An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension

Marwen Abbes[1,3], Foutse Khomh[2], Yann-Gaël Guéhéneuc[3], Giuliano Antoniol[3]
[1] Dépt. d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, Canada
[2] Dept. of Elec. and Comp. Engineering, Queen's University, Kingston, Ontario, Canada
[3] Ptidej Team, SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada
E-mails: marwen.abbes@umontreal.ca, foutse.khomh@queensu.ca
yann-gael.gueheneuc@polymtl.ca, antoniol@ieee.org

***Abstract*—Antipatterns are "poor" solutions to recurring design problems which are conjectured in the literature to make object-oriented systems harder to maintain. However, little quantitative evidence exists to support this conjecture. We performed an empirical study to investigate whether the occurrence of antipatterns does indeed affect the understandability of systems by developers during comprehension and maintenance tasks. We designed and conducted three experiments, with 24 subjects each, to collect data on the performance of developers on basic tasks related to program comprehension and assessed the impact of two antipatterns and of their combinations: Blob and Spaghetti Code. We measured the developers' performance with: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. Collected data show that the occurrence of one antipattern does not significantly decrease developers' performance while the combination of two antipatterns impedes significantly developers. We conclude that developers can cope with one antipattern but that combinations of antipatterns should be avoided possibly through detection and refactorings.**

***Keywords*-Antipatterns, Blob, Spaghetti Code, Program Comprehension, Program Maintenance, Empirical Software Engineering.**

I. Introduction

Context: In theory, antipatterns are "poor" solutions to recurring design problems; they stem from experienced software developers' expertise and describe common pitfalls in object-oriented programming, *e.g.*, Brown's 40 antipatterns [1]. Antipatterns are generally introduced in systems by developers not having sufficient knowledge and–or experience in solving a particular problem or having misapplied some design patterns. Coplien [2] described an antipattern as "something that looks like a good idea, but which back-fires badly when applied". In practice, antipatterns relate to and manifest themselves as code smells in the source code, symptoms of implementation and–or design problems [3].

An example of antipattern is the Blob, also called God Class. The Blob is a large and complex class that centralises the behavior of a portion of a system and only uses other classes as data holders, *i.e.*, data classes. The main characteristic of a Blob class are: a large size, a low cohesion, some method names recalling procedural programming, and its association with data classes, which only provide fields and–or accessors to their fields. Another example of antipattern is the Spaghetti Code, which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code classes have little structure, declare long methods with no parameters, and use global variables; their names and their methods names may suggest procedural programming. They do not exploit and may prevent the use of object-orientation mechanisms: polymorphism and inheritance.

Premise: Antipatterns are conjectured in the literature to decrease the quality of systems. Yet, despite the many studies on antipatterns summarised in Section II, few studies have empirically investigated the impact of antipatterns on program comprehension. Yet, program comprehension is central to an effective software maintenance and evolution [4]: a good understanding of the source code of a system is essential to allow its inspection, maintenance, reuse, and extension. Therefore, a better understanding of the factors affecting developers's comprehension of source code is an efficient and effective way to ease maintenance.

Goal: We want to gather quantitative evidence on the relations between antipatterns and program comprehension. In this paper, we focus on the system understandability, which is the degree to which the source code of a system can be easily understood by developers [5]. Gathering evidence on the relation between antipatterns and understandability is one more step [6] towards (dis)proving the conjecture in the literature about antipatterns and increasing our knowledge about the factors impacting program comprehension.

Study: We perform three experiments: we study whether systems with the antipattern Blob, first, and the Spaghetti Code, second, are more difficult to understand than systems without any antipattern. Third, we study whether systems with both Blob and Spaghetti Code are more difficult to understand than systems without any antipatterns. Each experiment is performed with 24 subjects and on three different systems developed in Java. The subjects are graduate students and professional developers with experience in software development and maintenance. We ask the subjects to perform three different program comprehension tasks covering three out of four categories

of usual comprehension questions [7]. We measure the subjects' performance with: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers.

Results: Collected data show that the occurrence of one antipattern in the source code of a system does not significantly reduce its understandability when compared to a source code without any antipattern. However, the combination of two antipatterns impacts negatively significantly subjects' comprehension; hinting that developers can cope with antipatterns in isolation but that combinations thereof should be avoided during development and maintenance.

Relevance: Understanding the impact of antipatterns on program comprehension, especially the understandability of systems, is important from the points of view of both researchers and practitioners. For researchers, our results bring further evidence to support the conjecture in the literature on the negative impact of antipatterns on the quality of systems. For practitioners, our results provide concrete evidence that they should pay attention to systems with a high number of classes participating in antipatterns, because these antipatterns would reduce their systems understandability and, consequently, increase their systems' aging [8]. Our results also support *a posteriori* the removal of antipatterns as early as possible from systems and, therefore, the importance and usefulness of antipatterns detection techniques.

Organisation: Section II relates our study with previous work. Section III describes our empirical study definition and design. Section IV presents the study results while Section V discusses them and threats to their validity. Section VII concludes with future work.

II. Related Work

We summarise previous works on the impact of antipatterns and on their relation to program comprehension.

