evolution, Vaucher et al. observed that, in some systems, God classes are created by design, as the best solution to a particular problem, for example, when a functionality is not easily decomposable or when there exist strong requirements on efficiency or performance. They studied the life-cycles of God classes in the source code of Eclipse JDT and Xerces; investigating how they arise, how prevalent they are, and whether they remain as the system evolves over time. They distinguished between those classes that are God classes by design from those that occurred by accident in the implementation. They concluded that some God classes are created by design but most are the result of a decay of the systems. They propose that developers use detection techniques and refactoring to track and prevent anti-patterns in their systems.

Following this previous work, Tufano et al. [86] studied the life-cycle of five anti-patterns in Android, Apache, and Eclipse and drew the following lessons: (1) classes often play roles in anti-patterns from their inception in the systems, (2) the metric values of the classes that started to play some roles in some anti-patterns only during the evolution of the systems have specific trends, (3) refactoring operations, in addition to other changes, may lead to the introduction of anti-patterns, and (4) time pressure is the main cause of the introduction of anti-patterns. With these lessons, in the particular lesson (1), they confirmed the hypothesis of this present study that anti-patterns are not necessarily the results of developers' lack of times/skills but could be due to the very designs of the systems.

Chatzigeorgiou and Manakos [30], who investigated the evolution of anti-patterns

Table 6.8: *Studied Software Systems*

Project Name	Descriptions	URLs
ArgoUML	An open source UML modeling tool	`http://argouml.sourceforge.net`
Annoyme	Adds beautiful typewriter sounds to Desktop keyboards	`https://github.com/dedeibel/annoyme`
JGAP	Package of Genetic Algorithm and Genetic Programming	`http://jgap.sourceforge.net`
Mars_Simulation	Project to create a simulation of future settlements on Mars	`http://mars-sim.sourceforge.net`
Msv_Poker	Poker Game (poker server and poker client)	`https://github.com/mihhailnovik/msvPoker`
Neuroph	Lightweight Java neural network framework to develop network architecture	`http://neuroph.sourceforge.net`
Wro4j	Web resource optimizer for Java	`http://code.google.com/p/wro4j`

in object-oriented systems reported that anti-patterns tend to linger in systems for multiple releases.

All these studies considered occurrences of anti-patterns in the source code of the studied systems (and their revisions) or design models reverse-engineered from their source code. They did not study the prevalence of anti-patterns in design models created before (and–or during) development in comparison to that in the source code implementing these design models. Our following study aims at confirming the observations that, in some designs, anti-patterns are present from the very beginning of the inception of the systems. In addition, these anti-patterns in the design have an impact on the implementation.

6.2.2.2 Experiment Design

The seven open-source systems selected from UML Repository are available from Github, SourceForge, and Google Code. The studied software systems are in Table 6.8. We notice that it is much easier to use UML Repository for finding such case studies that contain UML design with source code. It is hard to inside code repositories to find software projects that contain UML design. We conduct this study using both architects' and designers' models of software systems and the implementation of these models as source code.

6.2.2.3 Anti-patterns Identification

We use the Ptidej[2] tool suite, which implements the anti-pattern detection approach DECOR (Defect dEtection for CORrection) [85], to identify occurrences of anti-patterns in both models and source code. DECOR is an approach based on the automatic generation of detection algorithms from rule cards. It converts anti-patterns descriptions automatically into detection algorithms and identifies the occurrences of these anti-patterns in UML class diagrams and the source code of systems.

We apply DECOR in three steps: first, we reuse/define a rule card describing an anti-pattern through a domain analysis of the literature [84]. From the rule card, we generate a detection algorithm. Finally, we apply the detection algorithm on models of systems to detect the different occurrences of the anti-pattern in these systems. DECOR has appropriate performance, precision, and recall for our study.

DECOR can be applied to any object-oriented system through the use of the PADL [90] meta-model and POM framework [91]. PADL describes the structure of systems and a subset of their behavior, i.e., classes and their relationships. POM is a PADL-based framework that implements more than 60 structural metrics. We apply DECOR on models obtained either from the class diagrams available in the repository, by parsing the corresponding XMI files, or by parsing the corresponding C++ and Java source code.

6.2.2.4 Anti-patterns Specification

Concretely, we detect the occurrences of four anti-patterns which are: (1) Complex class, (2) Large class, (3) Lazy class, and (4) LongMethod. We are using the metrics available in Ptidej for both class diagrams and source codes.

