Revealing Design Problems in Stinky Code

A Mixed-Method Study

Willian Oizumi[1], Leonardo Sousa[1], Alessandro Garcia[1], Roberto Oliveira[1], Anderson Oliveira[1], O.I. Anne Benedicte Agbachi[1], Carlos Lucena[1]
[1]Opus Research Group, Informatics Department, PUC-Rio, Rio de Janeiro, Brazil
{woizumi, lsousa, afgarcia, rfelicio, aoliveira, oagbachi, lucena}@inf.puc-rio.br

ABSTRACT

Developers often have to locate design problems in the source code. Several types of design problem may manifest as code smells in the program. A code smell is a source code structure that may reveal a partial hint about the manifestation of a design problem. Recent studies suggest that developers should ignore smells occurring in isolation in a program location. Instead, they should focus on analyzing stinkier code, i.e. program locations – e.g., a class or a hierarchy – affected by multiple smells. The stinkier a program location is, more likely it contains a design problem. However, there is limited understanding if developers can effectively identify a design problem in stinkier code. Developers may struggle to make a meaning out of inter-related smells affecting the same program location. To address this matter, we applied a mixed-method approach to analyze if and how developers can effectively find design problems when reflecting upon stinky code – i.e., a program location affected by multiple smells. We performed an experiment and a survey with 11 professionals. Surprisingly, our analysis revealed that only 36.36% of the developers found more design problems when explicitly reasoning about multiple smells as compared to single smells. On the other hand, 63.63% of the developers reported much lesser false positives. Developers reported that analyses of stinky code scattered in class hierarchies or packages is often difficult, time consuming, and requires proper visualization support. Moreover, it remains time-consuming to discard stinky program locations that do not represent design problems.

KEYWORDS

design problem, software design, code smell, agglomeration

1 INTRODUCTION

The identification of design problems in the source code is not a trivial task [6, 31]. A code smell is a structure in the source code that may provide developers with a partial hint about the manifestation of a design problem [10]. Examples of code smells are *God Class*, *Feature Envy* and *Brain Method*. However, the occurrence of a single smell in isolation in a program often does not represent a design problem [18, 25, 26]. Recent studies reveal that design problems are much more often located in stinkier program locations, i.e., a class, a hierarchy or a package affected by multiple smells [1, 18, 25, 26, 38, 39]. For instance, a *Fat Interface* [19] is a design problem that often manifests as multiple smells in a program, affecting various classes that implement, extend and use the interface in a program [26].

The stinkier a program location is, more likely it contains a design problem [25, 26]. In fact, developers tend to focus on refactoring program locations with a high density of code smells, and ignore those locations affected by a single smell [4, 5]. However, there is limited understanding if developers can effectively identify design problems in stinkier code, i.e. program locations – e.g., a class or a hierarchy – affected by multiple smells. Indeed, existing techniques tend to focus on the detection and visualization of each single smell [9, 23, 27, 35]. They do not offer a summarized view of inter-related smells affecting a program location [26]. Moreover, previous studies focus on simply analyzing the correlation between design problems and code smells [17, 26]. They have not investigated if and how developers are indeed effective in the task of finding design problems in stinkier code.

Therefore, we do not know whether the analysis of multiple smells actually provides better precision for the identification of design problems. Developers may struggle to make a meaning out of inter-related smells affecting the same program location. To address this matter, we designed and executed a multi-method study with 11 professional developers. Our goal was to analyze if developers can effectively find design problems when reflecting upon multiple smells affecting program locations. Developers were asked to identify design problems in this context. Our study comprised both quantitative and qualitative analyses.

For the quantitative analysis, we compared the precision of the developers with a baseline, i.e. situations where only single smells were given to them. As we want to assess if multiple smells can help developers to reveal more design problems than single smells, we divided the developers into two groups. In the first group, we asked them to identify design problems through the analysis of stinky program locations. In the second group, we asked them to identify design problems with the analysis of single smells. After that, we inverted the groups, and we asked them to repeat the identification of design problems in a second system. In each identification task, we used the group that identified design problems with single smells as the control group. Thus, we could use the control group to measure if the analysis of stinkier program locations can improve the precision of design problem identification.

