The Impact of Antipatterns on the Change-Proneness of Java Systems

Master's Thesis

Paulius Raila

The Impact of Antipatterns on the Change-Proneness of Java Systems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Paulius Raila
born in Vilnius, Lithuania

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

The Impact of Antipatterns on the Change-Proneness of Java Systems

Author: Paulius Raila
Student id: 4121171
Email: paulius.raila@live.com

Abstract

Antipatterns are called *poor* solutions to design and/or implementation problems which are claimed to make object oriented systems hard to maintain. On the one hand, it has been shown that certain antipatterns negatively impact system comprehension, bug-proneness and change-proneness. On the other hand, previous studies reported that classes infected by certain design or code defects can remain stable and be even less change prone than other classes when changes are counted per lines of code (LOC). An example of such a case is a God Class acting as a parser. Because there are controversial results, more detailed investigations are needed. One limitation of the existing studies is that they count the changes in classes based on the number of file revisions or the number of changed lines, which, in our opinion, is not accurate.

In a recent study the authors analyzed a set of 12 antipatterns and demonstrated their negative impact on the change-proneness of four Java systems. We replicated a part of this study and extended it by performing an analysis based on fine-grained source code changes (*SCC*s) performed on Java classes during the evolution of the systems. Using *SCC*s extracted from the version control systems of 16 Java open-source projects, we investigate: (1) the change-proneness of Java classes participating in antipatterns, (2) change-proneness of Java classes affected by specific types of antipatterns, and (3) the likelihood that certain types of change are performed in classes affected by certain antipatterns.

Besides validating the recent findings with a larger data set and considering the *SCC*s, our results show that: (1) the distribution of *SCC*s performed in classes affected by antipatterns is statistically greater than the distribution of *SCC*s performed in other classes, (2) there is an association between the number of antipatterns affecting a class and number of *SCC*s performed in that class, (3) the classes participating in 3 antipatterns (e.g., *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife*) are more change-prone than classes affected by other antipatterns, and (4) certain types of change are more likely to be performed in classes affected by certain antipatterns (e.g., *API* changes are likely to be performed if the class participates in the *SwissArmyKnife*, *ComplexClass* or *SpaghettiCode* antipattern).

The results of our study are valuable for engineers. For example, software engineers can evaluate the quality of a system when they know which antipatterns have a negative impact on change-proneness. Moreover, they may want to avoid API changes in publicly exposed classes and, therefore they should consider refactoring classes affected by *SwissArmyKnife*, *ComplexClass* and *SpaghettiCode* antipatterns.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Pinzger, Faculty EEMCS, TU Delft
Daily supervisor:	Daniele Romano, Faculty EEMCS, TU Delft
Committee Member:	Dr. Jan Hidders, Faculty EEMCS, TU Delft

Preface

This is the result of my Master's thesis project, which I performed at the Software Engineering Research Group (SERG). I had an interesting opportunity to perform a study on the quality of software applications. This gave me both a good insight into current researches in the field of Software Engineering and contributes to my career as a software developer. For the opportunity to perform such a study I want to thank several people, who made a tremendous contribution.

During my master studies at TUDelft I enjoyed the courses tough by Martin Pinzger and Andy Zaidman. Namely, Software Reengineering, Software Evolution, Software Architecture. After that, I decided to write a literature study on the impact of design defects and their socio-technical aspects. Martin leaded me though this study and helped to discover many interesting topics for the master thesis. So, first of all I want to thank Martin and Andy for teaching very interesting courses at TUDelft and additionally Martin for helping to write a literature study, which was a good start for my thesis.

During my thesis I had a very good assistance from Daniele Romano. He provided valuable ideas to investigate, answered technical questions, assisted in writing and performing analysis and was a nice person to work with. Therefore, he significantly contributed to this work. Also together with Daniele Romano, Martin Pinzger and Foutse Khomh I have published my first paper [54], which this thesis is based on.

Paulius Raila
Delft, the Netherlands
November 25, 2012

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Software maintenance is the most expensive stage of software development processes [4]. Indeed, over the past two decades, maintenance costs have grown to more than 50% and up to 90% of the overall costs of software systems [12].

To help reduce the cost of maintenance, researchers have proposed several approaches to ease program comprehension, decrease change- and bug-proneness. These approaches include source code metrics (e.g., [53, 41, 6]), heuristics like usage of design patterns, code smells and antipatterns to assess the design of a software system (e.g., [51, 31, 61]). For instance, Romano *et al.* [53] showed that the interface usage cohesion metric (IUC) is correlated with the number of source code changes in Java interfaces. Cartwright et. al [6] found that defect density in the classes which use inheritance relations (measured by DIT and NOC metrics [7]) is significantly higher.

Recently, many researchers have focused their effort on analyzing the impact of antipatterns and code smells on software units (e.g., [31]). Antipatterns [5] are "poor" solutions to design and implementation problems. Many researchers do not distinguish antipatterns and code smells or bad smells, that are well described in Martin Fowler's book [40]. Antipatterns are often contrasted to design patterns [21] which are "good" solutions to recurring design problems. However, there exist studies that show opposing empirical evidence. Design patterns do not always impact the maintenance positively [28], [65], [52], [19]. Similarly, antipatterns do not necessary bring issues to the code. Current studies on the impact of antipatterns focus on code comprehension, change-proneness, bug-proneness and defect evolution.

Code comprehension is typically measured by performing an experiment where participants have to implement a functionality in different versions of classes (infected and not infected by antipatterns) [27], [26], [3], [1]. The performance of participants' works are typically measured in terms of time and quality of proposed modifications. The studies [27] [26], [3] showed that groups of participants who worked on the systems with refactored *GodClasses* were both faster and results were of a better quality. Similarly, Abbes et al. [1] showed that participants performed better on a system without antipatterns than on the one

infected by *Blob* and *SpaghettiCode*.

Existing studies on change-proneness and fault-proneness [30], [48], [17], [49] show that antipatterns and code smells infected classes are more change-prone and bug-prone than other classes. A number of such studies are presented in the Chapter 3, Section 3.3.

Because of the diversity and large number of antipatterns, support is needed for software engineers to identify the risky antipatterns that lead to errors and that increase development and maintenance costs. For this we need to obtain a deeper understanding of the change-proneness of antipatterns and the types of changes occurring in classes effected by antipatterns. Providing this insight is the main objective of this study.

In this thesis we investigate the extent to which antipatterns can be used as indicators of changes in Java classes. The *goal* of this study is to investigate which antipattern is more likely to lead to changes and which types of changes are likely to appear in classes affected by certain antipatterns. Differently to existing studies (*i.e.*, [30, 31]), our study proposes an approach based on fine-grained source code changes (*SCC*s) [14, 20] mined from the version control systems. Previous studies counted changes based on the class file revisions or the number of changed lines of code in each revision. We argue that, our approach is more accurate [14, 20]. It allows us to identify and count about 50 different types of code changes, many of which can occur on a single line. We also use *SCC*s to analyze the types of changes performed in classes affected by a particular antipattern. Moreover, we take into account the significance of the change types [13] and we filter out irrelevant change types (i.e., changes to comments and copyrights), that account for more than 10% of all changes in our dataset.

1.2 Research Questions

Using the fine-grained source code changes (*SCC*s) and antipatterns extracted from 16 open-source Java systems, we address the following research questions:

- ***RQ1:*** Are Java classes affected by antipatterns more change-prone than classes not affected by any antipattern? This research question is aimed at replicating the previous study [31], but this time considering fine-grained source code changes (*SCC*s). Additionally, we used another distribution test and performed correlation analysis (more in the Chapter 4). That allows us to show more evidence about the relation between antipatterns and change-proneness.

- ***RQ2:*** Are Java classes affected by certain antipatterns more change-prone than classes affected by other antipatterns - *i.e.*, does the kind of antipattern impact change-proneness? The results from this research question are meant to assist software engineers in identifying the more change-prone antipatterns. Defect detection tools could benefit from such information and give better suggestion for refactoring.

- ***RQ3:*** Are particular types of changes more likely to be performed in Java classes affected by certain antipatterns? The results of this question can assist software engineers in prioritizing antipatterns that need to be resolved in order to prevent certain

types of changes in a system. For example, changes in the method declaration of a class exposing a public API.

1.3 Results

Our study has confirmed the results of [31], counting *SCC*s as code changes and ignoring irrelevant changes (*e.g.*, changes to comments and copyrights). Additionally, we observed that:

- The number of *SCC*s performed in classes affected by antipatterns is statistically greater than the number of *SCC*s performed in other classes.
- There is an association between the number of antipatterns affecting a class and number of *SCC*s performed in that class.
- Classes affected by *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife* are more change-prone than classes affected by other antipatterns.
- Changes in *API*s are more likely to appear in classes affected by the *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife*; methods are more likely to be added/deleted in classes affected by *ComplexClass* and *SpaghettiCode*; changes in executable statements are likely in *AntiSingleton*, *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife*; changes in conditional statements and *else*-parts are more likely in classes affected by *SpaghettiCode*.

These findings suggest that software engineers should take into account the negative impact of antipatterns. For example, during the code review meetings such code fragments can be discussed. Besides that, they should consider detecting and resolving instances of certain antipatterns to prevent certain types of changes. For instance, they should resolve instances of *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns to prevent frequent changes in *API*s (declarations of classes, signatures of methods). To prevent changes in the condition expressions and *else*-parts they should resolve instances of *SpaghettiCode* antipattern as it might lead to bugs and complex unit tests. Likewise, to avoid method add/remove changes, *ComplexClass* and *SpaghettiCode* antipatterns should be avoided. And to prevent from the statement changes, software engineers should consider refactoring *AntiSingleton*, *SwissArmyKnife*, *ComplexClass* and *SpaghettiCode* antipatterns.

1.4 Structure

In the next chapter (Chapter 2), the concept of antipatterns is introduced and a couple of examples are given. After that (Chapter 3), we describe related work and concepts, including studies on the tools used for detecting design defects and the impact of those defects. The setup of the study and statistical analysis are explained in the Chapters 4, 5. The results for research questions *RQ1*, *RQ2*, *RQ3* are presented in the Chapter 6. Finally, we discuss the results, threads to validity (Chapter 8) and conclude our study (Chapter 9).

Chapter 2

Antipatterns

This chapter introduces the concept of design defects called antipatterns, code smells and related concepts. Additionally, we show that antipatterns can be identified and investigated during code review sessions, which are common software engineering practices. One of the purposes of such sessions is to maintain a high quality code by identifying and, later, refactoring the code which is hard to maintain.

2.1 Concept

According to Brown [5] antipatterns are repeated bad practices in software industry that initially appear to be beneficial, but eventually bring issues. He presented 40 antipatterns falling into 3 categories: software architectures, software development and software project management. 14 of antipatterns belong to the software development group. Only a subset of those is being investigated in this thesis. Antipatterns are generally introduced in software systems by the software engineers lacking the adequate knowledge and-or experience in solving a particular problem or having misapplied design patterns. Coplien [8] described an antipattern as "something that looks like a good idea, but which back-fires badly when applied".

Similar concepts to antipatterns are "bad smells" or "code smells", which were introduced by Kent Beck and gained more popularity after Martin Fowler wrote a a book about refactoring [40]. 22 bad code smells were shown and techniques to remove them (refactoring) were introduced. Later on, a study on classifying those code smells into categories appeared [36]. The purpose was to increase the comprehension of smells. The following 7 categories were introduced:

- Bloaters. Structures of the code that tends to grow in time. Eventually, they become too large to be effectively maintained. The examples are *LongMethod*, *LargeClass* code smells. *LongMethod* and *LargeClass* has too much code contained within method/class that it is hard to comprehend and reuse such a code.

- Object-orientation abusers. Object oriented (OO) design possibilities are not exploited. The examples are *SwitchStatement*, *ParallelInheritenceHierarchies* code smells.

According to the Fowler [40] the only good place for the switch statement is during the object initiation. Other situations, where this smell is used suggest improper OO design. *ParallelInheritenceHierarchies* indicates the inheritence structures which evolves parallel at the same time.

- Change preventer. Those code smells prevent or hinder the futher development of the code and violate the rule suggested by Beck and Fowler [40] that changes and classes should have a 1:1 relationship. It means that if there is a change in functionality it is preferred that it would effect a single class. *DivergentChange* and *ShotgunSurgery* smells belong to this group. *DivergentChange* indicates a single class that is modified multiple times during different types of changes. Contrary, *ShotgunSurgery* is a situation when many classes have to be modified on a single change.

- Dispensables. This group of smells represents unnecessary code structures, which can be removed from the code. Examples are *LazyClass*, *DataClass*, *DublicateCode*. *LazyClass* is a class that has little or no functionality. Similarly, *DataClass* is a class only holding the data and not performing any operations on it.

- Encapsulators. The code smells in this group concerns data communication mechanisms and encapsulation. There are only two smells: *MessageChain* and *MiddleMan*. When the class only delegates operations to other classes and does not perform any logic, then it suffers from *MiddleMan* code smell. Let's say class A wants to call class B, but doesn't have a direct reference to it. Instead it calls class C, which then calls B. If C performs only this kind of operations, then it is a *MiddleMan* smell. The *MessageChain* code smell occurs when there is a long method invocation chain. For example, a class A wants data from the class D. It has to call class B (there is a reference) to get an object C, then on that object class A calls class D, and finally class A calls a method to get some data on the class D. Basically, *MiddleMan* class could solve *MessageChain* code smell.

- Couplers. The code blocks, belonging to this group have a high coupling. For example, *FeatureEnvy* code smell is a method, having more relations to other classes than to the one it belongs to.

- Others. Those code smells are not classified anywhere else.

On one hand, there exist classifications of defects where code smells and antipatterns fall into different categories [66]. Smells are treated as code level defects and antipatterns as design level defects. On the other hand, the defects like *GodClass* are often referred both as a code smell and antipattern. Our study also mixes code smells and antipatterns and, therefore, we don't make a distinction.

2.2 Examples

This section presents three antipatterns, two of which we found later in the study (see more in the Chapter 7) to impact the change-proneness of the systems the most: *Blob*, *Spaghetti-*

Code and *SwissArmyKnife*.

2.2.1 SpaghettiCode

Spaghetti is a classical antipattern, since it existed from the invention of the programming languages [5]. Non-object programming languages are more prone to this antipattern. Large methods, global variables are the typical signs of it's presence. However, one can introduce this defect also in OO languages, when the concepts of OO are not used. Below a number of symptoms and consequences of *SpaghettiCode* antipattern [5] are provided:

- Methods are large and, therefore, hard to reuse.
- Objects and methods are process oriented. Often methods have no parameters and objects are named as processes.
- There exist little relations between the objects.
- A frequent form of code reuse in infected classes is code cloning.
- Inheritance is not used.
- Software quickly reaches a point where the effort of maintaining existing code is greater than the cost of developing a new solution.

We took an example of *SpaghettiCode* from the ArgoUML system - *Init* class from the *ArgoUML-0.10.1* release. As it is shown in the Figure 2.1, the class exhibits the following properties, which are typical for this antipattern:

- Inheritance is not used.
- The class has 86 attributes and most of them are static and global.
- The *Init()* method is the only method and it has 169 lines of code and has no parameters.
- Both class and method is named with a process name (*Init*).

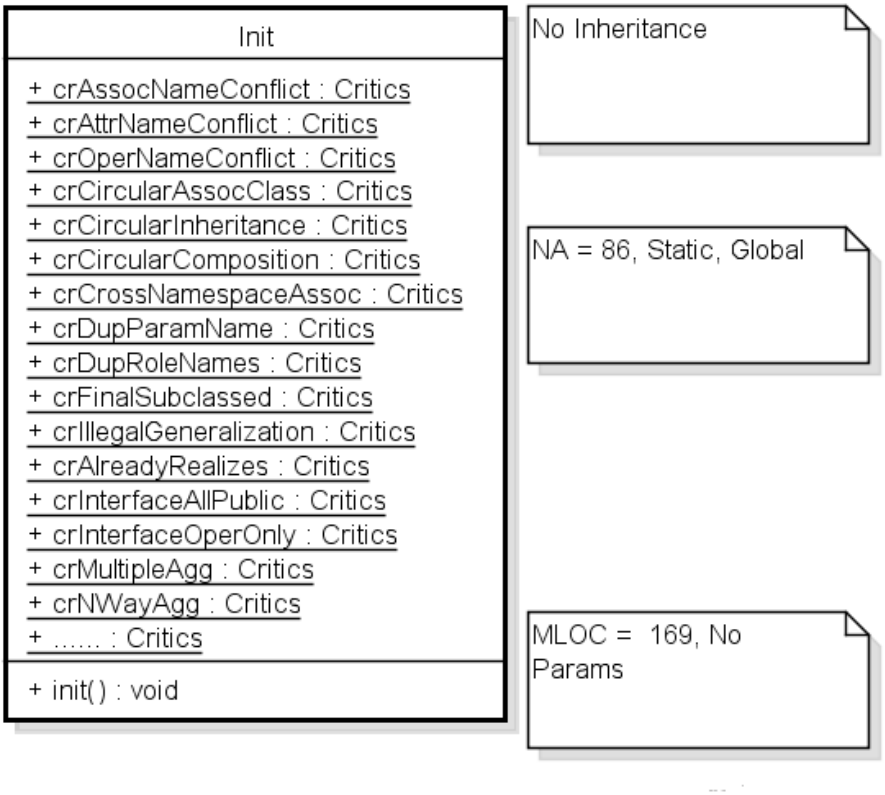

Figure 2.1: SpaghettiCode antipattern. The class org.argouml.uml.cognitive.critics.Init from ArgoUML-0.10.1 release of ArgoUML.

2.2.2 SwissArmyKnife

Another popular antipattern is the *SwissArmyKnife* (also known as *KitchenSink*). The designer of a class affected by this antipattern tries to solve different type of issues. The class has a number of responsibilities, provided by a number of interfaces. *SwissArmyKnife* are relevant in commercial software interfaces, where vendors are attempting to make their products applicable to all possible applications [5]. The antipattern is difficult to maintain and document. Developers find it hard to comprehend a class which has multiple responsibilities. However, we haven't found any empirical evidence to support such claim.

We took an example of *SwissArmyKnife* from the ArgoUML system - *Modeller* class from the *ArgoUML-0.10.1* release. As it is shown in the Figure 2.2 the class exhibits the following properties, which are typical for this antipattern:

- The class is not cohesive. It implements 16 interfaces and inherits functionality from 1 class.
- The class is too large. It has 2669 lines of code, 32 attributes and 122 methods.