Essentially, we specify the Lazy Class in terms of the number of methods defined in the classes. We define the Complex Class as the number of methods and the relationships among classes: a class that defines many methods and that has many relationships (in or out) is inherently complex. Long Method can only be computed on classes from the source code because we need the number of statements in the methods. Finally, we specify Large Class as the "opposite" of a Lazy Class, in terms of the number of methods in the classes. The details of the specification and detection of the anti-patterns are outside the scope of this chapter because we reuse the specifications and detection algorithms used in previous work and detailed in the presentation of DECOR to which we refer the reader [84].

6.2.2.5 Results

We now report the results of detecting anti-patterns in models and source code of seven systems from the UML Repository using Ptidej. First, we show some analysis of

[2]http://www.ptidej.net

Table 6.9: *Summary of number of Classes in class diagrams versus in source code*

Project Name	# Classes in class diagrams	# Classes in source code	Proportion of classes
Annoyme	17	59	0.29
ArgoUML	51	1722	0.03
JGAP	19	191	0.1
Mars_Simulation	32	953	0.03
Msv_Poker	22	55	0.4
Neuroph	26	179	0.15
Wro4j	28	598	0.05

the numbers of classes in the models and their source code. Then, we summarize the anti-patterns detected in both models and their source code.

Table 6.9 shows an overview of classes in the models and source code, which proportions are based on the equation 6.1:

$$\text{Proportion of classes} = \frac{\text{No. of classes in the Design}}{\text{No. classes in the source code}} \tag{6.1}$$

The numbers of classes in the source code are higher than in the models, but we found that some classes in the models were missing in the source code. This is also expected because the models, which are conceptual models, are often refined by developers during implementation. However, these refinements are not always documented back in the models. In Table 6.10, we show the proportions of classes that exist in models and source code. The proportions in Table 6.10 are measured using the following equations 6.2 and 6.3:

$$\text{C.C.C. to the Design} = \frac{\text{No. classes exist in both Design and source code}}{\text{Number of class in Design}} \tag{6.2}$$

$$\text{C.C.C.totheImplementation} = \frac{\text{No. of classes exist in both Design and source code}}{\text{Number of class in source code}} \tag{6.3}$$

C.C.C. = Common Classes Compared
Common Classes = Classes exist in class diagrams and the implementation

The majority of classes contained in the models are also present in the source code of the systems. However, classes in the class diagrams represent only a fraction of the total numbers of classes contained in the source code. Nevertheless, next we show that

Table 6.10: *Proportion of classes that exist in both class diagrams and source code*

Project Name	# Common Classes	Common classes compared to the Design	Common classes compared to the Implementation
Annoyme	14	0.82	0.24
ArgoUML	44	0.86	0.03
JGAP	18	0.95	0.09
Mars_Simulation	29	0.91	0.03
Msv_Poker	13	0.59	0.24
Neuroph	24	0.92	0.13
Wro4j	23	0.82	0.04

anti-patterns appear in class diagrams during the design phase are transferred to the implementation.

We can detect three anti-patterns in the class diagrams: Complex Class, Large Class, and Lazy class. We can only detect LongMethod in the source code because class diagrams are abstract representations of the systems, and they contain only method signatures without the implementation details needed to compute the lengths of the methods. In Table 6.11, we report the numbers of anti-patterns that we found in the class diagrams and source code of the studied systems. We calculate the proportion of classes that play the same roles in the same anti-patterns in class diagrams and the source code based on Equation 6.4:

$$\text{Proportion of classes} = \frac{\text{No. S.AP.in.D.and.I}}{\text{No. Anti} - \text{patterns in class diagrams}} \tag{6.4}$$

S.AP.in.D.and.I = Same anti-patterns in the same classes in Design and implementation

Table 6.12 shows the number of anti-patterns that exist in the same classes in class diagrams and the source code. We also show the proportion based on Equation 6.4. From Table 6.12 we show that there is a significant proportion of classes playing the same role in the same anti-patterns in the class diagrams, and the source code **(36%)**. We notice that some anti-patterns appear in class diagrams and the same classes in the source code have different anti-patterns. We relate this to a common mistake in both open-source and commercial software development that they update the source code and do not update the design.

