Electronic Communications of the EASST
Volume 59 (2013)

Special Issue of the First Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP 2013)

Analysing Anti-patterns Static Relationships with Design Patterns

Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh

26 pages

Guest Editors: Aminata Sabané, Wei Wu
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Analysing Anti-patterns Static Relationships with Design Patterns

Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh

Fehmi Jaafar
School of Computing, Queen's University, Ontario, Canada
Sylvie Hamel
DIRO, Université de Montréal, Québec, Canada
Yann-Gaël Guéhéneuc and Foutse Khomh
DGIGL, École Polytechnique de Montréal, Québec, Canada

Abstract: Anti-patterns are motifs that are usually thought to be good solutions to some design or implementation problems, but back-fires badly when applied. Previous studies have reported that anti-patterns make object oriented systems hard to maintain. Anti-patterns motifs usually have dependencies with other classes in the system. In this paper, we propose to analyse these dependencies (with in particular design patterns) in order to understand how developers can maintain programs containing anti-patterns. To the best of our knowledge, no substantial investigation of anti-pattern dependencies with design patterns has been done before. This paper presents the results of a study that we performed on three different systems, ArgoUML, JFreeChart, and XercesJ, written in Java, and of size ranges from 1,191 to 3,325 classes, to analyse the static relationships between anti-patterns and design patterns. We found that these relationships (1) exist, but (2) are temporary and (3) classes participating in such relationships are more change-prone but less fault-prone than other anti-pattern classes.

Keywords: Anti-patterns, Design Patterns, Static Relationships, Change-proneness, Fault-proneness, Software Evolution, Mining Software Repositories.

1 Context and Problem

Software systems continuously evolve in order to incorporate ever changing customers' requirements, performance improvements, and bug fixes. Without proper knowledge, developers may introduce anti-patterns in the system. In theory, anti-patterns [Web95] are "poor" solutions to recurring problems. In practice, an anti-pattern is a literary form that describes a bad solution to recurring design problems that leads to negative effects on code quality [BMMM98]. Opposite to anti-patterns, design patterns [GHJV94] are "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability and, above all, maintainability and resilience to changes. Large, long-lifespan systems often have both design patterns and anti-patterns and, consequently, anti-patterns and design patterns may have some relationships, *i.e.* the classes participating to some design patterns may be in relation with those participating to some anti-patterns. The static relationships among classes are typically use, association, aggregation, and composition relationships [GA04].

Research Problem Most previous work agree that anti-patterns render the maintenance of systems more difficult [KPGA12, AKGA11] and that design patterns can serve as guide in program exploration and, thus, ease maintenance [Vok04, ACC+07]. However, there are few previous work about the relationships between anti-patterns and design patterns. Yet, understanding these relationships can help developers to better understand and maintain software systems.

Contribution The contribution of this paper is three-fold. First, we extend the existing empirical knowledge by an empirical study that demonstrates that anti-patterns and design patterns can have static relationships in systems. Second, we provide knowledge about the evolution of these relationships. Finally, we present the results of an empirical study on the impact of such static relationships on change proneness and fault proneness. Indeed, in the study reported in this paper we observe that anti-patterns do have static relationships with design patterns, but that these relationships are temporary. Yet, anti-pattern classes participating in such relationships are more change-prone but less fault prone than other anti-pattern classes. Therefore, it seems that developers sometimes use design patterns as a temporary fix for anti-patterns (since both the anti-patterns and the relationships to design patterns) are removed later from the systems. The fix seems to be working since the fault-proneness of anti-patterns are reduced. Detecting and analysing static relationships of anti-patterns is important from the points of view of both researchers and practitioners. In fact, we bring evidence to researchers that (1) anti-patterns *do* statically relate to some design patterns, that (2) some anti-patterns have more relationships with design patterns than others, and that (3) these relationships indicate specific trends for the evolution of classes in term of fault-proneness and change-proneness.

Organisation Section 2 relates our study with previous work. Section 3 presents our method. Section 4 describes our empirical study. Section 5 presents the study results, while Section 6 discusses them, along with threats to their validity. Finally, Section 7 concludes the study and outlines future work.

2 Related Work

Several work studied the detection and the analysis of anti-patterns and design patterns. For lack of space, we only cite some relevant work, the interested readers can find more references in our previous work [MGDL10] and [GA08].

Anti-patterns Definition and Detection Code smells and anti-patterns both describe re-occurring software problems. Code smells are symptoms of problems existing in the source code [MGDL10]. Code smells are related to the inner workings of classes while anti-pattern include the relationships among classes and are more situated on a micro-architectural level. Concretely, code smells give warnings to software developers that the source code has some problems, while anti-patterns provide software managers, architects, designers, and developers a common vocabulary for recognizing possible sources of problems in advance. The first book on “anti-patterns” in object-oriented development was written in 1995 by Webster [Web95]. In this book, the author

reported that an anti-pattern describes a frequently used solution to a problem that generates ineffective or decidedly negative consequences. Riel [Rie96] defined 61 heuristics characterising good object-oriented programming to assess a program quality manually and improve its design and implementation. These heuristics are similar and–or precursor to code smells. Brown *et al.* [BMMM98] discussed 40 anti-patterns, which are often described in terms of lower-level code smells. These books provide in-depth views on heuristics, code smells, and anti-patterns aimed at a wide academic audience. They are the basis of all the approaches to detect anti-patterns.

The study presented in this paper relies on anti-patterns detection approach proposed in [MGDL10]. However several other approaches have been proposed in the past. For example, Van Emden *et al.* [EM02] developed the JCosmo tool. This tool parses source code into an abstract model (similar to the Famix meta-model). It uses primitives and rules to detect the presence of smells and anti-patterns. The JCosmo tool can visualize the code layout and anti-patterns locations. The goal of JCosmo is to help developers assess code quality and perform refactorings. The main difference compared with other detection tools is that JCosmo tries to visualize problems by visualizing the design. Marinescu *et al.* developed a set of detection strategies to detect anti-patterns using metrics [RDGM04]. They later refined their detection strategies by adding information collected from the documentation of problematic structures. They showed how to detect several anti-patterns, such as God Classes and Data Classes. Settas *et al.* explored the ways in which an anti-pattern ontology can be enhanced using Bayesian network [SCF12]. Their approach allow software developers to quantify the existence of an anti-pattern using Bayesian network; based on probabilistic knowledge contained in an anti-pattern ontology regarding relationships of anti-patterns through their causes, symptoms and consequences.

The Integrated Platform for Software Modeling and Analysis (iPlasma) described in [LM06] can be used for anti-patterns detection. This platform calculates metrics from C++ or Java source code and applies rules to detect anti-patterns. The rules combine the metrics and are used to find code fragments that exceed some thresholds.

Design Pattern Definition and Detection The first book on "design patterns" in object-oriented development was written in 1996 by Gamma et al. [GHJV94]. Since this book, several workshops and conferences have emerged to propose new patterns. Many papers have been published studying the use, impact of patterns. The study presented in this paper relies on design patterns detection approach proposed in [GA08]. However several other approaches have been proposed in the past. For example, one of the first papers about detecting design patterns was written by Kramer *et al.* [KP96] in 1996. They introduced an approach detecting design information directly from C++ header files. This information is stored in a repository. The design patterns are expressed as PROLOG rules which are used to query the repository with the extracted information. Their work focused on detecting five structural design patterns: Adapter, Bridge, Composite, Decorator, and Proxy. Recently, an approach based on similarity scoring has also been proposed [TCSH06], which provides an efficient means to compute the similarity between the graph of a design motif and the graph of a system to identify classes participating to a design motif. Iacob [Iac11] presented a method that aims at identifying proven solutions to recurring design problems through design workshops and systems analysis. Indeed, during a design workshop, a team of 3-5 designers is asked to design a system and the design issues they address

are collected. Moreover, a set of systems are analysed in order to identify in what measure the design issues discussed during the workshops are considered in the implementation of existing solutions. Candidates for being documented as design patterns are the most recurring design issues in both the workshops and the systems analysis.