Antipatterns Definition and Detection: The first book related to antipatterns in object-oriented development was written in 1995 by Webster [9]; his contribution includes conceptual, political, coding, and quality-assurance problems. Riel [10] defined 61 heuristics characterising good object-oriented programming to assess a system quality manually and improve its design and implementation. Beck [3] defined 22 code smells, suggesting where developers should apply refactorings. Mäntylä [11] and Wake [12] proposed classifications for code smells. Brown *et al.* [1] described 40 antipatterns, including the Blob and Spaghetti Code. These books provide in-depth views on heuristics, code smells, and antipatterns aimed at industrial and academic audiences.

Several approaches to specify and detect code smells and antipatterns exist in the literature. They range from manual approaches, based on inspection techniques [13], to metric-based heuristics [14], [15], [16], using rules and thresholds on various metrics [17] or Bayesian belief networks [18].

Antipatterns and Evolution: Olbrich *et al.* [19] 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 [20] studied the evolution of Long Method, Feature Envy, and State Checking 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 these smells persist 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.* [6] studied the impact of classes with code smells on change-proneness and the particular impact of certain code smells. They showed that the likelihood for classes with code smells to change is very high, except in a few explainable cases.

Antipatterns and Developers: Deligiannis *et al.* [21], [22] proposed the first quantitative study of the impact of antipatterns on software development and maintenance activities. They performed a controlled experiment with 20 students on two systems on the impact of Blob classes on the understandability and maintainability of the 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 God classes on the ease of their subjects to understand the systems and the subjects' ability to perform successful comprehension tasks on these systems.

Du Bois *et al.* [23] showed through a controlled experiment with graduate students that the decomposition of God classes into a number of collaborating classes using well-known refactorings can improve their understandability. In their experiment, students were asked to perform simple maintenance tasks on God classes and their decompositions. Du Bois *et al.* found that the students had more difficulties understanding the original God class than certain decompositions. However, their study did not reveal any objective notion of "optimal comprehensibility".

Summary: These previous works raised the awareness of the community towards the impact of code smells and antipatterns on software development and maintenance activities. We build on these previous works and propose experiments assessing the impact of the Blob and Spaghetti Code on the understandability of systems.

III. Experimental Design

We perform three experiments to assess the comprehension of source code by subjects in the presence of two antipatterns. Experiment 1 deals with the Blob, Experiment 2 deals with the Spaghetti Code, and Experiment 3 deals with both these antipatterns. We chose these antipatterns because they are well-known and have been used in the literature to perform other experiments, in particular in the previous work by [6], [23], [24], [25], [26]. In each experiment, we assign two systems to each subject: one

Experiments	Systems	Numbers of Classes	SLOCs	Release dates
1	YAMM 0.9.1	64	11,272	1999
	JVerFileSystem	167	38,480	2008
	AURA	95	10,629	2008
2 and 3	GanttProject 2.0.6	527	68,545	2008
	JFreeChart 1.0.13	989	302,844	2009
	Xerces 2.7.0	740	233,331	2008

Table I
OBJECT SYSTEMS

containing one occurrence of one (or both) antipattern and one without any occurrence. We then measure and compare the subjects' performances for both systems. We follow Wohlin *et al.*'s template [27] to describe the experimental design of Experiment 1, giving particulars of the other two experiments when appropriate.

A. Research Question

Our research questions stem from our goal of understanding the impact of antipatterns on program comprehension and is: "what is the impact of an occurrence of the Blob antipattern (respectively of the Spaghetti Code antipattern and of the two antipatterns) on understandability?"

B. Hypotheses

We want to assess the following null hypothesis when subjects perform comprehension tasks with source code:

- $H_{0_{Blob}}$: There is no statistically significant difference between the subjects' average performance when executing comprehension tasks on the source code of systems containing one occurrence of the antipattern Blob and their average performance with source code without any antipattern.

We have two identical null hypotheses $H_{0_{SpaghettiCode}}$ and $H_{0_{Blob+SpaghettiCode}}$ for the other antipattern and for the combination of one occurrence of each antipattern.

If we reject the previous null hypotheses, then we explain the rejection either as:

- either $E_{1_{Blob}}$: the subjects' average performance is better when executing comprehension tasks on systems containing no occurrence of the Blob;
- or $E_{2_{Blob}}$: the subjects' average performance is better when executing comprehension tasks on systems containing one occurrence of the Blob;

and similarly for the Spaghetti Code and the combination of the two antipatterns ($E_{1_{SpaghettiCode}}$, $E_{2_{SpaghettiCode}}$, $E_{1_{Blob+SpaghettiCode}}$, and $E_{2_{Blob+SpaghettiCode}}$).

We choose one explanation by comparing the subjects's average performance: $E_{1_{Blob}}$ if the average of developers' performance is better with systems containing no occurrence of the Blob, else $E_{2_{Blob}}$. We thus conclude on the impact of antipattern on understandability within the limits of the threats to the validity of our experiments in Section VI.

C. Objects

We choose three systems for each experiment, all developed in Java, and briefly described in Table I. We performed each experiment on 3 systems, because one system could be intrinsically easier/more complex to understand.

For Experiment 1, we use YAMM (Yet Another Mail Manager): an email client previously used in a similar study by Du Bois *et al.* [23]; JVerFileSystem: a system to model and analyse the content of version control systems, like CVS or SVN [28]; and, Aura: a tool implementing a hybrid approach to generate rules to upgrade a system when its underlying framework evolves [29].