Next, we focus on individual anti-patterns and occurrences of each one in class diagrams and source code.

Table 6.11: *Anti-patterns detection in both class diagrams and source code*

Project Name	# APs* in Design	# APs in the Implementation	# Long-Method APs in the source code	# Same APs in the same classes in Design and Implementation
Annoyme	10	16	0	5
ArgoUML	20	545	256	10
JGAP	14	252	130	5
Mars_Simulation	24	370	206	3
Msv_Poker	16	18	8	4
Neuroph	12	41	27	4
Wro4j	28	209	130	12

**APs = Anti-patterns*

Table 6.12: *Proportion of classes in class diagrams that transfer same anti-patterns to the source code*

Project Name	# APs* in Design	# Same APs in the same classes in Design and Implementation	Proportions of same classes have same APs in Design and Implementation
Annoyme	10	5	0.5
ArgoUML	20	10	0.5
JGAP	14	5	0.38
Mars_Simulation	24	3	0.12
Msv_Poker	16	4	0.25
Neuroph	12	4	0.33
Wro4j	28	12	0.43
Average			**0.36**

**APs = Anti-patterns*

6.2.2.6 Complex Class

Regarding the Complex Class anti-pattern, very few occurrences are found in class diagrams, which means that architects and designers' tend to avoid excessively complex classes. However, developers do not seem to follow the same care during implementation as we observe proliferations of occurrences of the Complex Class anti-pattern in the source code of ArgoUML, JGAP, Mars, and Wro4j. Figure 6.5 shows show the number of occurrences of Complex class anti-patterns detected in the class diagrams and source code. We explain this observation by two facts. On the one hand, models

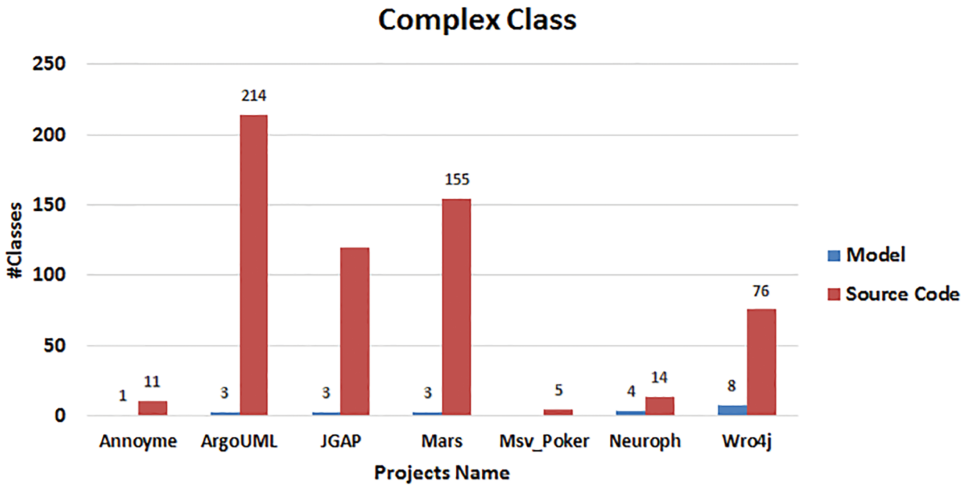

Figure 6.5: *Occurrences of the complex class anti-pattern in class diagrams and source code*

tend to be sketches of the actual implementation and, hence, do not contain all the details and complexity of the source code while the source code must, by its very definition, contain the actual algorithms, which may be intrinsically complex to implement. On the other hand, complex classes in source code tend to arise because of the lack of time for developers to research the best (i.e., simplest) implementation. Hence, it is our experience and observation that source code tends to be inherently more complex than necessary and, therefore, more complex that the models.

6.2.2.7 Large Class

Occurrences of the Large Class anti-pattern are absent from both models and source code, except for Wro4j, whose model contains five occurrences of the Large class anti-pattern, as shown in Figure 6.6. As reported by Vaucher et al. [87], Large Classes are sometimes present in systems because they are the best solution to some problems, for example when the problem is not easily decomposable. Such cases seem to be rare in models: only one system out of seven contains occurrences of the Large Class for the same reasons as mentioned above. So, modelers' focus on the essentials of classes, developers lack of time to introduce proper abstractions and, thus, their tendency to "grow" classes to implement new features.