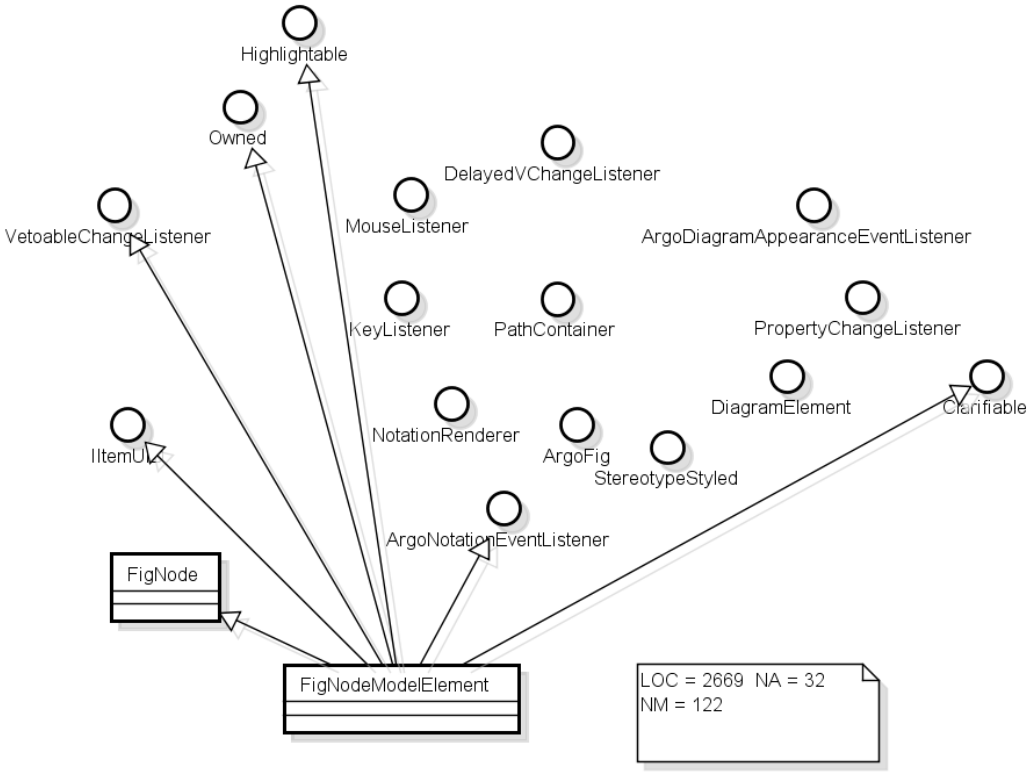

Figure 2.2: SwissArmyKnife antipattern. The class org.argouml.uml.reveng.java.Modeller from ArgoUML-0.10.1 release of ArgoUML.

2.2.3 Blob

Another popular antipattern is a *Blob* (also known as *GodClass*). It is used in the designs where one class monopolizes the processing, and other classes primarily encapsulate data. The class is composed out of the main complex controller class surrounded by data classes. Such classes are considered of a procedural design. However, *Blob* can also be implemented using OO languages [5].

Below are provided a number of symptoms and consequences of *Blob* antipattern [5]:

- *Blob* is a single class with a large number of attributes/operations.
- There is an overall lack of cohesiveness of the attributes and operations.
- The antipattern has many associations to data classes, which hold data and do not perform any operations.
- The *Blob* does not use a proper OO design. It is very centralized and, usually, does not use inheritance.

- The *Blob* class is hard to reuse and test because of the complexity. Additionally, the complexity of the class tends to grow in time.

- It can get hard to load *Blob* class into memory due to the huge size.

We took an example of *Blob* from the Rhino - *Parser* class from the *rhino1_6R3* release. As it is shown in the Figure 2.3, the class is large in terms of LOC, NA and NM. It also holds two data classes (*CompilerEnvirons*, *ScriptOrFnNode*). More than 90% of their methods are just getters/setters or overrides the standart functionality of the object class (*toString, clone, equals, finalize, hashCode*). Therefore, the class is treated as a *Blob* antipattern.

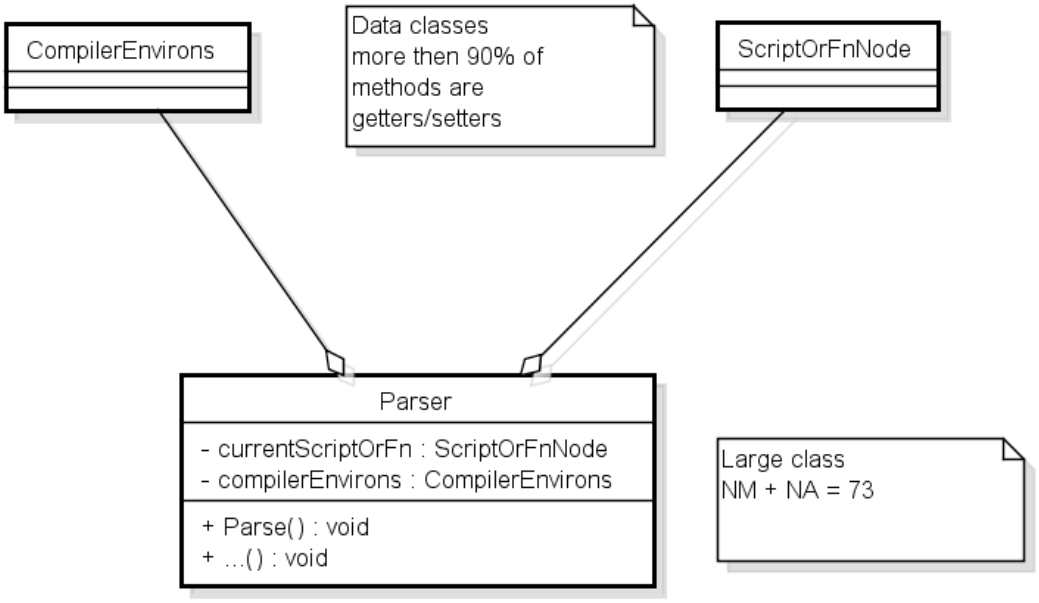

Figure 2.3: Blob antipattern. The class org.mozilla.javascript.Parser from rhino1_6R3 release of Rhino.

2.3 Application

One of the existing applications of antipatterns is code review sessions where software engineers in groups or individually check each others' code. Mäntylä *et al.* [38] spotted a significant importance of software maintainability issues, while observing code review sessions. During his research 75% of defects identified in the code review sessions were not related to a functionality but to the code evolution. This finding confirmed an opinion of Siy *et al.* [58], that the biggest part of defects found in code reviews are code evolvability issues, which adversely affect the code maintenance.

To come up with such results the author looked both into the industrial and the code written by student. A quick overview of the experiments is presented in the Figure 2.1:

	Industrial Reviews	Student Reviews
LOC reviewed per session	100-5000	100-300
N of reviews	9	23
Domain	Engineering	Questionnaire data analysis
QA-method prior to code review	Functional or automated testing	JUnit testing
Individuals (reviewers)	Industrial developers	Students
Review instructions	Find problems in the code Checklists (existed but mainly not used)	Find and classify defects as functionality, evolvability, or other Checklists (60% had utilized checklists in individual preparation)
N of reviewers	1-4 + author of code	6-7 + author of code
Data collection method	Observation of reviews	Submitted defects lists and source code

Table 2.1: Code review settings of the experiments. [38]

In general, the research showed three types of code review findings:

- Functional Defects constitute 13-21% of all defects.
- Evolutional defects constitute 71-77% of all defects.
- Others are false positives.

Evolutional defects found by the code reviews were further classified by Mäntylä into Documentation, Visual Representation and Structure. The last category includes code smells organized into categories. The full organization structure is presented in the Figure 2.4:

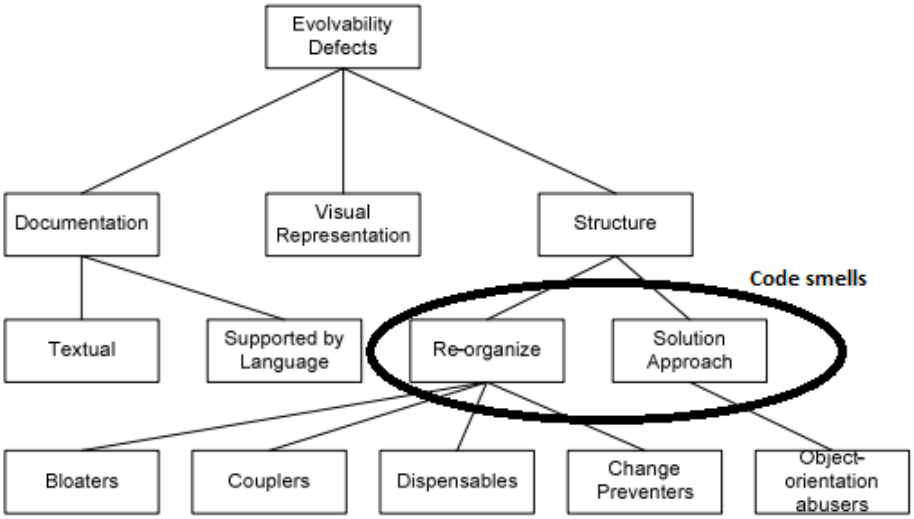

Figure 2.4: Classification of Maintainability issues. Code smells are outlined. [37]

About 50% of all the maintenance issues felt into Structure category and, therefore, are treated as code smells. To sum it up, we can see that code smells constitute a big part of code review findings at least for some systems.

First, this chapter showed that the concept of antipatterns is not completely clear and, therefore, many researchers use it with a different meaning. Second, antipatterns can be detected during code reviews. The next chapter will introduce the techniques and tools for detecting antipatterns from the source code.

Chapter 3

Related Work

This chapter overviews the existing studies on antipattern detection techniques, the impact of antipatterns on software maintainability and introduces the concept of RM (Repository Mining). The goal is to motivate our choice for the antipattern detection tool, introduce the related concepts similar studies.

3.1 Detection of design defects

While initially antipatterns were described in a textual form (see the Chapter 2), there were no strict rules for identifying them and a manual investigation was preferred. However, it is not an efficient approach, especially for the large scale systems. Therefore, many techniques and tools were introduced to assist software engineers and researchers in this process. This section presents four tools widely used in design and code defect detection: DECOR, JDeodorant, Evan, JCosmo and an overview of the available techniques. They are based on metric and meta-model detection techniques.

Some approaches for complex software analysis use visualization [11, 57, 10]. Although visualization is sometimes considered as an interesting compromise between fully automatic detection techniques, which are efficient but loose track of the context, and manual inspections, which are slow and subjective [33], visualization requires human expertise and is thus time-consuming. Sometimes, visualization techniques are used to present the results of automatic detection approaches [34, 63]. However, we chose not to describe those tools in details as they are not suitable candidates for this study.

3.1.1 DECOR

First, we start by presenting the technique we encountered most frequently during the literature study and chose to use for our study. Noha et al. [43] presented a study contributing to three aspects: it presented a novel method, called DECOR to detect code smells and antipatterns, first, a detection technique based on DECOR, DETEX, second, and the evaluation of DETEX on open-source projects (XERCES, QuickUML, PMD, NUTCH, Lucene, Log4J, GanttProject, Azureus, ArgoUML) in terms of precision and recall, third.
DECOR is composed out of 5 steps:

1. Description Analysis. Concepts are gathered from the descriptions in the literature to a reusable concept vocabulary.

2. Specification. Concepts gathered in step 1 are combined to form detection rules for smells.

3. Processing. The Specifications are translated into algorithms.

4. Detection. Potential smells on a given code are identified.

5. Validation. The Potential smells are manually verified.

DETEX uses the domain specific language (DSL) to create card rules for smells. From the cards smell detection algorithms are generated. Additional information about the code generation part is also available in [45]. A full mapping of DETEX and DECOR steps is visible in the Figure 3.1:

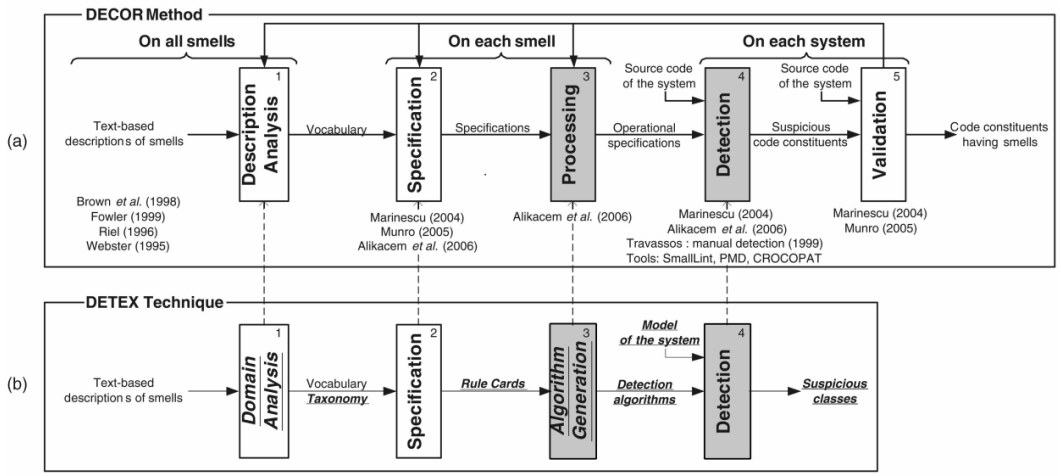

Figure 3.1: (a) Decor method compared to related work. (Boxes are steps and arrows connect the inputs and outputs of each step.) (b) DETEX detection technique. (The steps, inputs, and outputs in bold, italics, and underlined are specific to DETEX compared with DECOR.) [10]

Card rules are composed out of a list of properties, which can be either a code metric or a relation with other rules, a combination of other properties or an operator (intersection, union, etc.). A property can be:

- Measurable. The property is expressed by a numerical or an ordinal value for a specific metric. Full set of ordinal values is composed of: very high, high, medium, low, very low.

- Lexical. The property provides a rule for a class and properties/methods names.

- Structural. The property is related to the structure of classes, interfaces, methods, fields, etc. For example, NO POLYMORPHISM checks whether a class uses polymorphism.

An example of a rule card for the Spaghetti Code in the listing 3.1.

Listing 3.1: Rule card of the spaghetti code. [10]

```
1  RULE_CARD : SpaghettiCode {
2      RULE : SpaghettiCode { INTER NoInheritanceClassGlobalVariable
3          LongMethodMethodNoParameter };
4      RULE : LongMethodMethodNoParameter { INTER LongMethod MethodNoParameter };
5      RULE : LongMethod { (METRIC: METHOD_LOC, VERY_HIGH, 0) };
6      RULE : MethodNoParameter { (METRIC: NOParam, INF, 5, 0) };
7      RULE : NoInheritanceClassGlobalVariable { INTER NoInheritance ClassGlobalVariable };
8      RULE : NoInheritance { (METRIC: DIT, INF_EQ, 2, 0) };
9      RULE : ClassGlobalVariable { (STRUCT: GLOBAL_VARIABLE, 1) };
10 };
```

The evaluation of DETEX showed that generated detection algorithms has a recall of 100% and the precision varied depending on the system and antipattern. For instance, the highest precision was archived for the *Blob*. It varied from 67% to 100%, while the lowest one was measured for the *SwissArmyKnife*, varying from 11% to 42%. Furthermore, the algorithms were quit fast. For the chosen systems the most costly analysis was for a *Blob* that took several seconds. We chose to use DETEX for our experiment. More details are provided in the Section 4.5.

3.1.2 EvAn

Another metric based detection tool EvolutionAnalyzer (EvAn) was developed by Olbrich et al. [49]. The tool is made to extract code smells from Java code in SVN repositories. After the code is extracted, a meta-model, containing code metrics and defect information for a number of revisions is stored into the database. Finally, the stored data is analyzed and smells are identified using heuristic rules. Although it works similar to other metric based tools, it has a built in defect mapper which allowed the authors to investigate the impact of code smells on fault-proneness of the system. The overview of the system architecture is presented in the Figure 3.2. More information about the study is provided in the Section 3.3. The paper, however, does not provide any evaluation of the tool.

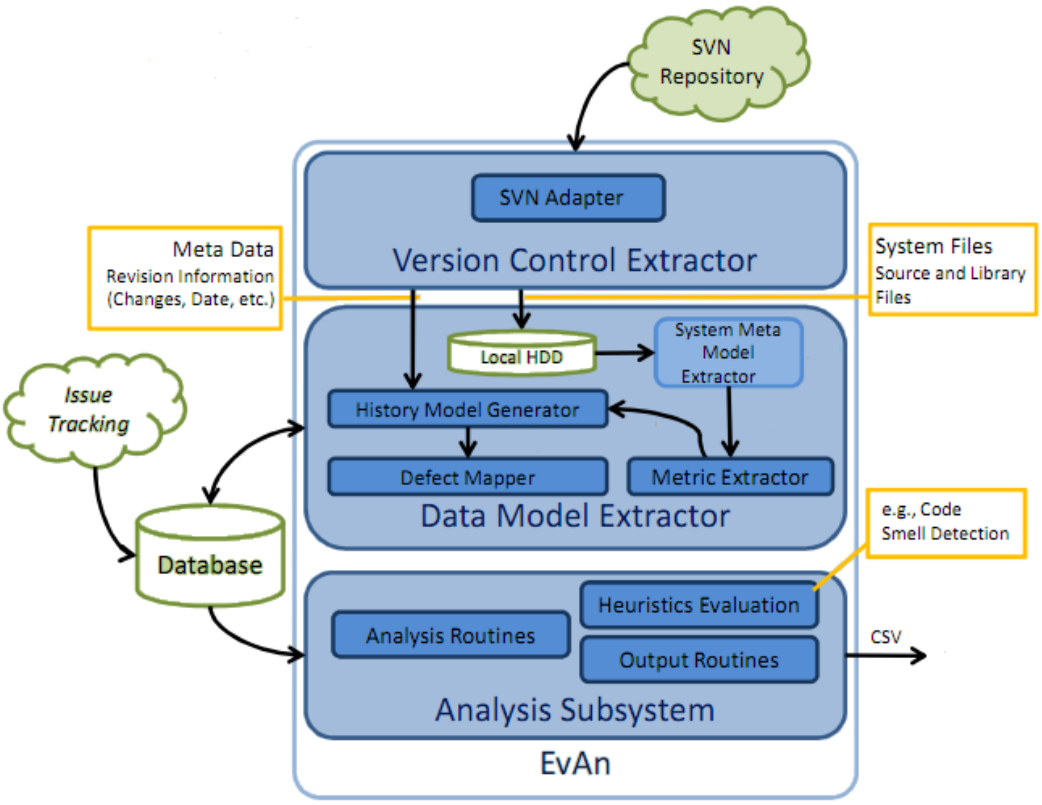

Figure 3.2: Schematic overview of the EvAn.[49]

3.1.3 jCosmo

Van Emden et al. [62] presented a prototype version of the tool jCosmo to detect code smells using the source code model together with visualization. Each code smell exposes a number of smell aspects, which are used to identify it. The authors pointed out two types of such code aspects:

- Primitive smell aspects. They can be observed directly from the code (e.g., A switch statement).
- Derived smell aspects. Those are inferred from other aspects (e.g., A class doesn't use inherited methods).

The detection technique used by jCosmo works as follows. First, the tool parses all interesting entities and inspects them for primitive smell aspects. The information is stored into the repository. Finally, jCosmo infers derived smell aspects from the repository. This technique is illustrated in the Figure 3.3.

