A Large Scale Empirical Study of the Impact of Spaghetti Code and Blob Anti-patterns on Program Comprehension

7 authors, including:

Cristiano Politowski
Concordia University Montreal
8 PUBLICATIONS   **19** CITATIONS

Giuseppe Scanniello
Università degli Studi della Basilicata
206 PUBLICATIONS   **1,811** CITATIONS

Fabio Petrillo
University of Québec in Chicoutimi
44 PUBLICATIONS   **219** CITATIONS

Some of the authors of this publication are also working on these related projects:

Project   Service Oriented Architecture - Patterns and Practices View project

Project   concern localization View project

A Large Scale Empirical Study of the Impact of Spaghetti Code and Blob Anti-patterns on Program Comprehension[*]

Cristiano **Politowski**[a,*], Foutse **Khomh**[b], Simone **Romano**[c], Giuseppe **Scanniello**[d], Fabio **Petrillo**[e], Yann-Gaël **Guéhéneuc**[a] and Abdou **Maiga**[f]

[a]*Concordia University, Montreal, Quebec, Canada*

[b]*Polytechnique Montréal, Montreal, Quebec, Canada*

[c]*University of Bari, Bari, Italy*

[d]*University of Basilicata, Potenza, Italy*

[e]*Université du Québec à Chicoutimi, Chicoutimi, Quebec, Canada*

[f]*Université Félix Houphouet Boigny, Abidjan, Ivory Coast*

ARTICLE INFO

Abstract

Context: Several studies investigated the impact of anti-patterns (*i.e.,* "poor" solutions to recurring design problems) during maintenance activities and reported that anti-patterns significantly affect the developers' effort required to edit files. However, before developers edit files, they must understand the source code of the systems. This source code must be easy to understand by developers.
Objective: In this work, we provide a complete assessment of the impact of two instances of two anti-patterns, Blob or Spaghetti Code, on program comprehension.
Method: We analyze the impact of these two anti-patterns through three empirical studies conducted at Polytechnique Montréal (Canada) with 24 participants; at Carlton University (Canada) with 30 participants; and at University Basilicata (Italy) with 79 participants.
Results: We collect data from 372 tasks obtained thanks to 133 different participants from the three universities. We use three metrics to assess the developers' comprehension of the source code: (1) the duration to complete each task; (2) their percentage of correct answers; and, (3) the NASA task load index for their effort.
Conclusions: We report that, although single occurrences of Blob or Spaghetti code anti-patterns have little effect on code comprehension, two occurrences of either Blob or Spaghetti Code significantly increases the developers' time spent in their tasks, reduce their percentage of correct answers, and increase their effort. Hence, we recommend that developers act on both anti-patterns, which should be refactored out of the source code whenever possible. We also recommend further studies on combinations of anti-patterns rather than on single anti-patterns one at a time.

1. Introduction

Program comprehension is central to an effective software maintenance and evolution [20]. In his theory of program comprehension, Brooks [4] propound that developers understand software programs in a top-down manner; formulating hypotheses about the domain of the program, mapping this knowledge to the source code, and refining it incrementally. Therefore, understanding the factors affecting developers's comprehension of the source code is essential to improve program comprehension and consequently software maintenance and evolution activities [1].

Anti-patterns, which are "poor" solutions to recurrent design problems, have been reported to significantly affect the effort required to explore and edit files [42]. Furthermore, the increase in the number of anti-patterns in a system is likely to generate faults [7]. One may therefore question whether certain anti-patterns and–or combinations thereof significantly impede code understandability. To understand whether and how anti-patterns affect code understandably, Abbes et al. [1] conducted three experiments, with 24 participants each, investigating whether the occurrences of the Blob[1] and Spaghetti Code[2] anti-patterns impact the understandability of source code for comprehension and maintenance tasks. They analyzed the Blob and Spaghetti Code anti-patterns individually and in combinations. They showed that the occurrence of one Blob or one Spaghetti Code in source code does not significantly reduce its understandability in comparison to source code without anti-pattern. However, they reported that the combination of one occurrence of the Blob anti-pattern with one of the Spaghetti Code significantly decreases understandability. They mentioned the following limitations to their study: (1) the experiments were conducted only with 24 students from the same computer-

---

---

[1]Blob anti-pattern is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data. The major issue here is that the majority of the responsibilities are allocated to a single class [5], violating the single responsibility principle [19].

[2]Considering the Object Oriented paradigm, the software with Spaghetti Code anti-pattern may include a small number of objects that contain methods with very large implementations that invoke a single, multistage process flow [5].

science program and (2) they did not test the impact of multiple occurrences of each anti-pattern individually. Thus, the results reported in this previous work are insightful but incomplete.

We extend this previous work [1] with two additional replications at two different universities. We conducted these replications with 30 students from Carleton University, Ottawa, Ontario, Canada, and 79 students from the University of Basilicata, Potenza, Italy, using six different Java software systems. In total, we collected data from 133 participants executing 372 tasks in three universities. We complemented the previous work with questions about the co-occurrences of the same anti-patterns: two occurrences of the Blob anti-patterns and two occurrences of the Spaghetti Code. We asked the participants to perform program comprehension tasks and we measured their performances using (1) the time that they spent performing their tasks; (2) their percentages of correct answers; and, (3) the NASA task load index of their effort.

Thus, we report in the following a complete, exhaustive study of the impact of the Blob and Spaghetti Code anti-patterns in isolation and in combinations. We show that the presence of two occurrences of either Blob or Spaghetti Code anti-pattern harms source-code understanding, weakening the performance of developers by increasing the time spent to finish the tasks, lowering the correctness of the answers, and increasing the effort to complete the tasks.

We bring further evidences with a larger experiment without setting aside previous results from Abbes et al. [1], which we describe in details in Section 2. Also, after presenting the new data in Section 4, we compare them with previous results in Section 4.3, in which Experiments #1 (Blob) and #2 (Spaghetti Code) correspond to the previous study. Thus, we name the new experiments Experiments #3, #4, #5, and #6.

The remainder of this paper is organized as follows: Section 2 briefly describes the previous paper by Abbes et al. [1]. Section 3 describes the definition and design of our empirical studies. Section 4 presents the study results from experiments with the Blob and Spaghetti Code anti-patterns as well the comparison with the previous data from Abbes et al. [1]. Sections 5 and 6 discuss the results and threats to their validity. Section 7 relates our study with previous work. Section 8 concludes the paper with future work.

2. Previous Work by Abbes et al. [1]

Abbes et al. [1] designed and conducted three experiments, with 24 participants each, to collect data on the performance of developers on tasks related to program comprehension and assessed the impact of anti-patterns: Blob and Spaghetti Code. The first and second experiments examined the impact of an occurrence of Blob and Spaghetti Code anti-pattern, independently, on system understandability. In the third experiment, they used a combination of one Blob and one Spaghetti Code anti-pattern to analyze its impact.

Research Question: Abbes et al. [1] addressed the following research questions: what is the impact of an occurrence

of the Blob anti-pattern (respectively of the Spaghetti Code anti-pattern and of the two anti-patterns mixed) on understandability?

Objects of the Study: Each experiment was performed using three software systems written in Java. Table 1 presents a short description of the systems. For the sake of clarity, we are omitting Abbes et al. [1] experiment #3 here because they combined *Blob+Spaghetti Code*, which we will not perform in our experiments (see next Section).

Table 1
Object systems used in Abbes et al. [1].

Exp	AP	System	Ver.	NoC	SLOC
#1	Blob	YAMM	0.9.1	64	11,272
#1	Blob	JVerFileSystem	-	167	38,480
#1	Blob	AURA	-	95	10,629
#2	Spaghetti	GanttProject	2.0.6	527	68,545
#2	Spaghetti	JFreeChart	1.0	989	302,844
#2	Spaghetti	Xerces	2.7.0	740	233,331

From each system, they selected randomly a subset of classes responsible for a specific feature to limit the size of the displayed source code. They performed manual refactorings [9] on each subset of each system to remove all other occurrences of (other) anti-patterns to reduce possible bias by other anti-patterns, while keeping the systems compiling and functional.

Finally, they refactored each subset of each system to obtain a subset of classes in which there was no occurrence of any anti-pattern. This subset was used as a base line to compare the participants' performance.

Variables of the Study: The independent variables for the first, second, and third experiments are an occurrence of Blob, an occurrence of Spaghetti Code, and a co-occurrence of Blob and Spaghetti Code, respectively. The dependent variables of the three experiments are: participants' performance, in terms of effort, time spent, and percentage of correct answers. The effort of a participant was measured using the NASA Task Load Index (TLX) [11]. They also investigated mitigating variables: participants knowledge level in Java; participant's knowledge level of Eclipse; and, participant's knowledge level in software engineering. They assessed the participants' knowledge levels using a post-mortem questionnaire that was administered to every participant at the end of the experiments.

Participants: Each experiment was performed by 24 anonymous participants. Some participants were enrolled in the M.Sc. and Ph.D. programs in computer and software engineering at Polytechnique Montréal or in computer science at Université de Montréal. Others were professionals working for software companies in the Montréal area, recruited through the authors' industrial contacts.

Experiment Questions: The questions used to elicit comprehension tasks and collect data on the participants' performances were related to: (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; and, (3) understanding a number of classes and their relations in some subset of the source code.