6.2.2.8 Lazy Class

Lazy class, which is the most frequent anti-pattern among the four anti-patterns under study, is more prevalent in models than source code (see Figure 6.7. We explain this

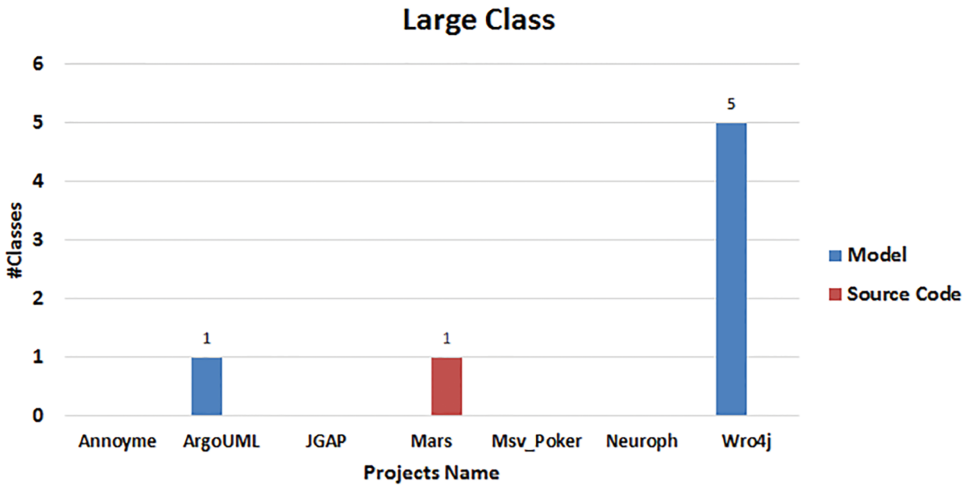

Figure 6.6: *Occurrences of the large class anti-pattern in class diagrams and source code*

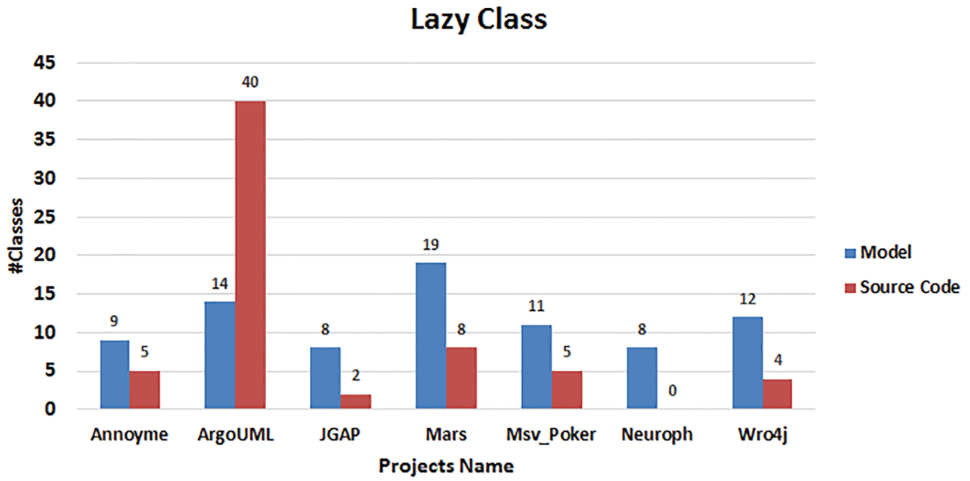

Figure 6.7: *Occurrences of the Lazy class anti-patterns in class diagrams and source code*

result by the fact that, during the design phase, developers try to anticipate future evolutions of the systems, which often lead to many abstract classes that do not contain necessarily enough behavior to justify their existence. These classes are considered Lazy classes by our detection technique and by definition of the anti-pattern. However, as Figure 6.7 shows, in all systems but ArgoUML, these excessive abstractions are corrected later by developers during the implementation of the systems.

6.2.2.9 LongMethod Class

Occurrences of the Long Method anti-pattern are also introduced in the source code in large number by developers. Again, we explain this observation and the difference between models and source code in two ways. First, models do not contain all the details necessary to identify Long Methods because of their very nature as sketches. Second, as previously mentioned, developers tend to implement features as fast as possible, under time pressure, and thus cannot take the time required to refactor their code and to avoid long methods.

