Delft University of Technology
Software Engineering Research Group
Technical Report Series

Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes

Daniele Romano, Paulius Raila, Martin Pinzger and Foutse Khomh

Report TUD-SERG-2012-017

TUD-SERG-2012-017

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Proceedings of the 19th Working Conference on Reverse Engineering, 2012.

Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes

Daniele Romano,[1] Paulius Raila,[1] Martin Pinzger,[1] Foutse Khomh[2]
[1] *Software Engineering Research Group, Delft University of Technology, The Netherlands*
[2] *Dept. of Elec. and Comp. Engineering, Queen's University, Kingston, Ontario, Canada*
daniele.romano@tudelft.nl, p.raila@student.tudelft.nl, m.pinzger@tudelft.nl, foutse.khomh@queensu.ca

***Abstract*—Antipatterns are *poor* solutions to design and implementation problems which are claimed to make object oriented systems hard to maintain. Our recent studies showed that classes with antipatterns change more frequently than classes without antipatterns. In this paper, we detail these analyses by taking into account fine-grained source code changes (SCC) extracted from 16 Java open source systems. In particular we investigate: whether classes with antipatterns are more change-prone (in terms of SCC) than classes without; (2) whether the *type of antipattern* impacts the change-proneness of Java classes; and (3) whether certain *types of changes* are performed more frequently in classes affected by a certain antipattern.**

Our results show that: 1) the number of SCC performed in classes affected by antipatterns is statistically greater than the number of SCC performed in classes with no antipattern; 2) classes participating in the three antipatterns *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife* are more change-prone than classes affected by other antipatterns; and 3) certain types of changes are more likely to be performed in classes affected by certain antipatterns, such as *API* changes are likely to be performed in classes affected by the *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife* antipatterns.

***Keywords*-Antipatterns; change-proneness; fine-grained source code changes; empirical software engineering**

I. Introduction

Over the past two decades, maintenance costs have grown to more than 50% and up to 90% of the overall costs of software systems [1]. To help reduce the cost of maintenance, researchers have proposed several approaches to ease program comprehension, and identify change- and bug-prone parts of the source code of software systems. These approaches include source code metrics (*e.g.*, [2], [3]) and heuristics to assess the design of a software system (*e.g.*, [4], [5], [6]).

Recently, we have started on analyzing the impact of antipatterns on the change-proneness of software units [5]. Antipatterns [7] are "poor" solutions to design and implementation problems. In contrast to design patterns [8] which are "good" solutions to recurring design problems. Antipatterns are typically introduced in software systems by developers lacking the adequate knowledge or experience in solving a particular problem or having misapplied some design patterns. Coplien [9] described an antipattern as "something that looks like a good idea, but which backfires badly when applied". Previous studies, such as ours [5], support this description by showing that software units, *i.e.*, classes, affected by antipatterns are more likely to undergo changes than other units.

Existing literature proposes many different antipatterns, such as the 40 antipatterns described by Brown *et al.* [7]. Furthermore, antipatterns occur in large numbers and affect large portions of some software systems. For instance, we found that more than 45% of the classes in the systems studied in [5] contained at least one antipattern. Because of the diversity and the large number of antipatterns, support is needed, for instance by software engineers, to identify the *risky* classes affected by antipatters that lead to errors and increase development and maintenance costs. For this, we need to obtain a deeper understanding of the change-proneness of different antipatterns and the types of changes occurring in classes affected by them. Providing this deeper understanding is the main objective of this paper.

In this paper 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.*, [10], [5]), the approach of our study is based on the analysis of fine-grained source code changes (*SCC*) mined from version control systems [11], [12]. This approach allows us to analyze the *types of changes* performed in classes affected by a particular antipattern which was not possible with our previous approach. Moreover, we take into account the significance of the change types [13] and we filter out irrelevant change types (*e.g.*, changes to comments and copyrights), that account for more than 10% of all changes in our dataset.

Using the data of fine-grained source code changes and antipatterns, we aim at providing answers to the following three research questions:

- **RQ1:** Are Java classes affected by antipatterns more change-prone than Java classes not affected by any antipattern?
  This research question is aimed at replicating our previous study [5] with fine-grained source code changes (*SCC*).