In the qualitative analysis, we performed a detailed and systematic evaluation through: (1) the careful observation of participants during the experiment execution, and (2) the conduction of a post-experiment survey. The objective of this analysis was to identify the main advantages and barriers of reflecting upon multiple smells along the task of identifying design problems. The outcomes of this analysis helped us to better understand ways to improve support for the identification of design problems.

By triangulating the results of both analyses, we noticed that 36.36% of the developers found more design problems when explicitly reasoning about multiple smells. We found that the understanding of complex stinky structures helped to confirm the occurrence of non-trivial design problems, such as *Scattered Concern* [12]. Furthermore, we found that 63.63% of the developers reported much less false positives when analyzing multiple smells than when single smells. Thus, developers that consider stinky program locations, instead of isolated smelly code, could identify design problems with higher precision. However, this study also showed that developers need better support to analyze stinky program locations for revealing design problems. We observed that the analysis of stinky code may be difficult and time consuming. For instance, a prioritization algorithm is required so that developers do not waste time analyzing stinky program locations not related to design problems. In addition, developers need better visualization support to analyze complex stinky code scattered across class hierarchies or packages.

The remainder of this paper is organized as follow. Section 2 introduces basic concepts and presents an illustrative example. Section 3 describes the settings of our study. Section 4 summarizes the main results. Sections 5 and 6 present the related work and the threats to validity, respectively. Finally, Section 7 concludes the paper.

Table 1: Types of code smell

Type	Description
God Class	Long and complex class that centralizes the intelligence of the system
Brain Method	Long and complex method that centralizes the intelligence of a class
Data Class	Class that contains data but not behavior related to the data
Disperse Coupling	The case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes
Feature Envy	Method that calls more methods of a single external class than the internal methods of its own inner class
Intensive Coupling	When a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes
Refused Parent Bequest	Subclass that does not use the protected methods of its superclass
Shotgun Surgery	This anomaly is evident when you must change lots of pieces of code in different places simply to add a new or extended piece of behavior
Tradition Breaker	Subclass that provides a large set of services that are unrelated to services provided by the superclass

2 CONTEXTUALIZATION

This section is organized into two subsections. Section 2.1 presents basic concepts. Section 2.2 brings up an illustrative example of analyzing stinky code to identify design problems.

2.1 Basic Concepts

Design Problem. A design problem is a design characteristic that negatively impacts maintainability [31]. Design problems affect program locations like packages, interfaces, hierarchies, classes and other structures that are relevant for the design of the system [3]. Examples of design problems are *Scattered Concern* [12] and *Fat Interface* [19]. The description of the 8 types of design problems considered in our study is available in our complementary material [7]. We opted by selecting these design problems since: (i) they are often considered as critical in the systems [26] chosen in our experiment, and (ii) other studies haven shown the relation between such design problems and code smells [15–18, 26].

Smelly Code. Code smell is a recurring micro structure in the source code that may indicate the manifestation of a design problem [10]. A design problem can manifest itself in a program by affecting multiple source code locations. Each of these locations are called here *smelly code*. Thus, the developers can analyze the smelly code to identify a design problem. There are several types of code smell, which may affect a method, a class or a hierarchy. In this paper, we used nine types of code smell, as described in Table 1. These types of smell were considered in this study as they occur in the systems of our experiment (Section 3.3).