6.2.2.10 Result Discussion

From the results presented in the results section, three of the four anti-patterns under study could be detected in class diagrams, i.e., during the design, which is considered an early stage of the software development life-cycle. We could not find occurrences of the Long Method anti-pattern in models because the detection of this anti-pattern is based on the numbers of Line of Code (LOC) of the methods, which is not available in class diagrams.

Table 6.10 shows that some classes contained in the models disappear in the source code, which can be considered two ways. First, having a class in a model and not having this class in the source code could be a design violation. For example, an architect or designer could have introduced a Facade between two subsystems, later the Facade is removed by developers for the sake of simplicity of implementation or performance of execution. Such a removal could yield to unintended accesses to some subsystems and also reduce information hiding.

However, having less information in models and not in source code such as classes, can also be the result of a lack of traceability in the project, because developers may have refined the model during implementation while failing to document the modifications and updating the models. Indeed, updates and changes to the source code without updating the models is a common practice observed in many software projects. Hence, our results confirm the software engineering lore that models are not synchronized with their source code by developers.

In addition, we observe that most of the classes that are in models and disappear in source code are Lazy Classes (see Figure 6.7, which confirms our intuition about developers refining the models because Lazy classes are the result of excessive abstractions. With a better knowledge of the system under development, developers may decide to remove some of the abstractions that result from architects' and designers' speculations about future evolutions of their systems. The average proportion of the same classes in the models and the source code that have same anti-patterns is 36%, which represents an important fraction of the total number of anti-patterns contained in the design. Hence, by acting early on these anti-patterns, architects, designers, and developers could improve the quality of their systems. Defects contained in design

models are known to be particularly expensive if they are not fixed quickly because classes in the models are the backbone of the source code and, in most models, are the most important classes in the source code. Thus, our results report and confirm that (1) classes in models may have anti-patterns, which propagate to the source code; and, (2) classes in both models and source code have the same anti-patterns in half the cases.

Also, following the broken windows theory [92], which states that a broken window may lead to a general degradation of the whole environment, we make the following argument. Similarly, we argue that when design problems are not fixed quickly, they tend to propagate in the system causing other problems. Therefore, it is important to track and fix design problems as early as possible in the development cycle. The results of this study show that software organizations can make use of anti-patterns detection tools like Ptidej during the design phase and track and fix anti-patterns in their software system as early as the design phase. Thus, anti-patterns detection tools will help prevent defects that could occur because of anti-patterns. Indeed, the refactoring of anti-patterns should be easier and less costly at modeling level than during implementation.

6.2.3 Effects of Anti-patterns in design on Software Changes and Faults

In this study, we focus on seven types of anti-patterns, Complex, Large, Lazy, Blob, ClassDataShouldBePrivate, RefusedParentBequest, and BaseClassShouldBeAbstract classes. We study the effect of anti-patterns of the classes appear in the design and the source code. We study the impact of anti-patterns appear in the design on the quality of the same classes in the source code measure based on the number of change- and fault-proneness the source code. Indeed, we use the Ptidej tool suite for detecting anti-patterns in the design (class diagrams).

6.2.3.1 Experiment Design

We categorized classes in the design into two categories:

- Anti-patterns category, which contains classes that have anti-patterns in design.
- No-Anti-patterns category, which contains classes that do not have anti-patterns in design.

Then we use the Mann-Whitney test to compare the means of the number of changes and the number of bugs between both categories.

Because the number of classes in designs is small as usual in open-source software projects, we did the same as in section 6.2.1.1 that we collected classes in the design and their mapped classes in the implementation. We use IntelliJ IDEA to find the mapped classes in the implementation to the classes in the design. Therefore, the Anti-patterns category becomes the classes that have anti-patterns in the design and their

Table 6.13: *Summary of classes used in the experiment*

Project Name	Anti-patterns Category	No-Anti-patterns Category	Total
ArgoUML	56	32	88
Wro4j	130	69	199

Table 6.14: *Means of classes changes in ArgoUML and Wro4j*

	Project Name	Categories	Mean
Changes	ArgoUML	Anti-pattens	30.8
		No-Anti-patterns	06.59
	Wro4j	Anti-pattens	16.6
		No-Anti-patterns	12.26

mapped classes in the implementation, and the No-Anti-patterns category contains the classes that do not have anti-patterns in the design and their mapped classes in the implementation. Table 6.13 shows both categories in both the ArgoUML and Wro4j projects. We take into account the number of changes and bugs occurred in different versions of both ArgoUML and Wro4j. We collected anti-patterns, changes and bugs for each class, then we entered the collected data into a database and made some queries for analyzing, filtering and organizing categories based on classes in the design (with their mapped classes) and occurrences of bugs and changes in the implementation of all versions.