- **RQ2:** Are Java classes affected by certain types of antipatterns more change-prone than Java classes affected by other antipatterns – *i.e.*, does the type of antipattern impact change-proneness?
  The results from this research question can assist software engineers in identifying the *risky* classes affected by antipatterns.
- **RQ3:** Are particular types of changes more likely to be performed in Java classes affected by certain types of antipatterns?
  The results of this question will assist software engineers in prioritizing antipatterns that need to be resolved to prevent certain types of changes in a system. For example changes in the method declarations of a class exposing a public *API*.

To answer our research questions, we perform an empirical study with data extracted from 16 Java open-source software systems. Our main outcomes are:

- The number of *SCC* performed in classes affected by antipatterns is statistically greater than the number of *SCC* performed in other classes.
- 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 consider detecting and resolving instances of certain antipatterns to prevent certain types of changes. For instance, they should resolve instances of the *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife* to prevent frequent changes in the APIs.

The remainder of this paper is organized as follows. Section II describes the approach used to mine fine-grained source code changes and to detect Java classes participating in antipatterns. The study design and our findings are presented in Section III. Section IV discusses threats to the validity of the results of our study. Section V presents related work. We draw our conclusions and outline directions for future work in Section VI.

II. Data Collection

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 1 shows an overview of our approach consisting of 4 steps. In the following we describe each step in details.

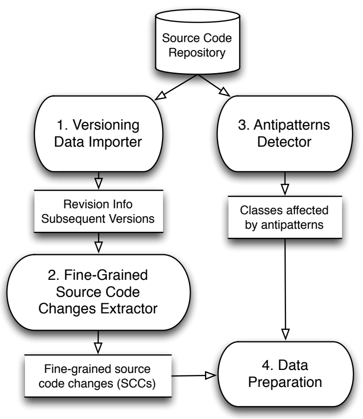
Source Code Repository
1. Versioning Data Importer
3. Antipatterns Detector
Revision Info Subsequent Versions
Classes affected by antipatterns
2. Fine-Grained Source Code Changes Extractor
Fine-grained source code changes (SCCs)
4. Data Preparation

Figure 1: Overview of the approach to extract fine-grained source code changes and antipatterns for Java classes.

A. Importing Versioning Data

The first step concerns retrieving the versioning data for the Java classes from the version control systems (*e.g.*, CVS, SVN or GIT). To perform this step we use the Evolizer Version Control Connector (*EVCC*) [12], belonging to the Evolizer[1] tool set. For each class EVCC fetches and parses the log entries from the versioning repository. Per log entry, EVCC extracts the revision numbers, the revision timestamps, the name of the developers who checked-in the revision, the commit messages, the total number of lines modified, and the source code. This information plus the source code of each revision of Java class is stored into the Evolizer repository.

B. Fine-Grained Source Code Changes Extraction

In the second step, ChangeDistiller is used [11] to extract the *fine-grained source code changes (SCC)* between the subsequent versions of a Java class. ChangeDistiller first parses the source code from the two subsequent versions of a Java class and creates the corresponding Abstract Syntax Trees (*ASTs*). Second, the two ASTs are compared using a tree differencing algorithm that outputs the differences in form of the tree-edit operations add, delete, update, and move. Next, each edit operation for a given node in the AST is annotated with the semantic information of the source code entity it represents and is classified as a specific change type based on a 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

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

subsequent versions of each Java class which is stored into the Evolizer repository.

C. Antipatterns Detection

The third step of our approach is detecting the antipatterns that occur in Java classes. This is achieved by DECOR (Defect dEtection for CORrection) [14], [15], [16]. DECOR provides a domain-specific language to describe antipatterns through a set of rules (*e.g.*, lexical, structural, internal, etc.) and an algorithm to detect antipatterns' in Java classes.

We use the predefined specifications of antipatterns and run DECOR on the different source code releases of our systems under analysis. Among the antipatterns detectable with DECOR we select the following twelve antipatterns:

- AntiSingleton: 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 to data classes.
- 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.
- 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 whose methods can be divided in disjunct set of many methods, thus providing many different unrelated functionalities.