Experiment Design: The design for each experiment is presented in Table 2. An Eclipse workspace was provided to each participant. This workspace contained a compiling and functional subset of classes, linked to JAR files containing the rest of the compiled code. It also contained a timer, a TLX program, a brief tutorial on Eclipse, a brief description of the system at hand, and the post-mortem questionnaire used to measure the participant knowledge. The three experiments were conducted in the same laboratory, with the same computer and software environments. The participants did not have prior knowledge of the systems on which they performed the comprehension tasks.

Table 2

Design of Experiments #1 and #2 [1], with Blob and Spaghetti Code anti-pattern, showing the participants IDs.

Exp	AP	Participants' ID							
#1	Blob	1	2	6	14	15	17	20	22
#1	Blob	3	7	9	11	12	18	21	24
#1	Blob	4	5	8	10	13	16	19	23
#1	-	4	7	9	11	13	18	19	23
#1	-	1	5	8	10	15	16	20	22
#1	-	2	3	6	12	14	17	21	24
#2	Spaghetti	1	2	6	14	15	17	20	22
#2	Spaghetti	3	7	9	11	12	18	21	24
#2	Spaghetti	4	5	8	10	13	16	19	23
#2	-	4	7	9	11	13	18	19	23
#2	-	1	5	8	10	15	16	20	22
#2	-	2	3	6	12	14	17	21	24

Summary: Results of Abbes et al. [ ]

Results of the three experiments showed that the occurrence of one Blob or one Spaghetti Code anti-pattern in the source code of a software system *does not* significantly make its comprehension harder for participants when compared to source code without anti-pattern. However, the combination of one Blob and one Spaghetti Code anti-pattern impacts negatively and significantly a systems' understandability.

3. Experiment Design

The previous section presented the results of two experiments, considered here #1 and #2. This paper aims to extend that work by adding four more experiments, called #3, #4, #5, and #6, in two different locations. Each one of these deal with two occurrences of Blob and two occurrences of Spaghetti Code in a software system. Our experiment does not consider combinations of anti-patterns, for example *Blob + Spaghetti Code*, as Abbes et al. [1] did.

Table 3 shows the experiments and its participants. In each experiment, we assign two systems to each participant: one containing two occurrences of the Blob or Spaghetti Code anti-pattern and one without any anti-pattern. We then measure and compare the participants' performances for both systems in term of program comprehension. Thus, we can determine the impact of the two anti-patterns on understandability, from the point of view of developers, in the context of Java systems.

3.1. Research Question

Our research question stems from our goal of understanding the impact of two occurrences of Blob and Spaghetti Code anti-patterns on program comprehension. The research questions that our study addresses are:

- **RQ1**: What is the impact of two occurrences of the Blob anti-pattern on the participants' *average time spent* understanding a software system?

- **RQ2**: What is the impact of two occurrences of the Blob anti-pattern on the participants' *average correct answers* to understanding questions?

- **RQ3**: What is the impact of two occurrences of the Blob anti-pattern on the participants' *average effort spent* understanding a software systems?

- **RQ4**: What is the impact of two occurrences of the Spaghetti Code anti-pattern on the participants' *average time spent* understanding software systems?

- **RQ5**: What is the impact of two occurrences of the Spaghetti Code anti-pattern on the participants' *average correct answers* to understanding questions?

- **RQ6**: What is the impact of two occurrences of the Spaghetti Code anti-pattern on the participants' *average effort spent* understanding a software systems?

3.2. Hypotheses

From Research Questions 1, 2, and 3, we formulate the following *null* hypotheses for the Blob anti-pattern:

- $H_{BlobTime}$ There is *no* statistically significant difference between the participants' *average time spent* when executing comprehension tasks on source code containing two occurrences of the *Blob anti-pattern* compared to participants executing the same tasks on source code without any occurrences of the Blob anti-pattern.

- $H_{BlobAnswer}$ There is *no* statistically significant difference between the participants' *correctness of the answers* when executing comprehension tasks on source code containing two occurrences of the *Blob anti-pattern* compared to participants executing the same tasks on source code without any occurrences of the Blob anti-pattern.

Table 3
Group of experiments.

Location	AP Type	#APs	Experiment ID	Participants	Data Origin
Montréal, Canada	Blob	1	#1	24	Abbes et al. [1]
Montréal, Canada	Spaghetti Code	1	#2	24	Abbes et al. [1]
Ottawa, Canada	Blob	2	#3	30	This paper
Ottawa, Canada	Spaghetti Code	2	#4	29	This paper
Potenza, Italy	Blob	2	#5	41	This paper
Potenza, Italy	Spaghetti Code	2	#6	38	This paper

#APs means how many occurrences of the anti-pattern are in the source code.

- $H_{BlobEffort}$ There is *no* statistically significant difference between the participants' *effort* when executing comprehension tasks on source code containing two occurrences of the *Blob anti-pattern* compared to participants executing the same tasks on source code without any occurrences of the Blob anti-pattern.

For the Spaghetti Code anti-pattern and on the basis of Research Questions 4, 5 and 6, we formulate the following *null* hypotheses:

- $H_{SpaghettiTime}$ There is *no* statistically significant difference between the participants' *average time spent* when executing comprehension tasks on source code containing two occurrences of the *Spaghetti Code anti-pattern* compared to participants executing the same tasks on source code without any occurrences of the Spaghetti Code anti-pattern.

- $H_{SpaghettiAnswer}$ There is *no* statistically significant difference between the participants' *correctness of the answers* when executing comprehension tasks on source code containing two occurrences of the *Spaghetti Code anti-pattern* compared to participants executing the same tasks on source code without any occurrences of the Spaghetti Code anti-pattern.

- $H_{SpaghettiEffort}$ There is *no* statistically significant difference between the participants' *effort* when executing comprehension tasks on source code containing two occurrences of the *Spaghetti Code anti-pattern* compared to participants executing the same tasks on source code without any occurrences of the Spaghetti Code anti-pattern.

3.3. Objects

We choose three systems for each experiment, all developed in Java, and briefly described in Table 4. We performed each experiment on the three different systems, because one system could be intrinsically either easier or harder to understand.

For Experiment #3 and #5, we use Azureus[3], a Bit Torrent client used to transfer files via the Bit Torrent protocol

(now known as Vuze); iTrust[4], a medical application that provides patients with a means to keep up with their medical history and records as well as communicate with their doctors; and, SIP[5], an audio/video Internet phone and instant messenger that supports some of the most popular instant messaging and telephony protocols (now known as Jitsi).

For Experiment #4 and #6, we use ArgoUML[6], an UML diagramming application written in Java; JHotDraw[7], a graphic framework for drawing 2D graphics; and, Rhino[8], an open-source JavaScript interpreter.

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 on which developers work normally. We also chose these systems because they are typical examples of systems having continuously evolved over 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 the anti-pattern detection technique DETEX, which stems from the DECOR method [22, 21] to ensure that each system has at least two occurrences of the Blob and–or the Spaghetti Code anti-pattern. We randomly assigned a set of three systems to each experiment.

We manually validated the detected occurrences. From each system, we selected randomly a subset of classes responsible for managing a specific feature to limit the size of the source code given to the participants. For example, in iTrust, we chose the source code of the classes providing patients with access to their medical history and records.

The difference between systems would not impact our results because, regardless of the sizes, a participant concentrates her efforts only on a small part of the subset of the source code in which the anti-pattern class plays a *central* role, *i.e.*, the Blob class and its surrounding classes. Therefore, we ensure that all participants perform comprehension tasks within, almost, the same piece of code. Then, we refactor each subset of each system to remove all other occurrences of (other) anti-patterns to reduce possible bias

---

[3] http://www.vuze.com/

[4] http://agile.csc.ncsu.edu/iTrust/
[5] http://www.jitsi.org/
[6] http://argouml.tigris.org/
[7] http://www.jhotdraw.org/
[8] http://www.mozilla.org/rhino/

Table 4
Object Systems.

Experiment	Anti-pattern	System	Version	NoC	SLOCs	Release	Commits
#3 and #5	Blob	Azureus	2.3.0.6	1,449	191,963	2005	27300
#3 and #5	Blob	iTrust	11	565	21,901	2010	256
#3 and #5	Blob	SIPComm	1	1,771	486,966	2010	12693
#4 and #6	Spaghetti Code	ArgoUml	0.2	1,230	113,017	2006	16144
#4 and #6	Spaghetti Code	JHotDraw	5.4b2	484	72,312	2004	765
#4 and #6	Spaghetti Code	Rhino	1.6R5	108	48,824	2009	3532

#Commits is the approximated number of commits of each repository, gathered on December $13^t h$, 2018. Some of the repositories where migrated, for example from SVN to Github and some historical data may have been lost.

by other anti-patterns, while keeping the system compiling and functioning. We perform manual refactoring following the guidelines from Fowler et al. [9]. For example, when dealing with a Blob class, we replace it by multiple smaller classes and move some methods to existing/new classes.

For experiments #3 and #5, each subset contains two occurrences of the Blob anti-pattern. For Experiment #4 and #6, each subset contains two occurrences of the Spaghetti Code anti-pattern. We refactor each subset of the systems to obtain new subsets in which no occurrence of the anti-patterns exist. We use these subsets as baselines to compare the participants' performances and test our *null* hypotheses.

3.4. Independent and Dependent Variables

The *independent* variables are those variables that we can control. The variables should have some effect on the dependent variables and must be controllable [48]. These are the independent variables of our experiments:

- *Experiment ID* is a nominal variable identifying the experiment. For example, a value for this variable equal to *#3* identifies the third experiment.

- *Object* is a nominal variable representing the system. For example, it assumes the values *Azureus*, *iTrust*, and so on.