For Experiment 2 and 3, we use GanttProject[1]: a cross-platform desktop tool for project scheduling and management; JFreeChart[1]: a chart library for Java to generate various kinds of charts, such as pie, bar, or time series charts; and, Xerces[1]: a parser to analyze XML documents written according to XML 1.1. It implements a number of standard API for XML parsing, including DOM, SAX, and SAX2.

We used the following criteria to select the systems. First, we selected open-source systems; therefore other researchers can replicate our experiment. Second, we avoided to select small systems that do not represent the ones that developers deal normally. We affected randomly a set of three systems to each experiment. We also chose these systems because they are typical examples of systems having continuously evolved on periods of time of different lengths. Hence, the occurrences of Blob and Spaghetti Code in these systems are not coincidence but are realistic. We use our antipattern detection technique, DEX, which stems from our DECOR method [24], [26] to ensure that each system has at least one occurrence of the Blob and–or the Spaghetti Code antipattern. We validate the detected occurrences manually. From each system, we selected randomly a subset of classes responsible for managing a specific task to limit the size of the displayed source code. For example, in JFreeChart, we chose the source code of the classes responsible for editing and displaying the properties of a plot. Hence the subsets used in our experiments have different sizes. However this difference would not impact our results because, regardless of the sizes, a subject concentrates his efforts only on a small part of the subset in which a Blob and–or Spaghetti Code class plays a *central* role, *i.e.*, the Blob class and its surrounding classes. Therefore, we ensure that all subjects perform the comprehension tasks within, almost, the same piece of code. Then, we refactor [30] each subset of each system to remove all other occurrences of (other) antipatterns to reduce possible bias by other antipatterns, while keeping the system compilable and functional. We performed manual refactorings following the guidelines of Fowler's book [30]. For example when dealing with a Blob class, we replace it by multiple smaller classes or just spread the methods to their right places. (In the course

[1]http://ganttproject.biz/index.php, http://www.jfree.org/jfreechart/, and http://xerces.apache.org/

of the refactorings, we have removed and introduced new classes, hence Aura with one occurrence of the Blob has 89 classes, see Table IV, while its original version has 95 classes, see Table I.)

Therefore, for Experiment 1, we obtain three subsets of the three systems, each containing one and only one occurrence of a Blob class. For Experiment 2, each subset contains only one occurrence of the Spaghetti Code. For Experiment 3, each subset contains one occurrence and only one occurrence of both antipatterns co-occurring in the same class.

We finally refactor each subset of the systems to obtain new subsets in which no occurrence of the antipatterns exist. We use these subsets as base line to compare the subjects' performance and test our null hypothesis.

D. Independent Variables

The independent variable in Experiment 1 is the presence of the occurrence of the Blob antipattern, which is a Boolean value stating whether there is such an occurrence or not. It is the value of this independent variable that should influence the subjects' performances. In Experiment 2 the independent variable is the presence of one occurrence of the Spaghetti Code while in Experiment 3 it is the presence of one occurrence of the Blob and Spaghetti Code antipattern.

E. Dependent Variables

The dependent variables measure the subjects' performance, in terms of effort, time spent, and percentage of correct answers. We measure the subjects' effort using the NASA Task Load Index (TLX) [31]. The TLX assesses the subjective workload of subjects. It is a multi-dimensional measure that provides an overall workload index based on a weighted average of ratings on six sub-scales: mental demands, physical demands, temporal demands, own performance, effort, and frustration. NASA provides a computer program to collect weights of six sub-scales and ratings on these six sub-scales. We combine weights and ratings provided by the subjects into an overall weighted workload index by multiplying ratings and weights; the sum of the weighted ratings divided by fifteen (sum of the weights) represents the effort [32].

We measure the time using a timer developed in Java that the subjects must start before performing their comprehension tasks to answer the questions and stop when done.

We compute the percentage of correct answers for each question by dividing the number of correct elements found by the subject by the total number of correct elements they should have found. For example, for a question on the references to a given object, if there are ten references but the subject find only four, the percentage would be forty.

F. Mitigating Variables

We retain three mitigating variables possibly impacting the measures of the dependent variables:

- Subject's knowledge level in Java.
- Subject's knowledge level of Eclipse.
- Subject's knowledge level in software engineering.

We assess the subjects' levels using a *post-mortem* questionnaire administered to subjects at the end of their participation to our study to avoid any bias, because some questions pertain to antipatterns. This questionnaire uses Likert scales for each of the mitigating variables and also include open questions about antipatterns, refactorings, and so on.

G. Subjects

Each experiment was performed by 24 anonymous subjects, S1 to S24. Some subjects were enrolled in the M.Sc. and Ph.D. programs in computer and software engineering in École Polytechnique de Montréal or in computer science in Université de Montréal. Others were professionals working for software companies in the Montréal area, recruited through the authors' industrial contacts. All subjects were volunteers and could withdraw at any time, for any reason.

H. Questions

We used comprehension questions to elicit comprehension tasks and collect data on the subjects' performances.

We consider questions in three of the four categories of questions regularly asked and answered by developers [7]: (1) finding a focus point in some subset of the classes and interfaces of some source code, relevant to a comprehension task; (2) focusing on a particular class believed to be related to some task and on directly-related classes; (3) understanding a number of classes and their relations in some subset of the source code; and, (4) understanding the relations between different subsets of the source code. Each category contains several questions of the same type [7].