According to the authors, people who inspect the code need to know which parts of the system are affected by the smells and, therefore, they have introduced visual views. Views

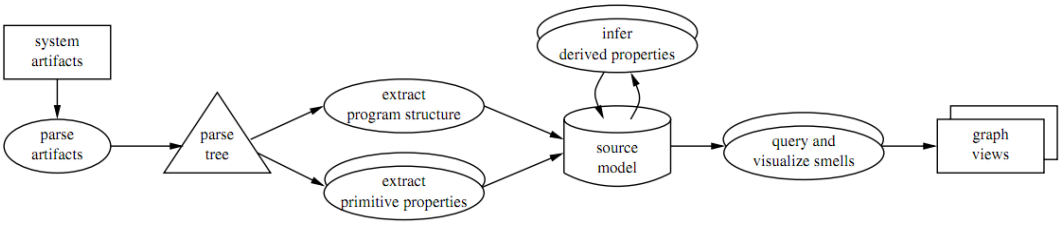

Figure 3.3: Architecture of code smell browser. [62]

represent code structure (packages, classes, interfaces, methods) and code smells, attached to particular code elements in trees.

The tool was evaluated on a research oriented CharToon system, which helps to develop facial expression animations. Although no detailed evaluations were performed, the maintainers of CharToons, found the results from jCosmo useful for conformance checking and refactoring support.

3.1.4 jDeodorant

Another non-metric based tool we came across is JDeodorant, a plug-in for the Eclipse IDE. Based on the entity meta-model, it identifies bad design smells, and removes them by applying appropriate refactoring. Currently the tool works with four bad smells: *FeatureEnvy*, *TypeChecking* (*SwitchStatement* according to Martin Fowler), *LongMethod* and *GodClass*. There are several researches concerning this tool. We will briefly describe three of them.

First, a study [15] presents how JDeodorant deals with *GodClasses*. The idea is the following. When a class does too much, a new class can be extracted from it, by performing Extract Class refactoring. To do so, however, we have to decide which functionality is less related to the class. Firstly, a set of candidate classes for refactoring is calculated by a hierarchical agglomerative clustering algorithm. Methods and attributes of the class are treated as entities and at the beginning they belong to different clusters. At each algorithm step closest clusters are merged. The closeness is determined by how similar entity sets, which contains all the members of classes that use or are used by the entity in question, of two entities are. Secondly, a user is suggested to perform a number of Extract-Class refactoring on each class. Experts confirmed the effectiveness of the tool after evaluating it on open-source JHotDraw system. They agreed they would perform 56% of suggested refactoring and even confirmed that in some cases they wouldn't have spotted a possible refactoring themselves.

Second, a research [39] was done on *FeatureEnvy* bad smells. The study showed how JDeodorant recognizes *FeatureEnvy* smells and aids to re-factor them. The tool employs the ASTParser of Eclipse Java Development Tools (JTD). Based on the similar technique as for the *GodClasses* removal tool suggests a new class for a method, based on the relation with classes. The evaluation has been conducted on two well-known refactoring examples,

Video Store [40] and Lan-simulator [55]. The results showed a high accuracy: 6 out of 6 and 7 out of 8 *FeatureEnvy* smells were successfully identified.

Finally, Tsantalis et al. [47] showed that JDeodorant is capable of identifying *Type-Checking* (*SwitchStatement* according to Martin Fowler) code smells. AST analysis allows the tool to suggest a corresponding refactoring, Replace Type Code with State/Strategy or Replace Conditional with Polymorphism. However, the method proposes the first refactoring whenever it finds a ”code-like” attribute on which conditional statements are executed, which doesn't conform to Fowlers opinion.

3.1.5 Other Tools

In general, we presented two main approaches to detect code smells: metric and meta-model based. There were also attempts to optimize current approaches by using domain specific information or the history of a system.

In the study [9] change history of the classes was used to filter out code smells which are relatively stable. As a metaphor human diseases were compared with the code smells. Some of diseases are present when a person is born, but if the organism is accustomed to them and they make no danger to one's health, we can ignore them and even not treat as diseases. The same holds for the code smells. For example, if we have a generated smelly code, which we are aware of and it doesn't need to be changed, we may not treat this code as a code smell. The idea of harmless code smells is also supported by a number of studies found that certain code smells may not adversely influence the change-proneness of code [59, 49].

Another study [67] showed that metric based rules for code smells detection can be adjusted for the domain to increase accuracy. The group of specialists evaluated the results from the CodeWizard tool applied on the certain system and suggested how the thresholds values for metrics can be adjusted. After the adjustment, the accuracy of code smell detection has increased. Bayesian belief networks can also be used [32] to assist metric based rule approach.

Although most of the section was focused on presenting metric based techniques, a number of tools were skipped. Examples are Incode[1], CodeWizard [68], iPlasma[2], inFussion[3]. For example, InCode is an eclipse plug-in system, which assists developers by pointing out code smells and explaining them. More information about these tools are available in [16].

For our study we chose a metric based detection tool - DECOR as it showed a high precision and recall, works well with Java source code, has a flexible rule engine and a unique semantic rule detector.

[1] http://www.intooitus.com/products/inCode.htm
[2] http://loose.upt.ro/iplasma/index.html
[3] http://www.intooitus.com/products/infusion.htm

3.2 Repository mining

Mining Software Repository (MSR) or just Repository Mining (RM) in our context refers to the investigation technique based on the software repositories. Repositories include code version control systems (e.g., SVN, CVS), issue tracking systems (e.g., Bugzilla), or communication channel archives (e.g., skype logs, email). A literature survey, showing a taxonomy of known approaches on MSR is presented by Huzefa et al. [29].

Such data sources usually exist through the life-circle of a software system and, therefore, thousands of records can be found. Including source code commits (date, user, code, comments, etc), bugs (severity, date, component, etc).

In our study, we used Evolizer with built-in repository miner, which supports CVS, SVN an GIT repositories and extract the information into database (see the Chapter 4).

3.3 The impact of design defects

This section presents the papers concerning the impact of design defects. Beck and Fowler [40] treated them as code illnesses that should be healed with appropriate refactoring. However, there is a lack of empirical evidence to support opinions about the impact of design defects. We, therefore, present papers, which tried to present such evidence. We discovered two main types of impacts: code evolution and code comprehension.

The first group contains design defect evolution and impact on change-proneness and fault-proneness of code. Most works use RM as a primary method for the code evolution analysis. Furthermore, bug tracking systems are combined with RM to map fault fixes and commits. As a result, fault-proneness can be evaluated.

Code comprehension impact studies try to answer how easy is it to understand the code containing the smells. Easier understandable code leads to less costly software changes and, therefore, is preferred in code maintenance.

3.3.1 Code Evolution

Code evolution related studies investigate how the code changes and identifies relations between code smells and such changes. First, we present the works about the code smell evolution. They try to answer whether a presence of code smells influences the appearance of the new ones. Second, we show how change-proneness and fault-proneness of the code is impacted by smells.

Code Smell Evolution

We found two papers focusing purely on the evolution of code smells. Additionally, there is one paper concerning both code smell evolution and the change-proneness.

Evolutional Trends The first one [64] looked deeper into the evolution of *GodClasses*. Source code repositories were mined from Eclipse and the Xerces open-source Java sys-

tems. To identify *GodClasses* a Bayesian belief network [18] was applied. For each class a probability of being a *GodClass* was calculated based on the following measures:

- Size (number of methods and attributes)
- Cohesion (using LCOM5 [25])
- Number of associated data classes
- Lexical analysis of a class and it's method names

Based on how these probabilities changes the evolution trends were created and classified into 7 categories (Constant, Sharp Improvement, Gradual Improvements, Temporary Badness, Temporary Relief, Sharp Degradation, and Gradual Degradation). Having all the required values, the author showed how often each evolution trend occurs in each project.

One of the interesting findings was that the largest group in both projects appeared to be the constant evolutionary trend. Most of the time godliness of the classes remained stable. Also the ratio of *GodClasses* compared to the normal classes remained more or less stable. Additionally, the Xerces primary developer was asked why the particular *GodClasses* were used. He mentioned that the reasons of those smells lies in the complexity of problems they deal with. To get rid of the doubt that the code containing *GodClasses* is poorly implemented the authors decided to investigate whether design patterns were used as it can indicate the quality of code. Interestingly, it was revealed that 82% of *GodClasses* were involved in design patterns.

As a conclusion it is stated that, although a smell is considered as a bad code, there are many cases when they cannot be improved and they remain relatively untouched. As an example a parser class is given which on one hand, has a complex logic, but on the other hand, is tightly coupled and hard to decompose.

Counting the Smells The second work concerning the evolution of the code smells is written by Chatzigeorgiou et al. [2]. He investigated how other code smells change over time. Namely, *LongMethod*, *FeatureEnvy* and *StateChecking* smells were mined from repositories and captured by the jDeodorant tool. As a source code input for the tool, two open-source Java systems, JFlex and JFreeChart were chosen. The authors measured how the number of each bad smell evolves through software revisions. A number of important results are presented below:

- The number of code smells increases over time. Authors think that this might be specific for the open-source systems, since they are not usually involved in a systematic preventive maintenance.
- The majority of code smells (for *LongMethod* in JFlex even 89,9%) persist from their appearance to the latest version of a system.
- A large number of code smells (59.59% for *LongMethod* in JFreeChart) were introduced during the initial design of a particular component.

- In the vast majority of the cases where code smells were removed, targeted refactoring activities were not the reasons for this.

Contrary to the previous work, from the first glance it may seem that this time researchers have discovered that the amount of code smells increase over time. However, one can miss an essential detail here.The total size of the projects also increases over time. Therefore, in the previous work a relative number of classes involved in the code smells was not taken into account.

The idea that code smells do increase over time was once more refuted by Olbrich et al. [48]. Besides change-proneness the authors also looked into the evolution of the smells. To be more precise, they tried to answer the following questions: Does the number of code smell increase over time? Does the fraction of components having code smells increase over time? The results showed the following:

- The total number of code smells did not increase steadily. Activities such as refactoring can influence them.

- The fraction of components having the code smells does not necessary increase over time. Reasons could be the same as to the previous point.

Change-proneness and Fault-proneness

A common way to measure the badness of a smell is to check whether the code containing the smell is related to more changes and faults than other code. We found four papers falling in this sub sub-section. Those works have analysed the same aspect as our study (change-proneness).

Change-proneness of 29 code smells A number of code smells were investigated by Khomh et al. [30]. He looked how code smells impacts change-proneness of two open-source projects, Eclipse and Azure. DECOR tool was chosen to identify 29 different code smells [30] from the code repositories. Identification was based on the rules combined from code metrics (McCabe complexity, Brian Henderson-Sellers cohesion metric LCOM5, etc.) and threshold values.

First, the authors tested whether there is a significant difference in the proportion of classes exhibiting at least one / none changes among the classes participating and not participating in code smells. The Fisher exact test [35] was used to measure if there is a difference between the two samples as well as odds ratio (OR) [50] to indicate if the probability of a change is greater in one sample than in another.

Second, they identified a relation between the number of code smells in classes and change-proneness. The numbers were compared between the classes which were changed at least once and the ones which were not. The parametric t-test and non-parametric Mann-Whitney tests were used to compare the two samples. Furthermore, the authors assessed the magnitude of difference with Cohen d effect size [35].

Finally, it was checked, if some code smells impact change-proneness more than the others. For that Khomh et al. applied a logistic regression model [50]. The results showed the following:

- Classes, containing code smells have a higher likelihood to be changed.
- Classes, containing more code smells are more likely to be changed.
- Class change likelihood depends on the particular code smell in the specific context (system, version). For example, the worst code smell in terms of change proneness for the Azureus appeared to be *NotAbstract* smell, while for the Eclipse *HasChildren*, *MessageChains* and *NotComplex* smells.

The research questions of this study are similar to our's. However, here, the authors focused on smaller scale code defects. In DECOR settings there are code smells and design defects. This study used code smells, while our study - design defects. Additionally, we counted changes as *SCC* and investigated what types of changes are common for each defect.

Change-proneness of GodClass and ShotgunSurgery Another research on changeproneness was conducted by Olbrich et al. [48]. He analyzes two code smells: *GodClass* and *ShotgunSurgery*. The source code from the SVN repositories of two open-source projects, Lucene, Apache Xerces was analyzed with the CodeWizard tool. Similarly to the other code smell detection tools, which are described in the Section 3.1, it combines code metrics (WMC, TCC, ATFD, CM, CC for *GodClass* and *ShotgunSurgery*) and thresholds values to identify smells.

The authors were interested in the following questions: Do classes infected by code smells change more frequently? Are infected classes modified in bigger code churns? Modification meant change, delete or add operation. A code churn stands for a number of code lines. To answer these questions, the numbers of entities containing and not containing a smell were compared over the change history. In order not to pay attention to insignificant changes threshold values were introduced. Findings are the following:

- Classes participating in the code smells were more change-prone.
- Classes containing code smells did not suffer from larger code churn changes.

Compared to our study, the authors used a limited set of defects and haven't performed the analysis on types of changes. Moreover, they counted changes as changed lines, which is not as precise as *SCC*.

Introducing Fault-proneness Another researched topic of the code evolution is fault-proneness. Khomh et al. [17], [31] has continued his work (previous section) on the impact of design defects. Fowlers code smells (*LargeClass*, *LazyClass*, *LongMethod*, *MessageChain*) and other design defects (13 in total) were referred to as antispatterns. They were examined from the 54 releases of open-source Java systems, ArguUML, Eclipse, Mylyn, and Rhino.

Besides investigating the change-proneness of bad smells as he did in the previous work (described in the first part of this subsection), he also analyzed how antipatterns are related to the fault-proneness of systems. Thus, several new questions were raised. Does the proportion of classes participating in at least one bug fixing-change between to releases differ between classes participating in antipatterns or not? Do classes participating in specific antipatterns are more bug-prone than others? Is LOC of a class a main cause of a higher fault or change proneness?

During the study a class was treated as fault-prone if it underwent at least one fault-fixing change between two subsequent releases. Those were identified in two different ways: by manually validating publicly available bug reports or by using issue tracking systems such as Bugzilla in case they were present.

The first question as well as the similar one concerning change-proneness was analyzed using Fishers exact test and odds ratio (OR) as in the previous work of Khomh. Similarly, the authors used logistic regression model to find which antipatterns differs from others in terms of fault-proneness. For the final question 3 step approach was used. Firstly, for each release the average size of classes participating in antipatterns and not was compared using Mann-Whitney test and Cohen d effect size. Second, the same was calculated for each class participating in each different antipattern. It is expected that for some of them there won't be any significant difference. Third, the same tests were executed to assess how the classes participating in size-related (*Blob*, *LargeClass*, etc.) antipatterns differ. To accomplish it, classes that had size above 75% percentile were divided into two groups (antipatterns infected/ not infected). Besides that, the study also looked into the types of changes: They considered two types of changes: structural and non-structural changes. Structural changes are changes that would alter a class interface while non-structural changes are changes to method bodies.

The overall results were the following:

- Classes that participate in antipatterns in most releases appeared to be significantly more change-prone.
- Some antipatterns appeared to be significantly more correlated with change-proneness than others. In particular, *MessageChain*, *LongMethod* in ArgoUML, Eclipse, and Mylyn; *LongParameterList* in ArgoUML and Eclipse; *AntiSingleton* and *RefusedParentBequest* in ArgoUML, *ComplexClass* and *LazyClass* in Eclipse.
- Classes that participate in antipatterns are more fault-prone.
- Some antipatterns appeared to be significantly more correlated with fault-proneness than others. In particular, *MessageChain* in Eclipse and Rhino; *AntiSingleton*, *ComplexClass*, *LazyClass* and *LongMethod* in Eclipse.
- Some antipatterns are related to the size of class, but the size alone doesn't explain the greater change and fault-proneness.
- Structural changes occurred more often in the classes, which participate in antipatterns.

We replicated this study by adding additional research questions, using *SCC* and a larger dataset.

Further study on how code smells impact fault and change proneness of code was conducted by Olbrich et al. [49]. He continued his work on change-proneness and also included fault aspects of code smells. This paper tries to explain why *GodClass* and *BrainClass* code smells are considered to increase defect rate, and negatively influence change-proneness of systems. A straightforward explanation would be that classes infected with such code smells simple have more lines of code and, therefore, they are both more likely to be changed in the future and to contain defects.

RM approach was used to investigate three open-source systems, Log4j, Apache Lucene, Apache Xerces. In order to mine repositories for Java code Evan tool was developed to extract code smells from a versioning system containing Java code. Similarly as other tools (see the section 3.1, code smell detection is based on a combination of code metrics and thresholds.

Besides EvAn, Jira, BugZilla were used as bug tracking systems. They had to map bug fixes to repository code commits. During the study calculated a WDR (weighted defect rate) was calculated. Weights were given to the bugs to indicate the severity. To be more precise, bug tracking systems classify bugs into Blocker, Critical, Major, Minor, Trivial. For each category numbers 16, 8, 4, 2, 1 were assigned correspondingly.

The same analysis method was applied as in the previous Olbrichs study, yet LOC metric was taken into account for change and fault proneness as well as WDR and WDR per LOC were calculated. WDR of a class is c the sum of all weighted defects. Authors came up with the following findings:

- CF (change frequency) of *GodClasses* and *BrainClasses* is higher compared with other classes.
- CF per LOC is lower of *GodClasses* but this doesn't hold for *BrainClasses*.
- CS (change size) for the *GodClasses* is higher compared with other classes. This doesn't hold for *BrainClasses*.
- CS per LOC is smaller for both *GodClasses* and *BrainClasses* compared with other classes.
- WDR and WDF per LOC have not brought any significant results for all the analyzed systems.

The authors concluded that when the presence of *GodClass* and *BrainClass* smells is not extreme, they can be even beneficial, as they are less change-prone when LOC is taken into account. The study differs from our study in the same way as the previous study of the authors.

3.3.2 Code Comprehension

As it is necessary to understand the code before a bug can be fixed or a new feature added, most of the papers here perform experiments that simulate this kind of situation. Also ques-

tionnaires were commonly used to provide a supplementary data. This topic contains four papers. All the studies measure the impact of *GodClass* smells. One, however, additionally investigated the *SpaghettiCode* antipattern. Our study didn't focus on the code comprehension.

God Class Studies with Experiment and a Questionnaire

Deligiannis et al. [27] looked into the impact of the *GodClass* smells on the program comprehension and maintainability. Two groups (A and B) of two people were involved in the experiment. During it, the subjects needed to understand the system and perform several design tasks. All the participants were not new in the field OO technologies. Tasks were performed on the small mobile tariff selector system (TSS). The system was implemented in Java programming language with AWT graphical user interface and contained 16 (Design B) or 18 (Design A) classes in total. The design B contained a centralized functionality, dedicated to one *GodClass*, while the Design A was a decentralized version of the same system.

First of all, groups were introduced to the system with a short session where the running software was demonstrated. It was assumed that after the session, participants would have an overall understanding of the system. Also, they were given a set of corresponding system design documents (Specifications, Use case diagram, Event flow diagram, Class diagram, Sequence diagram) and a required modification task (e.g., adding new functionality). The whole maintenance work was video-taped to record every activity. Lately, using the recorded video data, the authors came up with diagrams, showing how often participants switch activities and how much time do they spend on each one.