Anti-patterns and Design Patterns Static Relationships analysis There are few papers analyzing the relationships among anti-patterns and design patterns. Vokac [Vok04] analyzed the corrective maintenance of a large commercial program, comparing the defect rates of classes participating in design motifs against those that did not. He found that the Observer and Singleton motifs are correlated with larger classes; classes playing roles in Factory Method were more compact, less coupled, and less defect prone than other classes; and, no clear tendency exists for Template Method. Their approach showed correlation between some design patterns and smells like LargeClass but do not report an exhaustive investigation of possible correlations between these patterns and anti-patterns. Pietrzak and Walter [PW06] defined and analysed the different relationships that exist among smells and provide tips about how they could be exploited to alleviate the detection of anti-patterns. The authors performed an experiment to show that the use of knowledge about identified smells in Jakarta Tomcat code supports the detection process. They found examples of several smell dependencies, including aggregate relationships. The certainty factor for those relations in that code suggests the existence of correlation among the dependent smells and applicability of this approach to anti-patterns detection. Rather than focusing on the relationships among code smells and anti-patterns, our study focuses on analysing anti-patterns relationships with design patterns.

Yamashita et *al.* [YM13] reported that the interactions between code smells affect maintenance. They investigated the interactions amongst 12 code smells and analyzed how those interactions relate to maintenance problems. They revealed how smells that were co-located in the same artifact interacted with each other, and affected maintainability. Indeed, they found that code smell interactions occurred across coupled artifacts, with comparable negative effects as same-artifact co-location.

Anti-patterns and Design Patterns Evolution and Impact Bieman et *al.* [BSW+03] explored whether the relative stability of design pattern classes compared to other classes were being empirically realized. The authors showed that large classes were found to be the most change-prone and that pattern-based classes were more change-prone. Vokac [Vok04] found significant differences in the fault-proneness of different design patterns in a study of a large $C++$ industrial system. In the same direction, Gatrell et *al.* [GCH09] showed that pattern-based classes are more change-prone than non-pattern classes.

On the other hand, Olbrich *et al.* [OCBZ09] analysed the historical data of Lucene and Xerces over several years and concluded that Blob classes and classes subjected to Shotgun Surgery have a higher change frequency than other classes; with Blob classes featuring more changes. Similarly, Chatzigeorgiou and Manakos [CM10] studied the evolution of Long Method throughout successive versions of two open-source systems and concluded that a significant percentage of these smells are introduced during the addition of new methods to the system. They also found that this anti-pattern persists in systems and that their removal is often a side effect of

adaptive maintenance rather than the result of targeted refactoring activities. Using Azureus and Eclipse, Khomh *et al.* [KPGA12] investigated the impact of code smells on the change-proneness of classes and showed that in general, the likelihood for classes with code smells to change is very high. Khomh *et al.* [KPGA12] also investigated the relation between the presence of anti-patterns and the change- and fault-proneness of classes. They detected 13 anti-patterns from 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino, and analyzed the likelihood that a class with an anti-pattern will change in the future, in particular to fix a fault. They concluded that classes participating in anti-patterns are significantly more likely to be subject to changes and to be involved in fault-fixing changes than other classes.

Khomh *et al.* [KPGA12] also investigated the kind of changes experienced by classes with anti-patterns. Their study focused on 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. They concluded that structural changes are more likely to occur in classes participating in anti-patterns.

Deligiannis *et al.* [DSRS03] proposed the first quantitative study of the impact of anti-patterns on software development and maintenance activities. They performed a controlled experiment with 20 students on two systems. The experiment aimed to understand the impact of Blob classes on the understandability and maintainability of systems. The results of their study suggest that Blob classes affect the evolution of design structures and the subjects' use of inheritance. However, Deligiannis *et al.* did not assess the impact of anti-patterns static relationships on subjects' understandability and ability to perform the tasks successfully.

Yamashita and Moonen [YM12] investigated the extent to which code smells reflect factors affecting software maintainability and observed that using code smells definitions alone, developers cannot fully evaluate the overall maintainability of a software system. They conclude on the need to combine different analysis approaches in order to achieve more complete and accurate evaluations of the overall maintainability of a software system.

Taba *et al.* [SN13] argue that anti-patterns can tell developers whether a design choice is "poor" or not. They explored the use of anti-patterns for fault prediction, and strive to improve the accuracy of fault prediction models by proposing various metrics based on anti-patterns. They considered the history of anti-patterns in files from their inception into the system. They observed that files participating in anti-patterns have higher fault density than other files and that their proposed anti-pattern-based metrics can provide additional explanatory power over traditional metrics and improve fault prediction models. They suggest that developers use their proposed metrics to better improve fault prediction models and better focus testing activities and the allocation of support resources.

We share with all the above authors the idea that anti-patterns detection is a powerful mechanism to assess code quality, in particular indicating whether the existence of anti-patterns and the growth of their relationships makes the source code more difficult to maintain.

Summary: These previous works raised the awareness of the community towards the impact of anti-patterns and design patterns on software development and maintenance activities. In this paper, we build on these previous works and analyze the existence and evolution of anti-patterns static relationships with design patterns. We aim to understand if the negative effects of anti-patterns can transit to other classes through static dependencies and if design patterns can mitigate these negative effects. We also analyze the evolution and impact of relationships

between anti-patterns and design patterns in terms of fault proneness and change proneness.

3 Approach

This section describes the steps of our data collection process. We use two previous approaches DECOR[MGDL10] to detect anti-patterns, and DeMIMA[GA08] to detect design patterns.

3.1 Step 1: Detecting Anti-patterns

We use the Defect DEtection for CORrection Approach DECOR[MGDL10] to specify and detect anti-patterns. DECOR is based on a thorough domain analysis of anti-patterns defined in the literature, and provides a domain-specific language to specify code smells and anti-patterns and methods to detect their occurrences automatically. It can be applied on any object-oriented system through the use of the PADL [GA08] meta-model and POM framework. PADL is a meta-model to describe object-oriented systems [GA08]. POM is a PADL-based framework that implements more than 60 metrics.

Indeed, Decor proposes a domain-specific language to specify and generate automatically design defect detection algorithms. A domain-specific language offers greater flexibility than ad hoc algorithms because the domain experts, the software engineers, can specify and modify manually the detection rules using high-level abstractions, taking into account the context, environment, and characteristics of the analysed systems. Moreover, the language allows specifying defect detection algorithms at a high-level of abstraction using key concepts found in their text-based descriptions, not in the underlying ad hoc detection framework, as in previous work.

Moha *et al.* [MGDL10] reported that DECOR current anti-patterns' detection algorithms achieve 100% recall and have a precision greater than 31% in the worst case, with an average precision greater than 60%.