We choose questions only in the first three categories, because the last category pertains to different subsets of the source code and, in our experiments, we focus only on one subset containing the occurrence(s) of the antipattern(s). In each chosen category, we select the two most relevant questions through votes among the first three authors, which were validated by the last author. Selecting two questions in each category allows us to have, for each subject, a different question from the same category on the system with and without antipattern, hence reducing the possibility of a learning bias for the second system.

The six questions are the followings. The text in bold is a placeholder that we replace by appropriate behaviors, concepts, elements, methods, and types depending on the systems on which the subjects were performing their tasks.

- Category 1: Finding focus points:
  - Question 1: Where is the code involved in the implementation of **this behavior**?
  - Question 2: Which type represents **this domain** concept or **this UI element or action**?
- Category 2: Expanding focus points:
  - Question 1: Where is **this method** called or **this type** referenced?

  - Question 2: What data can we access from **this object**?
- Category 3: Understanding a subset:
  - Question 1: How are **these types or objects** related?
  - Question 2: What is the behavior that **these types** provide together and how is it distributed over **these types**?

For example, with AURA, we replace "**this behavior**" in Question 1, Category 1, by "differentiating callees" and the question reads: "Where is the code involved in the implementation of differentiating callees?"

For category 2, it may seem that the questions could be simply answered by subjects using Eclipse. Yet subjects still must identify and understand the classes or methods that they believe to be related to the task. Moreover, discovering classes and relationships that capture incoming connections prepare the subject for the questions of the third category.

I. Design

Our design is a 2×3 factorial design [27], presented in Table II. We have three different systems, each with two possibilities: containing or not the occurrence(s) of the antipattern(s). Hence, six combinations are possible. For each combination, we prepare a set of comprehension questions, which together form a *treatment*. We have six different groups of subjects, each one affected by each one treatment.

This design is a between-subject design [33] with a set of different groups of subjects, which allows us to avoid repetition by using a different group of subjects for each treatment. We take care of the groups to ensure their homogeneity and avoid bias in the results, for example we ensure that no group entirely contains male or female subjects. The use of balanced groups simplifies and enhances the statistical analysis of the collected data [27].

J. Procedure

We received the agreement from the Ethical Review Board of Université de Montréal to perform and publish this study. The collected data is anonymous. The subjects could leave any experiment at any time, for any reason, and without penalty of any kind. No subject left the study and took more than 45 minutes to perform the experiment. The subjects knew that they would perform comprehension tasks, but did know neither the goal of the experiment nor whether the system that they were studying contained or did not contain antipatterns. We informed them of the goal of the study after collecting their data, before they finished the experiment.

For each experiment, we prepare an Eclipse workspace packaging the target classes, on which the subjects must perform their comprehension tasks to answer the selected questions. The workspace contains compilable and functional subsets, linked to JAR files with the rest of the system compiled code. It also includes the timer, the TLX program, a brief tutorial on the use of Eclipse, a brief explanation about the system at hand, and the post-mortem questionnaire. We conduct the experiments in the same lab, with the same computer and software environments to avoid any kind of environmental bias. No subjects knew the systems on which they perform comprehension tasks, thus we eliminate the mitigating variable relative to the subject's knowledge of the system.

K. Analysis Method

We use the (non-parametric) Mann-Whitney test to compare two sets of dependent variables and assess whether their difference is statistically significant. The two sets are the subjects' data collected when they answer the comprehension questions on the system with antipattern(s) and without. For example, we compute the Mann-Whitney test to compare the set of times measured for each subject on the system with antipattern(s) with the set of times measured for each subject on the system without antipattern(s). Non-parametric tests do not require any assumption on the underlying distributions.

We also test the hypothesis with the (parametric) t-test. Other than testing the hypothesis, performing the t-test is of practical interest to estimate the magnitude of the differences, for example in the time spent by subjects on systems with and without antipattern(s): we use the Cohen d effect size [34], which indicates the magnitude of the effect of a treatment on the dependent variables. The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$ and large for $d \geq 0.8$. For independent samples and un-paired analysis, as in our study, it is defined as the difference between the means (M_1 and M_2), divided by the pooled standard deviation ($\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$) of both sets: $d = (M_1 - M_2)/\sigma$.

We use Analysis Of Variance (ANOVA) to test if the means of the subjects' groups are identical. ANOVA generalises the t-test to more than two groups. We use ANOVA to assess the dependence between the six sets of dependent variables, as we have six different groups affected to the different treatments. We investigate if there is significant difference between groups for each of our dependent variables. For example, we compute ANOVA to compare the efforts of the six different groups and assess whether there is statistical significant difference due to the treatments.

IV. Study Results

A. Descriptive Statistics

We now describe the collected data and present the results of our measured dependent variables as well as explain our hypotheses. Table IV summarises the averages of collected data. It presents, for all the systems, the average of each dependent variable: efforts, times, and percentages of correct answers. For example, for Experiment 1 and System 1, the subjects took on average 261 seconds to answer one comprehension question on the subset of the source code containing one occurrence of the Blob while they took on average 149 seconds to answer the

	With Antipattern(s)	Without Antipattern(s)
System 1	$S_3, S_7, S_9, S_{11}, S_{12}, S_{18}, S_{21}, S_{24}$	$S_1, S_5, S_8, S_{10}, S_{15}, S_{16}, S_{20}, S_{22}$
System 2	$S_1, S_2, S_6, S_{14}, S_{15}, S_{17}, S_{20}, S_{22}$	$S_4, S_7, S_9, S_{11}, S_{13}, S_{18}, S_{19}, S_{23}$
System 3	$S_4, S_5, S_8, S_{10}, S_{13}, S_{16}, S_{19}, S_{23}$	$S_2, S_3, S_6, S_{12}, S_{14}, S_{17}, S_{21}, S_{24}$