The performances of the group works were measured in terms of time and quality of proposed system designs containing required modifications. The quality was indicated by code metrics such as COF (coupling factor), WMC (weighted method per class), Class coupling, etc.

The group A performed tasks significantly faster. Furthermore, output of their work was of a similar quality compared to the original, which was relatively high. Contrary, the design produced by one (out of 2) of the participants from the group B was of a much lower quality. It was also noted that the group B more frequently looked at the sequence diagrams which may indicate that it was harder for them to understand an interaction between objects. This was also confirmed by the questioner.

The groups were given a questioner about the difficulties they faced during the work. As main difficulties they mentioned tight coupling between classes, cohesion, and syntax and naming conventions.

Deligiannis et al. [26] published another paper based on the previous one. The study also focused on how the *GodClass* smells influence maintainability and understanding. The similar experiment on the same TTS system was conducted with 20 students belonging to two groups (A and B, 10 to each). This time, however, there was no video recording, and analysis was done differently. Maintainability was measured in terms of completeness, correctness, and consistency. All the measures were expressed in simple formulas. To evaluate

the understandability of the program groups were given a questioner. Statistical methods were used to test both maintainability and program comprehension.

The evaluation of the changes performed by the participants revealed that the system A, which did not have a *GodClass*, is easier to maintain according to all three criterias. After giving the questionnaire regarding the design understanding to the participants, it was observed that the system without a *GodClass* was easier to work with. Thus, the findings confirmed the previous study.

God Class Comprehension and Education

A slightly different approach to evaluate the impact of *GodClass* smells to the program comprehension was used in another study conducted by Du Bois et al. [3]. The author brought forward the question whether decomposing *GodClass* smells affects program comprehensibility in terms of accuracy and speed. To answer it, he applied four different refactoring on *GodClass* smells and included one case where the *GodClass* smells remained. After that, he asked the students to perform simple changes on the code.

63 master level students formed 5 groups based on their course and university. Only one of the groups was formed not from the computer science students, but ICT electronics. The experiment was performed on the open source Java main client (Yamm). Students were given the source code and tasks to be performed. A task is finished when a student submits a new code to the Concurrent Versioning System (CVS). It was measured how accurately and fast participants solve tasks, which require localizing the relevant attributes and algorithmic steps. For the analysis step a non-parametric test was applied.

The observed results indicate that certain refactoring had positively influenced the tasks requiring a code understanding. However, results depend on the group. For example, ICT electronic students were more successful in understanding *GodClass* smells than comfputer science students. Authors interpret this finding as confirmation that object-oriented training influences the way of understanding code.

Study of Two antipatterns, Blob and Spaghetti Code

The final work in this subsection is done by Abbes et al. [1]. He investigated the impact of two antipatterns, *Blob* and *SpaghettiCode*, on the program comprehension. The aim was to show that antipatterns negatively influence the understanding of code. The study consisted of three experiments: the tests were performed to find out whether the code containing *Blob*, *SpaghettiCode* and both antipatterns is harder to understand than the code without antipatterns.

Each experiment was conducted by 24 subjects consisting of professional developers and graduate students. They were given different comprehension tasks and the performance was measured with: NASA task load index [24], time, and correctness. Three different open-source systems were chosen for each experiment. DECOR tool aided to identify systems containing specific antipatterns. After that, only the code sample with a one random instance of the antipattern was extracted together with related classes performing a particular task. The subjects were given original code samples and refactored ones (using Fowler's

refactoring techniques). However, none of them received two versions of the same sample. This was done in order to prevent from learning code before an experiment.

The results did not show a statistically significant difference on program comprehension with and without a single antipattern. Yet, experiment 3, which studied the combination of the Blob and the Spaghetti Code, revealed a significant difference between the subjects efforts, correct answers and time. This showed that a single antipattern did not cause comprehension issues. However, this didn't hold for multiple antipatterns. Yet, authors were not sure whether the results were due to the high density of antipatterns or their specifics.

3.4 Overview

This chapter has introduced important concepts for our study. We used Evolizer as a RM tool together with DECOR as an antipattern detector. Evolizer retrieves source code version history from publicly available source repositories and analyzes it. DECOR is a very flexible defect detection tool, working on Java source code, detecting a wide range of antipatterns and is available free of charge for research purpose. More information about the tools and reasons why they have been chosen are provided in the Chapter 4.

Our study is similar to the ones performed by Khomh et al. [30], [17], [31] and Olbrich et al. [48], [49]. The main difference is that we use *SCC* to count source code changes and, we are able to distinguish about 50 different types of them. Therefore, we believe that we count changes more precise. Moreover, we related different type of defects with different types of changes. The focus of our study is to contribute and extend current knowledge on the impact of antipatterns on the change-proneness of software systems.

Chapter 4

Research Framework

This work aims to investigate the relation between the antipatterns and changes-proneness of software classes. This chapter describes the research framework used to answer the research questions, which were introduced in the previous chapter, Section 1.2.

4.1 Overview

In this section, we describe the approach used to gather the data needed to perform our study. The data consist of the fine-grained source code changes (*SCC*), performed in each Java class along the history of the systems under analysis, and the type and number of antipatterns in which a class participates during its evolution. Figure 4.1 shows an overview of our approach consisting of 6 steps.

4.2 Versioning data importer

The first step (step 1 in Figure 4.1) is to retrieve the versioning data from the versioning control systems (CVS, SVN or GIT). To perform this step we use the Evolizer Version Control Connector EVCC[20], belonging to the Evolizer[1] tool set. For each cass, EVCC fetches and parses log entries from the versioning repository. The extracted information together with the source code is stored in the Evolizer repository. From the log entries EVCC extracts:

- Revision numbers
- Revision timestamps
- The name of the developer who checked-in the revision
- The commit messages
- The total number of lines modified

[1] http://www.evolizer.org/

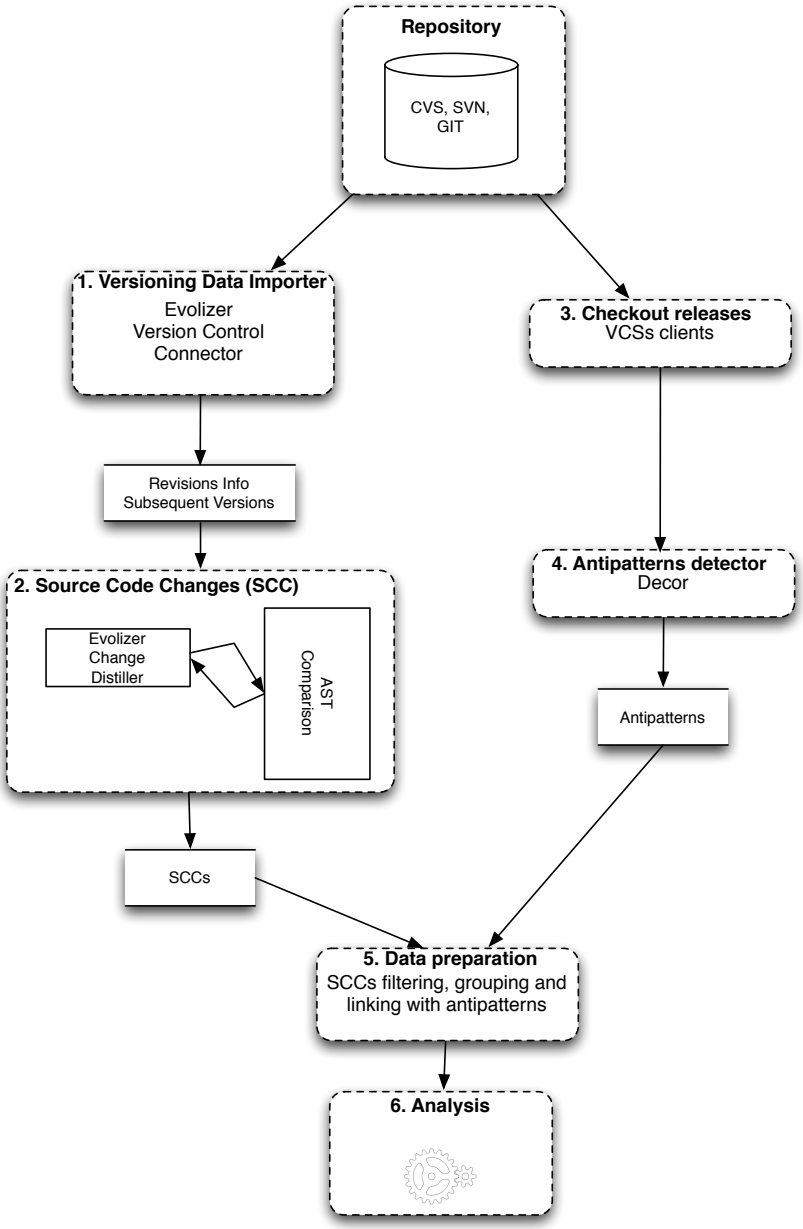

Figure 4.1: Overview of the source code fine-grained changes and antipatterns extraction process [53].

- The source code

We found that retrieving data from SVN takes too long (at least several weeks for a smaller system). Therefore, before the retrieval, the exact copies of publicly available SVN

repositories were created locally. After that, the data was extracted from the local repositories.

4.3 Source Code Changes *(SCC)*

The source code of subsequent versions of each Java file is used by ChangeDistiller [14] to extract the fine-grained source code changes *SCC* (step 2 in the Figure 4.1). ChangeDistiller first parses the source code from the two subsequent versions of Java class and creates the corresponding Abstract Syntax Trees ASTs. Second, the two ASTs are compared using a tree differencing algorithm, which outputs the differences in form of tree edit operations add, delete, update and move. Next, each edit operation for a given node in AST is annotated with a semantic information of the source code entity it represents and is classified as a specific change type based on the taxonomy of code changes [13]. For instance, the insertion of a node representing an *else-part* in the AST is classified as *else-part insert* change type. The result is a list of change types between two subsequent versions of each Java class which is stored into Evolizer repository.

The alternative ways to find source code changes would be to check for the total number of revisions for a class file. However, this would not provide any information about the extent of changes. Another approach would be to count Lines Modified (LM). However, this still would not show what kind of changes has been made. For example, if we change a return type and one parameter type of the same method, it will be treated as one change in terms of file revisions and lines modified. *SCC*, however, would indicate two changes and show their type and location as it is shown in the Figure 4.2.

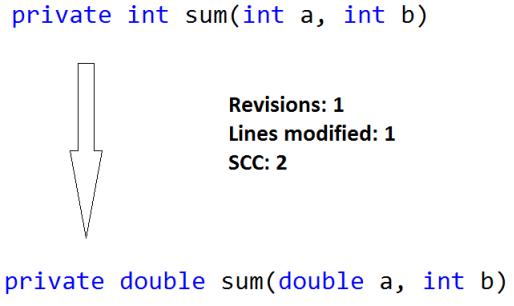

Figure 4.2: *SCC*, file revision and lines modified count of the same change.

4.4 Get release

In the third step of our approach (step 3 in the Figure 4.1) we get releases used later on by the antipattern detector. A range of releases for each software system were downloaded from the public websites as *.jar* files. Another way would be to check out releases from the source code. We chose the first option for a number of reasons:

- Releases are compiled which is mandatory for the next step.
- Releases do not include unit tests and obsolete code, which otherwise would need to be manually removed as it does not represent the functional code.
- Only functioning releases are available from the main branch. So we avoid dealing with experimental releases.

4.5 AntiPettern detector

The fourth step of our approach (step 4 in the Figure 4.1) is aimed to detect all antipatterns existing in the releases. This is achieved by *DECOR (Defect dEtection for CORrection)* [45, 43, 44, 46]. It is described more in the Chapter 3.1. The following reasons influenced our choice of DECOR:

1. The former study [31] used this tool. Therefore, we are sure that we can detect the same antipatterns, while using the same detection heuristics.
2. DECOR is free and open for modifications, in case it is used for the research purposes.
3. DECOR supports Java source code.
4. Detection rules are available and flexible for modifications.
5. During the literature survey we found DECOR to be the most used tool for detecting design defects.

To analyze the participation of a class in antipatterns along its history we run the *DECOR* detection algorithm on each different release for each system under analysis. Among all the antipatterns detectable with DECOR we select the following ones:

- *AntiSingleton* (*AS*): A class that provides mutable class variables, which consequently could be used as global variables.
- *Blob*: A class that is too large and not cohesive enough, that monopolises most of the processing, takes most of the decisions, and is associated with data classes. Also known as *GodClass*.
- *ClassDataShouldBePrivate* (*CDSBP*): A class that exposes its fields, thus violating the principle of encapsulation.
- *ComplexClass* (*ComplexC*): A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
- *LargeClass*: A class that has a very high number of methods and attributes.
- *LazyClass* (*LazyC*): A class that has few fields and methods (with little complexity).
- *LongMethod* (*LongM*): A class that has a method that is overly long, in term of LOCs.

- *LongParameterList* (*LPL*): A class that has (at least) one method with a too long list of parameters with respect to the average number of parameters per methods in the system.

- *MessageChain* (*MsgC*): A class that uses a long chain of method invocations to realise (at least) one of its functionality.

- RefusedParentBequest (*RPB*): A class that redefines inherited method using empty bodies, thus breaking polymorphism.

- *SpaghettiCode* (*Spaghetti*): A class declaring long methods with no parameters and using global variables. These methods interact too much using complex decision algorithms. This class does not exploit and prevents the use of polymorphism and inheritance.

- *SpeculativeGenerality* (*SG*): A class that is defined as abstract but that has very few children, which do not make use of its methods.

- *SwissArmyKnife* (*Swiss*): A class which has multiple responsibilities, defined by multiple interfaces.

Metric and syntatic rules used by *DECOR* to detect antipatterns can be found in Appendix A. We chose this subset of antipatterns because of the following reasons:

1. They appear frequently in the different releases of the systems under analysis.

2. They are representative of design and implementation problems with data, complexity, size, and the features provided by Java classes.

3. They allow us to compare our findings with those of Khomh [31].

4.6 Data preparation

Once we have the *SCCs* performed in each class and the number and type of antipatterns affecting each class in each different release, we prepare the data for the analyses (step 5 in the Figure 4.1). This task is performed in two steps:

1. Merging releases: If between two subsequent releases the number of changes is less than 200, they are merged. The operation is repeated until the condition is satisfied between all the pairs of two subsequent (k and $k+1$) releases. The information about all releases is available in AppendixB.

2. Linking: For each release we link the *SCC* performed in between two subsequent releases (k and $k+1$) with the type and number of antipatterns appearing in the release k. Only the classes which appear both in source control and downloaded release are considered.

3. Clustering *SCCs* types: the *SCCs* are clustered in seven different categories as proposed by Giger [22]. The different categories are shown in the Table 4.1 together with the description of the changes grouped in each category. Clustering the *SCCs* allows us to analyze the contingency between different types of changes and different antipatterns.

As a result, we came up with a list containing for each release *k* a list of Java classes with a number of detected instances of the twelve antipatterns at release *k* plus the number of fine-grade changes per change type category that occurred between two subsequent releases *k* and *k+1*.

Category	Description
API	Changes that involve the declaration of classes (e.g., class renaming and class API changes) and alter the signature of methods (e.g., modifier changes, method renaming, return type changes, changes of the parameter list)
oState	Changes that affect object states of classes (e.g., fields addition and deletion).
func	Changes that affect the functionality of a class (e.g., methods addition and deletion)
stmt	Changes that modify executable statements (e.g., statements insertion and deletion)
cond	Changes that alter condition expressions in control structures and the modification of *else*-parts

Table 4.1: Categories of types of change used to test *H3*

4.7 Analysis

The last step of our approach (step 6 in the Figure 4.1) is analysis. A number of statistical tests were executed on top of the collected data. The analyses performed in this study will be extensively described in the next Chapter (5).

Chapter 5

Analysis

The *goal* of this empirical study is to evaluate the impact of antipatterns on the change proneness of Java classes. The focus is the ability to highlight change-prone Java classes based on whether they participate in antipatterns.

5.1 Overview

This chapter presents the approach we used to test our hypothesis. For All three research questions (RQ1, RQ2, RQ3) we present the hypotheses and methods how those hypothesie are tested. The raw data used to perform our analysis are available at our website.[1]

5.2 Investigation of RQ1

5.2.1 Hypotheses

The *goal* of *RQ1* is to analyze the change-proneness of Java classes participating in antipatterns, compared to the change-proneness of classes not participating in antipatterns. We decided to answer this question to test whether the Khomh et al. findings [31] are confirmed when we analyze the *SCC*s performed between two subsequent releases. Moreover, we provide a further validation testing the difference between the distributions of *SCC*s performed in classes participating and not participating in antipatterns. We address RQ1 by testing the following two null hypotheses:

- $\mathbf{H1}_{\mathrm{a}}$: The proportion of classes changed at least once between two releases is *not* different between classes that are affected by antipatterns and classes not affected by antipatterns.
- $\mathbf{H1}_{\mathrm{b}}$: The distribution of *SCC* performed in classes between two releases is *not* different for classes affected by antipatterns and classes not affected by antipatterns.
- $\mathbf{H1}_{\mathrm{c}}$: There is *no* association between the number of antipatterns affecting a class and number of *SCC* performed in that class.

[1] http://swerl.tudelft.nl/wiki/pub/DanieleRomano/WebHome/WCRE12rawData.zip

5.2.2 Method

To test $H1_a$ we classify the Java classes in *change-prone* if they undergo at least one change in between two subsequent releases (*k* and *k+1*). Otherwise they are considered *not change-prone*. This binary variable (we refer to it as *change-proneness (k, k+1)*) is the *dependent* variable. As *independent* variable we choose a binary variable that indicates if a Java class participates or not in any antipattern in a given release *k* and we refer to it as *antipatterns(k)*. We test H_a examining the significance of the contingency between these two variables. The significance of the association between *antipatterns(k)* and *change-proneness (k, k+1)* for each class and for each given release *k* is tested with the Fisher's exact test [56]. Moreover we use the *odds ratio* (ORs) [56] to measure the probability that a Java class will be changed between two releases (*k* and *k+1*) if it is affected by at least one antipattern in the release *k*. OR is defined as $OR = \frac{p/(1-p)}{q/(1-q)}$ and it measures the ratio of the odds *p* of an *event* occurring in one group (i.e., *experimental group*) to the odds *q* of it occurring in another group (i.e., *control group*). In this case, the *event* is a change in a Java class, the *experimental group* is the set of classes affected by at least one antipattern and the *control group* is the set of classes not affected by any antipattern. ORs equal to 1 indicates that a change can appear with the same probability in both groups. ORs greater than 1 indicates that the change is more likely to appear in a class affected by at least one antipattern. ORs less than 1 indicate that classes not participating in antipatterns are more likely to be changed.