- *Treatment* is a nominal variable indicating whether or not the source code contains an anti-pattern. For example, in the experiments #3 and #5, this variable assumes as values *Blob* or *NoBlob*, *i.e.,* source code with-/out the Blob anti-pattern.

The *dependent* variables measure the participants' performance, in terms of time spent, percentage of correct answers, and effort.

We measure the *time* using a timer developed in Java that the participants must start before performing their comprehension tasks to answer the questions and stop when done. The time variable assumes values between 0 seconds and "N" seconds, where a value close to zero indicates that the task was performed quickly by the participant.

We compute the percentage of *correct answers* for each question by dividing the number of correct elements found by a participant 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 participant finds only four, the percentage would be forty. Then, we average these percentages to get an overall percentage for each participant. This variable assumes values between 0% and 100% where a value close to 100% indicates a high correctness in the answers.

We measure the participants *effort* using the NASA Task Load Index (TLX) [11]. The TLX assesses the subjective workload of participants. 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 participants 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 [25] with values between 0% and 100% where a value close to 100% indicates a high effort for a participant.

Finally, Table 5 shows a sample of the stored data with the dependent and independent variables.

3.4.1. Participants' Technical Expertise

The participants' knowledge levels were assessed using a post-mortem questionnaire that was administered to every participant at the end of its participation to an experiment. The questions were in five-point-scale, as described below. Table 6 shows a subset of the data regarding the skills.

1. How do you rate your skills and knowledge in *Software Engineering*?
2. How do you rate your level in *Java*?
3. How do you rate your level in *Eclipse*?

3.5. Participants

For each group of experiments shown in Table 3, we use a random sampling to select and assign participants. The population is a convenient sampling of students participating in classes at two universities.

Table 5
Example of the data, the variables and its structure.

N	Place	Exp	Global ID	Local ID	Object	Treatment	Time	Answer	Effort
97	Carleton	e3	s25	s1	iTrust	Blob	305	23	71
98	Carleton	e3	s25	s1	Azureus	NoBlob	85.7	67	57
99	Carleton	e3	s26	s2	iTrust	Blob	355	33	58
100	Carleton	e3	s26	s2	Azureus	NoBlob	143	72	48
101	Carleton	e3	s27	s3	SIPComm	Blob	273	33	57
102	Carleton	e3	s27	s3	Azureus	NoBlob	150	83	43
103	Carleton	e3	s28	s4	SIPComm	Blob	392	11	67
104	Carleton	e3	s28	s4	iTrust	NoBlob	95	78	52
105	Carleton	e3	s29	s5	Azureus	Blob	277	22	54
106	Carleton	e3	s29	s5	iTrust	NoBlob	92	89	49

Table 6
Example of the data regarding the skills of the participants.

Place	Exp	G.ID	L.ID	Treat.	SE	Java	Eclipse
Montr.	e1	s1	s1	Blob	3	3	3
Montr.	e1	s2	s2	Blob	3	3	3
Montr.	e1	s3	s3	Blob	3	3	3
Montr.	e1	s4	s4	Blob	3	3	3
Montr.	e1	s5	s5	Blob	3	3	4
Montr.	e1	s6	s6	Blob	3	5	3

Experiments #3 and #4: Each experiment was performed by 30 (30 for Blob tasks and 29 for Spaghetti Code tasks) anonymous participants, with some from the M.Sc. and Ph.D. programs of computer and software engineering at the computer science department of Carleton University.

Experiments #5 and #6: 79 anonymous participants performed the experiments at the University of Basilicata, 41 out of 79 dealt with the Blob anti-pattern in the experiment #5 and 38 dealt with the Spaghetti Code anti-pattern in the experiment #6. Participants were enrolled to either the B.Sc. or the M.Sc. program at University of Basilicata. Seven participants were enrolled to the M.Sc. in the first year in computer engineering. The remaining students were enrolled in the third year in computer science. Different participants participated in each experiment.

3.6. Questions

We use comprehension questions to elicit comprehension tasks and collect data on the participants' performances. We consider questions in three of the four categories of questions regularly asked and answered by developers [39]:

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;

4. Understanding the relations between different subsets of the source code. Each category contains several questions of the same type.

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 anti-pattern(s).

The six questions are the followings. The text in bold is a placeholder that we replace by appropriate behaviors, concepts, UI elements, methods, and types depending on the systems on which the participants perform 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 Azureus, we replace "**this behavior**" in Question 1, Category 1, by "the health status of the resource to be downloaded, by calculating the number of seeds and peers" and the question reads: "Where is, in this project, the method involved in the implementation of the health status of the resource to be downloaded, by calculating the number of seeds and peers?"

For Category 2, it could seem that participants could simply answer the questions using Eclipse but they 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 participants for the questions of the third category.

Below are some examples of related questions for the ArgoUML system:

- *Where is, in this project, the method involved in the implementation of parsing a line of text (containing the notation UML for an operation) to align the Operation to the specification given?*

- *What data can we access from an object of the type OperationNotationUml?*

- *How are the types OperationNotationUml and NotationUtilityUml related?*

3.7. Design

The distribution of the participants was random, as long as the same participant did not perform a task on the same system more than once to avoid learning biases. We have three different systems, each with two possibilities: containing or not containing the occurrences of an anti-pattern. Hence, six combinations are possible for each experiment. In each combination, we prepare a set of comprehension questions, which together form a *treatment*.

Table 7 shows the design of the experiments #3, #4, #5 and #6 for the Blob and Spaghetti Code anti-patterns, respectively, as well as the participants who performed the tasks and the respective systems.

3.8. Procedure

We received the agreement from the Ethical Review Boards of Université de Montréal and Carleton University to perform and publish this study. The collected data is anonymous. The participants could leave any experiment at any time, for any reason, and without penalty of any kind. No participant left the study or took more than 45 minutes to perform the experiment. The participants knew that they would perform comprehension tasks but did know the goal of the experiment or the system that they were studying or whether the system contained or not anti-pattern(s). We informed participants of the goal of the study after collecting their data, before they left their experiments.

For each experiment, we prepared an Eclipse workspace packaging the classes on which the participants performed their comprehension tasks to answer the questions. The workspace contained *compilable* and functional subsets of the source code, linked to JAR files containing the rest of the compiled code of the systems. It also included a timer, the TLX program, brief tutorials on the use of Eclipse and about the systems, and a post-mortem questionnaire. No participants knew the systems on which they performed their tasks, thus we eliminated the mitigating variable relative to the participants' prior knowledge of the systems.

After the experiment, we computed 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 about the references to a given object, if there are ten references but the participant find only four, the percentage would be forty.

3.9. Analysis Method

For the analyses, we use the R software environment[9] for statistical computing and graphics in two steps:

Preliminary Analysis: We apply *descriptive statistics* on the dependent variables. We also make use of *boxplots*[10] to analyze the data.

Statistical Hypothesis Test: We test the null hypotheses using *linear mixed models (LMM)* analysis method. We use this method because it allows analyzing models with random effects (as it is the case for participants in our experiments) and data dependency due to repeated measures (as it is our case) [45]. Moreover, thanks to *LMM* analysis method, we analyze the effect of the independent variables (*i.e., experiment*, *object*, and *treatment*) modelled as fixed effects on the dependent variables (*i.e., time spent*, *correctness of answers*, and *effort*). The built models also include a random effect for the participants in the experiments. As it is customary with statistical hypotheses tests, we accept a probability of 5% of committing a Type-I error (*i.e.* $\alpha = 0.05$).

Considering a matrix notation, LMM can be represented as: $y = X\beta + Zu + e$. Where β describe the fixed effects, the variables we want to test, that is, *time*, *answer*, and *effort*; u describe the random effects, in this our case the *GlobalID*, which correspond to a participant's ID; X and Z are the respective column values; Finally, e describe the random errors.

The requirements to apply the *LMM* analysis method are: (1) residuals must follow a normal distribution and (2) the mean of the residuals must be equal to zero [45]. To check the normality of residuals, we use Shapiro-Wilk W test [38], where a *p-value* less than $\alpha = 0.05$ indicates data not normally distributed. Without normality, we use *Friedman Test* on the *treatment* variable. This analysis method does not allow analyzing the effect of *experiment* and *object*.

4. Results

Table 8 and Table 9 report the descriptive statistics regarding the *time* spent (in seconds) during the tasks, correctness of the *answers* (in percentage), and the *effort* (in percentage) used to complete the tasks.

The tasks performed on code containing the anti-pattern Blob are, on average, *33.85% more time consuming* when compared to code without the anti-pattern. The answers for the tasks containing the anti-pattern Blob is, on average, *22.07% less accurate* when compared to task with code without the anti-pattern. Finally, the participants, on average, reported

---

[9]www.r-project.org

[10]The box-plots are drawn with the box height corresponding to the 25th and 75th percentile, with the 50th percentile (the median) marked in the box. The whiskers correspond to the 10th and 90th percentile [48].

Table 7
Design of the experiments #3, #4, #5 and #6, for the Blob and Spaghetti Code anti-patterns, respectively, showing the participants IDs.