6.2.3.2 Results

Tables 6.14 shows the mean of changes for both categories in the both ArguUML and Wro4j. Tables 6.15 shows the mean of bugs for both categories in the both ArguUML and Wro4j. We use the Mann-Whitney test because the test fits in our cases. The result of the Mann-Whitney test shows that there is a significant difference between the Anti-patterns category and the No-Anti-patterns category in ArgoUML in terms of changes and bugs, where p-value = 0.000 and p-value = 0.015 respectively.

In Wro4j, the results are the same, where the results of the Mann-Whitney test show that there is significant difference between the Anti-patterns category and the No-Anti-patterns category in both numbers of changes and bugs, where p-value = 0.000 and p-value = 0.004 respectively.

Table 6.16 and Table 6.17 show the correlations between numbers of changes and faults in Anti-patterns classes and No-anti-patterns classes with both LOC and AvgCyc. Table 6.16 shows that the numbers of changes have significantly higher correlations with LOC in No-anti-patterns classes than Anti-patterns classes in both ArgoUML and Wro4j: in systems with anti-patterns, size is not the only factor affecting change-proneness. The occurrence of anti-patterns also contributes to the occurrence of changes.

Table 6.15: *Means of classes faults in ArgoUML and Wro4j*

	Project Name	Categories	Mean
Bugs	ArgoUML	Anti-pattens	1.2
		No-Anti-patterns	0.15
	Wro4j	Anti-pattens	3.16
		No-Anti-patterns	2.38

Table 6.16: *Correlation between Changes, LOC and AvgCyc*

	Project Name	Categories	Correlation		R^2	Formulas
			AvgCyc	LOC		
Changes	ArgoUML	Anti-patterns classes	0.41	0.67	0.44	y=0.997+0.017
		No-anti-patterns classes	0.33	0.85	0.72	Y=0.149+0.028(LOC)
	Wro4j	Anti-patterns classes	0.06	0.59	0.35	Y=0.989+0.245(LOC)
		No-anti-patterns classes	0.02	0.7	0.48	Y=-0.698+0.465(LOC)

Table 6.17 shows the correlations between numbers of faults, LOC, and AvgCyc in ArgoUML and Wro4j: the numbers of faults in No-anti-patterns classes have significantly higher correlations with LOC than Anti-patterns classes in both ArgoUML and Wro4j. More faults occur in bigger No-antipatterns classes. For Anti-pattern classes, there is no strong correlation with LOC, which means that faults exist no matter the size of the classes. For AvgCyc, the correlations with changes is higher in Anti-patterns classes, which means that complex classes have more changes.

6.2.3.3 Discussion

The results show that there is a significant difference between Anti-patterns category and Non-Anti-patterns category. Therefore, and because of the mean of changes and mean of bugs of Anti-patterns category are bigger than No-Anti-patterns category in both ArgoUML and Wro4j, we conclude that the classes that have anti-patterns at design in ArgoUML and Wro4j, have more changes and bugs in the implementation.

From this experiment, we observe that classes that have antipatterns in the designs and corresponding classes in the source code of ArgoUML and Wro4j have more changes and faults in the implementation.

The broken windows theory [93] states that a broken window may lead to a general degradation of the whole environment and we argue that developers should solve these design problems before transferring them to the source code to reduce implementation and maintenance effort.