Per release, we obtain a list of detected antipatterns for each Java class. We choose this subset of antipatterns because (1) they are well-described by Brown [7], (2) they appear frequently in the different releases of the systems under analysis and (3) they are representative of design and implementation problems with data, complexity, size, and the features provided by Java classes. Moreover they allow us to compare our findings with those of our previous study [5].

Table I: Categories of source code changes [17].

Category	Description
API	Changes that involve the declaration of classes (*e.g.*, class renaming and class API changes) and 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

D. Data Preparation

In this step, the fine-grained source code changes are grouped and linked with the antipatterns. ChangeDistiller currently supports more than 40 types of source code changes that cover the majority of modifications to entities of object oriented programming languages [13]. We group these change types into five categories. Grouping them facilitates the analysis of the contingency between different types of changes and the interpretation of the results. The different categories are shown in Table I together with a short description of each category. Per Java class revision we count the number of changes for each category. Per Java class we compute the sum for each change type category over the Java class revisions between two subsequent releases *k* and *k+1*. Finally, for each Java class we add the number of antipatterns detected in the Java class at release *k*. We did not normalize the number of changes in classes by the number of lines of code, because we wanted our results to be comparable to previous studies. Furthermore, one of our previous studies [5] has shown that size alone is not the dominating factor affecting the change proneness of classes with antipatterns.

The resulting list contains for each release *k* a list of Java classes with the number of detected instances of the twelve antipatterns at release *k* plus the number of fine-grained changes per change type category that occurred between the two subsequent releases *k* and *k+1*. The analyses performed on these data will be described in the next section.

III. Empirical Study

The *goal* of this empirical study is to investigate the association between antipatterns and the change proneness of Java classes. We performed the empirical study with 16 open-source systems from different domains, implemented in Java and widely used in academic and industrial communities. Table II shows an overview of the dataset. *#Files* denotes the number of Java files in the last release, *#Releases* denotes the number of releases analyzed, *#SCC* denotes the number of fine-grained source code changes in the given time period (*Time*) and *#SCC'* denotes the number of fine-grained source code changes without counting changes performed in the comments and copyrights. In total, changes due to comments and copyrights modifications account for approximately 11% of all the changes (*i.e.*, 64021 out of 585614). This high percentage highlights the necessity to filter out changes related to comments and copyrights, in order to avoid biasing the results.

Table II: Dataset used in our empirical study.

System	#Files	#Releases	#SCC	#SCC'	Time [M,Y]
argo	1716	9	97767	79414	Oc02-Mar09
hibernate2	494	10	26099	23638	Jan03-Mar11
hibernate3	970	20	37271	34440	Jun04-Mar11
eclipse.debug.core	188	12	7600	6555	May01-Mar11
eclipse.debug.ui	793	22	40551	37306	May01-Mar11
eclipse.jface	381	17	14072	11789	Sep02-Mar11
eclipse.jdt.debug	469	16	14983	13647	Jun01-Mar11
eclipse.team.core	172	6	2318	1790	Nov01-Mar11
eclipse.team.cvs.core	189	11	13070	11544	Nov01-Mar11
eclipse.team.ui	293	13	9787	8948	Nov01- Mar11
jabref	1996	30	41665	37983	Dec03-Oct11
mylyn	1288	17	67050	63601	Dec06-Jun09
rhino	184	8	14795	13693	May99-Aug07
rapidminer	2061	4	9899	9277	Oct09-Aug10
vuze	3265	29	119138	113570	Dec06-Apr10
xerces	710	20	69549	54398	Dec00-Dec12

Table III shows the number of antipatterns detected by DECOR in the first and last release of the analyzed systems. Basically, all 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*), *MessageChain* (*MsgC*), and *RefusedParentBequest* (*RPB*) occur most frequently while *SpaghettiCode* (*Spaghetti*), *SpeculatigeGenerality* (*SG*), and *SwissAmryKnife* (*Swiss*) occur less frequently. Overall, the frequency of antipatterns and changes allows us to investigate the three research questions stated in the Section I.

The raw data used to perform our analysis are available on our web site.[2] In the following, we state the hypotheses, explain the analysis methods, and report on the results for each research question.