Table II
EXPERIMENTAL DESIGN

	Times				Answers				Efforts			
	M.-W. p	t-Test p	ANOVA p	Cohen d	M.-W. p	t-Test p	ANOVA p	Cohen d	M.-W. p	t-Test p	ANOVA p	Cohen d
Experiment 1	0.46	0.30	**0.02**	0.18	0.41	0.21	0.58	0.27	0.25	0.17	0.29	0.24
Experiment 2	0.89	0.97	0.87	0.01	0.26	0.24	0.60	0.35	0.64	0.57	0.41	0.09
Experiment 3	**<0.01**	**<0.01**	**<0.01**	**1.54**	**<0.01**	**<0.01**	**<0.01**	**1.61**	**<0.01**	**<0.01**	**0.01**	**1.20**

Table III
p-VALUES AND COHEN'S d EFFECT SIZE RESULTS FOR EACH EXPERIMENT

Experiments	Systems	Characteristics	Average Times (s.)	Average Efforts	Average % of Correct Answers	# of Classes	# of SLOCs
Experiment 1: Blob	System 1: YAMM 0.9.1	With Blob	261	43.42	71%	65	11,241
		Without Blob	149	27.99	71%	62	10,923
	System 2: JVerFileSystem	With Blob	251	43.30	67%	83	8,971
		Without Blob	206	33.68	79%	85	8,750
	System 3: Aura	With Blob	189	33.31	58%	89	10,629
		Without Blob	271	42.55	66%	99	11,836
Experiment 2: Spaghetti Code	System 1: GanttProject 2.0.6	With Spaghetti	190	47.31	67%	69	6,171
		Without Spaghetti	194	53.44	51%	69	4,441
	System 2: Xerces 2.7.0	With Spaghetti	215	39.27	52%	63	4,370
		Without Spaghetti	182	34.48	52%	63	4,393
	System 3: JFreeChart 1.0.13	With Spaghetti	195	42.37	52%	76	36,949
		Without Spaghetti	218	45.84	45%	76	36,976
Experiment 3: Both APs	System 1: GanttProject 2.0.6	With both APs	187	45.36	44%	72	4,628
		Without any APs	107	27.32	83%	69	4,441
	System 2: Xerces 2.7.0	With both APs	184	42.83	73%	64	11,634
		Without any APs	140	29.33	86%	63	4,393
	System 3: JFreeChart 1.0.13	With both APs	208	48.68	43%	78	37820
		Without any APs	138	31.27	78%	76	36,976

Table IV
SUMMARY OF THE COLLECTED DATA FOR EACH EXPERIMENT (AP = ANTIPATTERN)

other question in the same category on the system without antipattern.

For Experiment 1, including one occurrence of the Blob, and Experiment 3, including both antipatterns Blob and Spaghetti Code, collected data show that the means of the three dependent variables, between systems with the antipattern(s) and systems without, show meaningful and consistent differences, in particular in Experiment 3. We notice that systems with antipatterns are more time consuming, need more effort, and lead subjects to answer less correctly. The subjects' performances for their comprehension tasks are better when the system does not contain any antipattern. For Experiment 2, including one occurrence of the Spaghetti Code, results show varying differences with no directly-explainable reasons. For example, for Xerces, on average, subjects took less time and effort performing comprehension tasks in the subset of the source code without Spaghetti Code than in the subset with this antipattern, while in GanttProject, we observe the opposite difference.

B. Hypothesis Testing

Table III report the p-values obtained by comparing the differences between the data collected for each experiment. We use the Mann-Whitney test to compare our dependent variables between the systems with and without antipattern(s) and the ANOVA test to compare data between the different groups assigned to the different treatments.

For Experiment 1, systems with one occurrence of the Blob seem to be more time consuming, to need more effort, and to lead to more incorrect answers, but there are no statistically significant differences between the subjects' efforts, times, and percentages of correct answers when comparing systems with and without the Blob antipattern, as shown by high p-values in Table III: we cannot reject $H_{0_{Blob}}$ and therefore must disapprove both explanations $E_{1_{Blob}}$ and $E_{2_{Blob}}$.

For Experiment 2, p-values are also high and, for some systems, we observe better performances when subjects performed their comprehension tasks on systems with one occurrence of the Spaghetti Code. Again, we cannot reject $H_{0_{SpaghettiCode}}$ and therefore must disapprove both explanations $E_{1_{SpaghettiCode}}$ and $E_{2_{SpaghettiCode}}$.