To test H_b we analyze the differences between the distributions of *SCC* performed in Java classes participating in antipatterns and classes not participating. To perform this analysis, as *dependent* variable we choose a variable that counts the number of *SCC* for a given Java class between two subsequent releases and we refer to it as *#SCC(k, k+1)*. The *independent* variable is the same used in the Fisher's exact test. We use the Mann-Whitney test to test whether the two distributions of *SCC* performed in between two subsequent releases are significantly different for classes participating and not participating in antipatterns. We apply the Cliff's Delta *d* effect size [23] to measure the magnitude of the difference. Cliff's Delta estimates the probability that a value selected from one group is greater than a value selected from the other group. Cliff's Delta ranges between +1 if all selected values from one group are higher than the selected values in the other group and -1 if the reverse is true. 0 expresses two overlapping distributions. For independent samples (as in our case) the effect size is defined as the difference between the means (M_1 and M_2), divided by the pooled standard deviation ($\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$) of both groups: $d = (M_1 - M_2)/\sigma$. The effect size is considered negligible for $d < 0.147$, small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.47$ and large for $d \geq 0.47$ [23]. We chose the Mann-Whitney test and Cliff's Delta effect size because the values of the SCC per class are non-normally distributed. Furthermore, our different levels (small, medium, and large) facilitate the interpretation of the results. The Cliff's Delta effect size has been computed with the *orddom* package[2] available for the *R* environment.[3]

[2] http://cran.r-project.org/web/packages/orddom/index.html
[3] http://www.r-project.org/

H_c is aimed at investigating whether there is a correlation between the number of antipatterns a Java class is participating in and the number of *SCC* performed in the class. The *independent* variable is the number of antipatterns in a given release *k* (*#antipatterns(k)*), while the *dependent* variable is the number of *SCC* performed in between two subsquent releases (*#SCC(k, k+1)*). We use the Spearman rank correlation test to analyze the correlation between these variables. Spearman compares the ordered ranks of the variables to measure a monotonic relationship. We chose to use the Spearman rank correlation test because it doesn't make any assumptions about the distribution, variances and the type of the relationship [60]. A high positive or negative correlation is shown by respectively a Spearman *rho* value of +1 and -1, while 0 indicates that the two variable do not correlate at all. Values greater than +0.5 or smaller than -0.5 are considered substantial; values greater than +0.7 and smaller than -0.7 are considered strong correlations.

5.3 Investigation of RQ2

5.3.1 Hypotheses

The *goal* of *RQ2* is to test whether certain antipatterns lead to more changes in Java classes than other antipatterns. The basic idea is to assist software engineers in identifying the most change-prone antipatterns in a system that should be resolved first. We address RQ2 by testing the following null hypothesis:

- $\mathbf{H}2_{\mathrm{a}}$: The proportion of classes changed at least once between two releases is *not* different for classes affected by different antipatterns.
- $\mathbf{H}2_{\mathrm{b}}$: The distribution of *SCC* is *not* different for classes affected by different antipatterns.

5.3.2 Analysis Method

For the $\mathrm{H}2_{\mathrm{a}}$ the dependent variable is *change-proneness (k, k+1)*, already introduced in the investigation of $H1_a$. As independent variables we use a binary variable for each antipattern that indicates whether the class is affected by that antipattern. To test $H2_a$ first we measure the contingency between the *dependent* and the *indipendent* variables with the Fisher's exact test. Then we use the ORs to measure the likelihood that a class affected by a particular antipattern will be changed between two subsequent releases.

For the $\mathrm{H}2_{\mathrm{b}}$ we use the number of *SCC* performed in a class between two releases, already introduced in the investigation of $H1_b$ (*SCC (k, k+1)*) as dependent variable. As independent variable we use a binary variable for each antipattern that indicates whether the class is affected by that antipattern or it doesn't participate in any antipattern. To test $H2_b$ we use the Mann-Whitney test to analyze whether there is a difference in the distributions of *SCC* performed in Java classes not affected by different antipatterns and *SCC* performed in classes not affected by any antipattern. As already introduced testing $H1_b$, we

also used Cliff's Delta *d* effect size over all releases for a system. We selected all releases per system since several releases had too few data points (e.g., there have been only 6 *SCC* between releases 1.6R3 and 1.6R4 of Rhino). The *orddom* package used to compute Cliff's Delta *d* is not optimized for very big data sets. Therefore, in cases of systems with more than 5000 data points (i.e., more than 5000 classes experiencing changes over the revision history), we randomly sampled 5000 data points 30 times and computed the average of the obtained Cliff's Delta values. This sampling allows us to compute Cliff's Delta values for each system with a confidence level of 99% and a confidence interval of 0.004; which is a very precise estimation.

5.4 Investigation of RQ3

5.4.1 Hypotheses

Addressing *RQ3* we analyze the relationship between different antipatterns and different types of changes. The *goal* is to further assist software engineers by verifying whether a particular type of change is more likely to be performed in classes affected by a specific antipattern. This knowledge can help engineers to avoid or fix certain antipatterns that frequently lead to changes that impact large parts of the rest of a system, such as changes in the method declarations of a class that exposes a public API. We address RQ3 by testing the following null hypothesis:

- **H**3: The distribution of different types of *SCC* performed in classes affected by different antipatterns are *not* different.

5.4.2 Analysis Method

To test *H3* we categorize the changes mined with *ChangeDistiller* in five different categories as illustrated in the Section 4.6 (see 4.1). In order to have enough data about each change type category we use the data from all systems as input for this analysis. As *dependent* variables we use the change type categories representing the number of *SCC* that fall in each category. As for *H2*, the *independent* variables are the set of binary variables that denote whether a class is affected by a specific antipattern or not. We test the difference in the distributions of SCC per category using the Mann-Whitney test and compute the magnitude of the difference with the Cliff's Delta *d* effect size. In order to have enough data about each change type category we use the data from all systems as input for this analysis. Similar to $H2_b$, we use the random sampling approach for computing Cliff's Delta and we report the mean effect size of the 30 random samples.

Chapter 6

Project Selection

This chapter details the process of project selection. The selection criteria and our dataset are introduced here.

6.1 Overview

The next section will provide the criteria for the project selection. After that, we will introduce the dataset used in this study.

6.2 Selection criteria

The project selection criteria depend on our chosen approach (see the Chapter 4) and the former study conducted by Khomh et.al [31]. We chose to use Java open source systems. First, they support OO (Object Oriented) design which is misused by a number of antipatterns. Second, the former study investigated four open source Java systems: Rhino, Mylyn, ArgoUML, Eclipse. Our chosen projects had also to have publicly available code source repositories and system releases available for public download. Many open source projects satisfied those requirements.

Additionally, we chose three requirements to provide enough information for our study:

1. More than 2000 revisions
2. More than 200 classes in any release
3. More than 100 antipatterns in any release

6.3 Selected projects

The *context* of this study consists of the 16 open-source systems from different domains, implemented in Java and widely used in academic and industrial communities. The Table 6.1 shows an overview of the dataset in our empirical study. *#Files* is the number of Java files in the last release, *#Classes* is the number of classes in the last release, *#Rel* is the

number of releases analyzed, *#Rev* is the number of revisions, *#SCC* the number of fine-grained source code changes performed in the given time period (*Time*). The Tables 6.2, 6.3 show the number of antipatterns detected by DECOR in first and last release of the analyzed systems.

Project	#Files	#Classes	#Rel	#Rev	#SCC	Time[M,Y]
argo	1716	1859	9	875137	128949	Oc02-Mar09
hibernate2	537	494	10	13584	22960	Jan03-Mar11
hibernate3	1036	970	20	30774	34960	Jun04-Mar11
eclipse.debug.core	188	201	12	8295	11670	May01-Mar11
eclipse.debug.ui	793	907	22	41860	55259	May01-Mar11
eclipse.jface	381	442	17	22136	27041	Sep02-Mar11
eclipse.jdt.debug	469	509	16	11711	33895	Jun01-Mar11
eclipse.team.core	172	199	6	3726	4551	Nov01-Mar11
eclipse.team.cvs.core	189	210	11	12343	23311	Nov01-Mar11
eclipse.team.ui	293	405	13	20183	32267	Nov01- Mar11
jabref	1996	2597	30	6614	56936	Dec03-Oct11
mylyn	1288	1596	17	34816	100405	Dec06-Jun09
rhino	184	219	8	8669	20873	May99-Aug07
rapidminer	2061	2907	4	6207	7786	Oct09-Aug10
vuze	3265	3733	29	22651	176891	Dec06-Apr10
xerces	710	869	20	12577	119238	Dec00-Dec12

Table 6.1: Dataset used in the empirical study

Basically, all of the systems contain instances of most of the 12 antipatterns. In particular, Rapid miner and Vuze contain the largest number of antipatterns which is not surprising, since they also are the largest systems in our sample set. According to our numbers, the antipatterns *LongMethod (LongM)* and *MessageChain (MsgC)* and *RefusedParentBequest (RPB)* occur most frequently, while *SpathettiCode (Spaghetti)*, *SpeculatigeGenerality (SG)* and *SwissAmryKnife (Swiss)* occur less frequently. Overall, the frequency of antipatterns and changes allow us to investigate the relationship with fine-grained source changes.

Unfortunately, due to the lack of time and computational resources we could not include the full Eclipse project. Instead the seven subprojects (eclipse.debug.core, eclipse.debug.ui, eclipse.jface, eclipse.jdt.debug, eclipse.team.core, eclipse.team.cvs.core eclipse.team.ui) of Eclipse were analysed in our study. The other projects from the the former study (Rhino, ArgoUML, Mylyn) are a part of our dataset.

The project eclipse.team.core have not met the requirements of having at least 200 classes but was close (199). We still included it in our dataset. In the next chapter we will present the results on the chosen dataset.

Project	#AS	#Blob	#CDBSP	#ComplexC	#LazyC	#LongM
argo	352-3	26-169	136-51	56-195	16-53	172-354
hibernate2	113-104	34-37	33-17	30-37	5-3	56-72
hibernate3	176-232	52-75	31-50	58-8	9-12	121-194
eclipse.debug.core	1-22	7-14	0-12	1-8	0-9	5-22
eclipse.debug.ui	18-146	13-70	0-70	11-50	0-22	30-176
eclipse.jface	8-25	7-22	6-32	5-13	6-22	22-60
eclipse.jdt.debug	17-44	26-27	1-74	30-33	8-42	68-78
eclipse.team.core	1-12	2-7	1-10	1-5	0-4	8-33
eclipse.team.cvs.core	9- 64	1-21	2-6	1-21	0-0	17-79
eclipse.team.ui	9-64	1-21	2-6	1-21	0-0	17-79
jab	12-139	10-136	8-400	9-144	1-126	21-365
mylyn	4-70	43-101	61-174	43-83	2-16	132-300
rhino	16-18	5-11	4-18	9-19	4-9	11-33
rapidminer	11-19	130-161	145-203	152-156	10-15	450-568
vuze	179-145	199-282	189-270	138-193	29-215	381-473
xerces	10-22	8-59	14-134	13-44	6-21	29-96

Table 6.2: Number of antipatterns in first and last releases of the analyzed systems. Table 1

Project	#LPL	#MsgC	#RPB	#Spaghetti	#SG	#Swiss	#Total
argo	195-334	130-197	65-513	22-1	9-34	3-4	1182-1908
hibernate2	34-19	51-101	93-97	15-4	2-1	0-0	466-492
hibernate3	48-74	157-236	123-202	9-12	3-8	3-9	790-1112
eclipse.debug.core	0-18	3-6	0-11	0-1	1-1	0-2	18-126
eclipse.debug.ui	25-41	6-53	6-73	3-8	2-24	0-7	114-740
eclipse.jface	19-45	22-34	5-14	0-2	7-21	0-2	107-292
eclipse.jdt.debug	37-40	78-80	80-82	3-3	1-2	1-1	350-506
eclipse.team.core	0-26	1-15	0-7	0-1	3-10	0-0	17-130
eclipse.team.cvs.core	1-51	4-45	0-13	0-1	2-10	0-0	37-311
eclipse.team.ui	1-51	4-45	0-13	0-1	2-10	0-0	37-311
jab	2-169	2-332	2-295	1-16	0-17	0-1	68-2140
mylyn	43-66	98-135	34-165	2-0	12-35	1-1	475-1146
rhino	9-8	15-51	3-7	0-0	0-2	0-1	76-177
rapidminer	214-270	583-674	781-1068	1-1	12-28	3-1	2492-3164
vuze	217-295	514-773	476-637	22-16	21-27	35-70	2400-3396
xerces	16-130	19-99	3-37	2-1	5-4	10-11	135-658

Table 6.3: Number of antipatterns in first and last releases of the analyzed systems. Table 2

Chapter 7

Results of analysis

The results of our study are of interest for software engineers. They can use the number of antipatterns as a quality indicator to highlights change prone classes. As it was discussed in the Chapter 2, antipatterns are good targets for the code review sessions. This chapter reports the results of our empirical study presented in the Chapter 6.

7.1 Overview

This chapter is structured as follows: each section presents results to the research questions (more in the Chapter 1, starting from *RQ1* and finishing with *RQ3*.

7.2 Results: *RQ1*

The *goal* of $H1_a$ is to test whether *Java classes affected by antipatterns are more change-prone than classes not affected by antipattern*. The odds ratios are summarized in the Table 7.1. The Table 7.1 shows for each system the total number of releases (#Releases) and the number of releases that showed a p-value for the Fisher's exact test smaller than 0.01 and odds ratios greater than 1 (*ORs>1*). Fisher's exact test p-values smaller than 0.01 indicate that there is significant difference between change-proneness of classes affected and not affected by antipatterns. The ORs greater than 1 indicate that Java classes participating in at least one antipattern are more change-prone than the other classes. The results show that, except for three systems (*eclipse.team.cvs.core*, *Jabref* and *Rhino*), in most of the analyzed releases Java classes participating in at least one antipattern are more change-prone than other classes. In total in 190 out of 244 releases ($\approx$82%) classes affected by at least one antipattern are more change-prone.

To test $H1_b$ we analyzed the difference between the distributions of *SCC* performed in Java classes participating in antipatterns and in other classes. We applied the Mann-Whitney test to test whether the distributions are different (Mann-Whitney p-value $<$0.01) and we measured the Cliff's Delta d effect size to quantify the magnitude of the difference. The Table 7.2 shows the p-values of the Mann-Whitney tests and values of the Cliff's Delta d effect size for testing $H1_b$. Only in 18 releases ($\approx$7%) there is no significant difference

System	#Releases	ORs ≥1
argo	9	9
hibernate2	10	10
hibernate3	20	19
eclipse.debug.core	12	8
eclipse.debug.ui	22	20
eclipse.jface	17	16
eclipse.jdt.debug	16	16
eclipse.team.core	6	4
eclipse.team.cvs.core	11	5
eclipse.team.ui	13	9
jabref	30	3
mylyn	17	17
rhino	8	2
rapidminer	4	4
vuze	29	29
xerces	20	19
Total	244	190

Table 7.1: Total number of releases (*#Releases*) and number of releases where Fisher's exact test shows significant differences (p-values<0.01) and ORs>1 (*ORs ≥1*) for each system under analysis.

(Mann-Whitney p-value≥0.01) between the distributions of *SCC* performed in classes affected by antipatterns and in other classes. In the other 226 releases (≈93%) the difference is significant (Mann-Whitney p-value<0.01). Concerning the effect size we found that this difference is small (0.147≤d<0.33) in 102 releases (≈42%), medium (0.33≤d<0.47) in 26 releases (≈11%), large (0.47≤d) in 9 releases (≈4%) and negligible ($d < 0.147$) in 89 releases (≈36%). Based on these results we reject $H1_b$ and accept the alternative hypothesis that in most cases Java classes with antipatterns undergo more changes during the next release than classes that are free of antipatterns.

The last hypothesis on *RQ1* is $H1_c$. We tested it by computing the Spearman rank correlation test between the number of *SCC* and the number of antipatterns affecting a Java class. The results are reported in the Table 7.3. The results show that there is no strong correlation because only in 15 releases out of 144 (≈6%) the *rho* value is greater than 0.5. However, 177 releases (≈72%) show a *rho* value in between 0.3 and 0.8. This result give us enough evidence of the association between number of antipatterns and number of *SCC*.

Based on these results we accept *H1* and we conclude that Java classes participating in antipatterns are more change-prone than other classes. Also, classes affected by antipatterns undergo a number of changes statistically greater than the number of changes performed in other classes. Moreover we conclude that there is a relationship between the number of antipatterns and the number of *SCC* even though we do not have the evidence of linear relationship (i.e., correlation) between the two variables.

		Mann-Whitney p-value<0.01				Mann-Whitney p-value≥0.01
System	#Releases	0.47≤d	0.33≤d<0.47	0.147≤d<0.33	d≤0.147	
argo	9	0	1	6	2	0
hibernate2	10	0	1	6	3	0
hibernate3	20	0	3	7	10	0
eclipse.debug.core	12	4	2	4	1	1
eclipse.debug.ui	22	0	0	14	8	0
eclipse.jface	17	0	0	12	4	1
eclipse.jdt.debug	16	0	1	8	5	2
eclipse.team.core	6	0	1	3	0	2
eclipse.team.cvs.core	11	1	3	4	3	0
eclipse.team.ui	13	1	4	3	1	4
jabref	30	0	3	11	16	0
mylyn	17	0	2	9	6	0
rhino	8	2	0	0	0	6
rapidminer	4	0	0	0	4	0
vuze	29	0	2	7	20	0
xerces	20	1	3	8	6	2
total	244	9	26	102	89	18

Table 7.2: p-values of the Mann-Whitney tests and Cliff's Delta *d* showing the magnitude of the difference between the distribution of *SCC* in classes affected and not affected by antipatterns.

		p-value<0.01						p-value≥0.01
System	#Releases	1<rho≤0.8	0.8<rho≤0.5	0.5<rho≤0.4	0.4<rho≤0.3	0.3<rho≤0.2	0.2<rho≤0	
argo	9	0	0	1	5	1	2	0
hibernate2	10	0	0	1	2	5	2	0
hibernate3	20	0	0	1	6	8	5	0
eclipse.debug.core	12	0	9	2	1	0	0	0
eclipse.debug.ui	22	0	0	0	12	10	0	0
eclipse.jface	17	0	0	1	10	6	0	0
eclipse.jdt.debug	16	0	0	2	8	6	0	0
eclipse.team.core	6	0	0	2	2	0	0	2
eclipse.team.cvs.core	11	0	2	3	5	1	0	0
eclipse.team.ui	13	0	0	3	6	1	0	3
jab	30	0	2	2	0	5	21	0
mylyn	17	0	0	0	3	10	4	0
rhino	8	0	2	0	0	0	0	6
rapidminer	4	0	0	0	0	0	4	0
vuze	29	0	0	0	2	12	15	0
xerces	20	0	0	2	6	9	2	1
totale	244	0	15	20	68	74	55	12

Table 7.3: Spearman rank correlation between number of antipatterns and number of *SCC*.