3.2 Step 2: Detecting Design Patterns

We use the Design Motif Identification Multilayered Approach (DeMIMA)[GA08] to specify and detect design patterns. DeMIMA ensures traceability between motifs and source code by first identifying idioms related to binary class relationships to obtain an idiomatic model of the source code and then, by using this model, it can identify design motifs and generate a design model of the system. Indeed, DeMIMA makes it possible to recover two kinds of design choices from source code: idioms pertaining to the relationships among classes and design motifs characterizing the organization of the classes. DeMIMA depends on a set of definitions for unidirectional binary class relationships. The formalizations define the relationships in terms of four language-independent properties that are derivable from static and dynamic analyses of systems: exclusivity, type of message receiver, lifetime, and multiplicity. DeMIMA keeps track of data and links to identify and ensure the traceability of these relationships. DeMIMA also uses explanation-based constraint programming to identify microarchitectures similar to design motifs. This technique makes it possible to identify microarchitectures similar to a model of a design motif without having to describe all possible variants explicitly. We also use DeMIMA to detect motifs's relationships. In fact, DeMIMA distinguishes use, association, aggregation, and

composition relationships because such relationships exist in most notations used to model systems, for example, in UML. Gueheneuc and Antoniol [GA08] reported that DeMIMA can detect design patterns with a recall of 100% and a precision greater than 34%. While, for the detection of relationships among classes, the DeMIMA approach ensures 100% recall and precision.

3.3 Step 3: Analysing Motifs Static Relationships

Table 1 summarizes anti-patterns considered in this paper. To perform the empirical study, we choose to analyse the relationships of the well known anti-patterns and six design patterns belonging to three categories: creational patterns (Factory method and Prototype), structural patterns (Composite and Decorator), and behavioral patterns (Command and Observer). Thus, we choose these motifs because they are representative of problems with data, complexity, size, and the features provided by classes. We choose, also, these motifs because they have been used and analysed in previous work [MGDL10] and [KPGA12]. Definitions and specifications are outside of the scope of this paper and are available in [GHJV94] and [KPGA12].

We assume that a design pattern P has a static relationships with the anti-pattern A if at least one class belonging to P has a use, association, aggregation, or composition relationships with one class belonging to A.

3.4 Step 4: Analysing the Evolution of Motifs and their Static Relationships

Our analysis goes as follows: we detect all static relationships between anti-patterns and design patterns in the first studied version of a system. Then, we investigate whether these relationships persist in the following versions. We hold that a relationship will be ignored by our approach if the classes playing roles in anti-patterns in the first studied version are restructured and corrected and, thus, they do not belong to anti-patterns any more in the next versions.

To analyse the evolution of static relationships between anti-patterns and design patterns, we need to detect class renamings, class changes, and fault fixing. If we considered only class name to identify classes in different versions, then a same class in two different versions is considered as two different classes if it is renamed. To overcome this issue, we use the structure-based and text-based similarities to identify class renamings in programs.

Concretely, we use previous approaches: ADvISE [HGHG12] and Macocha [JGHA11].

ADvISE identifies class renamings using structure-based and text-based metrics, to assess the similarities between original and renamed classes, as follows:

3.4.1 Structure-based and Text-based Similarities

The structure-based similarity ($StrS$), between a candidate renamed class C_A and a target class C_B, is defined as the percentage of their common methods, attribute types, and relationships. We compute the text similarity, between a candidate renamed class C_A and each of the target classes C_{B_i} $i \in [1,n]$, using a Camel similarity ($CamelS$), and the Normalized Levenshtein Edit Distance (ND). The $CamelS$ similarity between C_A and C_B represents the percentage of their common

tokens. The Normalized Edit Distance (*ND*) between C_A and C_B is defined as:

$$ND(C_A, C_B) = \frac{Levenshtein(C_A, C_B)}{sum(length(C_A), length(C_B))} \in [0,1]$$

Let $S(C_A)$ (respectively $S(C_B)$) be the set of methods, attributes, and relationships of C_A (respectively C_B). The structure-based and the text-based similarities of C_A and C_B are computed by comparing $S(C_A)$ to $S(C_B)$ using the Jaccard index of similarity [RV96]. In fact, when we compare the similarities of a candidate renamed class C_A to many target classes $\{C_{B_1}, ..., C_{B_n}\}$, we first compare their structure-based similarity *StrS*. We select the set of target classes having the highest *StrS* value. Then, we compute their textual similarities (*ND* and *CamelS*).

Finally, we combine *ND* and *CamelS* to compare the text similarity between names of the candidate class and the target class, because *ND* and *CamelS* assess to two different aspects of string comparison: *ND* is concerned with the difference between strings but cannot tell if they have something in common, while *CamelS* focuses on their common tokens but cannot tell how different they are. ADvISE reports the $S(C_B)$ with the highest *CamelS* and the lowest *ND* scores as the class renamed from $S(C_A)$. The score combined of *ND* and *CamelS* is equal to:
$S = \frac{CamelS}{ND}$
We also use Macocha [JGHA11] to identify the set of changes performed on each class by mining version-control systems as follow:

3.4.2 Analysing the Change Proneness and the Fault Proneness

We explore whether classes belonging to an anti-pattern and related to a design pattern are less change prone than other anti-pattern classes.

Indeed, Macocha mines version-control systems (CVS and SVN), to detect changes committed in each class. A commit contains several attributes: the changed class names, the dates of changes, the name of the developer who committed the changes. Macocha takes as input a CVS/SVN change log and creates a profile that describes the evolution of each class.

First, using Fisher's exact test [She07] and Odds ratios (OR), we investigate whether static relationships between anti-pattern and design pattern are significantly correlated to a higher class's change-proneness. We present more details about this investigation in Section 4.

Second, we compute the fault-proneness of a class by relating fault reports and commits to the class. Indeed, fault fixing changes are documented in textual reports that describe different kinds of problems in a program. We parse the SVN/CVS change logs of our subject systems and apply the heuristics by Sliwersky *et al.* [SZZ05] to identify fault fix locations. We parse commit log messages using a Perl script and extract bug IDs and specific keywords, such as "fixed" or "crash" to identify fault fixing commits. From each fault fixing commit, we extract the list of files that were changed to fix the fault. This list of files tells us which classes where changed to fix the faults.

4 Study Definition and Design

The *goal* of our study is to analyse the existence, the evolution and the impact of the static relationships between anti-patterns and design patterns in software systems. The *quality focus* is

	ArgoUML	JFreeChart	XercesJ
# of classes	3,325	1,615	1,191
# of AntiSingleton	3	38	24
# of Blob	100	49	12
# of CDSP	51	3	6
# of ComplexClass	158	52	7
# of LongMethod	336	75	7
# of LongParameterList	281	76	4
# of MessageChains	162	59	8
# of RefusedParentBequest	123	5	7
# of SpaghettiCode	1	2	6
# of SpeculativeGenerality	22	3	29
# of SwissArmyKnife	13	26	29

Table 1: Descriptive statistics of the object systems (CDSP: ClassDataShouldBePrivate)

related to the quality of systems and to the evolution of classes participating in anti-patterns. The *perspective* is detecting the impact of relationships between anti-patterns and design patterns on class change-proneness and fault-proneness. The *context* of our experiment is three open-source Java programs: ArgoUML, JFreeChart, and XercesJ.

4.1 System Under Analysis

We apply our approach on three Java systems: ArgoUML[1], JFreeChart[2], and XercesJ[3]. These systems can be classified as large, medium, and small systems, respectively. We use these systems because they are open source, have been used in previous work, are of different domains, span several years and versions, and have between hundreds and thousands of classes. Table 1 summarises some statistics about these systems.

ArgoUML is an UML diagramming system written in Java and released under the open-source BSD License. For anti-patterns dependencies analysis, we extracted a total number of 4480 snapshots from release 0.26 to release 0.34, in the time interval between September 27th, 2008 and December 15th, 2011.

JFreeChart is a Java open-source framework to create charts. For macro co-change analysis, we considered an interval of observation ranging from June 15th, 2007 (release 1.0.6) to November 20th, 2009 (release 1.0.13 ALPHA). In such interval we extracted 2010 snapshots.

XercesJ is a collection of software libraries for parsing, validating, serialising, and manipulating XML. It is developed in Java and managed by the Apache Foundation. For anti-patterns dependencies analysis, we extracted a total number of 159196 snapshots from release 1.0.4 to release 2.9.0, in the time interval between October 14th, 2003 and November 23th, 2006.