Exp	Anti-pattern	Object	Participants' ID													
#3	Blob	iTrust	1	12	44	45	46	11	13	15	17	24				
#3	Blob	Azureus	42	43	2	6	8	9	14	18	21	22				
#3	Blob	SIPComm	23	33	3	4	5	7	10	16	19	20				
#3	-	iTrust	33	42	2	3	5	7	8	18	19	21				
#3	-	Azureus	1	12	23	44	45	4	10	11	16	20	24			
#3	-	SIPComm	43	46	6	9	13	14	15	17	22					
#5	Blob	iTrust	1	12	43	6	7	9	15	21	27	29	36	37		
#5	Blob	Azureus	44	46	3	4	10	14	17	18	22	26	28	34	38	40
#5	Blob	SIPComm	33	42	45	2	8	11	16	19	20	24	30	31	35	39	41
#5	-	iTrust	33	44	46	3	10	11	16	18	19	24	31	35	38	40
#5	-	Azureus	1	42	45	2	7	8	15	20	27	29	30	36	39	41
#5	-	SIPComm	12	43	4	6	9	14	17	21	22	26	28	34	37	
#4	Spaghetti C.	ArgoUML	1	33	45	2	6	7	10	11	15	19				
#4	Spaghetti C.	JHotDraw	12	23	46	3	9	13	16	17	18	22				
#4	Spaghetti C.	Rhino	42	43	44	4	5	8	14	20	21					
#4	-	ArgoUML	12	23	43	3	5	8	13	14	16	18				
#4	-	JHotDraw	1	33	42	44	2	4	10	15	19	20	21			
#4	-	Rhino	45	46	6	7	9	11	17	22						
#6	Spaghetti C.	ArgoUML	23	46	3	4	10	14	17	18	26	28	32	34	38	40
#6	Spaghetti C.	JHotDraw	1	43	6	9	13	15	21	25	27	29	36			
#6	Spaghetti C.	Rhino	33	42	45	2	5	8	11	16	19	24	31	35	39	
#6	-	ArgoUML	1	42	45	2	8	13	15	25	27	29	36	39		
#6	-	JHotDraw	33	46	3	5	10	11	16	18	19	24	31	32	35	38	40
#6	-	Rhino	23	43	4	6	9	14	17	21	26	28	34			

using *more effort (around 10% more)* when performing tasks involving code that contained the anti-pattern Blob.

Table 8
Descriptive statistics showing the aggregated results for the anti-pattern *Blob*.

D.Var.	Treat.	Mean	Median	SD	Min	Max
Time	-	155.1	153.1	60.0	38.2	319.6
Time	Blob	234.5	234.9	81.5	87.5	434.1
Answer	-	60.3	56.1	18.0	25.0	95.8
Answer	Blob	38.3	37.3	18.0	5.5	71.0
Effort	-	52.6	52.1	10.8	34.1	75.7
Effort	Blob	62.4	63.3	12.0	37.7	89.5

The tasks performed on code containing the anti-pattern Spaghetti Code are, on average, *39.50% more time consuming* when compared to code without the anti-pattern. The answers for the tasks containing the anti-pattern Spaghetti Code are, on average, *25.30% less accurate* when compared to task with code without the anti-pattern. Finally, the participants reported, on average, using *more effort (around 10% more)* when performing tasks involving code that contained the anti-pattern Spaghetti Code.

The *boxplots* in Figure 1 and Figure 2 summarizes these

Table 9
Descriptive statistics showing the aggregated results for the anti-pattern *Spaghetti Code* (SC).

| D.Var.	Treat.	Mean	Median	SD	Min	Max
Time	-	184.5	183.9	56.5	86.6	314.5
Time	SC	257.5	253.8	74.6	119.9	457.6
Answer	-	52.8	52.5	13.0	29.1	79.1
Answer	SC	27.5	26.5	17.1	4.1	66.8
Effort	-	55.6	56.5	10.3	35.5	74.7
Effort	SC	65.4	64.2	11.9	41.5	89.5

results. Aside from the spread of the values regarding the variable *answer* and the outliers in the variable *time*, in general, the performance of the participants are better when the anti-patterns are not present.

Before applying the *LMM* analysis method, we assess the normality of the residuals with the *Shapiro Test*, indicating that the data are normally distributed for any dependent variable. As an example, Listing 1 is the R script executed for the data about the Blob anti-pattern and the variable *time*.

The variable *answer*, for the experiments with the Spaghetti Code anti-pattern (#4 and #6) has a value less than 0.05, in-

(a) Blob - Time spent.   (b) Blob - Correctness of Answers.   (c) Blob - Overall Effort.
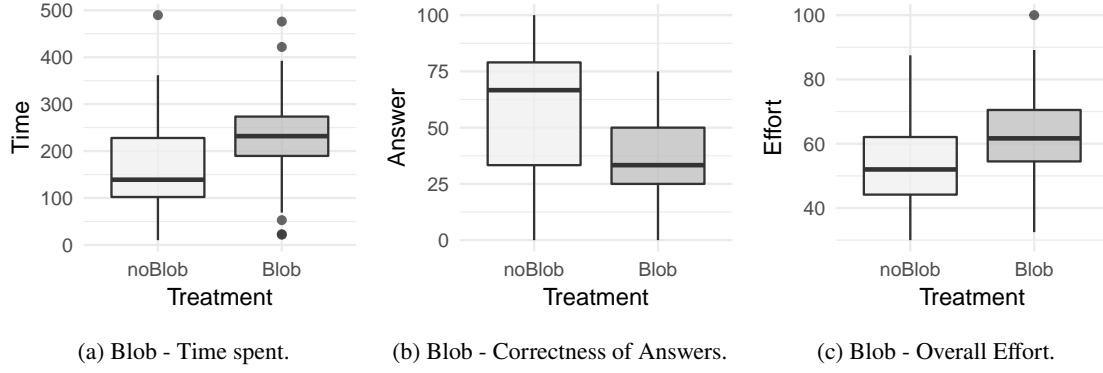
Figure 1: Boxplot of the exploratory analysis with three dependent variables (time, answer, and effort) for the Blob anti-pattern.
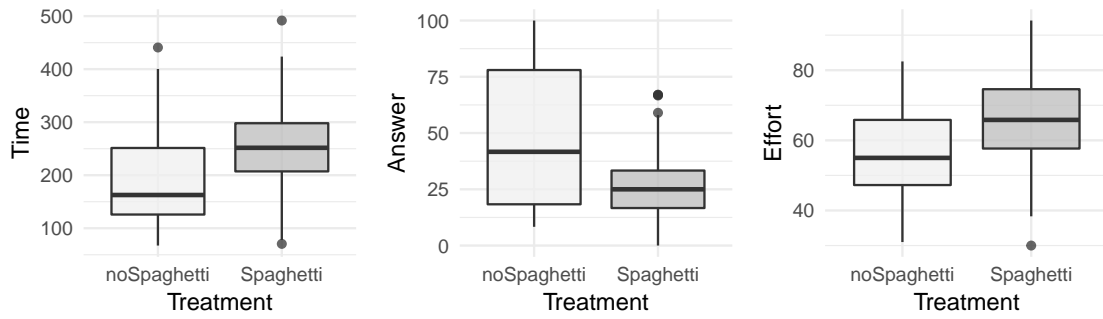
Figure 2: Boxplot of the exploratory analysis with three dependent variables (time, answer, and effort) for the Spaghetti Code anti-pattern.

```
# linear mixed model
LMM.Time <<- lmer(Time ~ Treatment +
    Experiment + Object + Treatment:Object +
    Treatment:Experiment + (1|GlobalID), data=
    dataset_blob)
# residuals check
r_blob_time <- residuals(LMM.Time)
# normality check
shap_blob_time <- shapiro.test(r_blob_time)
# perform anova in the LMM
anova.Time <- Anova(LMM.Time)
```

Listing 1: Example of the R code for the Linear Mixed Model analysis.

dicating non-normally distributed residuals (while the residuals for the other dependent variables were normally distributed). Consequently, we cannot apply the *LMM* analysis method and use the *Friedman Test* for the variable answer.

Table 10 reports the results from the LMM analysis method (left of the vertical line) as well as those from the testing of the LMM assumptions. Each row corresponds to a LMM. For each LMM, we show the p-values for *Treatment*, *Experi-*

ment, *Object*, *Treatment:Object* (*i.e.,* the interaction between *Treatment* and *Object*) and *Treatment:Experiment* (*i.e.,* the interaction between *Treatment* and *Experiment*).

For example, the LMM built for the dependent variable *time* indicates that *Treatment*, *Experiment*, *Treatment:Object*, and *Treatment:Experiment* have significant effects because their p-value is less than 0.05. In particular, a significant effect of *Treatment* means that there is a significant difference between *Blob* and *NoBlob*. Similarly, a significant effect of *Experiment* means that there is a significant difference between the experiments, and so on.

Table 10 show the results of the *LMM* analysis method for Blob and Spaghetti Code anti-patterns. The variables *Treatment* (anti-pattern Blob), *Experiment* (Experiments #3 and #5), as well as their interaction (denoted as *Treatment: Experiment* in Table 10) affect the time spent, correctness of answers, and participants' effort (their p-values are less than $\alpha = 0.05$). The variable *Object*, *e.g.,* the Java system used in the tasks, impact the correctness of the answers (while it does not impact the time spent and participants' effort). This observation could mean that some systems/tasks were more difficult to perform. Lastly, the interaction between *Treatment* and *Object* (*e.g., Treatment:Object* in Table 10) affects the time spent (while the correctness of the answers and participants' effort are not affected).

Table 10
Output of the execution of the *LMM* analysis method, *Shapiro Test* and *Mean of Residuals* for the Blob and Spaghetti Code anti-patterns. Significant effects are displayed in bold font.