We explain the lack of statistically significant difference, in Experiment 1 and 2, by the fact that one antipattern, in a system of about 75 classes, is not enough to impede the subjects' comprehension of the system. Figure 1 illustrates these results: it shows that the shapes of the two curves (one representing the system with antipatterns,

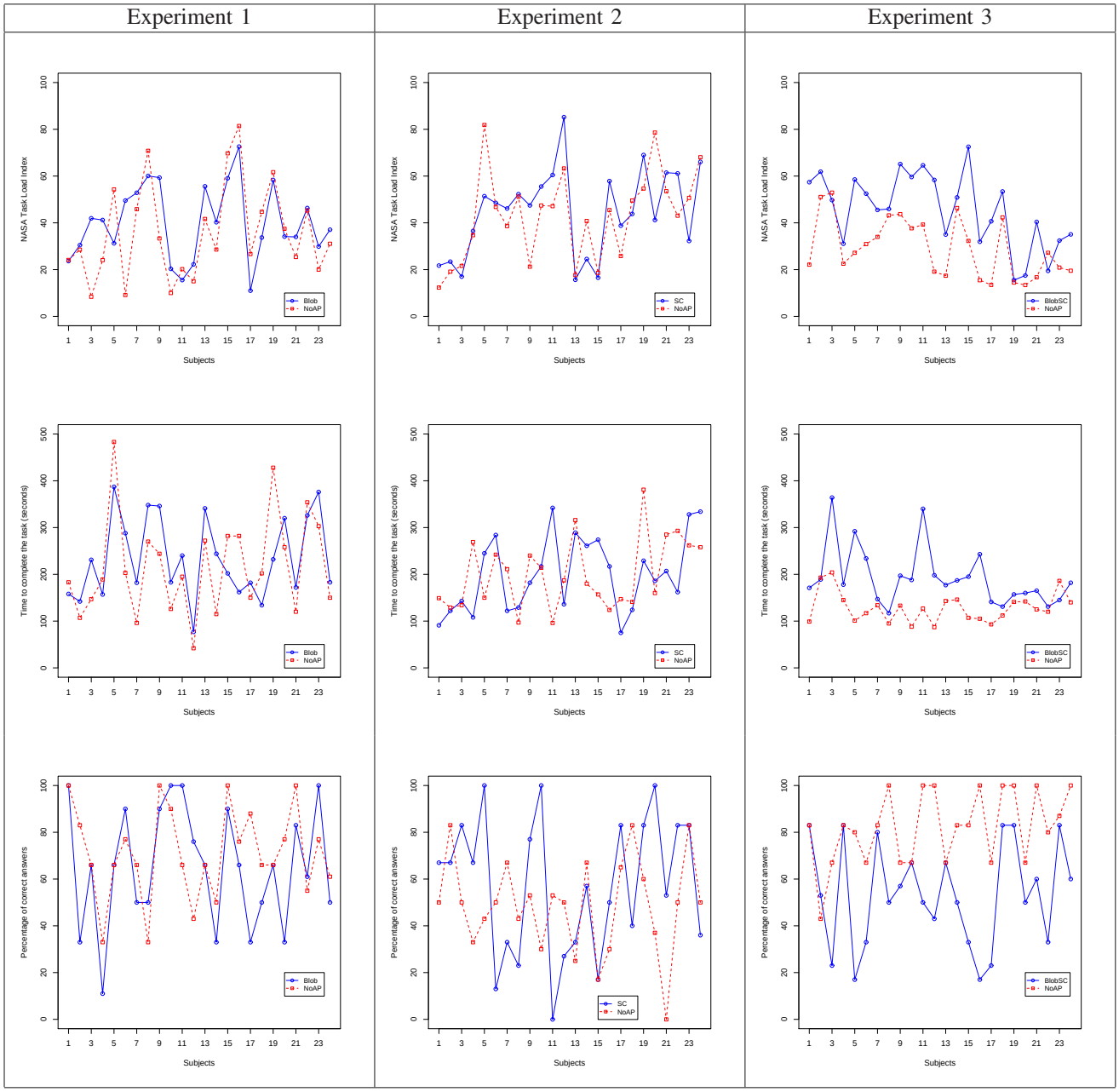

Figure 1. Graphical representations of the collected data

blue solid lines, and one without, red dash lines) for each dependent variable (effort, time, and percentage of correct answers) are almost the same in the two first columns (representing Experiment 1 and 2), which show that all subjects are performing similarly on both systems and are not disturbed by one antipattern.

For Experiment 3, results show statistically significant differences between subjects' efforts, times, and percentages of correct answers between source code with and without the combination of Blob and Spaghetti Code, as shown in Table III. Figure 1 illustrates the statistical differences of the dependent variables: (1) the subjects' efforts are higher in the system with the combination of antipatterns; (2) the times spent by the subjects to perform the comprehension tasks are higher; and, (3) the percentages of correct answers are lower. Moreover, Cohen's d effect size values are large (**1.45** in average). Therefore, a combination of the Blob and Spaghetti Code antipatterns has a strong impact on subjects' efforts, times, and percentages of correct answers. We thus can reject $H_{0_{Blob+SpaghettiCode}}$. A possible explanation is that *the occurrences of both antipatterns (Blob and Spaghetti Code) impedes the subjects' comprehension of the system*. We thus approve $E_{1_{Blob+SpaghettiCode}}$.

V. Discussion

We now discuss the results of our study.

A. Results

The results of Experiment 1 presented in Table IV show an increase in subjects' average time and effort on systems

with Blob and a decrease in their average percentage of correct answers. Therefore, systems with an occurrence of the Blob seem to be more time consuming, to need more effort, and to lead to more incorrect answers. These results confirm the finding by Du Bois *et al.* [23] that students have more difficulties comprehending Blob classes than other classes. However, we did not find a statistically significant difference. Nevertheless, the results of ANOVA suggest a significant difference in the amount of times spent by subjects between the systems with and without the Blob.