[1] http://argouml.tigris.org/
[2] http://www.jfree.org/
[3] http://xerces.apache.org/xerces-j/

4.2 Research Questions

We break down our study into three steps: first, we perform a preliminary study to verify the existence of relationships between anti-pattern and design pattern classes. Next, we study the evolution of these relationships. Finally, we analyse the fault proneness and the change proneness of classes containing anti-patterns and related to design patterns:

- **RQ1:** *Are there static relationships between anti-patterns and design patterns?*
- **RQ2:** *Are the static relationships between anti-patterns and design patterns casual?*
- **RQ3:** *Do static relationships between anti-patterns and design patterns impact change proneness and fault proneness?*

In the first question, we check if static relationships between anti-patterns classes and classes playing roles in design patterns exists. **RQ1:** is a preliminary study and its results present an initial evidence for the other research questions.

In the second question, we verify if these relationships are persistent during the evolution of the software systems.

The third research question is designed to check whether classes participating in such relationships are more change-prone and fault prone than other anti-pattern and design patterns classes.

To answer the third research question, we test the following null hypotheses:

- H_{1_0}: The proportion of classes changed at least once between two releases is not different between anti-pattern classes participating or not in a static relationship with at least one design pattern.
- H_{2_0}: the proportion of classes undergoing at least one fault-fixing change between two releases does not differ between anti-pattern classes participating or not in a static relationship with at least one design pattern.

If we reject the null hypothesis H_{1_0}, we explain the rejection as:

- H_{1_1}: The proportion of classes changed at least once between two releases is different between anti-pattern classes participating or not in static relationship with at least one design pattern.

If we reject the null hypothesis H_{2_0}, we explain the rejection as:

- H_{2_1}: The proportions of faults carried by anti-patterns classes which have static relationships with design patterns and faults carried by other anti-pattern classes are not the same.

Then, we check whether anti-patterns have effects transitive via static dependencies, *i.e.* we are attempting to compare the fault proneness and change proneness across the following groups of classes: (1) classes belonging to a design pattern and related to an anti-pattern and (2) classes belonging to a design pattern and not related to an anti-pattern.

We test the following two null hypotheses:

- H_{3_0}: The proportion of classes changed at least once between two releases is not different between design pattern classes participating or not in a static relationship with at least one anti-pattern.

- H_{4_0}: the proportion of classes undergoing at least one fault-fixing change between two releases does not differ between design pattern classes participating or not in a static relationship with at least one anti-pattern.

If we reject the null hypothesis H_{3_0}, we explain the rejection as:

- H_{3_1}: The proportion of classes changed at least once between two releases is different between design pattern classes participating or not in static relationship with at least one anti-pattern.

If we reject the null hypothesis H_{4_0}, we explain the rejection as:

- H_{4_1}: The proportions of faults carried by anti-patterns classes which have static relationships with design patterns and faults carried by other anti-pattern classes are not the same.

4.3 Analysis Method

The analysis reported in Section 5 have been performed using the R statistical environment[4]. We use the contingency tables to assess the direction of the difference of fault proneness and change proneness across different group of classes. In statistics, a contingency table is a table in a matrix format that displays the frequency distribution of the variables. We use Fisher's exact test [She07], to check whether the difference is significative between anti-pattern classes having static relationships with design patterns and other anti-pattern classes in term of change proneness and fault proneness. Fisher's exact test is a statistical significance test used in the analysis of contingency tables. Although in practice it is employed when sample sizes are small, it is valid for all sample sizes. The test is useful for categorical data that result from classifying objects in two different ways. It is used to examine the significance of the association (contingency) between the two classifications. We also compute the Odds ratio [She07] that indicates the likelihood for an event to occur. The Odds ratio (OR) is defined as the ratio of the odds p of an event occurring in one sample, *e.g.* the odds that anti-patterns having static relationships with design patterns are identified as fault-prone, to the odds q of the same event occurring in the other sample, *i.e.* the odds that the rest of anti-pattern classes are identified as fault-prone. An odds ratio greater than 1 indicates that the event (*i.e.* a fault) is more likely in the first sample (*i.e.* anti-patterns having static relationships with design patterns), while an odds ratio less than 1 indicates that it is more likely in the second sample. The Odds ratio is calculated as follows: $OR = \frac{p/(1-p)}{q/(1-q)}$.

[4] http://www.r-project.org

5 Study Results

We now present the results of our empirical study. We use data collected from the three programs and some external sources (*e.g.* bug reports) to answer our three research questions and to discuss typical examples of our findings as follows:

RQ1: Are there static relationships between anti-patterns and design patterns?

Yes Table 2 shows that, for the majority of anti-patterns, we detect static relationships with design patterns. On the one hand, we notice that different anti-patterns can have different proportions of static relationships with design patterns. This observation is not surprising because these systems have been developed in three unrelated contexts, under different processes. Thus, they have different complexities and different usages of design motifs. Consequently, they have different number of static relationships among their design motifs. On the second hand, the design pattern that has the most relationships with anti-patterns is the Command design pattern. For example, we noted that 50% of static relationships among SpeculativeGenerality and design patterns in ArgoUML, are with the Command design pattern. In XercesJ, we observe that 41% of relationships among ClassDataShouldBePrivate was with the Command design pattern.

No clear tendency exists for ComplexClass and RefusedParentBequest. For example, ComplexClasses have static relationships with six analysed design patterns with equivalent proportions in ArgoUML, JFreeChart, and XercesJ.

But.. In the three systems, if a class participates in a design pattern, it does not have a relationship with the SpaghettiCode anti-pattern, as shown in Table 2. In fact, in all three systems, we do not detect any class playing a role in a SpaghettiCode and having static dependencies (use, association, aggregation, and composition relationships) with one of the six design patterns (Command, Composite, Decorator, FactoryMethod, Prototype, and Observer). We discuss these observations in details in 6.

Relevance Design patterns are naturally geared to improve adaptability and maintainability. Each design pattern aims to make specific changes easier [JY01]. For example, in XercesJ, the class `org.apache.xerces.validators.common.XMLValidator` is an excessively complex class interface. The developer attempts to provide for all possible uses of this class. In her attempt, she adds a large number of interface signatures to meet all possible needs. The developer may not have a clear abstraction or purpose for `org.apache.xerces.validators.common.XMLValidator`, which is represented by the lack of focus in its interface. Thus, we claim that this class belongs to a SwissArmyKnife anti-pattern. This anti-pattern is problematic because the complicated interface is difficult for other developers to understand and obscures how the class is intended to be used, even in simple cases. Other consequences of this complexity include the difficulties of debugging, documentation, and maintenance. We detect that this class has a use-relationship with the class `org.apache.xerces.validators.dtd.DTDImporter`, which belongs to the Command design pattern. Using Command classes makes it easier to delegate method calls without knowing the owner

Table 2: Proportion of the relationships between anti-patterns and design patterns, in each first version of our analysed systems (SR: Static relationship among anti-patterns and design patterns)

Anti-patterns	Systems	# of SR
AntiSingleton	ArgoUml	68
	JFreeChart	92
	XercesJ	83
Blob	ArgoUml	161
	JFreeChart	72
	XercesJ	42
ClassDataShouldBePrivate	ArgoUml	83
	JFreeChart	31
	XercesJ	44
ComplexClass	ArgoUml	182
	JFreeChart	84
	XercesJ	66
LongMethod	ArgoUml	212
	JFreeChart	290
	XercesJ	142
LongParameterList	ArgoUml	290
	JFreeChart	188
	XercesJ	204
MessageChains	ArgoUml	192
	JFreeChart	94
	XercesJ	77
RefusedParentBequest	ArgoUml	146
	JFreeChart	72
	XercesJ	48
SpaghettiCode	ArgoUml	0
	JFreeChart	0
	XercesJ	0
SpeculativeGenerality	ArgoUml	20
	JFreeChart	34
	XercesJ	67
SwissArmyKnife	ArgoUml	35
	JFreeChart	84
	XercesJ	86

of the method or the method parameters. Thus, developer can correct `org.apache.xerces.validators.common.XMLValidator`, by using the related Command pattern, to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method, and values for the method parameters. Thus, by using the relationships of an anti-pattern with a specific design pattern, we explain how developers maintained the anti-pattern classes while reducing its influence on the system. An external information from the changelog file support this idea (see Section 6).