Anti-Pattern	Dep. Var.	Treatment	Experiment	Object	Treat.:Obj	Treat.:Exp.	Shapiro	Residual
Blob	Time	**< 0.01**	**0.03**	0.59	**0.03**	**< 0.01**	0.52	0
Blob	Answer	**< 0.01**	**< 0.01**	**< 0.01**	0.22	**< 0.01**	0.82	0
Blob	Effort	**< 0.01**	**0.04**	0.74	0.96	**< 0.01**	0.43	0
Spaghetti	Time	**< 0.01**	**< 0.01**	0.79	0.74	**< 0.01**	0.11	0
Spaghetti	Answer	**< 0.01**	-	-	-	-	**0.02**	0
Spaghetti	Effort	**< 0.01**	0.08	**< 0.01**	0.64	**< 0.01**	0.49	0

The variable *Treatment* (anti-pattern Spaghetti Code) affects the time spent, correctness of answers, and participants' effort. We also observe that the variable *Experiment* (Experiments #4 and #6) and the interaction between *Treatment* and *Experiment* affects the time spent (while the variable *Object* and its interaction with the variable *Treatment* do not affect the time spent). Finally, the variable *Object* has an impact on the participants' effort as well as the interaction between the variable *Treatment* and *Experiment* (while *Experiment* and the interaction between *Treatment* and *Object* do not impact the participants' effort).

4.1. Analyzing Results by Experiments

Figure 3 shows the boxplots grouped by experiments using the *Blob* anti-pattern. Although the experiments had the same setup and design, the results present a clear gap in their values. Experiment #3, with the tasks performed with the anti-pattern Blob, shows *inferior performance* from the participants in all variables (time, accuracy, and effort) while this difference is not visible in Experiment #5.

Similarly, Figure 5 shows the boxplots grouped by experiments with the Spaghetti Code anti-pattern and presents disparities among values. Experiment #4, with the Spaghetti Code, shows *inferior performance* from the participants in all the variables (time, accuracy, and effort) while Experiment #6 shows less clear difference. Also, for the variable effort, the results are slightly contradictory, with better results when the anti-pattern is present.

4.2. Analyzing Participants' Skills

To explain the disparities of the results between the experiments, Figure 4 and Figure 6 show *violin plots* with the skill levels of each participant in *Software Engineering*, *Java*, and *Eclipse*, gathered in the post-mortem questionnaire.

The participants of Experiment #5 and #6 have lower skills levels in Software Engineering and Eclipse but similar levels for Java. Thus, there could be a relationship between the expertise and the performance of the participants in tasks with anti-patterns. For example, participants with better technical skills had better results with source code without the anti-patterns Blob and Spaghetti Code.

To check if participants in different experiments are similar regarding their skills, we used the Mann-Whitney test. The null hypothesis was that the two samples come from the same distribution, which we would reject if the p-value is
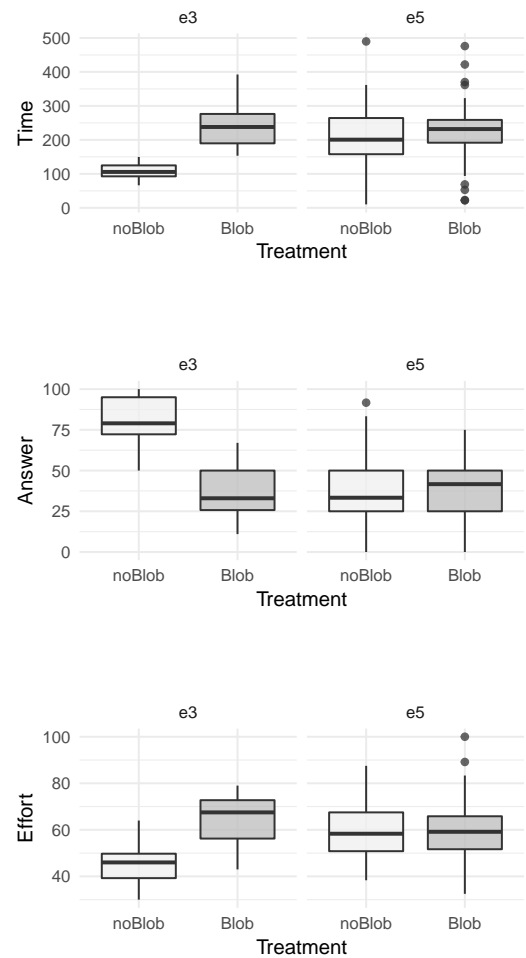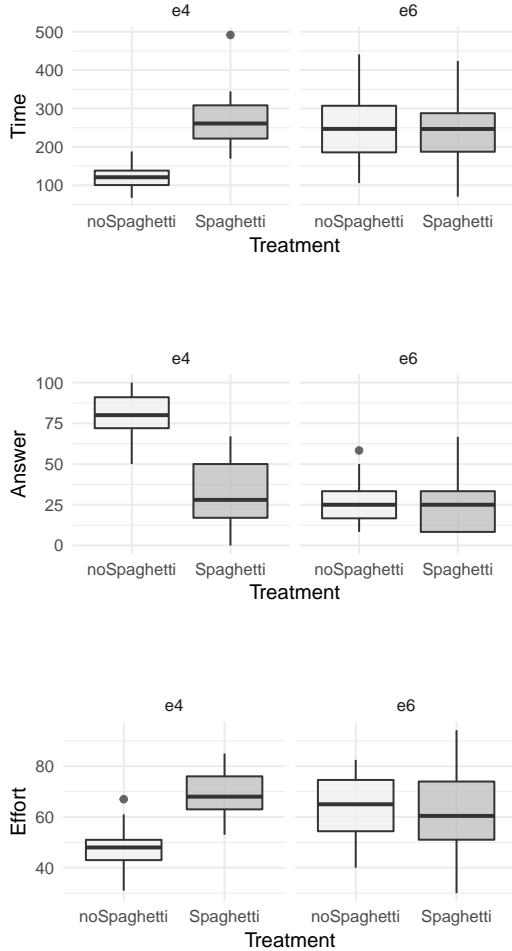

Figure 3: Boxplot of the exploratory analysis with dependent variables and the Blob anti-pattern (treatment) grouped by experiments.

less than 0.05 (5%). We also computed the effect sizes in both samples based on dominance matrices using Cliff's δ. Listing 2 is an example of the R script to applied the tests on Experiments #4 and #6 (Spaghetti Code) Java programming skills.
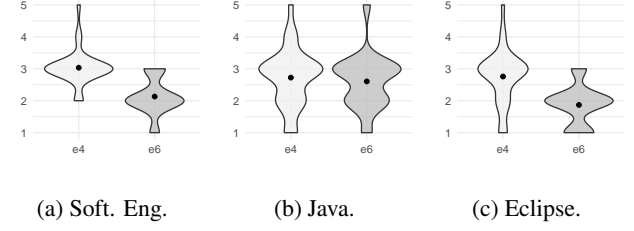
(a) Soft. Eng. (b) Java (c) Eclipse
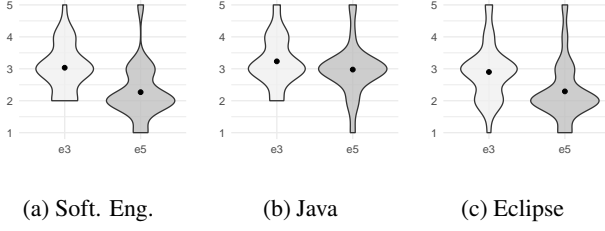
Figure 4: Violin plot showing the distribution of participants' expertise levels in *Software Engineering*, *Java*, and *Eclipse*, for Experiments #3 and #5, with Blob anti-pattern.

(a) Soft. Eng. (b) Java. (c) Eclipse.

Figure 6: Violin plots showing the distribution of participants' skills levels in *Software Engineering*, *Java*, and *Eclipse*, for Experiments #4 and #6, with the Spaghetti Code anti-pattern.

```
# Mann-Whitney test
wilcox.test(x = datasetExp4$Java, y =
    datasetExp6$Java, paired = FALSE)
# Cliff's Delta test
cliff.delta(datasetExp4$Java, datasetExp6$Java
    , conf.level=.95)
```

Listing 2: Example of the R code for the Mann-Whitney and Cliff's Delta test.

Table 11
Results of the Mann-Whitney and Cliff's δ tests. The values in bold font means that the null hypothesis was rejected and, therefore, that the samples are different.

A. P.	Exp.	Skill	P-value	Effect	Magnitude
Blob	3 & 4	SE	**0.00**	0.70	large
Blob	3 & 4	Java	0.15	0.17	small
Blob	3 & 4	Eclipse	**0.00**	0.42	medium
Spaghetti	4 & 6	SE	**0.00**	0.70	large
Spaghetti	4 & 6	Java	0.45	0.10	negligible
Spaghetti	4 & 6	Eclipse	**0.00**	0.59	large

Figure 5: Boxplot of the exploratory analysis with dependent variables and the Spaghetti Code anti-pattern (treatment) grouped by experiment.

Table 11 shows the results for each skill: p-value (Wilcoxon test), effect (Cliff's δ), and magnitude[11] (qualitative assessment of the magnitude of effect size). There is significant difference in the participants' skills regarding *Software Engineering* and *Java*, for both anti-patterns.