A. Investigation of RQ1

The *goal* of *RQ1* is to analyze the change-proneness of Java classes affected by antipatterns, compared to the change-proneness of classes not affected by antipatterns. We address RQ1 by testing the following two null hypotheses:

- $\mathbf{H1_a}$: 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.
- $\mathbf{H1_b}$: The distribution of *SCC* performed in classes between two releases is *not* different for classes affected by antipatterns and classes not affected by antipatterns.

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

1) Analysis Method: For investigating $H1_a$ we classify the Java classes of each system and release *k* into *change-prone* if there was at least one change in between two subsequent releases (*k* and *k+1*). Otherwise they are classified as *not change-prone*. This binary variable (we refer to it as *change-proneness(k,k+1)*) denotes the *dependent* variable. As *independent* variable we also use a binary variable that denotes whether a Java class is affected by at least one antipattern in a given release *k*. We refer to this variable as *antipatterns(k)*.

Next, we use the Fisher's exact test [18] to test for each release *k* of each system whether there is an association between *antipatterns(k)* and *change-proneness(k,k+1)* of classes. We then use the *odds ratio* (ORs) [18] 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 indicate that a change can appear with the same probability in both groups. ORs greater than 1 indicate 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 affected by antipatterns are more likely to be changed.

Concerning $H1_b$ we use the Mann-Whitney test to analyze for each release *k* whether there is a significant difference in the distributions of *#SCC(k,k+1)* performed in Java classes affected by antipatterns and in Java classes not affected by any antipattern. We apply the Cliff's Delta *d* effect size [19] 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. 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 $d \geq 0.47$ [19]. 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

Table III: Number of antipatterns detected with DECOR in the first and last releases of the analyzed systems.

System	#Antisingleton	#Blob	#CDBSP	#ComplexC	#LazyC	#LongM	#LPL	#MsgC	#RPB	#Spaghetti	#SG	#Swiss
argo	352-3	26-169	136-51	56-195	16-53	172-354	195-334	130-197	65-513	22-1	9-34	3-4
hibernate2	113-104	34-37	33-17	30-37	5-3	56-72	34-19	51-101	93-97	15-4	2-1	0-0
hibernate3	176-232	52-75	31-50	58-8	9-12	121-194	48-74	157-236	123-202	9-12	3-8	3-9
eclipse.debug.core	1-22	7-14	0-12	1-8	0-9	5-22	0-18	3-6	0-11	0-1	1-1	0-2
eclipse.debug.ui	18-146	13-70	0-70	11-50	0-22	30-176	25-41	6-53	6-73	3-8	2-24	0-7
eclipse.jface	8-25	7-22	6-32	5-13	6-22	22-60	19-45	22-34	5-14	0-2	7-21	0-2
eclipse.jdt.debug	17-44	26-27	1-74	30-33	8-42	68-78	37-40	78-80	80-82	3-3	1-2	1-1
eclipse.team.core	1-12	2-7	1-10	1-5	0-4	8-33	0-26	1-15	0-7	0-1	3-10	0-0
eclipse.team.cvs.core	9- 64	1-21	2-6	1-21	0-0	17-79	1-51	4-45	0-13	0-1	2-10	0-0
eclipse.team.ui	9-64	1-21	2-6	1-21	0-0	17-79	1-51	4-45	0-13	0-1	2-10	0-0
jabref	12-139	10-136	8-400	9-144	1-126	21-365	2-169	2-332	2-295	1-16	0-17	0-1
mylyn	4-70	43-101	61-174	43-83	2-16	132-300	43-66	98-135	34-165	2-0	12-35	1-1
rhino	16-18	5-11	4-18	9-19	4-9	11-33	9-8	15-51	3-7	0-0	0-2	0-1
rapidminer	11-19	130-161	145-203	152-156	10-15	450-568	214-270	583-674	781-1068	1-1	12-28	3-1
vuze	179-145	199-282	189-270	138-193	29-215	381-473	217-295	514-773	476-637	22-16	21-27	35-70
xerces	10-22	8-59	14-134	13-44	6-21	29-96	16-130	19-99	3-37	2-1	5-4	10-11

with the *orddom* package[3] available for the *R* environment.[4]

2) Results: The odds ratios computed to test *H1*$_a$ are summarized in Table IV. Table IV 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*). The results show that, except for three systems (*eclipse.team.cvs.core*, *jabref* and *rhino*), in most of the analyzed releases, Java classes affected by at least one antipattern are more change-prone than other classes. In total, for 190 out of 244 releases (≈82%), classes affected by at least one antipattern are more change-prone. These results allow us to reject *H1*$_a$ and accept the alternative hypothesis that Java classes affected by antipatterns are more likely to be changed than classes not affected by them.