In conclusion, we observed as an initial evidence, that the majority of anti-pattern analysed share static relationships with some design patterns.

RQ2: Are the static relationships between anti-patterns and design patterns casual?

No In **RQ2**, we observed that static relationships between anti-patterns and designs patterns exist in all versions of our analysed systems. Figure 1 illustrates that static relationships are continuously growing. Indeed, each new version contains more design patterns, more anti-patterns (which was observed in previous work in [ACC+07] and in [KPGA12]) and also contains more static relationships between these motifs.

But. we observe that even if the static relationships continue to exist between anti-patterns and design patterns (detected in the first version), the classes playing roles in anti-patterns are refactored, restructured and fixed in future versions. Anti-patterns are removed from the majority of anti-patterns classes that had a static relationship with a design pattern.

Figure 2 illustrates that, in the three analysed systems, static relationships detected in the first version disappear with the realization of each new release, following refactoring operations and maintenance tasks. This fact does not mean that such static relationships are casual. First, a class that belong to an anti-pattern *X* in the release *S* could appear as a part of an anti-pattern *Y* in the next release.

As a concrete example, we observe that the class `.DocumentTypeImpl.java` was detected in the version 1.0.4 of XercesJ as *Blob* class. In the next version, this class was changed and detected as a part of another anti-pattern, *LongParameterList*.

Second, the fact that these dependencies have short life-spans can explain the main motive of introducing such relationships by the developers. Indeed, in cases where the anti-pattern class is corrected (*i.e.*, the class is not part of an anti-pattern anymore), this may suggest that developers use design patterns as 'temporary fix' for the negative impact of the anti-patterns, and later on refactoring such anti-patterns to remove design smells.

Thus, future work includes conducting an empirical study to analyse the evolution of anti-pattern classes across the different release of each system to explain how an anti-pattern class could evolve to be a "normal" class or to belong to another anti-pattern.

Indeed, classes participating in anti-patterns are more subject to changes impacting their structure and, thus, possibly their states (*i.e.* the anti-pattern can be removed). This observation confirm, also, previous results reported in [KPGA12]. In fact, while studying the relation between kinds of anti-patterns and change- and fault-proneness, Khomh *et al.* also studied the kinds of changes impacting classes participating in anti-patterns. They reported that structural changes

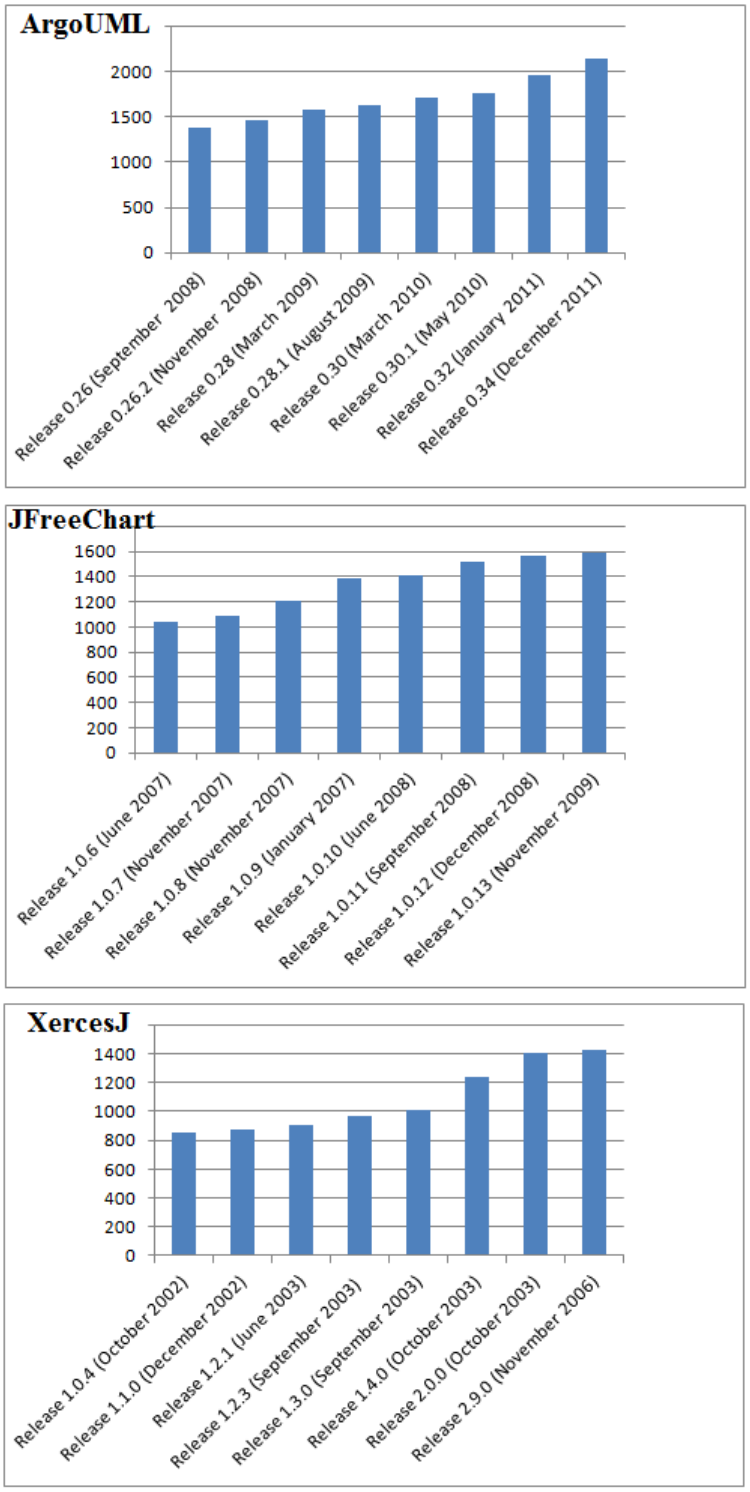

Figure 1: Evolution of the number of static relationships between anti-patterns and designs patterns.

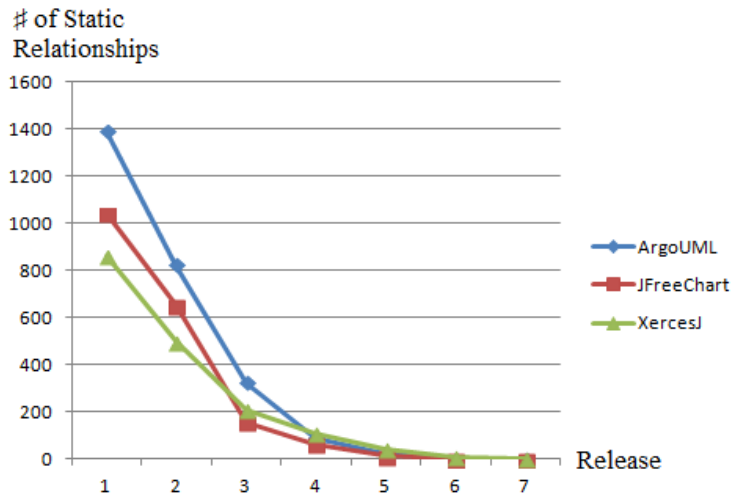

Figure 2: The persistence of static relationships detected in the first software version between anti-patterns and designs patterns.

occur more often on classes participating to anti-patterns than other changes.