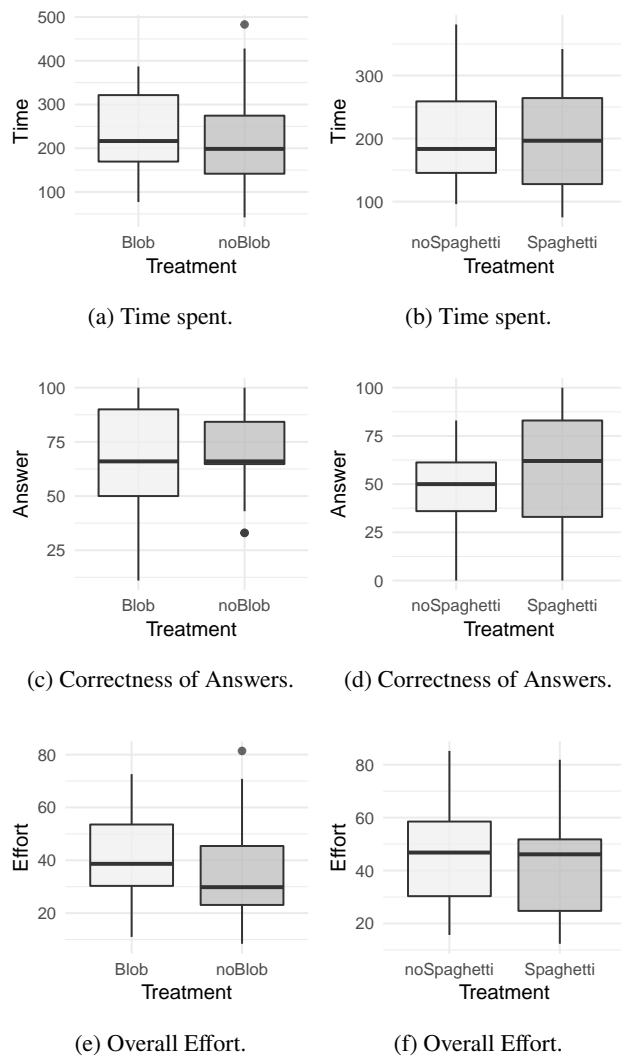[11]Magnitude is assessed using the thresholds: $|\delta| < 0.147$ "negligible",

4.3. Results from Abbes et al. [1]

Previous experiments by Abbes et al. [1], which used only one instance of the anti-patterns, concluded that there is no significant impact on the source code understandability. Yet, the combination of both anti-patterns, Blob and Spaghetti Code, did decrease understandability.

Figure 7 shows the boxplots of the results of Experiment #1 and #2 with Blob and Spaghetti Code anti-patterns. The difference regarding the mean is clear when compared with the new experiments shown in Figure 1 and Figure 2.

Nonetheless, the results of our new experiments, with two anti-patterns instead of only one, did change the outcome. The combinations of the anti-patterns from [1], Blob + Spaghetti Code, and the combinations of more than one occurrence of the same anti-patterns, Blob + Blob and Spaghetti Code + Spaghetti Code, harm the understandability of the source code for all the tested variables.

$|\delta| < 0.33$ "small", $|\delta| < 0.474$ "medium", otherwise "large".

(a) Time spent.   (b) Time spent.

(c) Correctness of Answers.   (d) Correctness of Answers.

(e) Overall Effort.   (f) Overall Effort.

Figure 7: Boxplot of the analysis with the dependent variables (time, answer, and effort) for Experiment #1 and #2 [1].

Summary: Results

The results of the *descriptive statistics*, boxplots, and LMM analysis method show that either anti-pattern Blob or Spaghetti Code, when present in a java source code, decreases the developers' productivity by increasing the time spent in the tasks, reducing the accuracy of their answers, and increasing their needed effort.

5. Discussion

The results from the experiments with the Blob anti-pattern show that the null hypotheses can be rejected for all dependent variables. Thus, we conclude that the Blob anti-pattern, when co-occurring twice in source code, impacts the participants' performances by increasing the time spent when performing their tasks, lowering the correctness of their an-

swers, and increasing their effort.

Similarly, results from the experiments with the Spaghetti Code anti-pattern indicate that the null hypotheses can be rejected for all dependent variables: when co-occurring twice in source code, it also impacts the participants' performances by increasing their time spent on the tasks, lowering the correctness of their answers, and increasing their efforts.

Time Spent: The time spent in any task is always a concern and usually is used as a metric of productivity, *e.g.,* to estimate how long it will take to implement a new feature or fix a bug. Our results show that the time spent could be reduced by 33% if the source code does not contain the Blob anti-pattern and by 39% without the Spaghetti Code anti-pattern.

Correctness of Answers: Developers often face source code that is hard to understand, especially when working with legacy code or third-party libraries. The poor understandability of the source code may lead to occurrences of bugs, as developers misunderstand the source code. Our results bring evidence that, when a source code contains two co-occurrences of the Blob anti-pattern or Spaghetti Code, the developers' accuracy could be reduced by 22% and 25%.

Participants' effort: The effort used to complete a task is also a metric commonly used to measure developers' productivity. The more effort a task demands, the harder it is. Our results show that the anti-patterns Blob and Spaghetti Code could increase by 10% developers' efforts.

Differences between Anti-patterns The difference is small between the anti-patterns used in this study. However, the Spaghetti Code has a slightly bigger impact on the understandability of the source code, resulting in about 6% more time spent and 3% less correct answers.

Differences among Participants When grouped by experiments, the results showed different outcomes. The difference in the participants' performance is clear in Experiments #3 and #4 (Canada) while is quite similar in the Experiments #5 and #6 (Italy). The former even has a contradictory better performance on effort regarding Spaghetti Code. The participants' expertise is similar for Java but different for Software Engineering and Eclipse. A first observation would be that skilled programmers work better with clean/organized code. However, we argue that we need more robust methods to model the participants' skills and measure their performance.

Other Consequences The previous extended work by Abbes et al. [1] demonstrated that a single occurrence of any anti-pattern, Blob or Spaghetti Code, does not affect source-code understanding. However, when both were mixed, understandability decreased. These latter results are confirmed by this present work, using two occurrences of each anti-pattern in the source code.

These previous observations raise the question whether the problem is the number of occurrences of anti-patterns,

not necessarily their types: Is Blob + Blob more impactful than Blob + Spaghetti Code? Future work is need to answer this question even if the presence of the anti-patterns and their effect on the participants' performance show the importance of well-designed source code.

Understanding the impact of Blob and Spaghetti Code anti-patterns on program comprehension 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 anti-patterns on the quality of software systems. For practitioners, our results provide concrete evidence that they should pay attention to software systems with high numbers of classes participating in anti-patterns, because these anti-patterns reduce understandability and, consequently, increase their systems' aging [35]. Our results also support *a posteriori* the removal of anti-patterns as early as possible from systems and, therefore, the importance and usefulness of anti-patterns detection techniques and related refactorings.

Summary: Discussion

The data provides preliminary evidence that anti-patterns do not affect developers with low skill level but a source code without anti-pattern improves the developers' performance. It thus confirms the recommendation of following good practices and patterns to keep the code "clean" and applying refactorings to maintain it, especially for the Spaghetti Code anti-pattern.

---

6. 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 [48].

6.1. 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 participants' performance. These measures are objective, even if small variations due to external factors, such as fatigue, could impact their values. We also use NASA TLX to measure the participants' effort. TLX is subjective by nature because participant were asked to rate their own efforts. Thus, it is possible that some participants did not answer truthfully.

The participants filled the questionnaire of skills by self-reporting/accessing their knowledge level, which is subjective and varied. Future research must address this issue in more details by using other techniques, for example, using eye-tracking tools.

Also, it might be possible that some occurrences of the Blob or Spaghetti Code anti-patterns are more potent in affecting source-code understandability than others. Therefore, two separate pieces of source code, both containing

some anti-patterns, may have different impact of participants' performance, which should be further studied.

6.2. Internal Validity

We identify two threats to the internal validity of our study: learning and instrumentation.

Learning threats should not affect our study for a specific experiment because each participant performed comprehension tasks on different systems with different questions for each system. We also took care to randomize the participants to avoid bias (*e.g.,* gender bias).

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

We strove to keep the systems/tasks as equivalent as possible, however, we accept somoe differences given the number of tasks used and the different systems.

6.3. Conclusion Validity

Conclusion validity threats concern the relation between the treatment and the outcome. We chose Linear Mixed Model approach to better isolate random effects, the participants in our case. Also, we added more types of analyses in the same data, as descriptive statistics, boxplots, violin plots, and Friedman test. We paid attention not to violate assumptions of the performed statistical tests as well not to blindly trust the LMM outcome.

6.4. External Validity

We performed our extended study on six different real systems belonging to different domains and with different sizes. Our design, *i.e.,* providing only on average 75 classes of each system to each participant, 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 a large code base.

We must consider that participants have different programming skills and the tasks may not suite them well. We mitigated this threat by measuring their expertise in Java and suggest that they are equivalent (see Figure 4 and Figure 6).

7. Related Work

We now discuss works related to anti-patterns. Table 12 shows the summary of the papers in each category.

Anti-patterns are poor solutions to recurrent design problems. Webster [47] wrote the first book related to anti-patterns in object-oriented development, which discussed political, conceptual, implementation, and quality-assurance problems. Fowler et al. [9] defined 22 code smells and suggested refactorings that developers can apply to remove the code smells. Mantyla [17] and Wake [46] proposed classifications for code smells. Brown et al. [5] described 40 anti-patterns, including the Blob and Spaghetti Code. Riel [36] defined 61 heuristics

Table 12
Related works grouped by research topic.