Table IV: Total number of releases (*#Releases*) and number of releases for which Fisher's exact test and OR show a significant association between change-proneness and antipatterns in Java classes.

System	#Releases	Fisher p-value < 0.01 & OR >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 V 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 (≈7%) there is no significant difference (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.

Based on these findings we can answer RQ1: Java classes affected by antipatterns are more change-prone than other classes. The results confirm the findings of our previous study [5], this time taking into account the type of changes, and filtering out non source code changes such as changes to indentations and comments.

B. Investigation of RQ2

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 classes affected by antipatterns. They should be resolved first. We address RQ2 by testing the following null hypotheses:

- **H2:** The distribution of *SCC* is *not* different for classes affected by different antipatterns.

1) Analysis Method: As dependent variable we use the number of *SCC* performed in a class between two releases *#SCC(k,k+1)*. As independent variable we use a binary variable for each antipattern that denotes whether a class is affected by a particular antipattern. To test *H2* we use the Mann-Whitney test and Cliff's Delta d effect size over all releases for a system. We selected all releases per system since some 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

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

Table V: 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.

		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

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.

2) Results: Table VI shows the values for 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 *SpeculativeGenerality* (*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 of Table VI, this difference is large for SwissArmyKnife (*Swiss*), medium for 2 antipatterns (0.33≤d<0.47), small for 5 antipatterns (0.147≤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.

Looking at the values in bold we can see that classes affected by the *ComplexClass* (*ComplexC*), *SpaghettiCode* (*Spaghetti*) and *SwissArmyKnife* (*Swiss*) antipatterns are more change-prone than classes affected by any other antipattern. More specifically, 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 these results we reject *H2* and we conclude that among all classes the classes affected by the *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife* antipatterns are more change-prone. These results detail our previous findings in [5] by highlighting three antipatterns that are more change-prone than the other antipatterns. Moreover, the new findings allow us to advice software engineers to focus on detecting instances of these three change-prone antipatterns and fix them first.

C. Investigation of RQ3

To address *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 changes is more likely to be performed in classes affected by a specific antipattern. This knowledge can help engineers to avoid or fix certain antipatterns leading to changes that impact large parts of the rest of a software system, such as changes in the method declarations of a class that exposes a public *API*. We answer RQ3 by testing the following null hypothesis:

- **H3:** The distributions of different types of *SCC* performed in classes affected by different antipatterns are *not* different.

1) Analysis Method: To test *H3* we categorize the changes mined with *ChangeDistiller* in five different categories as listed in Table I. 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*, we use the random sampling approach for computing

Table VI: 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	#AS	#Blob	#CDBSP	#ComplexC	#LazyC	#LongM	#LPL	#MsgC	#RPB	#Spaghetti	#SG	#Swiss
argo	0.311	0.098	0.331	0.226	-0.012	0.192	0.148	0.248	0.035	0.354	0.030	**0.528**
hibernate2	0.143	0.112	0.193	0.500	NA	0.149	0.347	0.250	-0.032	0.262	NA	**0.654**
hibernate3	0.171	0.086	0.064	0.386	-0.110	-0.172	0.169	0.170	0.016	0.191	NA	**0.662**
eclipse.debug.core	0.553	0.352	0.419	**0.889**	NA	0.544	0.691	0.289	0.435	NA	0.298	0.650
eclipse.debug.ui	0.169	0.299	0.150	0.454	0.147	0.231	0.169	0.227	NA	0.377	0.009	**0.514**
eclipse.jface	**0.461**	NA	NA	0.411	NA	0.266	NA	0.385	NA	NA	NA	NA
eclipse.jdt.debug	0.277	0.182	0.078	0.485	0.103	0.250	0.295	0.137	0.051	0.361	NA	**0.919**
eclipse.team.core	0.422	0.433	NA	**0.581**	NA	0.33	0.107	0.315	NA	NA	0.373	NA
eclipse.team.cvs.core	0.026	0.374	0.085	**0.723**	NA	0.331	0.172	0.329	NA	NA	NA	NA
eclipse.team.ui	0.290	0.293	0.212	0.395	NA	0.265	0.163	0.187	NA	**0.642**	0.183	NA
jabref	0.089	0.001	0.019	0.094	NA	0.072	0.044	0.042	-0.006	0.356	NA	**0.966**
mylyn	-0.020	0.150	0.177	**0.388**	NA	0.192	0.232	0.228	0.063	NA	NA	NA
rhino	0.276	NA	0.393	0.119	NA	0.067	0.025	0.100	NA	**0.928**	NA	NA
rapidminer	0.051	0.060	-0.001	0.141	NA	0.051	0.080	0.051	-0.002	NA	NA	**0.600**
vuze	0.151	0.076	0.079	0.211	NA	0.121	0.106	0.140	-0.021	**0.308**	0.028	0.213
xerces	0.302	0.104	0.044	0.541	NA	0.269	0.327	0.122	0.036	0.153	0.307	**0.565**
Median	0.223	0.131	0.117	**0.403**	0.045	0.211	0.169	0.207	0.025	**0.355**	0.183	**0.625**