Relevance The development and maintenance of a system include repairing anti-pattern classes. Indeed, A common perception of maintenance is that it merely involves fixing defects as anti-patterns.

We detected that changes performed on anti-pattern classes are usually motivated by the removal of code smells using some good recognized solution such as design patterns. For example, we detect seven LongMethod anti-patterns in the first analysed version of XercesJ. These anti-patterns were corrected in following versions through refactoring of the source code using design patterns such as Strategy pattern which encapsulates alternative strategies, or approaches, in separate classes so that each implements a common operation. Developers also used such design pattern to transform LongMethod anti-patterns into a new form that behaves the same as before but that no longer "smells". In fact, for a long routine, one or more smaller subroutines can be extracted, encapsulated separately from the validating object without code duplication. For duplicated routines, the duplication can be removed and replaced with one shared function.

There are two general categories of benefits of using design patterns to fix anti-pattern classes during program evolution: (1) improving the maintainability because the source code become easier to read and to understand, and (2) improve the extensibility because it is easier to extend defective classes if developers use recognizable design patterns which provide some flexibility where none may have existed before.

We conclude that static relationships between anti-patterns and designs patterns exist in all versions of our analysed systems, but they are temporary.

Table 3: Change-proneness Odd ratios of the Fisher's exact test for anti-pattern classes related to design patterns

ArgoUML		JFreeChart		XercesJ	
Version	OR	Version	OR	Version	OR
0.26	5.16	1.0.6	4.66	1.0.4	5.49
0.26.2	2.48	1.0.7	2.3	1.1.0	4.2
0.28	2.12	1.0.8	1.64	1.2.1	4.38
0.28.1	0.86	1.0.9	2.86	1.2.3	2.86
0.30	1.44	1.0.10	1.88	1.3.0	3.4
0.30.1	1.94	1.0.11	1.62	1.4.0	3.26
0.32	1.63	1.0.12	6.48	2.0.0	2.44
0.34	2.86	1.0.13	3.28	2.9.0	2.86

RQ3: Do static relationships between anti-patterns and design patterns impact change proneness and fault proneness?

Table 3 summarises ORs obtained when testing H_{1_0}. Each row shows, for each system, a version number and the ORs of classes (1) participating in at least one anti-pattern, (2) having static relationship with a design pattern, and (3) exhibiting at least one change before the next version. Table 4 summarises ORs obtained when testing H_{2_0}. Each row shows, for each system, a version number and the ORs of classes (1) participating in at least one anti-pattern, (2) having static relationship with a design pattern, and (3) exhibiting at least one fault before the next version.

Table 5 summarises ORs obtained when testing H_{3_0}. Each row shows, for each system, a version number and the ORs of classes (1) participating in at least one design pattern, (2) having static relationship with an anti-pattern, and (3) exhibiting at least one change before the next version. Table 6 summarises ORs obtained when testing H_{4_0}. Each row shows, for each system, a version number and the ORs of classes (1) participating in at least one design pattern, (2) having static relationship with an anti-pattern, and (3) exhibiting at least one fault before the next version.

In all releases, except ArgoUML 28.1, the Fisher's exact test indicates a significant difference of proportions of changes among classes participating and not participating in a relationships between anti-patterns and design patterns. Odds ratios vary across systems and, within each system, across versions. While in few cases, ORs are close to 1 (*i.e.* the odds is even that a class participating in a static relationship between an anti-pattern and a design patterns changes or not), in some pairs of systems/versions, such as ArgoUML 0.26, JFreeChart 1.0.12, or XercesJ 1.0.4, ORs are greater than 5. Overall, ORs for ArgoUML are lower than those of other systems, by one or two orders of magnitude. The odd ratios of classes participating in static relationships between anti-patterns and design patterns are, in most cases, higher than those of other classes with anti-patterns and design pattern.

We therefore conclude that, in most cases, there is a relation between the involvement in a static relationships between anti-pattern and design patterns, and change-proneness: a greater proportion of classes participating in these relationships change with respect to other anti-pattern

Table 4: Fault-proneness odd ratios of the Fisher's exact test for anti-pattern classes related to design patterns

ArgoUML		JFreeChart		XercesJ	
Version	OR	Version	OR	Version	OR
0.26	0.42	1.0.6	0.49	1.0.4	0.56
0.26.2	0.89	1.0.7	0.63	1.1.0	0.49
0.28	0.67	1.0.8	0.96	1.2.1	0.46
0.28.1	0.94	1.0.9	0.69	1.2.3	0.48
0.30	0.23	1.0.10	0.43	1.3.0	0.39
0.30.1	0.44	1.0.11	0.29	1.4.0	0.52
0.32	0.29	1.0.12	0.37	2.0.0	0.90
0.34	0.76	1.0.13	0.46	2.9.0	0.44

or design patterns classes. The rejection of H_{1_0} and H_{1_0}, and the ORs provide a posteriori concrete evidence of the impact of static relationships between anti-patterns and design patterns on change-proneness. In fact, we observe that classes involved in relationships between anti-pattern and design patterns are more change prone. We believe that this observation is an effect of change propagated through these motifs and that blends in perfectly with previous results on motif evolution stated (1) that anti-pattern classes are in general more change prone than other classes [KPGA12] and (2) that design patterns are in general more change prone than other classes [GCH09].

For fault proneness, in all systems, Fishers exact test indicates that class participating in relationships between anti-patterns and design patterns are less fault-prone than other anti-pattern classes, but more fault-prone than other design pattern classes. Odds ratios vary again across systems and, within each system, across versions. We notice that in few cases, ORs are close to 1. However, in most cases, the odd ratios of classes participating in relationships between design patterns and anti-patterns are lower than other anti-pattern classes and higher than other design pattern classes. The rejection of H_{RQ2_0} and H_{RQ4_0},and the ORs provide a posteriori concrete evidence of the impact of static relationships between anti-patterns and design patterns on fault-proneness.

Results suggest that being involved in a static relationship between design pattern and anti-pattern has an impact on the possibility of occurring faults and change in the class.

6 Discussion

This section discusses the results reported in Section 5 as well as the threats to their validity.

6.1 Observations

From Table 2, we note that many anti-patterns in ArgoUML, JFreechart, and XercesJ have relationships with design patterns. To the best of our knowledge, we are the first to report these relationships, thanks to our use of state-of-the-art detection algorithms, which detects occurrences

Table 5: Change-proneness odd ratios of the Fisher's exact test for design pattern classes related to anti-patterns

ArgoUML		JFreeChart		XercesJ	
Version	OR	Version	OR	Version	OR
0.26	3.22	1.0.6	2.12	1.0.4	2.83
0.26.2	3.40	1.0.7	2.18	1.1.0	3.5
0.28	3.18	1.0.8	2.14	1.2.1	2.6
0.28.1	2.61	1.0.9	1.96	1.2.3	2.46
0.30	3.02	1.0.10	1.38	1.3.0	3.2
0.30.1	2.69	1.0.11	1.42	1.4.0	1.99
0.32	1.87	1.0.12	3.23	2.0.0	1.86
0.34	2.36	1.0.13	1.89	2.9.0	1.25

Table 6: Fault-proneness odd ratios of the Fisher's exact test for design pattern classes related to anti-patterns

ArgoUML		JFreeChart		XercesJ	
Version	OR	Version	OR	Version	OR
0.26	2.21	1.0.6	1.06	1.0.4	1.46
0.26.2	1.63	1.0.7	1.82	1.1.0	1.02
0.28	1.46	1.0.8	1.19	1.2.1	2.79
0.28.1	2.03	1.0.9	1.11	1.2.3	1.59
0.30	1.56	1.0.10	1.84	1.3.0	1.47
0.30.1	1.89	1.0.11	2.06	1.4.0	1.69
0.32	1.77	1.0.12	2.12	2.0.0	1.88
0.34	2.36	1.0.13	2.59	2.9.0	1.63

of 11 anti-patterns and six design patterns. Moreover, we do not consider that an anti-pattern is necessarily the result of a "bad" implementation or design choice; only the concerned developers can make such a judgement. We do not exclude that, in a particular context, an anti-pattern can be practical ways to implement or design a (part of a) class. For example, automatically-generated parsers are often very large and complex classes. Only developers can evaluate their impact according to the context: it can be perfectly sensible to have these large and complex classes if they come from a well-defined grammar.