Anti-Pattern on...	Papers
Detection	[43, 24, 30, 18, 23, 16, 26]
Quality	[28, 6, 2]
Fault and–or change	[14, 15, 15, 13, 10, 31, 29]
S.L.R.	[37, 41]
Maintenance	[12, 8, 49, 40]
Refactoring and understand.	[3]
Origin in projects	[44, 32]
Developers reaction	[34]
Mobile energy consumption	[33]
Impact in design	[27]

characterizing good object-oriented programming to assess the quality of systems manually and improve their designs and implementations. These books provide in-depth views on heuristics, code smells, and anti-patterns for industrial and academic audiences. In program comprehension, anti-patterns concern the design level, code smells the implementation level, and heuristics describe idioms.

Several approaches to specify and detect code smells and anti-patterns exist in the literature. They range from manual approaches, based on inspection techniques [43], to metric-based heuristics [18, 24, 30], using rules and thresholds on various metrics [23] or Bayesian belief networks [16]. Commercial tools such as Borland Together[12] and AI Reviewer[13] also offer the automatic detection of some code smells and anti-patterns.

Many works investigated the impact of anti-patterns on software quality. Olbrich et al. [28] analyzed anti-patterns in Lucene and Xerces and found that classes subjected to the Blob and Shotgun Surgery anti-patterns have a higher change frequency than other classes. Similarly, Chatzigeorgiou and Manakos [6] investigated the evolution of classes with the Long Method, Feature Envy, and State Checking anti-patterns in two open-source systems and reported that a significant proportion of these anti-patterns were introduced during the addition of new methods in the systems. They also observed that these anti-patterns remain in the systems for long periods of time and that their removals are often due to adaptive maintenance rather than refactorings. Bavota et al. [2] who analyzed 12,922 refactorings in three open-source systems also reported that refactorings are rarely aimed at removing anti-patterns.

Khomh et al. [14] investigated the impact of several code smells on the change-proneness of classes in Azureus and Eclipse. They showed that, in general, the likelihood for classes with code smells to change is higher than classes without. Khomh et al. [15] investigated the relation between the presence of anti-patterns and class change- and fault-proneness. They detected 13 anti-patterns in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino, and analyzed the likelihood that a class with an anti-pattern changes in the fu-

---

ture, in particular to fix a fault. They concluded that classes participating in anti-patterns are significantly more likely to be changed and to be involved in fault-fixing changes than other classes.

Khomh et al. [15] also investigated the kind of changes experienced by classes with anti-patterns. They considered two types of changes: structural and non-structural. Structural changes are changes that would alter a class interface while non-structural changes are changes to method bodies. They concluded that structural changes are more likely to occur in classes participating in anti-patterns. Jaafar et al. [13] investigated the change- and fault-proneness of classes sharing a static or co-change dependency with an anti-pattern class and found that classes that are dependent on an anti-pattern class are more fault-prone than others.

Hall et al. [10] investigated the relationship between faults and five anti-patterns (Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man) in three software systems (Eclipse, ArgoUML, and Apache Commons) and showed that some smells do indicate fault-prone code but the effect that these smells have on faults is small. They also found that some smells even lead to a small reduction in fault-proneness.

Santos et al. [37] carried out a systematic literature review on code smells impact in software development. They reported a strong correlation between smells and software attributes, like effort in maintenance. They observed a low human agreement on smell detection. They concluded "that there are not strong evidences motivating the adoption of the code smells to evaluate the design quality.".

Previous work also investigated the impact of anti-patterns on software maintenance activities. Ignatios et al. [12], Deligiannis et al. [8] proposed the first quantitative study of the impact of anti-patterns on software development and maintenance activities. They performed a controlled experiment with 20 students and two systems to understand the impact of Blob classes on the understandability and maintainability of software systems. They reported that Blob classes affect the evolution of the system designs and the participants' use of inheritance. They did not assess the impact of Blob classes on the participants' understanding and their ability to perform successfully comprehension tasks on these systems.

Yamashita and Moonen [49] investigated the extent to which code smells affect software maintainability and observed that code-smell definitions alone do not help developers evaluate the maintainability of a software system. They concluded on the need to combine different analysis approaches to achieve more complete and accurate evaluations of the overall maintainability of a software system.

Bois et al. [3] showed through a controlled experiment with 63 graduate students that the decomposition of Blob classes into a number of collaborating classes using well-known refactorings improves their understandability. They asked students to perform simple maintenance tasks on God classes and their decompositions. They found that the students had more difficulties understanding the original Blob classes than certain decompositions. However, they did not

report any objective notion of "optimal comprehensibility".

Sobrinho et al. [41] performed a literature review of code smell research conducted between 1990 and 2017. The results show that Duplicate Code is the most studied anti-pattern while the other types are not well spread, also, this research topic has been rising and attracting different researchers over the time. There are multiple interpretations concerning anti-patterns concept and the lack of open science (standards and open data/tools) lead to contradictory results. There is a big concentration of papers regarding Duplicate Code in some venues, which can explain the gap between the research groups.

Tufano et al. [44] analyzed 200 open source projects to investigate "when and why code smells are introduced in software projects, how long they survive, and how they are removed". After analyzing more than 10,000 commits, they argued that smells are introduced when the artifact is created and not when it evolves. Moreover, 80% of these smells remains in the source code, while only 9% are removed through direct refactoring.

Palomba et al. [31] assessed the impact of code smells on change- and fault-proneness by empirically analyzing 395 releases of 30 Java open source projects. Among the 17,350 instances of 13 different code smell types, they found that (1) the most diffused smells are related to size, like Long Method, Spaghetti Code; (2) classes affected by code smells are more suitable to changes and faults; (3) removing code smells does affect change-proneness in the class but it is beneficial to the occurrence of changes.

Palomba et al. [32] investigated the nature of code smell co-occurrences in 395 projects. The results showed that 59% of the smelly classes are affected by more than one code smell. Among them, there are six code smell types frequently co-occurring together, being the *Message Chain* the most common. Still, the method-level smells may be the root cause of the class-level ones. Finally, the co-occurring smells tend to be removed together.

Palomba et al. [34] conducted a quantitative and qualitative study to check how developers react to smells depending of the tool used to its detection, that is, textual-base or structural-base. Among the results, they reported that both types are considered equally harmful for the project.

Palomba et al. [33] conducted an experiment with 9 Android-specific code smells on 60 Android apps to assess its energy consumption. The results showed the existence of four specific energy code smells which, when refactored, increase the energy efficiency.

Nucci et al. [26] replicate a study using 32 machine-learning techniques to detect four types of code smell by extending the dataset. Results showed differences in the previous output, which has 95% accuracy, being 90% less accurate.

Olbrich et al. [29] analyzed historical data from 7 to 10 years of the development of three open-source software systems. They found that after controlling for differences in class size, Brain and God Classes, are less change- and fault-prone than other classes.

Sjøberg et al. [40] hired six Java developers which modified 298 Java files in different tasks in the four systems while measuring the time spent. They essentially found that code smells do not have a consistent effect on maintenance effort if differences in file size are controlled for.

Oizumi et al. [27] analyzed more than 2,200 agglomerations (inter-related code anomalies) found in seven software systems of different sizes and from different domains. Their findings suggest that a single code smell seldom indicates a design problem, but if two or more smells co-occur, they indicate design problems with high accuracy.

> **Summary: Related Works**
>
> Table 12 shows that the literature on anti-patterns has its roots in detection and fault/change proneness. Few studies investigated the impact of anti-patterns on the understandability. In this paper, we built on previous work and propose experiments assessing the impact of the Blob and Spaghetti Code on source-code understandability.

8. Conclusion and Future Work

Anti-patterns were conjectured in the literature to negatively impact the quality of systems. Some studies empirically investigated the impact of anti-patterns on program comprehension. We revisited the studies on the impact of the Blob and Spaghetti Code on program comprehension by complementing a previous work by Abbes et al. [1].

We designed and conducted two replications with 133 participants and 372 program-comprehension tasks to assess the impact of combinations of occurrences of the Blob and Spaghetti Code anti-patterns. We measured developers' performances using: (1) the time that they spent performing their tasks; (2) their percentages of correct answers; and, (3) the NASA task load index for their effort.

We showed, in complement to the results of the previous work, that either two instances of the Blob anti-pattern or two instances Spaghetti Code anti-pattern negatively impact the understandability of source code by increasing the time spent by participants in performing the tasks, by reducing the correctness of their answers, and by increasing their efforts.

Consequently, developers should be wary with growing numbers of Blob and Spaghetti Code anti-patterns in their systems and, therefore, refactor these occurrences out of the source code of their systems. Researchers should study *combinations* of anti-patterns (and other practices, like code smells) rather than single anti-patterns, one at a time.

Future work includes investigating the relation between the number of occurrences of anti-patterns and understandability. We plan to investigate how to quantify the impact of occurrences of an anti-pattern on understandability. We also plan to replicate these studies in other contexts, with other participants, other anti-patterns, and other systems.

Acknowledgements

References

[1] Abbes, M., Khomh, F., Gueheneuc, Y.G., Antoniol, G., 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: Mens, T., Kanellopoulos, Y., Winter, A. (Eds.), 2011 15th European Conference on Software Maintenance and Reengineering, IEEE. pp. 181–190. doi:10.1109/csmr.2011.24.

[2] Bavota, G., Lucia, A.D., Penta, M.D., Oliveto, R., Palomba, F., 2015. An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software 107, 1–14. doi:10.1016/j.jss.2015.05.024.

[3] Bois, B.D., Demeyer, S., Verelst, J., Mens, T., Temmerman, M., 2006. Does god class decomposition affect comprehensibility?, in: IASTED Conf. on Software Engineering, pp. 26–49.