Experiment 2 reveals no significant difference between subject's efforts, times, and percentages of correct answers on systems with and without the Spaghetti Code. Surprisingly, subjects appear to perform better on JFreeChart and GanttProject, when there is an occurrence of the Spaghetti Code. Future work includes explaining this observation.

Experiment 3, which studies the combination of the two antipatterns Blob and Spaghetti Code, shows strong statistically significant differences between subjects' efforts, times, and percentages of correct answers. ANOVA results confirm that these differences are significant across our six groups of subjects, in Table II. Moreover, Cohen's d effect size values of the magnitude of the relation between, on the one hand, the presence of the Blob and Spaghetti Code antipattern and, on the other hand, these differences in efforts, times, and percentages of correct answers are large; suggesting the strong relation. Subjects spent more time and effort on systems with the combination of antipatterns and their percentages of correct answers is significantly lower.

In future work, we will investigate whether this statistically significant differences are due to the density of antipatterns in the system and–or to the occurrences of specific antipatterns together.

B. Impact of the Mitigating Variables

We investigated if the three mitigating variables: Java knowledge, Eclipse knowledge, and software engineering knowledge, impacted our results. We set 5 levels, using Lickert scales, corresponding to the subjects' respective levels (bad, neutral, good, excellent, expert).

Table V presents some descriptive statistics of the data collected for these three mitigating variables. As the groups are non-equivalent in terms of size, we used Mann-Whitney Test, which deals with the problem of un-equal sample sizes [35]. We found no significant differences between the different levels; p-values are high, expressing a null influence of the levels of the subjects on the data, the systems too did not affect our results.

We performed an ANOVA test to assess the impact of the mitigating variables on the three measured variables (time, effort, and % of correct answers), which shows that the mitigating variables do not impact our results, as shown by the high p-values in Table V. The lack of impact of these mitigating variables is consistent with previous findings [36], in which the authors assessed the impact of some subjects' knowledge on design patterns on the understandability of various representations of software systems.

VI. Threats to Validity

Some threats limit the validity of our study. We now discuss these threats and how we alleviate or accept them following common guidelines provided in [27].

A. Construct Validity

Construct validity threats concern the relation between theory and observations. In this study, they could be due to measurement errors. We use times and percentages of correct answer to measure the subjects' performances. These measures are objective, even if small variations due to external factors, such as fatigue, could impact their values. However, the observed impact on time and correctness for the Blob antipattern may be just caused by its size. Further experiments should be done to test wether the impact is really due to the Blob antipattern or simply to the large amount of code within the class. We also use the TLX to measure the subjects' effort. The TLX is by its very nature subjective and, thus, it is possible that our subjects provided us with particular effort values.

The degree of seriousness of the antipatterns is also a threat to construct validity. The Blob and Spaghetti Code, in each system, were validated through a voting process for decisions. The two first authors and two other Ph.D. students voted for the antipatterns, the third author validated them. We follow the definitions provided in the book of Brown *et al.* [1] to deal with antipatterns. Yet the occurrences used could have been more or less serious than in other systems. Future work should mitigate this threat.

Construct validity threats could also be due to a mistaken relation between antipatterns and program comprehension. We believe that this threat is mitigated by the facts that many authors discussed this relation, that this relation seems rational, and that the results of our analysis tend to show that, indeed, antipatterns impact program comprehension.

B. Internal Validity

We identify four threats to the internal validity of our study: learning, selection, instrumentation, and diffusion.

Learning: Learning threats do not affect our study for a specific experiment because we used a between-subject design. A between-subject design uses different groups of subjects, to whom different treatments are assigned. We also took care to randomize the subjects to avoid bias (*e.g.*, gender bias). Each subject performed comprehension tasks on two different systems with different questions for each system. However, the same subjects performed Experiment 1 and Experiment 3. The learning effect is minimal because Experiment 3 was performed 5 months after Experiment 1 and used different systems and different questions.

	Systems	Efforts: p-values	Time: p-values	% of Correct Answers: p-values
Java Knowledge	With Blob	0.39	0.10	0.09
	Without Blob	0.93	0.79	0.60
Eclipse Knowledge	With Spaghetti	0.78	0.23	0.47
	Without Spaghetti	0.84	0.67	0.40
Software Engineering Knowledge	With both APs	0.76	0.83	0.17
	Without any APs	0.78	0.88	0.07

Table V
p-VALUES OF THE IMPACT OF KNOWLEDGE LEVELS (AP = ANTIPATTERN)

Selection: Selection threats could impact our study due to the natural difference among the subjects' abilities. We tried to mitigate this threat by asking only volunteers, therefore with a clear willingness to participate. We also studied the possible impact of their levels of knowledge in Java, of Eclipse, and in Software engineering, through three mitigating variables without obtaining any statistically significant results.

Instrumentation: Instrumentation threats were minimized by using objective measures like times and percentages of correct answers. We observed some subjectivity in measuring the subjects' effort via TLX because, for instance, one subject 100% effort could correspond to another's 50% of effort. However, this subjectivity illustrates the concrete feeling of effort of the subjects.

Diffusion: Diffusion threats do not impact our study because we asked subjects not to talk about the study among themselves and the systems and questions among experiments were different. However, it is possible that a few subjects exchanged some information.

C. Conclusion Validity

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. Also, we mainly used non-parametric tests that do not require to make assumption about the data set distribution.