7.3 Results: *RQ2*

RQ2 is aimed at testing whether *Java classes affected by certain antipatterns are more change-prone than classes affected by other antipatterns*. We used both Fishers test for ($H2_a$) and Mann-Whitney test for ($H2_b$).

The Tables 7.4, 7.5, 7.6 and 7.7 shows the Odds ratios and Cliff's Delta *d* effect size for which the p-value of the Mann-Whitney is significant (p-value<0.01). NA denotes p-values for Mann-Whitney greater than 0.01 and consequently Cliff's Delta is not computed.

The results of the Mann-Whitney tests show that, except for the *LazyClass* and *Specu-*

Project	#AS	#Blob	#CDBSP	#ComplexC	#LazyC	#LongM
argo	3.60	1.44	3.70	2.35	NA	2.23
hibernate2	2.01	1.66	2.35	8.13	0.30	1.99
hibernate3	2.60	1.81	1.94	5.41	0.18	2.63
eclipse.debug.core	12.83	4.90	5.89	**109.10**	NA	12.77
eclipse.debug.ui	2.90	3.93	2.41	**8.85**	0.11	3.58
eclipse.jface	4.70	3.05	2.68	**13.75**	0.10	4.62
eclipse.jdt.debug	6.02	3.31	NA	9.77	0.05	5.86
eclipse.team.core	6.27	6.32	NA	**10.36**	NA	4.48
eclipse.team.cvs.core	NA	3.88	NA	**20.76**	0.35	3.60
eclipse.team.ui	3.92	3.41	2.64	5.06	NA	3.58
jab	3.88	NA	1.62	4.05	0.26	4.12
mylyn	NA	2.17	2.19	**4.86**	0.07	3.02
rhino	3.73	NA	4.59	NA	NA	NA
rapidminer	NA	1.95	NA	4.70	NA	2.29
vuze	3.80	2.44	2.39	**8.10**	0.39	3.21
xerces	3.54	2.12	1.31	10.3	0.57	3.84
Median	3.8	2.745	2.4	8.13	0.22	3.58

Table 7.4: ORs of changes in classes participating in different antipatterns. NA indicates a p-value for Fisher's exact test greater than 0.01 or lack of sufficient data to perform the tests.

Project	#LPL	#MsgC	#RPB	#Spaghetti	#SG	#Swiss
argo	1.85	2.79	1.24	3.84	NA	**5.05**
hibernate2	3.87	2.98	NA	3.07	NA	**22.40**
hibernate3	3.09	3.61	NA	3.04	NA	**5.59**
eclipse.debug.core	24.58	3.91	7.23	NA	NA	20.14
eclipse.debug.ui	2.80	3.97	0.64	5.88	NA	4.36
eclipse.jface	3.31	7.97	0.38	6.99	NA	4.45
eclipse.jdt.debug	5.63	3.15	0.66	5.60	NA	**12.18**
eclipse.team.core	NA	4.08	NA	NA	5.45	NA
eclipse.team.cvs.core	2.12	3.76	NA	NA	NA	NA
eclipse.team.ui	2.25	2.48	NA	**Inf**	2.46	NA
jab	3.12	2.34	NA	**15.27**	0.08	NA
mylyn	2.39	3.34	NA	NA	NA	NA
rhino	NA	1.68	NA	**Inf**	NA	NA
rapidminer	1.50	2.19	NA	NA	NA	**15.05**
vuze	3.48	4.50	1.24	7.56	NA	5.96
xerces	4.58	1.98	NA	4.05	3.41	**11.08**
Median	3.105	3.245	0.95	5.88	2.935	8.52

Table 7.5: ORs of changes in classes participating in different antipatterns. NA indicates a p-value for Fisher's exact test greater than 0.01 or lack of sufficient data to perform the tests.

lativeGenerality (*SG*), the distributions of *SCC* performed in classes affected by a specific antipattern are different from the distribution of *SCC* performed in classes not affected by that antipattern. According to the median values for Cliff's Delta shown in the last row, this

System	#AS	#Blob	#CDBSP	#ComplexC	#LazyC	#LongM
argo	0.311	0.098	0.331	0.226	-0.012	0.192
hibernate2	0.143	0.112	0.193	0.500	NA	0.149
hibernate3	0.171	0.086	0.064	0.386	-0.110	-0.172
eclipse.debug.core	0.553	0.352	0.419	**0.889**	NA	0.544
eclipse.debug.ui	0.169	0.299	0.150	0.454	0.147	0.231
eclipse.jface	**0.461**	NA	NA	0.411	NA	0.266
eclipse.jdt.debug	0.277	0.182	0.078	0.485	0.103	0.250
eclipse.team.core	0.422	0.433	NA	**0.581**	NA	0.33
eclipse.team.cvs.core	0.026	0.374	0.085	**0.723**	NA	0.331
eclipse.team.ui	0.290	0.293	0.212	0.395	NA	0.265
jabref	0.089	0.001	0.019	0.094	NA	0.072
mylyn	-0.020	0.150	0.177	**0.388**	NA	0.192
rhino	0.276	NA	0.393	0.119	NA	0.067
rapidminer	0.051	0.060	-0.001	0.141	NA	0.051
vuze	0.151	0.076	0.079	0.211	NA	0.121
xerces	0.302	0.104	0.044	0.541	NA	0.269
Median	0.223	0.131	0.117	**0.403**	0.045	0.211

Table 7.6: Cliff's Delta *d* effect sizes of cases for which Mann-Whitney shows a significant difference (p-value<0.01) or NA otherwise. Values in bold denote the largest difference per system. For the underlined systems we applied random sampling.

System	#LPL	#MsgC	#RPB	#Spaghetti	#SG	#Swiss
argo	0.148	0.248	0.035	0.354	0.030	**0.528**
hibernate2	0.347	0.250	-0.032	0.262	NA	**0.654**
hibernate3	0.169	0.170	0.016	0.191	NA	**0.662**
eclipse.debug.core	0.691	0.289	0.435	NA	0.298	0.650
eclipse.debug.ui	0.169	0.227	NA	0.377	0.009	**0.514**
eclipse.jface	NA	0.385	NA	NA	NA	NA
eclipse.jdt.debug	0.295	0.137	0.051	0.361	NA	**0.919**
eclipse.team.core	0.107	0.315	NA	NA	0.373	NA
eclipse.team.cvs.core	0.172	0.329	NA	NA	NA	NA
eclipse.team.ui	0.163	0.187	NA	**0.642**	0.183	NA
jabref	0.044	0.042	-0.006	0.356	NA	**0.966**
mylyn	0.232	0.228	0.063	NA	NA	NA
rhino	0.025	0.100	NA	**0.928**	NA	NA
rapidminer	0.080	0.051	-0.002	NA	NA	**0.600**
vuze	0.106	0.140	-0.021	**0.308**	0.028	0.213
xerces	0.327	0.122	0.036	0.153	0.307	**0.565**
Median	0.169	0.207	0.025	**0.355**	0.183	**0.625**

Table 7.7: Cliff's Delta *d* effect sizes of cases for which Mann-Whitney shows a significant difference (p-value<0.01) or NA otherwise. Values in bold denote the largest difference per system. For the underlined systems we applied random sampling.

difference is large for SwissArmyKnife (*Swiss*), medium for 2 antipatterns ($0.33 \leq d < 0.47$), small for 5 antipatterns ($0.147 \leq d < 0.33$) and negligible for 4 antipatterns. Note, that for classes affected by *LazyClass* and *SG* the Mann-Whitney test was significant only in 4 and

respectively 7 systems. ORs table mean values show that only classes participating in *LazyClass* and *RPB* antipatterns have less chance to be changed (or < 1).

Moreover, analyzing the results for each system we can notice that that classes participating in the *ComplexClass* (*ComplexC*), *SpaghettiCode* (*Spaghetti*) and *SwissArmyKnife* (*Swiss*) antipatterns are more change-prone than classes participating in any other antipattern. In 7 systems out of 16 the odds ratios of changes in classes affected by the *ComplexClass* antipattern are the highest. In 6 systems the odds ratios of changes in classes affected by the *SwissArmyKnife* antipattern are the highest even though in 6 systems we do not have enough occurrences of this antipattern. In the other 3 systems the highest odds ratios are of changes in classes participating in the *SpaghettiCode* antipattern. Similarly, in 8 systems out of 16 the Cliff's Delta effect size is highest for classes affected by *SwissArmyKnife*. In 4 systems the Cliff's Delta effect size is higher for classes affected by *ComplexClass*. In the other 3 systems the highest effect size is for classes affected by *SpaghettiCode*. Only in one system, namely *eclipse.jface*, the *Antisingleton* antipattern shows the highest value for Cliff's Delta.

Based on this result we reject *H2* and we conclude that among all classes the classes participating in the *ComplexClass* (*ComplexC*), *SpaghettiCode* (*Spaghetti*) and *SwissArmyKnife* (*Swiss*) antipatterns are more change-prone. These findings detail the results obtained by Khomh et al. [31] by highlighting three antipatterns that are more change-prone than the other antipatterns. Based on our findings we can advice software engineers to detect instances of these three change-prone antipatterns and fix them first.

7.4 Results: *RQ3*

The *goal* of *RQ3* is to test whether *Java classes affected by certain antipatterns are more likely to undergo particular types of changes*. We tested *H3* by analyzing the Cliff's Delta *d* effect sizes of different types of change (see the Table 4.1) performed in classes participating in different antipatterns. The Table 7.8 lists the results of this analysis.

Group	#AS	#Blob	#CDBSP	#ComplexC	#LazyC	#LongM	#LPL	#MsgC	#RPB	#Spaghetti	#SG	#Swiss
API	0.131	0.077	0.038	**0.213**	-0.043	0.073	0.095	0.075	0.001	**0.207**	0.029	**0.150**
oState	0.080	0.048	0.031	0.144	NA	0.042	0.060	0.045	-0.001	0.126	-0.001	0.109
func	0.084	0.057	0.019	**0.153**	-0.040	0.053	0.076	0.054	-0.002	**0.149**	NA	0.142
stmt	**0.157**	0.077	0.051	**0.252**	NA	0.140	0.146	0.120	0.100	**0.308**	0.007	**0.245**
cond	0.080	0.035	0.028	0.138	-0.020	0.059	0.081	0.058	0.001	**0.178**	0.100	0.136

Table 7.8: Cliff's Delta *d* effect sizes of cases for which Mann-Whitney shows a significant difference (p-value$<$0.01) or NA otherwise. Values in bold denote an effect size that is at least small ($d > 0.147$).

They show that changes in the class and methods declaration (*API*) are more likely to appear in classes participating in the *SwissArmyKnife (Swiss)*, *ComplexClass* and *SpaghettiCode (Spaghetti)* antipatterns. In fact, only for these antipatterns the distribution of *API* changes is larger ($0.8 \leq$d) than the distribution of the same changes in classes not affected by antipatterns. Changes in the functionalities (*func*) are likely in classes affected by the *Com-*

plexClass and *SpaghettiCode* antipatterns. Changes in the executable statements (*stmt*) are frequent in classes affected by *Antisingleton*, *SwissArmyKnife (Swiss)*, *ComplexClass* and *SpaghettiCode (Spaghetti)*.Finally, changes in the condition expressions and in the *else-parts (cond)* are more frequent in classes affected by the *SpaghettiCode* antipattern.

Based on these results we reject H3 and conclude that classes affected by different antipatterns undergo different types of changes.

7.5 Manual Inspection

To further highlight the relationship between antipatterns and change-proneness we manually inspected several classes affected by antipatterns that have been resolved. For these classes we analyzed the number of changes before and after the removal of the antipatterns. The analysis clearly shows that when classes are affected by an antipattern they undergo a considerably higher number of changes.

The example is the class **.xerces.StandardParserConfiguration* from the Xerces system. The Figure 7.1 illustrates how many total and *API* changes the class had during the evolution. This class was affected by the *ComplexClass* antipattern until the release 2.0.2. Before release 2.0.2, the class underwent on average 64.5 changes per release. The average number of changes decreased to 5.2 after the antipattern was removed. Furthermore, the average number of *API* changes decreased from 2 to 0.07, while the LOC dropped from around 370 to 150.

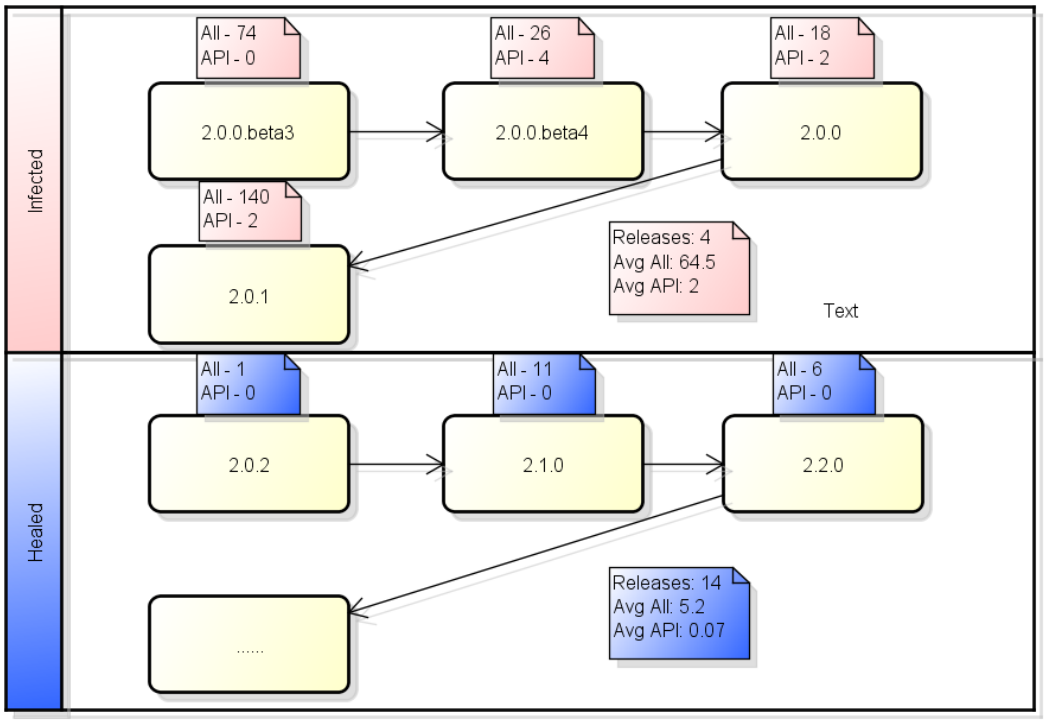

Figure 7.1: Complex antipattern evolution. the class **.xerces.StandardParserConfiguration* from the Xerces system.

As another example, consider the **.debug.ui.views.memory.AddMemoryBlockAction* class from the *eclipse.debug.ui* system. The Figure 7.2 illustrates how many total and conditional changes the class had during the evolution. This class was affected by the *SpaghettiCode* antipattern until the release 3.2. The average number of changes decreased from 79.83 to 1.5 after the release 3.2. Moreover, the average number of *cond* changes decreased from 2.67 to 0.1.

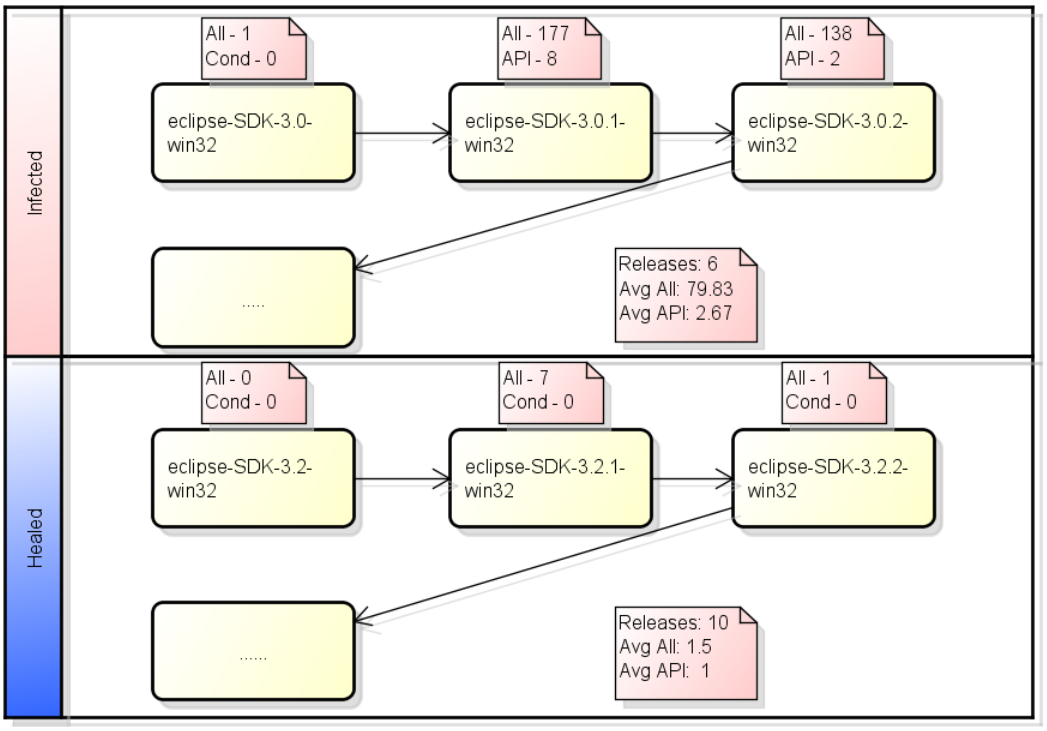

Figure 7.2: SpaghettiCode antipattern evolution. The class *.uml.cognitive.critics.Init from the ArgoUML releases.

7.6 Summary

The results of our study help researchers and software engineers to gain a better insight into the effects of antipatterns on change-proneness of Java classes.

We confirmed the Kohmh et.al findings [31] taking into account the *SCC* performed in between two subsequent releases ($H1_a$). Our approach overcomes the threats to validity that affect analyses based on source code changes mined from two subsequent releases without considering the actual changes performed in between them.

The distribution of *SCC* performed in classes affected by antipatterns is statistically greater than the distribution of *SCC* performed in other classes ($H1_b$). This result can be used by software engineers to optimize the allocation of the resources needed to evolve a system and to perform testing before it is released.

There is an association between the number of antipatterns affecting a class and number of *SCC* performed in that class ($H1_c$). Even though there is not a linear association, the results of our study show that the number of antipatterns a class participates in is an indicator of the number of *SCC* performed in that class.

Classes participating in *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* patterns are

more change-prone than classes participating in other antipatterns *(H2)*. This result is useful to narrow down the set of risky classes. Software engineers can highlight classes affected by these antipatterns in order to allocate more resources for their maintainance. Moreover, researchers can focus on investigating techniques and tools aimed at avoiding and refactoring classes affected by these antipatterns.

Changes in *API*s are more likely to appear in classes affected by the *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife*; methods are more likely to be added/deleted in classes affected by *ComplexClass* and *SpaghettiCode*; changes in executable statements are likely in *AntiSingleton*, *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife*; changes in conditional statements and *else*-parts are more likely in classes affected by *SpaghettiCode* (*H3*).