Similarly, we argue that when design problems are not fixed quickly, they tend to propagate in the system causing other problems. It is therefore, important to track and fix design problems as early as possible in the development cycle. The results of this study show that software organizations can make use of anti-patterns

Table 6.17: *Correlation between faults, LOC and AvgCyc*

	Project Name	Categories	Correlation		R^2	Formulas
			AvgCyc	LOC		
Bugs	ArgoUML	Anti-patterns classes	0.48	0.61	0.42	Y=-0.90+0.001(LOC) +0.054(AvgCyc)
		No-anti-patterns classes	0.35	0.81	0.65	Y=-0.072+0.001(LOC)
	Wro4j	Anti-patterns classes	-0.01	0.57	0.49	Y=0.377+0.046(LOC)
		No-anti-patterns classes	0.01	0.7	0.48	Y=-1.534+0.097(LOC)

detection tools like Ptidej during the design phase and track and fix anti-patterns in their software system as early as the design phase. Thus, anti-patterns detection tools will help prevent defects that could occur because of anti-patterns. Indeed, the refactoring of anti-patterns should be easier and less costly at modeling level than during implementation.

6.3 Threat to Validity

This section discusses the threats to validity of our study following common guidelines for empirical studies [94].

6.3.1 Construct Validity

In our anti-patterns study, we assumed implicitly that each anti-pattern is of equal importance, when in reality, this may not be the case. Future work must study the impact of the anti-patterns found in models in well-used dependent variables, such as class change- and fault-proneness, to assert whether all anti-patterns in models have a similar impact in the source code during implementation and maintenance.

6.3.2 Internal Validity

The UML Repository contains class diagrams collected from different categories. However, we still miss industrial models. Therefore, we ask companies to share their models for educational purpose.

The accuracy of Ptidej impacts our results. However, Ptidej has been successfully used in multiple studies [85][86][87][89], which have been ported to achieve high precision and recall [84]. However, other anti-pattern detection techniques and tools should be used to confirm our results.

In addition, the level of details of UML models affects the detection of anti-patterns in the class diagrams. It is possible that the lack of detailed information in class diagrams also affected the detection of anti-patterns. However, because the detection of these anti-patterns requires a high level of details in design (classes, methods, relations, and hierarchies (which are contained in the model)), we are confident about

the validity of our results. Yet, we will replicate our study with other techniques and tools in the future.

6.3.3 External Validity

We used two open source software projects in two studies, and made our conclusion based on these two systems. We use seven open source projects in another study and our conclusion is based on this data set. These systems that used in our studies are available in the UML Repository. It has different sizes and belong to different domains. Nevertheless, further validation on a larger set of systems is desirable, considering systems from different domains as well as systems from same domains.

6.4 Conclusion and Future Work

In this chapter, we describe a corpus study on the diagrams in the UML Repository. We show examples of an interesting relation between maximum coupling and size of the diagrams. More studies can be performed with this dataset, which can show behaviors of the designers, good patterns and bad patterns, also common patterns and anti-patterns.

We performed three experiments to investigate the relation between quality of the design and the source code.

In our study, we find that classes in the design have more changes and bugs than others. In the second, we investigated whether the design models produced by architects and designers before the implementation of software systems contain anti-patterns. We also examined whether the occurrences of the anti-patterns in models translate into the source code, affecting the same classes in models and the source code implementing these models. We conducted an empirical study on the prevalence of four anti-patterns: Complex Class, Large Class, Lazy Class, and Long Method, using both the architects' and designers' models of the seven systems (selected from the UML Repository) and the source code of these systems.

Our results showed that on average, 36% of the classes in the models that belong to anti-patterns also exist in the source code and also play roles in the same anti-patterns. Hence, we showed that anti-patterns appeared very early and concluded that architects and designers would benefit from help to identify and control these anti-patterns as early as possible.

Seven types of anti-patterns could be detected in the design phase: Complex, Large, Lazy, Blob, ClassDataShouldBePrivate, RefusedParentBequest, and BaseClassShouldBeAbstract classes. These anti-patterns mostly reappeared again in the source code in the same classes. Hence, it would be wise for maintenance teams to detect these anti-patterns early to save time and effort.

We found that classes in the design that have anti-patterns had more changes and bugs in the implementation. Therefore, anti-patterns should be detected and solved early in the design phase because in the source code it makes more changes and faults.

Future work includes analyzing more pairs of designers' models and their corresponding source code as well as analyzing more projects to propose prevention techniques. Because refactoring is easier at the design level, we aim to propose a technique to automatically refactor anti-patterns detected in models. For example, we envision that a Complex Class can be divided into two or more classes in models as well as in the source code. We will also consider anti-patterns as benchmarks for models quality and we plan to apply anti-patterns detection for whole models and systems in the UML Repository. We will make this information public to foster replications and contrasting studies.