Cliff's Delta and we report the mean effect size of the 30 random samples.

2) Results: Table VII lists the results of this analysis. Values in bold denotes differences that are at least small according to Cliff's Delta. They show that changes in the class and methods declaration (*API*) are more likely to appear in classes affected by the *ComplexClass*, *SpaghettiCode* and *SwissArmKnife* antipatterns. Changes in the functionalities (*func*) are likely in classes affected by the *ComplexClass* and *SpaghettiCode* antipatterns. Changes in the execution statements (*stmt*) are likely to appear in classes affected by the *Antisingleton*, *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns. Finally, changes in the condition expressions and *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.

D. 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. For instance, the class *org.apache.xerces.StandardParserConfiguration* from the Xerces system. 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. As another example, consider the *org.eclipse.debug.ui.views.memory.AddMemoryBlockAction* class from the *eclipse.debug.ui* system. 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.

E. Implications of Results

In summary, we see two main implications of our results that concern software engineers and researchers. Concerning the researcher, our results provide a deeper insight into the effects of antipatterns on the change-proneness of Java classes. First, we confirmed the results from our previous study [5] but this time taking into account the type of changes (see *RQ1*). Second, we identified three antipatterns, namely *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* that lead to change-prone classes (see *RQ2*). Third and most of all, we showed that certain antipatterns lead to certain types of changes (see *RQ3*). This helps to focus our research on a sub-set of antipatterns, namely the most change-prone ones.

Regarding the software engineer, the results of our study have several implications. In particular, the results for RQ2 and RQ3 show that software engineers should focus on detecting and resolving the three antipatterns *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife*. Classes affected by these antipatterns turned out to be the most change-prone ones, therefore resolving instances of these antipatterns helps to prevent changes in their *API*s. In particular, because *API* changes can have a significant impact on the implementation of the other parts of a software system 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 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.

Table VII: 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 ($d > 0.147$).

Group	#Antisingleton	#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

IV. Threats to Validity

This section discusses the threats to validity that can affect the results of our empirical study.

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 for which 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 [20]. 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 the raw data to allow other researches to replicate our study and to test other hypotheses on our dataset.

V. Related Work

In this section, we discuss the related literature on antipatterns in relation to software evolution.

Code Smells/Antipatterns Detection Techniques: The first book on "antipatterns" in object-oriented development was written in 1995 by Webster [21]. The book made several contributions on conceptual, political, coding, and quality-assurance problems. Beck [22] defined 22 code smells, suggesting where developers should apply refactorings. Mäntylä [23] and Wake [24] proposed classifications for code smells. Brown *et al.* [7] described 40 antipatterns, including the Blob, the Spaghetti Code, and the MessageChain. These books provide in-depth views on heuristics, code smells, and antipatterns, and are the basis of all approaches to detect (semi-)automatically code smells and antipatterns, such as DECOR [25] used in this study.