Chapter 8

Discussion

8.1 Summary of the Results

In this thesis we performed a study to help researchers and software engineers to gain a better insight into the effects of antipatterns on change-proneness of open-source systems. We used fine grade source changes to test whether 1) classes infected by antipatterns change more than other classes; 2) there are specific antipatterns which differently influence change-proneness; 3) there is a relation between the types of changes and the presence of specific antipatterns.

First, we confirmed the Khomh et al.'s findings [31] taking into account the *SCC*s performed in between two subsequent releases (RQ1). Our approach overcomes the threats to validity that affect analysis based on changes mined from two releases without considering the actual changes performed in between them. Moreover using *SCC*s we were able to count changes more precisely filtering out changes in the comments and copyrights that accounted for more the 10%. Our results also partly confirm Olbrich et al. results [48], [49]. Differently from us, they analyzed a couple of code smells and showed that infected classes can even change less when counting changes per line of code. Khomh et al. [31] addressed this question by showing that classes affected by antipatterns are no larger than non-infected classes and, therefore, the change-proneness of the infected classes are not due to the size, when a larger set of antipatterns is taken into account. The confirmation of this research question was out of scope of our study. Additionally, we showed that there is an association between the number of antipatterns and the number of *SCC*s.

Second, we showed that classes affected by *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns are more change-prone than classes affected by other antipatterns (*RQ2*). We think that such results were due to the high complexity and low cohesion of the antipatterns. In the former study, the authors concluded that the most change-prone antipatterns vary per system and weren't able to come up with a list of the most change-prone antipatterns. Considering a bigger dataset we found different results showing the greater impact of *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns on change proneness.

Third, we showed that certain antipatterns are linked with certain changes (*RQ3*):

- *API* changes are more frequently performed in classes affected by the *SwissArmyKnife*, *SpaghettiCode* and *ComplexClass* patterns.
- Conditional (*cond*) changes are more frequent in classes affected by the *SpaghettiCode* antipattern.
- Changes in the functionalities (func) are likely in classes affected by the *ComplexClass* and *SpaghettiCode* antipatterns.
- Changes in executable statements are likely in *AntiSingleton*, *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife*;

This is a novel and very dependent on the use of *SCC* research question. By looking at the definitions of the antipatterns and the examples from the source code we present the possible explanations of the results:

- *SwissArmyKnife* implements many interfaces and has a lot of responsibilities. Therefore, it is likely that new responsibilities are added and removed by modifying the API. The *API* changes in *SpaghettiCode* can be explained by the usage of long and complex methods which time after time get refactored by extracting new functionalities, exposed through the API. Similarly, *ComplexClass* is a target for many changes, including *API* related due to the high complexity.
- Frequent conditional changes in the *SpaghettiCode* antipattern may be caused by the high complexity and length of the methods.
- The high rate of functional changes in *ComplexClass* and *SpaghettiCode*, execution statements in *Antisingleton*, *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* can also be influenced by the higher complexity, low cohesion and the general high rate of changes in those classes.

8.2 Implications of the Results

The results of our study have several implications on software engineers and researchers. From the perspective of software engineers, first, *RQ1* suggests that antipatterns should be targets of refactoring due to the high change-proneness and, consequently, the higher maintenance cost. Second, the results for *RQ2* show that detecting and resolving the three antipatterns *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* is even more important. Classes affected by these antipatterns turned out to be the most change-prone. Third, among all the analyzed antipatterns, engineers should particularly focus on resolving instances of these antipatterns in order to prevent changes in *APIs* (*RQ3*). Those changes can have a significant impact on the implementation of the other parts of a software system and, therefore, should be prevented.

For instance, consider the scenario in which *API*s are made available through web services. The responsible software engineers want to assure the robustness of these classes to minimize the possibility of breaking the clients of the web services. Based on the results of

our study they can use DECOR during the code review sessions to detect instances of the *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns in the set of *API* classes. These are the antipatterns they should resolve first in order to reduce the probability that *API*s are changed and, hence, that clients are broken. This can be especially useful when implementing SOA architectures containing many services and clients. Similarly, software engineers may want to refactor *ComplexClass* and *SpaghettiCode* antipatterns as they are linked with method add/remove changes. In case a public method is added or removed the service *API* is changed and the clients may need to be updated.

In another situation, software engineers may prefer to have code with less conditional statements to ease the unit testing. One change in a conditional statement may impact a number of code paths and cause a huge effort in updating the test cases. In case there are tests for many of the impacted paths, they might need to be modified to reflect the change. Therefore, developers end up not only maintaining a highly change-prone class but also a bunch of frequently modified unit tests. Additionally, they may even stop supporting them which would impact the system quality. In order to prevent from such situations we suggest to refactor *SpaghettiCode* antipatterns.

Similarly, *ComplexClass* and *SpaghettiCode* antipatterns could be refactored as they are linked with the method add/remove changes. Consequently, after adding or removing methods in a class, corresponding unit tests are also added/removed/modifed. Although more unit tests are written for the public methods and we didn't distinguish private and public method changes in the study, we believe that there is a link between those antipatterns and modifications in unit tests. Its important to note that refactoring infected classes in naive way by moving complexity to other classes may not ease the unit testing a lot. The complexity will still remain in the system as the same execution paths will persist. Therefore, it is important to perform refactoring wisely. Many good refactoring cases are described in the book written by Beck and Fowler [40].

From the perspective of a researcher, our results are a step forward on analyzing the impact of antipatterns on the change-proneness of classes. As there is no clear agreement on the impact of design defects, there was a need for the further investigation. In particular, we showed that certain antipatterns are more change-prone than others and that specific antipatterns are more likely to lead to specific types of changes. These results can form basis for developing recommendation techniques that point out the most change-prone classes and how to refactor them. Furthermore, we showed that fine-grade source changes allow us to obtain a more detailed understanding of the evolution of classes affected by antipatterns. Based on our results, researchers can perform studies on investigating specific antipatterns. For instance, as *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns were shown to be the most change-prone ones, software engineers might find hard to understand the classes containing those defects. Therefore, a study on comprehension can be performed. Similarly, the bug-pronennes and *SpaghettiCode* antipattern link can be investigated as conditional changes may lead to bugs.

8.3 Threats to validity

This section discusses the threats to validity that can affect our empirical study and the results described in the previous sections.

Threats to *construct validity* concern the relationship between theory and observation. In our study, this threat can be due to the fact that we considered *SCC* performed in between two subsequent releases. However, the effects of antipatterns can manifest themselves after the next immediate release whenever the class affected by antipatterns needs to be changed. We mitigated this threat by testing all the hypotheses taking into account all the *SCC* performed after a release (available from the reported raw data). We obtained similar results.

Threats to *internal validity* concern factors that may affect an independent variable. In our study, both the independent and dependent variables are computed using deterministic algorithms (implemented in ChangeDistiller and Decor) delivering always the same results.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. To mitigate these threats our conclusions have been supported by proper statistical tests, in particular by non-parametric tests that do not require any assumption on the underlying data distribution.

Threats to *external validity* concern the generalization of our findings. Every result obtained through empirical studies is threatened by the bias of their datasets [42]. To mitigate these threats we tested our hypotheses over 16 open-source systems of different size and from different domains.

Threats to *reliability validity* concern the possibility of replicating our study and obtaining consistent results. We mitigated these threats by providing all the details necessary to replicate our empirical study. The systems under analysis are open-source and the source code repositories are publicly available. Moreover, we published on-line[1] the raw data to allow other researchers to replicate our study and to test other hypotheses on our dataset.

[1] http://swerl.tudelft.nl/twiki/pub/DanieleRomano/WebHome/
/WCRE12rawData.zip

Chapter 9

Conclusions and Future work

This chapter summarizes the project's contributions, concludes our findings and shows the implications. After that, some ideas for future work are discussed.

9.1 Conclusions

Antipatterns have been defined to denote "poor" solutions to design and implementation problems. Previous studies have showed that classes participating in antipatterns are more change-prone than other classes. In this paper we extended existing studies providing a better insight into the change-proneness of Java classes affected by antipatterns. We analyzed the change proneness of these classes extracting and analyzing 40 types of fine-grained source code changes (*SCC*) from the version control repositories of 16 Java open-source systems. Besides confirming existing studies, our results show that:

- Classes participating in antipatterns change more frequently along the evolution of a system (see RQ1).
- There is an association between the number of antipatterns affecting a class and number of *SCC* performed in that class (see RQ1), but no correlation existss.
- Classes participating in *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns are more likely to be changed than classes participating in other antipatterns (see RQ2).
- Certain antipatterns lead to certain types of source code changes (see RQ3). Conditional changes are more frequent in classes affected by the *SpaghettiCode* antipattern; Method add/remove changes are likely in classes affected by the *ComplexClass* and *SpaghettiCode* antipatterns; Changes in executable statements are likely in *AntiSingleton*, *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* .

To sum it up, we contributed to the existing knowledge on the impact of design defects by confirming the high change-proneness of antipattern infected classes, identifying the most change prone antipatterns and showing that specific changes are more common

for specific antipatterns. The study was performed while ignoring unimportant source code changes (comments, copyrights) and counting changes more precisely compared to the previous studies.

9.2 Future work

Based on the results of this study and discussed related works (see the Chapter 3), we suggest a number of possible future works:

- Perform a qualitative analysis of antipatterns. This would give a better understanding of the antipattern evolution. The study may also be restricted to focus only on highly change-prone antipatterns: *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife*.
- Enlarge our dataset and analyze a number of industrial systems. Currently only open-source systems were taken into investigation.
- Perform a study focused on specific domains, specific responsibilities of the classes. As it has been discovered in the previous studies, code defects might be good solutions in particular applications (e.g., GodClass in compilers) or unavoidable in auto-generated code, such as GUI.
- Analyze the types of changes performed when antipatterns are introduced and when they are resolved. This could indicate dangerous changes in antipatterns and solutions to them.
- Evaluate the cost of particular changes in the classes infected by different antipatterns.
- Analysis are needed to further estimate the development and maintenance costs caused by antipatterns.
- Analyze the relation between antipatterns and modifications in the unit tests. Such a study would show if antipatterns impact the change-proneness of the unit tests.

Besides the empirical studies, the knowledge gained in this and future works could be used to enhance existing tools for antipattern detections (see the Chapter 3, section 3.1). For example, during the code review sessions or the process of reengineering engineers would be given better suggestions for the code quality and possible refactoring. Furthermore, they could be warned for the possible consequences before inserting the dangerous changes.

Bibliography

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190, Washington, DC, USA, 2011. IEEE Computer Society.

[2] Chatzigeorgiou Alexander and Manakos Anastasios. Investigating the evolution of bad smells in object-oriented code. *QUATIC*, pages 106–115, 2010.

[3] Du Bois Bart, Demeyer Serge, Verelst Jan, Mens Tom, and Temmerman Marijn. Does god class decomposition affect comprehensibility? In *Proceedings of IASTED Conference on Software Engineering*, pages 346–355, Anaheim, California, 2006. IASTED/ACTA Press.

[4] Fred P. Brooks, Jr. The mythical man-month. In *Proceedings of the international conference on Reliable software*, pages 193–, New York, NY, USA, 1975. ACM.

[5] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[6] Michelle Cartwright and Martin Shephard. An empirical investigation of an object-oriented software system. *IEEE Trans. Softw. Eng.*, 26(8):786–796, 2000.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 120(6):476 – 493, 1994.

[8] James O. Coplien and Neil B. Harrison. *Organizational Patterns of Agile Software Development*. Prentice-Hall, Upper Saddle River, NJ (2005), 1^{st} edition, 2005.

[9] Ratiu Daniel, Ducasse Stéphane, Gîrba Tudor, and Marinescu Radu. Using history information to improve design flaws detection. In *Proceedings of the Conference on Software Maintenance and Reengineering*, pages 223–232, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 175–186, Washington, DC, USA, 1999. IEEE Computer Society.

[11] Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Visual detection of design anomalies. In *Proceedings of the* 12^{th} *European Conference on Software Maintenance and Reengineering, Tampere, Finland*, pages 279–283. IEEE CS Press, April 2008.

[12] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17 –23, may/jun 2000.

[13] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society.

[14] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33:725–743, November 2007.

[15] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Proceeding of the 33rd International Conference on Software Engineering*, pages 1037–1039, New York, NY, USA, 2011. ACM.

[16] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops*, pages 450–457, Washington, DC, USA, 2011. IEEE Computer Society.

[17] Khomh Foutse, Di Penta Massimiliano, Guéhéneuc Yann-Gaell, and Antoniol Guiliano. An exploratory study of the impact of antipatterns on class change- and fault-pronenesss. *Empirical Software Engineering*, 2011.

[18] Khomh Foutse, Vaucher Stéphane, Guéhéneuc Yann-Gael, and A. Sahraoui Houari. A bayesian approach for the detection of code and design smells. In *Proceedings of the International Conference on Quality Software*, pages 305–314, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Khomh Foutse and Gueheneuc Yann-Gael. An empirical study of design patterns and software quality. Technical report, University of Montreal, 2008.

[20] Harald C. Gall, Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE Softw.*, 26:26–33, January 2009.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[22] Emanuel Giger, Martin Pinzger, and Harald C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 83–92, New York, NY, USA, 2011. ACM.

[23] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[24] S. G. Hart and L. E. Stateland. Development of nasa-tlx (task load index): Results of emperical and theoretical research. *P. A. Hancock and N. Meshkati (Eds.) Human Mental Workload*, pages 139–183, 1988.

[25] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[26] S. Deligiannis Ignatios, Stamelos Ioannis, Angelis Lefteris, Roumeliotis Manos, and J. Shepperd Martin. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72:129–143, 2004.

[27] S. Deligiannis Ignatios, J. Shepperd Martin, Roumeliotis Manos, and Stamelos Ioannis. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65:127–139, 2003.

[28] M. Bieman James, Straw Greg, Wang Huxia, Munger P. Willard, and T. Alexander Roger. Design patterns and change proneness: An examination of five evolving systems. In *Proceedings of the International Software Metrics Symposium*, pages 40–49, Washington, DC, USA, 2003. IEEE Computer Society.

[29] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19:77–131, 2007.

[30] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 75–84, Washington, DC, USA, 2009. IEEE Computer Society.

[31] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[32] Foutse Khomh, Stphane Vaucher, Yann-Gal Guhneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In Choi Byoung-ju, editor, *Proceedings of the 9¡sup¿th¡/sup¿ International Conference on Quality Software (QSIC)*. IEEE Computer Society Press, August 2009. 10 pages.

[33] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *proceedings of the* 20^th *international conference on Automated Software Engineering*. ACM Press, Nov 2005.

[34] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[35] S Nash Maliha. Handbook of parametric and nonparametric statistical procedures:handbook of parametric and nonparametric statistical procedures. *Technometrics*, 43:374–374, 2001.

[36] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance*, pages 381–384, Washington, DC, USA, 2003. IEEE Computer Society.

[37] Mika V. Mäntylä. Empirical software evolvability - code smells and human evaluations. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 1–6, Espoo, Finland, 2010. Helsinki University of Technology.

[38] Mika V. Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Trans. Softw. Eng.*, 35:430–448, 2009.

[39] Fokaefs Marios, Tsantalis Nikolaos, and Chatzigeorgiou Alexander. Jdeodorant: Identification and removal of feature envy bad smells. In *International Conference on Software Maintenance*, pages 519–520, New York, NY, USA, 2007. ACM.

[40] Fowler Martin, Beck Kent, Brant John, and Roberts Don. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[41] Bernhart M. Mauczka A., Grechenig T. Predicting code change by using static metrics. In *Software Engineering Research, Management and Applications*, pages 64–71, 2009.

[42] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33:2–13, January 2007.

[43] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.*, 36:20–36, 2010.

[44] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, and Laurence Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 276–291, Berlin, Heidelberg, 2008. Springer-Verlag.

[45] Naouel Moha, Yann-Gael Gueheneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 297–300, Washington, DC, USA, 2006. IEEE Computer Society.

[46] Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. Refactorings of design defects using relational concept analysis. In *Proceedings of the 6th international conference on Formal concept analysis*, ICFCA'08, pages 289–304, Berlin, Heidelberg, 2008. Springer-Verlag.

[47] Tsantalis Nikolaos, Chaikalis Theodoros, and Chatzigeorgiou Alexander. Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the Conference on Software Maintenance and Reengineering*, pages 329–331, Washington, DC, USA, 2008. IEEE Computer Society.

[48] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 390–400, Washington, DC, USA, 2009. IEEE Computer Society.

[49] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I. K. Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Proceedings of the International Conference on Software Maintenance*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[50] Andersen Per Kragh. 3. applied logistic regression. 2nd edn. *Statistics in Medicine*, 21:1963–1964, 2002.

[51] Daryl Posnett, Christian Bird, and Prem Dévanbu. An empirical study on the influence of pattern roles on change-proneness. *Empirical Softw. Engg.*, 16(3):396–423, June 2011.

[52] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Trans. Softw. Eng.*, 27:1134–1144, 2001.

[53] Daniele Romano and Martin Pinzger. Using source code metrics to predict change-prone java interfaces. In *ICSM*, pages 303–312, 2011.

[54] Daniele Romano, Paulius Raila, Martin Pinzger, and Foutse Khomh. Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, Washington, DC, USA, 2012. IEEE Computer Society.

[55] Demeyer Serge, Van Rysselberghe Filip, Gîrba Tudor, Ratzinger Jacek, Marinescu Radu, Mens Tom, Du Bois Bart, Janssens Dirk, Ducasse Stéphane, Lanza Michele, Rieger Matthias, Gall Harald, and El-ramly Mohammad. The lan-simulation: A refactoring teaching example. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[56] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.

[57] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01)*, page 30. IEEE CS Press, 2001.

[58] Harvey Siy and Lawrence Votta. Does the modern code inspection have value? In *Proceedings of the International Conference on Software Maintenance*, pages 281–289, Washington, DC, USA, 2001. IEEE Computer Society.

[59] Vaucher Stéphane, Khomh Foutse, Moha Naouel, and Guéhéneuc Yann-Gael. Tracking design smells: Lessons from a study of god classes. In *Proceedings of the Working Conference on Reverse Engineering*, pages 145–154, Washington, DC, USA, 2009. IEEE Computer Society.

[60] S. Dowdy S.Weardon and D. Chilko. *Statistics for Research. Probability and Statistics*. John Wiley and Sons, 2004.

[61] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.

[62] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*, pages 97–115, Washington, DC, USA, 2002. IEEE Computer Society.

[63] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE CS Press, October 2002.

[64] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gael Gueheneuc. Tracking design smells: Lessons from a study of god classes. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 145–154, Washington, DC, USA, 2009. IEEE Computer Society.

[65] Marek Vokác. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Softw. Eng.*, 30:904–917, 2004.

[66] Luo Yixin, A Hoss, and D.L Carver. An ontological identification of relationships between anti-patterns and code smells. In *Proceedings of the Aerospace Conference*, pages 1–10, Baton Rouge, LA, USA, 2010. Software Eng. Lab., Louisiana State Univ.