D. Reliability Validity

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study. The systems, questionnaires, and raw data to compute the statistics are on-line[2].

E. External Validity

We performed our study on six different real systems belonging to different domains and with different sizes, see Table I. Our design, *i.e.*, providing only on average 75 classes of each system to each subject, is reasonable because, in real maintenance projects, developers perform their tasks on small parts of whole systems and probably would limit themselves as much as possible to avoid getting "lost" in large code base. However, we cannot assert that our results can be generalised to other Java systems, systems in other programming languages, and to other subjects; future work includes replicating this study in other contexts, with other subjects, other questions, other antipatterns, and other systems.

[2]http://www.ptidej.net/downloads/experiments/csmr11a/

VII. CONCLUSION AND FUTURE WORK

Antipatterns are conjectured in the literature to negatively impact the quality of systems. We performed three experiments to gather quantitative evidences on the relations between antipatterns and program comprehension.

We studied whether systems with the antipattern Blob, first, and the Spaghetti Code, second, are more difficult to understand than systems without any antipattern. Third, we studied whether systems with both the Blob and Spaghetti Code antipatterns are more difficult to understand than systems without any antipatterns. Each experiment was performed with 24 subjects and on three different Java systems.

We measured the subjects' performance with: (1) the NASA task load index for their efforts; (2) the times that they spent performing their tasks; and, (3) their percentages of correct answers. Collected data showed that the occurrence of one antipattern in the source code of a system does not significantly make its comprehension harder for subjects when compared to a source code without any antipattern. However, the combination of two antipatterns impacted negatively and significantly the system understandability; hinting that developers can cope with antipatterns in isolation but that combinations thereof should be avoided, possibly through detection and refactorings.

Consequently, developers and quality assurance personnel should be wary with growing numbers of antipatterns in their systems as they could reduce their system understandability and, therefore, increase the risks of the systems aging and also the introduction of faults. Indeed, D'Ambros *et al.* [37] found that an increase in the number of antipatterns in a system is likely to generate faults.

Future work includes investigating whether these statistically significant differences are due to the density of antipatterns in the system and–or to the occurrences of specific antipatterns together. We also plan to replicates this study in other contexts, with other subjects, other questions, other antipatterns, and other systems.

Acknowledgement

This work has been partly funded by the Canada Research Chairs on Software Patterns and Patterns of Software and on Software Change and Evolution.

REFERENCES

[1] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998. [Online]. Available: www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase_theantipatterngr/103-4749445-6141457

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

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

[4] A. von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[5] F. Khomh and Y.-G. Guéhéneuc, "Do design patterns impact software quality positively?" in *Proceedings of the* 12^{th} *Conference on Software Maintenance and Reengineering*, C. Tjortjis and A. Winter, Eds. IEEE Computer Society Press, April 2008.

[6] Foutse Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, October 2009. [Online]. Available: http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/WCRE09a.doc.pdf

[7] J. Sillito, "Asking and answering questions during a programming change task," Ph.D. dissertation, Vancouver, BC, Canada, Canada, 2007.

[8] D. L. Parnas, "Software aging," in *ICSE '94: Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 279–287.

[9] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995. [Online]. Available: www.amazon.com/exec/obidos/ASIN/1558513973

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

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

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

[13] 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.

[14] 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, 2004, pp. 350–359.

[15] 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. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/METRICS.2005.38

[16] 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. F. Rafael Capilla and J. C. Dueas, Eds. IEEE Computer Society Press, March 2010.

[17] 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. [Online]. Available: http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/TSE09.doc.pdf

[18] 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. [Online]. Available: http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/QSIC09.doc.pdf

[19] 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.

[20] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *QUATIC '10: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology*. IEEE Computer Society Press, 2010.

[21] 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.

[22] 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.

[23] 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.

[24] N. Moha and Y.-G. Guéhéneuc, "On the automatic detection and correction of software architectural defects in object-oriented designs," in *In Proceedings of the 6 th ECOOP Workshop on Object-Oriented Reengineering. Universities of Glasgow and Strathclyde*, 2005.

[25] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Gueheneuc, "Tracking design smells: Lessons from a study of god classes," *Reverse Engineering, Working Conference on*, vol. 0, pp. 145–154, 2009.

[26] N. Moha, Y. G. Guéhéneuc, L. Duchien, and A. F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[27] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[28] J. Tantéri, "Eyes of darwin : une fenêtre ouverte sur l'évolution du logiciel," Master's thesis, Université de Montréal, septembre 2009, master's thesis.

[29] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, 2010, pp. 325–334.

[30] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[31] S. G. Hart and L. E. Stavenland, "Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research," pp. 139–183, 1988.

[32] "Nasa task load index (tlx) v. 1.0," p. 1.

[33] A. T. Duchowski, *Eye Tracking Methodology: Theory and Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[34] S. D.J., *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[35] D. W. Zimmerman, "Comparative power of student t test and mann-whitney u test for unequal sample sizes and variances," *IEEE Trans. Softw. Eng.*, vol. 55, 1987.

[36] G. Cepeda Porras and Y.-G. Guéhéneuc, "An empirical study on the efficiency of different design pattern representations in uml class diagrams," *Empirical Softw. Engg.*, vol. 15, no. 5, pp. 493–522, 2010.

[37] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *QSIC '10: Proceedings of the 2010 10th International Conference on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 23–31.