Several approaches to specify and detect code smells and antipatterns exist in the literature. They range from manual approaches, based on inspection techniques [26], to metric-based heuristics [27], [28], [29], using rules and thresholds on various metrics or Bayesian belief networks [**?**]. Some approaches for complex software analysis use visualization [31], [32]. 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], [35]. This previous work significantly contributed to the specification and detection of antipatterns. The approach used in this study, DECOR, builds on this previous work.

Code Smells/Antipatterns and Software Evolution: Deligiannis *et al.* [36], [37] proposed the first quantitative study of the relation between antipatterns and software quality. They performed a controlled experiments with 20 students on two software systems to understand the impact of *Blobs* on the understandability and maintainability of software systems. The results of their study suggested that *Blob* classes considerably affect the evolution of design structures, in particular the use of inheritance. Du Bois *et al.* [38] showed that the decomposition of *Blob* classes into a number of collaborating classes using refactorings can improve comprehension. Abbès *et al.* [39] conducted three experiments, with 24 subjects each, to investigate whether the occurrence of antipatterns does affect the understandability of systems by developers during comprehension and maintenance tasks. They concluded that although the occurrence of one antipattern does not significantly decrease developers' performance, a combination of two antipatterns impedes significantly developers' performance during comprehension and maintenance tasks.

Li *et al.* [40] investigated the relationship between the probability of a class to be faulty and some antipatterns based on three versions of Eclipse and showed that classes with antipatterns *Blob*, *Shotgun Surgery*, and *Long Method* have a higher probability to be faulty than other classes. Olbrich *et al.* [41], analyzed the historical data of Lucene and Xerces over several years and concluded that classes with the antipatterns *Blob* and *Shotgun Surgery* have a higher change frequency than other classes; with *Blob* classes featuring more changes. However, they did not investigated the kinds of changes performed on the antipatterns.

Using Azureus and Eclipse, we 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 [10]. In [5] we also investigated the relation between the presence of antipatterns and the change- and fault-proneness of classes. We found that classes participating in antipatterns are significantly more likely to be subject to changes and to be involved in fault-fixing changes than other classes. Furthermore, we also investigated the kind of changes, namely structural and non-structural changes, experienced by classes with antipatterns. Structural changes are changes that alter a class interface while non-structural changes are changes to method bodies. We found that in general structural changes are more likely to occur in classes participating in antipatterns. The main difference with this work is that we detailed the changes into 40 types of source code changes classified in 5 change type categories. This detailed information about changes allowed us to analyze which antipatterns lead to which types of source code changes. Also, this work is performed with more systems, namely 16, compared to our previous work which was done with only 4 systems.

VI. Conclusion and Future Work

Antipatterns have been defined to denote “poor” solutions to design and implementation problems. Previous studies have shown that classes affected by antipatterns are more change-prone than other classes. In this paper we provide a deeper insight into which antipatterns lead to which types of changes in Java classes. We analyzed the change-proneness of these classes taking into account 40 types of fine-grained source code changes (*SCC*) extracted from the version control repositories of 16 Java open-source systems. Our results show that:

- Classes affected by antipatterns change more frequently along the evolution of a system, confirming our previous findings (see RQ1).
- Classes affected by the *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns are more likely to be changed than classes affected by other antipatterns (see RQ2).
- Certain antipatterns lead to certain types of source code changes, such as *API* changes are more likely to appear in classes affected by the *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns (see RQ3).

Our results have several implications on software engineers and researchers. Regarding researchers our results suggest to focus our efforts on understanding a subset of antipatterns that lead to change-prone classes or changes with a high impact on the other parts of a software system. Concerning software engineers, our results provide strong evidence to use antipatterns detection tools, such as DECOR, to detect and resolve *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns. Resolving them shows to be beneficial in terms of preventing source code changes, such as *API* changes, that impact other parts of a system.

In future work, we plan to perform a more extended qualitative analysis of antipatterns. We also plan to enlarge our data set and analyze industrial software systems. Another direction of future work is to analyze the types of changes performed when antipatterns are introduced and when they are resolved. These analysis are needed to further estimate the development and maintenance costs caused by antipatterns.

Acknowledgment

We would like to thank the anonymous reviewers of WCRE for their valuable feedback that helped us to improve the quality of the paper. This work has been partially funded by the NWO-Jacquard program within the ReSOS project.