[67] Guo Yuepu, B. Seaman Carolyn, Zazworka Nico, and Shull Forrest. Domain-specific tailoring of code smells: an empirical study. In *Proceedings of the International Conference on Software Engineering*, pages 167–170, New York, NY, USA, 2010. ACM.

[68] Nico Zazworka and Christopher Ackermann. Codevizard: a tool to aid the analysis of software evolution. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 1–1, New York, NY, USA, 2010. ACM.

Appendix A

Decor rules

In this appendix we provide DECOR detection rules to detect 13 antipatterns. More information is provided in section 4.5.

Listing A.1: AntiSingleton detection rule

```
RULE_CARD : AntiSingleton {
    RULE : NotClassGlobalVariable { (STRUCT: GLOBAL_VARIABLE, 1) };
};
```

Listing A.2: Blob detection rule

```
RULE_CARD : Blob {
    RULE : Blob { ASSOC: associated FROM: mainClass ONE TO: DataClass MANY };
    RULE : mainClass { UNION LargeClassLowCohesion ControllerClass };
    RULE : LargeClassLowCohesion { UNION LargeClass LowCohesion };
    RULE : LargeClass { (METRIC: NMD + NAD, VERY_HIGH, 0) };
    RULE : LowCohesion { (METRIC: LCOM5, VERY_HIGH, 20) };
    RULE : ControllerClass { UNION
                (SEMANTIC: METHODNAME, {Process, Control, Ctrl, Command, Cmd,
                                        Proc, UI, Manage, Drive})
                (SEMANTIC: CLASSNAME, {Process, Control, Ctrl, Command, Cmd,
                                        Proc, UI, Manage, Drive, System, Subsystem}) };
    RULE : DataClass { (STRUCT: METHOD_ACCESSOR, 90) };
};
```

Listing A.3: CDSBP detection rule

```
RULE_CARD : ClassDataShouldBePrivate {
    RULE : FieldPublic { (STRUCT: PUBLIC_FIELD, 1) };
};
```

Listing A.4: ComplexClass detection rule

```
RULE_CARD : ComplexClass {
    RULE : ComplexClass { UNION LargeClassOnly ComplexClassOnly } ;
    RULE : LargeClassOnly { (METRIC: NMD + NAD, VERY_HIGH, 0) } ;
    RULE : ComplexClassOnly { (METRIC: McCabe, VERY_HIGH, 20) } ;
};
```

Listing A.5: LazyClass detection rule

```
RULE_CARD : LazyClass {
        RULE : LazyClass { INTER NotComplexClass FewMethods };
        RULE : NotComplexClass { (METRIC: WMC, VERY_LOW, 20) };
        RULE : FewMethods { (METRIC: NMD + NAD, VERY_LOW ,5) };
};
```

Listing A.6: LPL detection rule

```
RULE_CARD : LongParameterList {
        RULE : LongParameterListClass { (METRIC: NOParam, VERY_HIGH, 20) } ;
};
```

Listing A.7: MessageChain detection rule

```
RULE_CARD : MessageChains {
        RULE : MessageChainsClass { (METRIC: NOTI, SUP_EQ, 4, 0) } ;
};
```

Listing A.8: RPB detection rule

```
RULE_CARD : RefusedParentBequest {
        RULE : RefusedParentBequest {
                    INHERIT: inherited FROM: ParentClassProvidesProtected
                    ONE TO: RareOverriding ONE } ;
        RULE : ParentClassProvidesProtected { (METRIC: USELESS, EQ, 1,0) } ;
        RULE : RareOverriding { (METRIC: IR, VERY_LOW, 0)};
};
```

Listing A.9: SpaghettiCode detection rule

```
RULE_CARD : SpaghettiCode {
    RULE : SpaghettiCode { INTER NoInheritanceClassGlobalVariable
        LongMethodMethodNoParameter };
    RULE : LongMethodMethodNoParameter { INTER LongMethod MethodNoParameter };
    RULE : LongMethod { (METRIC: METHOD_LOC, VERY_HIGH, 0) };
    RULE : MethodNoParameter { (METRIC: NOParam, INF, 5, 0) };
    RULE : NoInheritanceClassGlobalVariable { INTER NoInheritance ClassGlobalVariable };
    RULE : NoInheritance { (METRIC: DIT, INF_EQ, 2, 0) };
    RULE : ClassGlobalVariable { (STRUCT: GLOBAL_VARIABLE, 1) };
};
```

Listing A.10: SG detection rule

```
RULE_CARD : SpeculativeGenerality {
        RULE : SpeculativeGenerality { INTER AbstractClass OneChildClass };
            RULE : AbstractClass {(STRUCT: IS_ABSTRACT, 1) };
            RULE : OneChildClass { (METRIC: NOC, EQ, 1, 0) };
};
```

Listing A.11: SwissArmyKnife detection rule

```
RULE_CARD : SwissArmyKnife {
    RULE : MultipleInterface {(STRUCT: MULTIPLE_INTERFACE, 3) };
};
```

,

Appendix B

Releases

In this appendix we list the releases of software applications which forms the dataset of our study. Columns *#Classes*, *#Antipatterns*, *#SCC* shows the number of classes, antipatterns and *SCC* in each release.

Release	#Classes	#Antipatterns	#SCC
ArgoUML-0.10.1	896	1278	14182
ArgoUML-0.12	984	1421	7551
ArgoUML-0.14	1282	1838	7237
ArgoUML-0.16	1263	1765	27022
ArgoUML-0.18.1	1332	1702	11569
ArgoUML-0.20	1483	1906	15548
ArgoUML-0.22	1579	2109	13279
ArgoUML-0.24	1639	2147	352
ArgoUML-0.26	1859	2089	1027

Table B.1: argo releases.

,

Release	#Classes	#Antipatterns	#SCC
hibernate-2.0beta1	428	559	2822
hibernate-2.0beta3	437	489	1374
hibernate-2.0beta4	476	576	3247
hibernate-2.0rc2	414	494	1399
hibernate-2.0	437	483	853
hibernate-2.0.1	401	513	1091
hibernate-2.1final	492	604	12876
hibernate-2.1.4	512	620	720
hibernate-2.1.5	518	666	1276
hibernate-2.1.7	537	638	441

Table B.2: hibernate2 releases.

Release	#Classes	#Antipatterns	#SCC
hibernate-3.0alpha	654	855	7778
hibernate-3.0beta1	748	982	3655
hibernate-3.0beta2	795	1020	1575
hibernate-3.0beta3	802	805	2985
hibernate-3.0beta4b	828	1026	3905
hibernate-3.0rc1	876	1082	1979
hibernate-3.0	877	1066	758
hibernate-3.0.1	887	1088	954
hibernate-3.0.2	893	1077	562
hibernate-3.0.3	894	1093	1461
hibernate-3.0.5	907	1125	2671
hibernate-3.1alpha1	934	915	1651
hibernate-3.1beta1	946	1162	2827
hibernate-3.1beta2	975	1195	527
hibernate-3.1beta3	972	1207	628
hibernate-3.1rc2	994	1204	786
hibernate-3.1rc3	1015	1235	710
hibernate-3.1	1019	1256	526
hibernate-3.1.1	1034	1247	380
hibernate-3.1.2	1036	1273	953

Table B.3: hibernate3 releases.

Release	#Classes	#Antipatterns	#SCC
eclipse-SDK-2.0-win32	63	19	872
eclipse-SDK-2.1-win32	79	22	188
eclipse-SDK-2.1.1-win32	79	22	450
eclipse-SDK-2.1.2-win32	79	21	191
eclipse-SDK-2.1.3-win32	79	21	392
eclipse-SDK-3.0-win32	159	68	826
eclipse-SDK-3.0.2-win32	159	67	565
eclipse-SDK-3.1-win32	159	75	237
eclipse-SDK-3.1.2-win32	159	75	383
eclipse-SDK-3.2-win32	160	77	1754
eclipse-SDK-3.3-win32	197	109	551
eclipse-SDK-3.4-win32	201	118	1191

Table B.4: eclipse.debug.core releases.

Release	#Classes	#Antipatterns	#SCC
eclipse-SDK-2.0-win32	157	125	1078
eclipse-SDK-2.0.2-win32	157	125	2621
eclipse-SDK-2.1-win32	212	186	810
eclipse-SDK-2.1.1-win32	212	186	1628
eclipse-SDK-2.1.2-win32	212	186	986
eclipse-SDK-2.1.3-win32	212	187	1672
eclipse-SDK-3.0-win32	416	343	1183
eclipse-SDK-3.0.1-win32	416	338	4434
eclipse-SDK-3.0.2-win32	416	331	2250
eclipse-SDK-3.1-win32	561	579	452
eclipse-SDK-3.1.1-win32	561	586	3909
eclipse-SDK-3.1.2-win32	561	583	4369
eclipse-SDK-3.2-win32	740	638	605
eclipse-SDK-3.2.1-win32	741	645	3663
eclipse-SDK-3.2.2-win32	741	648	2352
eclipse-SDK-3.3-win32	820	722	1600
eclipse-SDK-3.3.1-win32	820	714	812
eclipse-SDK-3.3.2-win32	820	713	943
eclipse-SDK-3.4-win32	859	722	1070
eclipse-SDK-3.4.2-win32	858	717	748
eclipse-SDK-3.5-win32	907	742	2535
eclipse-SDK-3.5.2-win32	907	740	831

Table B.5: eclipse.debug.ui releases.

Release	#Classes	#Antipatterns	#SCC
eclipse-SDK-2.1.1-win32	189	111	479
eclipse-SDK-2.1.2-win32	189	116	1023
eclipse-SDK-2.1.3-win32	189	114	1247
eclipse-SDK-3.0-win32	217	136	280
eclipse-SDK-3.0.1-win32	217	137	1220
eclipse-SDK-3.0.2-win32	217	136	977
eclipse-SDK-3.1-win32	305	198	1235
eclipse-SDK-3.1.2-win32	305	199	1140
eclipse-SDK-3.2-win32	371	237	652
eclipse-SDK-3.2.1-win32	371	238	1105
eclipse-SDK-3.2.2-win32	371	241	1491
eclipse-SDK-3.3-win32	416	286	1208
eclipse-SDK-3.3.2-win32	416	285	678
eclipse-SDK-3.4-win32	434	296	369
eclipse-SDK-3.4.2-win32	434	297	409
eclipse-SDK-3.5-win32	441	308	559

Table B.6: eclipse.jface releases.

Release	#Classes	#Antipatterns	#SCC
eclipse-SDK-2.0-win32	428	389	1075
eclipse-SDK-2.0.1-win32	428	380	1100
eclipse-SDK-2.0.2-win32	428	388	1146
eclipse-SDK-2.1-win32	441	367	754
eclipse-SDK-2.1.1-win32	441	373	435
eclipse-SDK-2.1.2-win32	441	368	837
eclipse-SDK-2.1.3-win32	441	378	325
eclipse-SDK-3.0-win32	468	418	650
eclipse-SDK-3.0.1-win32	468	426	2207
eclipse-SDK-3.0.2-win32	468	427	2135
eclipse-SDK-3.1-win32	471	484	237
eclipse-SDK-3.1.1-win32	471	463	942
eclipse-SDK-3.2-win32	476	484	1327
eclipse-SDK-3.3-win32	500	547	515
eclipse-SDK-3.4-win32	500	528	591
eclipse-SDK-3.5-win32	508	534	707

Table B.7: eclipse.jdt.debug releases.

Release	#Classes	#Antipatterns	#SCC
eclipse-SDK-2.0-win32	48	18	517
eclipse-SDK-2.1.2-win32	53	35	247
eclipse-SDK-3.0-win32	102	82	459
eclipse-SDK-3.1-win32	119	96	280
eclipse-SDK-3.1.2-win32	119	96	322
eclipse-SDK-3.2-win32	199	147	493

Table B.8: eclipse.team.core releases.

Release	#Classes	#Antipatterns	#SCC
eclipse-SDK-2.0-win32	132	123	989
eclipse-SDK-2.0.2-win32	132	123	3834
eclipse-SDK-2.1-win32	147	161	2772
eclipse-SDK-2.1.2-win32	147	163	488
eclipse-SDK-2.1.3-win32	147	163	893
eclipse-SDK-3.0-win32	184	183	1010
eclipse-SDK-3.0.2-win32	184	183	745
eclipse-SDK-3.1-win32	186	202	387
eclipse-SDK-3.1.2-win32	186	202	407
eclipse-SDK-3.2-win32	208	234	1100
eclipse-SDK-3.3-win32	210	231	445

Table B.9: eclipse.team.cvs.core releases.

Release	#Classes	#Antipatterns	#SCC
eclipse-SDK-2.0-win32	80	38	684
eclipse-SDK-2.1.1-win32	88	51	165
eclipse-SDK-2.1.2-win32	88	51	102
eclipse-SDK-2.1.3-win32	88	51	223
eclipse-SDK-3.0-win32	171	114	630
eclipse-SDK-3.0.1-win32	171	113	824
eclipse-SDK-3.0.2-win32	171	113	670
eclipse-SDK-3.1-win32	222	145	638
eclipse-SDK-3.1.2-win32	222	145	1456
eclipse-SDK-3.2-win32	360	279	2387
eclipse-SDK-3.2.2-win32	361	278	457
eclipse-SDK-3.3-win32	395	275	1047
eclipse-SDK-3.5-win32	405	288	504

Table B.10: eclipse.team.ui releases.

Release	#Classes	#Antipatterns	#SCC
JabRef-1.0	174	77	644
JabRef-1.1	192	101	1264
JabRef-1.19	248	130	1342
JabRef-1.2	238	142	1806
JabRef-1.4	505	453	1479
JabRef-1.5	663	563	1528
JabRef-1.55	850	743	818
JabRef-1.6	874	742	3764
JabRef-1.7b	1038	875	1570
JabRef-1.7	1057	876	2582
JabRef-1.8b	1099	885	1873
JabRef-1.8	981	765	2795
JabRef-2.0b	1284	912	558
JabRef-2.0.1	1276	928	1155
JabRef-2.1b	1355	1015	1105
JabRef-2.1	1350	1006	1619
JabRef-2.2b	1859	1559	355
JabRef-2.2b2	2095	1837	724
JabRef-2.2	2101	1896	1672
JabRef-2.3b	2196	1832	3751
JabRef-2.3b2	2202	1849	369
JabRef-2.3b3	2208	1864	1828
JabRef-2.3	2212	1849	976
JabRef-2.4b	2508	2183	627
JabRef-2.4	2517	2171	199
JabRef-2.4.2	2540	2194	1125
JabRef-2.5b	2567	2214	951
JabRef-2.6b	2581	2241	675
JabRef-2.6	2588	2235	1239
JabRef-2.7b	2597	2244	1272

Table B.11: jabref releases.

Release	#Classes	#Antipatterns	#SCC
mylar-site-1.0.1-e3.3	776	512	7100
mylar-site-2.0M1-e3.3	936	625	3203
mylar-site-2.0M2-e3.3	966	651	4301
mylar-site-2.0M3-e3.3	1037	727	10795
mylyn-2.0.0-e3.3	882	593	3990
mylyn-2.1-e3.3	916	639	4244
mylyn-2.2.0-e3.3	971	697	3343
mylyn-2.3.0-e3.3	1060	773	1120
mylyn-2.3.1	1062	772	905
mylyn-2.3.2	1062	770	14559
mylyn-3.0.0	1106	751	510
mylyn-3.0.1	1109	754	859
mylyn-3.0.2	1121	760	1462
mylyn-3.0.3	1124	767	4035
mylyn-3.0.4	1124	761	1098
mylyn-3.0.5	1125	761	847
mylyn-3.1.0	1596	1236	4679

Table B.12: mylyn releases.

Release	#Classes	#Antipatterns	#SCC
Rhino140R3_RELEASE	95	85	4631
rhino15R1	125	108	4031
rhino15R2	187	164	780
rhino15R4	206	147	741
rhino1_5R5	209	139	1423
rhino1_6R1	218	163	426
rhino1_6R2	218	159	234
rhino1_6R3	219	163	2529

Table B.13: rhino releases.

Release	#Classes	#Antipatterns	#SCC
rapidminer-4.6-community	2261	2690	239
rapidminer-5.0	2839	3412	6702
rapidminer-5.0.008	2861	3433	840
rapidminer-5.0.009	2907	3321	2118

Table B.14: rapidminer releases.

Release	#Classes	#Antipatterns	#SCC
Azureus3.0.0.3	2838	2525	11380
Azureus3.0.0.8	2901	2905	915
Azureus3.0.1.0	2911	2917	2754
Azureus3.0.1.4	2961	2983	2836
Azureus3.0.1.6	2980	2999	7463
Azureus3.0.2.0	3038	3045	3346
Azureus3.0.3.0	3063	3092	593
Azureus3.0.3.4	3065	3095	3278
Azureus3.0.4.0	3096	3134	5161
Azureus3.0.4.2	3082	3115	4780
Azureus3.0.5.0	3164	3185	8839
Azureus3.0.5.2	3180	3178	2831
Azureus3.1.0.0	3381	3406	3781
Azureus3.1.1.0	3458	3586	13592
Azureus4.0.0.0	3557	3599	1172
Azureus4.0.0.2	3550	3593	1629
Azureus4.0.0.4	3562	3591	1690
Azureus4.1.0.0	3605	3598	4880
Azureus4.1.0.2	3605	3592	685
Azureus4.1.0.4	3606	3602	852
Azureus4.2.0.0	3671	3649	4723
Azureus4.2.0.2	3671	3653	2129
Azureus4.2.0.4	3694	3669	3611
Azureus4.2.0.8	3746	3724	5058
Azureus4.3.0.0	3680	3559	9770
Azureus4.3.0.2	3679	3553	3815
Azureus4.3.1.0	3696	3570	1818
Azureus4.3.1.2	3698	3565	2247
Vuze_4400	3733	3597	3510

Table B.15: vuze releases.

Release	#Classes	#Antipatterns	#SCC
Xerces-J-bin.2.0.0.alpha	247	147	2216
Xerces-J-bin.2.0.0.beta	536	370	1914
Xerces-J-bin.2.0.0.beta3	633	490	3951
Xerces-J-bin.2.0.0.beta4	608	553	4641
Xerces-J-bin.2.0.0	579	439	2014
Xerces-J-bin.2.0.1	579	477	8511
Xerces-J-bin.2.0.2	617	512	4381
Xerces-J-bin.2.1.0	633	553	1811
Xerces-J-bin.2.2.0	642	524	4944
Xerces-J-bin.2.3.0	673	544	1416
Xerces-J-bin.2.4.0	678	555	3310
Xerces-J-bin.2.5.0	692	545	4259
Xerces-J-bin.2.6.0	700	565	3656
Xerces-J-bin.2.6.2	723	569	9233
Xerces-J-bin.2.7.0	795	668	1941
Xerces-J-bin.2.8.0	802	643	616
Xerces-J-bin.2.8.1	804	642	1675
Xerces-J-bin.2.9.0	808	647	1340
Xerces-J-bin.2.9.1	815	645	7121
Xerces-J-bin.2.10.0	869	700	599

Table B.16: xerces releases.