[4] Brooks, R., 1983. Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies 18, 543 – 554. URL: http://www.sciencedirect.com/science/article/pii/S0020737383800315, doi:https://doi.org/10.1016/S0020-7373(83)80031-5.

[5] Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J., 1998. Antipatterns: Refactoring Software, Architectures, and Projects in Crisis. 1st ed., John Wiley & Sons.

[6] Chatzigeorgiou, A., Manakos, A., 2010. Investigating the evolution of bad smells in object-oriented code, in: 2010 Seventh International Conference on the Quality of Information and Communications Technology, IEEE, Washington, DC, USA. pp. 106–115. doi:10.1109/quatic.2010.16.

[7] D'Ambros, M., Bacchelli, A., Lanza, M., 2010. On the impact of design flaws on software defects, in: 2010 10th International Conference on Quality Software, IEEE, Washington, DC, USA. pp. 23–31. doi:10.1109/qsic.2010.58.

[8] Deligiannis, I., Stamelos, I., Angelis, L., Roumeliotis, M., Shepperd, M., 2004. A controlled experiment investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software 72, 129–143. doi:10.1016/S0164-1212(03)00240-1.

[9] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 2012. Refactoring: Improving the Design of Existing Code. Addison-Wesley Object Technology Series, Pearson Education.

[10] Hall, T., Zhang, M., Bowes, D., Sun, Y., 2014. Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology 23, 1–39. doi:10.1145/2629648.

[11] Hart, S.G., Stavenland, L.E., 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research, in: Hancock, P.A., Meshkati, N. (Eds.), Human Mental Workload. Elsevier. chapter 7, pp. 139–183. doi:10.1016/s0166-4115(08)62386-9.

[12] Ignatios, D., Martin, S., Manos, R., Ioannis, S., 2004. An empirical investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software 72.

[13] Jaafar, F., Guéhéneuc, Y.G., Hamel, S., Khomh, F., Zulkernine, M., 2015. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. Empirical Software Engineering 21, 896–931. doi:10.1007/s10664-015-9361-0.

[14] Khomh, F., Penta, M.D., Gueheneuc, Y.G., 2009a. An exploratory study of the impact of code smells on software change-proneness, in: 2009 16th Working Conference on Reverse Engineering, IEEE. pp. 75–84. doi:10.1109/wcre.2009.28.

[15] Khomh, F., Penta, M.D., Guéhéneuc, Y.G., Antoniol, G., 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empirical Software Engineering 17, 243–275. doi:10.1007/s10664-011-9171-y.

[16] Khomh, F., Vaucher, S., Guéhéneuc, Y., Sahraoui, H., 2009b. A bayesian approach for the detection of code and design smells, in: 2009 Ninth International Conference on Quality Software, pp. 305–314. doi:10.1109/QSIC.2009.47.

[17] Mantyla, M., 2003. Bad Smells in Software - a Taxonomy and an Empirical Study. Ph.D. thesis. Helsinki University of Technology.

[18] Marinescu, R., 2004. Detection strategies: metrics-based rules for detecting design flaws, in: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., pp. 350–359. doi:10.1109/ICSM.2004.1357820.

[19] Martin, R., Rabaey, J., Chandrakasan, A., Nikolic, B., 2003. Agile Software Development: Principles, Patterns, and Practices. Alan Apt series, Pearson Education. URL: https://books.google.ca/books?id=0HYhAQAAIAAJ.

[20] Mayrhauser, A.V., Vans, A., 1995. Program comprehension during software maintenance and evolution. Computer 28, 44–55. doi:10.1109/2.402076.

[21] Moha, N., Gueheneuc, Y., Duchien, L., Meur, A.L., 2010. Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering 36, 20–36. doi:10.1109/TSE.2009.50.

[22] Moha, N., Guéhéneuc, Y.G., 2005. 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, pp. 1–7.

[23] Moha, N., Guéhéneuc, Y.G., Duchien, L., Meur, A.F.L., 2009. DECOR: A method for the specification and detection of code and design smells. Transactions on Software Engineering (TSE) doi:10.1109/tse.2009.50. 16 pages.

[24] Munro, M., 2005. Product metrics for automatic identification of "bad smell" design problems in java source-code, in: 11th IEEE International Software Metrics Symposium (METRICS'05), IEEE. pp. 9–18. doi:10.1109/metrics.2005.38.

[25] NASA, 1988. NASA TASK LOAD INDEX (TLX) v. 1.0. Moffett Field, California.

[26] Nucci, D.D., Palomba, F., Tamburri, D.A., Serebrenik, A., Lucia, A.D., 2018. Detecting code smells using machine learning techniques: Are we there yet?, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 612–621. doi:10.1109/saner.2018.8330266.

[27] Oizumi, W., Garcia, A., d. S. Sousa, L., Cafeo, B., Zhao, Y., 2016. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 440–451. doi:10.1145/2884781.2884868.

[28] Olbrich, S., Cruzes, D.S., Basili, V., Zazworka, N., 2009. The evolution and impact of code smells: A case study of two open source systems, in: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 390–400. doi:10.1109/ESEM.2009.5314231.

[29] Olbrich, S.M., Cruzes, D.S., Sjøberg, D.I.K., 2010. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems, in: 2010 IEEE International Conference on Software Maintenance, pp. 1–10. doi:10.1109/ICSM.2010.5609564.

[30] Oliveto, R., Khomh, F., Antoniol, G., Gueheneuc, Y., 2010. Numerical signatures of antipatterns: An approach based on b-splines, in: 2010 14th European Conference on Software Maintenance and Reengineering, pp. 248–251. doi:10.1109/CSMR.2010.47.

[31] Palomba, F., Bavota, G., Penta, M.D., Fasano, F., Oliveto, R., Lucia, A.D., 2017. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering 23, 1188–1221. doi:10.1007/s10664-017-9535-z.

[32] Palomba, F., Bavota, G., Penta, M.D., Fasano, F., Oliveto, R., Lucia, A.D., 2018a. A large-scale empirical study on the lifecycle of code smell co-occurrences. Information and Software Technology 99, 1–10. doi:10.1016/j.infsof.2018.02.004.

[33] Palomba, F., Nucci, D.D., Panichella, A., Zaidman, A., Lucia, A.D., 2019. On the impact of code smells on the energy consumption of mobile applications. Information and Software Technology 105, 43–55. doi:10.1016/j.infsof.2018.08.004.

[34] Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., Lucia, A.D., 2018b. The scent of a smell: An extensive comparison between textual and structural smells. IEEE Transactions on Software Engineering 44, 977–1000. doi:10.1109/tse.2017.2752171.

[35] Parnas, D.L., 1994. Software aging, in: Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA. pp. 279–287.

[36] Riel, A.J., 1996. Object-Oriented Design Heuristics. 1st ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[37] Santos, J.A.M., Rocha-Junior, J.B., Prates, L.C.L., do Nascimento, R.S., Freitas, M.F., de Mendonça, M.G., 2018. A systematic review on the code smell effect. Journal of Systems and Software 144, 450–477. doi:10.1016/j.jss.2018.07.035.

[38] Shapiro, S.S., Wilk, M.B., 1965. An analysis of variance test for normality (complete samples). Biometrika 52, 591–611. doi:10.1093/biomet/52.3-4.591.

[39] Sillito, J., Murphy, G.C., De Volder, K., 2008. Asking and answering questions during a programming change task. IEEE Transactions on Software Engineering 34, 434–451. doi:10.1109/TSE.2008.26.

[40] Sjøberg, D.I.K., Yamashita, A., Anda, B.C.D., Mockus, A., Dybå, T., 2013. Quantifying the effect of code smells on maintenance effort. IEEE Transactions on Software Engineering 39, 1144–1156. doi:10.1109/TSE.2012.89.

[41] Sobrinho, E.V., Lucia, A.D., d. Maia, M., 2018. A systematic literature review on bad smells — 5 w's: which, when, what, who, where. IEEE Transactions on Software Engineering , 1–1doi:10.1109/tse.2018.2880977.

[42] Soh, Z., Yamashita, A., Khomh, F., Gueheneuc, Y.G., 2016. Do code smells impact the effort of different maintenance programming activities?, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE. pp. 393–402. doi:10.1109/saner.2016.103.

[43] Travassos, G., Shull, F., Fredericks, M., Basili, V.R., 1999. Detecting defects in object-oriented designs: Using reading techniques to increase software quality, in: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM, New York, NY, USA. pp. 47–56. doi:10.1145/320384.320389.

[44] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M.D., Lucia, A.D., Poshyvanyk, D., 2017. When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering 43, 1063–1088. doi:10.1109/tse.2017.2653105.

[45] Vegas, S., Apa, C., Juristo, N., 2016. Crossover designs in software engineering experiments: Benefits and perils. IEEE Transactions on Software Engineering 42, 120–135. doi:10.1109/tse.2015.2467378.

[46] Wake, W., 2004. Refactoring Workbook. Addison-wesley Object Technology Series, Addison-Wesley.

[47] Webster, B.F., 1995. Pitfalls of Object-Oriented Development. 1st ed., M & T Books.

[48] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A., 2012. Experimentation in Software Engineering. Springer Publishing Company, Incorporated. doi:10.1007/978-3-642-29044-2.

[49] Yamashita, A., Moonen, L., 2012. Do code smells reflect important maintainability aspects?, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 306–315. doi:10.1109/ICSM.2012.6405287.