References

[1] L. Erlikh, “Leveraging legacy system dollars for e-business,” *IT Professional*, vol. 2, no. 3, pp. 17 –23, may/jun 2000.

[2] D. Romano and M. Pinzger, “Using source code metrics to predict change-prone java interfaces,” in *ICSM*, 2011, pp. 303–312.

[3] B. M. Mauczka A., Grechenig T., “Predicting code change by using static metrics,” in *Software Engineering Research, Management and Applications*, 2009, pp. 64–71.

[4] D. Posnett, C. Bird, and P. Dévanbu, “An empirical study on the influence of pattern roles on change-proneness,” *Empirical Softw. Engg.*, vol. 16, no. 3, pp. 396–423, Jun. 2011.

[5] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[6] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta, “An empirical study on the maintenance of source code clones,” *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010.

[7] W. J. Brown, R. Malveau, H. McCormikk III, and T. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] J. O. Coplien and N. B. Harrison, *Organizational Patterns of Agile Software Development*, 1^{st} ed. Prentice-Hall, Upper Saddle River, NJ (2005), 2005.

[10] F. Khomh, M. D. Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *WCRE*, 2009, pp. 75–84.

[11] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 725–743, November 2007.

[12] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Softw.*, vol. 26, pp. 26–33, January 2009.

[13] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ser. ICPC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 35–45.

[14] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. 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*, ser. FASE'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 276–291.

[15] N. Moha, A. M. R. Hacene, P. Valtchev, and Y.-G. Guéhéneuc, "Refactorings of design defects using relational concept analysis," in *Proceedings of the 6th international conference on Formal concept analysis*, ser. ICFCA'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 289–304.

[16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[17] E. Giger, M. Pinzger, and H. 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*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 83–92.

[18] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4th ed. Chapman & Hall/CRC, 2007.

[19] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[20] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 2–13, January 2007.

[21] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995.

[22] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1^{st} ed. Addison-Wesley, June 1999.

[23] M. Mantyla, "Bad smells in software - a taxonomy and an empirical study." Ph.D. dissertation, Helsinki University of Technology, 2003.

[24] W. C. Wake, *Refactoring Workbook*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[25] Naouel Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *Transactions on Software Engineering (TSE)*, 2009.

[26] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *Proceedings of the 14^{th} Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1999, pp. 47–56.

[27] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20^{th} International Conference on Software Maintenance*. IEEE CS Press,

[28] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the 11^{th} International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[29] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *Proceedings of the 14^{th} Conference on Software Maintenance and Reengineering*, R. Capilla, R. Ferenc, and J. C. Dueas, Eds. IEEE Computer Society Press, March 2010.

[30] Foutse Khomh, Stéphane Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the 9^{th} International Conference on Quality Software (QSIC)*. IEEE CS Press, August 2009, 10 pages.

[31] K. Dhambri, H. Sahraoui, and P. Poulin, "Visual detection of design anomalies." in *Proceedings of the 12^{th} European Conference on Software Maintenance and Reengineering, Tampere, Finland*. IEEE CS Press, April 2008, pp. 279–283.

[32] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01)*. IEEE CS Press, 2001, p. 30.

[33] G. Langelier, H. A. Sahraoui, and P. 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] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. [Online]. Available: http://www.springer.com/alert/urltracking.do?id=5907042

[35] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE CS Press, Oct. 2002.

[36] D. Ignatios, S. Ioannis, A. Lefteris, R. Manos, and S. Martin, "A controlled experiment investigation of an object oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, no. 2, February 2003.

[37] D. Ignatios, S. Martin, R. Manos, and S. Ioannis, "An empirical investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, 2004.

[38] B. D. Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, "Does god class decomposition affect comprehensibility?" in *Proceedings of the IASTED International Conference on Software Engineering*. IASTED/ACTA Press, 2006, pp. 346–355.

[39] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *CSMR, 15th European Conference on Software Maintenance and Reengineering*, T. Mens, Y. Kanellopoulos, and A. Winter, Eds. IEEE Computer Society, 2011, pp. 181–190.

[40] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, no. 7, 2007.

[41] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Third International Symposium on Empirical Software Engineering and Measurement*, 2009.

TUD-SERG-2012-017
ISSN 1872-5392