We observe, also, that the number of static relationships among anti-pattern classes and design patterns classes increases over time, following the increase of the complexity of each new version, although classes playing roles in these relationships are maintained, fixed and refactored.

6.2 Potential Explanations

The majority of static relationships among anti-patterns and design patterns comes from the Command pattern. This design pattern is implemented as a motif in which an object is used

to present and encapsulate all the information needed to call a method at a later time. Thus, developers use this design pattern, possibly unintentionally, when there is a proliferation of similar methods and the user-interface code becomes difficult to maintain. This characteristic can explain, predominately, the static relationships of this design pattern with the following anti-patterns:

- ClassDataShouldBePrivate because commands must access the data of other objects to function, and developers may have used public instance variables to allow this access;
- LongMethod and LongParameterList because commands must access the functionalities provided by other classes, which typically can perform lots of processing, in long methods and-or with long parameter lists;
- SpeculativeGenerality because classes in relation to commands may have been engineered with extension in mind, but the command does not use it.

SpaghettiCodes have no static relationships (use, association, aggregation, and composition) with design patterns. This observation is not surprising because a SpaghettiCode is revealed by classes with no structure, declaring long methods with no parameters, and using global variables for processing. A SpaghettiCode does not take the advantage of object-orientation mechanisms: polymorphism and inheritance. Many object methods have no parameters, and utilize class or global variables for processing. Thus, a SpaghettiCode is difficult to reuse and to maintain, and when it is, it is often through cloning. In many cases, however, code is never considered for reuse. The findings of our analysis indicate that no relation is detected between the different occurrence of SpaghettiCode anti-pattern and design patterns. However, it could be possible that they have no relations because they constitute DeadCode. Dead code means unnecessary, inoperative code that can be removed. It is a code in the program which can never be executed or a code that is executed but has no effect on the output of a program [KRS94]. Dead code analysis can be performed using live variable analysis, a form of static code analysis and data flow analysis [CGK98]. However, in large programming projects, it is sometimes difficult to recognize and eliminate dead code [DP98].

Lanza *et al.* [LD02] presented an evolution matrix to display the evolution of the files of a program. The authors presented a categorisation of files based on the visualisation of different versions of a file. They reported that an idle file does not change over several versions and that can be explained by the fact that such file can present a dead code or a good design. Thus, we decide to use this observation by mining version-control systems (Concurrent Versions System named CVS[5] and Apache Subversion System named SVN[6]), to identify, for example, which SpaghettiCode class were never changed after their introduction in the analyzed systems. We noted for example, that in ArgoUML, more than 80% of classes were maintained three times at most. On the other hand, less than 1% of classes were maintained 50 times at least. Based on change analysis, it is neither possible to conclude that SpaghettiCode classes have no relations with design patterns because they constitute DeadCode nor is the opposite true. Indeed, from

[5] http://cvs.nongnu.org/
[6] http://subversion.apache.org/

the results of this case study it is impossible to definitely exclude the possibility that there is in fact no statistically relevant correlation between SpaghettiCode and DeadCode. However, it could be true that the spaghetti code classes have no dependencies because of the lower number of instances in the analysed systems (9 instances).

6.3 Outcome

On the one hand, the notion of static dependency can be used to assess the architecture of a software system. In fact, we can assume that systems with more static relationships among design patterns and anti-patterns are more stable, since these relationships can be explained by developers making use of recognized and stable solutions (design patterns) to refactor and correct anti-patterns. For example, by mining software version-control systems, we found that the design pattern Command described in Section 5 and containing the class `org.apache.xerces.validators.dtd.DTDImporter` was created by the developer *jeffreyr* on 2000-04-04 15:38:39, to *Factoring Validators code* implemented in `org.apache.xerces.validators.common.XMLValidator`. From this result, we are working on combining static dependency with other quality attributes to improve the assessment of the code quality.

On the other hand, in this paper we study correlations among collocated anti-patterns and design patterns because there might be an interaction effect that could explain the existence of such motifs. In fact, our results show that the presence of some anti-patterns (*LongMethod*, *LongParameterList*, etc.) may increase the likelihood of the presence of a specific design pattern (Command design pattern). While, the presence of spaghetti code do not have any direct correlation with the presence of design patterns.

Moreover, it would be desirable to use anti-patterns and their static relationships, other than metrics, to build more accurate/informative change- and fault-prediction methods. Last, but not least, further investigation, devoted to mine change logs, mailing lists and issue reports, is desirable to seek evidence of cause effect relationships between the presence of anti-patterns static relationships, or the need to remove and refactor design defects, and the class change- and fault-proneness.

6.4 Threats to Validity

We now discuss in details the threats to the validity of our results, following the guidelines provided in [Yin02].

Internal validity, in our context, they are mainly due to errors introduced in measurements. We are aware that the detection techniques used include some subjective understanding of the definitions of anti-patterns and design patterns. However, as discussed, we are interested in relating anti-patterns "as they are defined in DECOR" [MGDL10] with design patterns "as they are defined in DeMIMA" [GA08]. For this reason, the precision of the anti-patterns and design patterns detection is a concern that we agree to accept. However, threshold-based detection techniques such as DECOR do not handle the uncertainty of the detection results and, therefore, miss borderline classes, *i.e.*, classes with characteristics of anti-patterns surfacing slightly above or sinking slightly below the thresholds because of minor variations in the characteristics of these classes. We also notice that Sylversky method relies on heuristics so it can presents errors when

extracting faults. However, this method has been used successfully in previous studies.

Reliability validity threats concern the possibility of replicating this study. We attempted here to provide all the necessary details to replicate our study. Moreover, both ArgoUML, JFreeChart, and XercesJ source code repositories are available. Finally, the data sets on which we computed our statistics are available on the Web[7].

Threats to *external validity* concern the possibility to generalise our observations. First, although we performed our study on three different, real systems belonging to different domains and with different sizes and histories, we cannot assert that our results and observations are generalisable to other systems and the facts that all the analysed systems are in Java and open-source may reduce this generability. Second, we used particular, yet representative, sets of anti-patterns and design patterns. Different anti-patterns and design patterns could have lead to different results, which are part of our future work.

7 Conclusions and Future Work

In this paper, we provide empirical evidence of the relationships between anti-patterns and design patterns. We show that some anti-patterns are significantly more likely to have relationships with design patterns than other. This study raises a question, within the limits of the threats to its validity, about the conjecture in the literature that anti-patterns and design patterns have an impact on system quality. We provide a basis for future research to understand more precisely the causes and the consequences of the relationships between anti-patterns and design patterns, *i.e.* if developers use design patterns to encapsulate anti-patterns. The advantages of knowing these relations are (1) spotting how developers strive to maintain a system containing anti-patterns by using design patterns and (2) detecting correlations among collocated anti-patterns and design patterns to identify the causes of the co-existence of such motifs.

This empirical study provided, within the limits of its validity, evidence that classes participating in static relationships between anti-patterns and design patterns are more change-prone and less fault-prone than classes not participating in such relationships. The study also provided evidence to practitioners that they should pay attention to systems with a high number of classes participating in such relationships, because these classes are more likely to be the subject of their change efforts. More specifically, managers and developers can use these results to guide maintenance activities: for example, they can recommend their developers to use Command design pattern as a temporary fix (when necessary) for some anti-patterns, since our results show that it can reduce the fault-proneness of the anti-patterns.

Future work includes (i) replicating our study on other systems to assess the generality of our results and (ii) analysing change propagation among anti-patterns and design patterns related by static relationships.

We are also interested in studying the extend to which similar systems exhibit similar likelihood of having anti-patterns, and how they relate to systems' change-/fault-proneness. We also plan to study the categorisation of classes as change-prone, error-prone, or none, and compute Types I and II errors to assess whether anti-patterns perform better than complexity metrics.

[7] http://www.ptidej.net/download/experiments/eceasst13/

Acknowledgements This work has been partly funded by a FQRNT Team grant, the Canada Research Chair in Software Patterns and Patterns of Software, and the Tunisian Ministry of Higher Education and Scientific Research.

Bibliography

[ACC+07] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, M. Di Penta. An empirical study on the evolution of design patterns. In *proceedings of ESEC-FSE '07*. Pp. 385–394. ACM Press, New York, NY, USA, 2007.

[AKGA11] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*. Pp. 181–190. IEEE Computer Society, Washington, DC, USA, 2011.

[BMMM98] W. Brown, H. McCormick, T. Mowbray, R. Malveau. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley Press, 1998.

[BSW+03] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, R. T. Alexander. Design Patterns and Change Proneness: An Examination of Five Evolving Systems. In *Proceedings of the 9th International Symposium on Software Metrics*. Pp. 40–50. IEEE Computer Society, Washington, DC, USA, 2003.

[CGK98] Y.-F. Chen, E. R. Gansner, E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. *IEEE Transaction Software Engineering* 24(9):682–694, Sept. 1998.

[CM10] A. Chatzigeorgiou, A. Manakos. Investigating the Evolution of Bad Smells in Object-Oriented Code. In *Proceedings of the Seventh International Conference on the Quality of Information and Communications Technology*. Pp. 106–115. IEEE Computer Society, Washington, DC, USA, 2010.

[DP98] F. Damiani, F. Prost. Detecting and Removing Dead-Code using Rank 2 Intersection. In *Selected papers from the International Workshop on Types for Proofs and Programs*. Pp. 10–32. Springer-Verlag, London, UK, UK, 1998.

[DSRS03] I. Deligiannis, M. Shepperd, M. Roumeliotis, I. Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal System Software* 65(2):127–139, Feb. 2003.

[EM02] E. V. Emden, L. Moonen. Java Quality Assurance by Detecting Code Smells. In *in Proceedings of the 9th Working Conference on Reverse Engineering. IEEE Computer*. Pp. 97–107. Society Press, 2002.

[GA04] Y.-G. Guéhéneuc, H. Albin-Amiot. Recovering Binary Class Relationships: Putting Icing on the UML Cake. In *Proceedings of the* 19^{th} *Conference on Object-Oriented*

Programming, Systems, Languages, and Applications (OOPSLA). Pp. 301–314. ACM Press, 2004.

[GA08] Y.-G. Guéhéneuc, G. Antoniol. DeMIMA: A Multi-layered Framework for Design Pattern Identification. *Transactions on Software Engineering (TSE)*, pp. 667–684, 2008.

[GCH09] M. Gatrell, S. Counsell, T. Hall. Design Patterns and Change Proneness: A Replication Using Proprietary C sharp Software. *Reverse Engineering, Working Conference on*, pp. 160–164, 2009.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, first edition, 1994.

[HGHG12] S. Hassaine, Y.-G. Guéhéneuc, S. Hamel, A. Giuliano. ADvISE: Architectural Decay In Software Evolution. In *16th European Conference on Software Maintenance and Reengineering*. Pp. 267–276. ACM, 2012.

[Iac11] C. Iacob. A design pattern mining method for interaction design. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '11, pp. 217–222. ACM, 2011.

[JGHA11] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, G. Antoniol. An Exploratory Study of Macro Co-changes. In *Working Conference on Reverse Engineering (WCRE)*. Pp. 325–334. 2011.

[JY01] D. Jain, H. J. Yang. OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study. In *Proceedings of the IEEE International Conference on Software Maintenance*. Pp. 580–590. IEEE Computer Society, 2001.

[KP96] C. Kramer, L. Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proceeding of the 3rd working conference on reverse engineering*. Pp. 208–215. IEEE Computer Society Press, 1996.

[KPGA12] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*. 17(3):243–275, June 2012.

[KRS94] J. Knoop, O. Rüthing, B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. Pp. 147–158. ACM, 1994.

[LD02] M. Lanza, S. Ducasse. Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics. In *In Proceedings of LMO 2002 (Langages et Modeles Objets)*. Pp. 135–149. Lavoisier, 2002.

[LM06] M. Lanza, R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[MGDL10] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *Transactions on Software Engineering (TSE)* 36(1):20–36, 2010.

[OCBZ09] S. Olbrich, D. S. Cruzes, V. Basili, N. 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*. Pp. 390–400. IEEE Computer Society, Washington, DC, USA, 2009.

[PW06] B. Pietrzak, B. Walter. Leveraging code smell detection with inter-smell relations. *Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering*, pp. 75–84, 2006.

[RDGM04] D. Ratiu, S. Ducasse, T. Gîrba, R. Marinescu. Using History Information to Improve Design Flaws Detection. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering*. Pp. 223–233. IEEE Computer Society, 2004.

[Rie96] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[RV96] R. Real, J. Vargas. The probabilistic basis of Jaccard's index of similarity. *Systematic Biology* 45(3):380–385, 1996.

[SCF12] D. Settas, A. Cerone, S. Fenz. Enhancing ontology-based antipattern detection using Bayesian networks. *Expert Systems with Applications*, pp. 9041–9053, 2012.

[She07] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2007.

[SN13] A. E. H. Seyyed Ehsan Taba, Foutse Khomh Ying Zou, M. Nagappan. Predicting Bugs Using Antipatterns. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*. Pp. 122–131. IEEE Computer Society, Washington, DC, USA, 2013.

[SZZ05] J. Sliwerski, T. Zimmermann, A. Zeller. When do changes induce fixes? *SIGSOFT Software Engineering Notes*, pp. 1–5, 2005.

[TCSH06] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. Halkidis. Design Pattern Detection Using Similarity Scoring. *Transactions on Software Engineering* 32(11):896–909, 2006.

[Vok04] M. Vokac. Defect Frequency and Design Patterns: An Empirical Study of Industrial Code. *IEEE Tranaction Software Engineering* 30(12):904–917, December 2004.

[Web95] B. F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, first edition, February 1995.

[Yin02] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.

[YM12] A. F. Yamashita, L. Moonen. Do code smells reflect important maintainability aspects? In *Proceedings of the IEEE International Conference on Software Maintenance*. Pp. 306–315. IEEE Computer Society Press, Washington, DC, USA, 2012.

[YM13] A. Yamashita, L. Moonen. Exploring the impact of inter-smell relations on software maintainability: an empirical study. In *Proceedings of the International Conference on Software Engineering*. Pp. 682–691. IEEE Press, Piscataway, NJ, USA, 2013.