Stinky Program Location. Developers can rely on the analysis of code smells to identify design problems [14, 17, 30]. In fact, recent studies [1, 18, 26, 39] suggest that the stinkier a program location is, the more likely it is to be affected by a design problem. Stinky code is the manifestation of multiple code smells in a program location. In this paper, we are especially interested in stinky code indicated by *smell agglomerations* [26]. A smell agglomeration is a group of inter-related code smells affecting the same program location, such as a method, a class, a hierarchy or a package [26]. Thus, the agglomeration is determined in the program by the co-occurrence of two or more code smells in the same method, class, hierarchy or package (or component). For code smells that co-occur in the three last cases, we only consider they are part of a (class-, hierarchy- or package-level) agglomeration if they are syntactically related [26]. For instance, two classes can be related through structural relationships in the program, such as method calls and inheritance relationships. In this paper, we considered five categories of agglomeration, namely *intra-method, intra-class, hierarchical, and intra-component* [26]. For instance, a method that contains several code smells represents an intra-method agglomeration. A full description of the categories of agglomeration considered in our study is available in our complementary material [7]. The agglomerations used in this study were detected with the Organic tool [24]. The Organic tool is a plug-in developed for the Eclipse IDE.

2.2 Identifying Design Problem in Stinky Code

As explained in previous section, the identification of design problems can be based on code smells. For instance, let us consider the example illustrated in Figure 1. This figure presents some classes that belong to the *Workflow Manager* substystem – a subsystem of the Apache OODT (Object Oriented Data Technology) system [20]. It is responsible for description, execution, and monitoring of workflows. Supposing that a developer is in charge of identifying design problems in the Workflow Manager. She can rely on the analysis of code smells to spot program locations that may contain a design problem. If she is analyzing the repository package, she

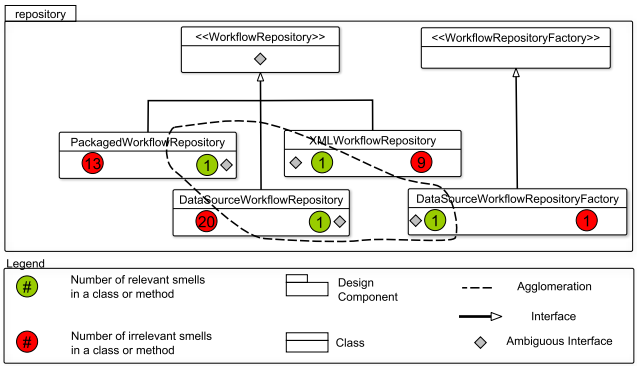

Figure 1: Example of agglomeration in the Workflow system

will notice that this package contains several code smells as indicated by a smell agglomeration. This agglomeration is formed by 4 instances of the *Feature Envy* smell. As illustrated by Figure 1, each of the *Feature Envy* occurrences affects a different class. In this case, 3 classes implement the *WorkflowRepository* interface. When the developer analyze these classes based on the *Feature Envy* smell, she will realize that these classes contain the smell because one of their methods is more interested in other classes than in its own hosting class. This happens because these methods are forced to implement a method that was defined in the *WorkflowRepository* interface. That is, the smells in the agglomeration are indicating that (the corresponding method in) the interface may contain a design problem. In fact, this "forced implementation" becomes a problem because these methods are implementing a concern that should not have been implemented in their hosting classes. That happens because of the fact that the *WorkflowRepository* interface processes multiple services; thus, any class that implements this interface needs to handle more services than it actually should have to.

In this example, the developer knows that the code smells in the agglomeration have the same type (*Feature Envy*). Also, she knows that 3 classes affected by the code smells implement the same interface, as reified in a *hierarchical* agglomeration. This interface, in its turn, seems to provide non-cohesive services. Thus, the developer can infer that a design problem, called *Ambiguous Interface*, is affecting the *WorkflowRepostiory* interface. On the other hand, if she did not reflect upon the code smell agglomeration, it would be harder to her to identify the same design problem. One of the reasons is the number of code smells spread over the 6 classes and 2 interfaces within the package. Although the package contains only 8 classes (Figure 1 only shows some of them), it has more than 50 code smells. Thus, she has to analyze many smelly code snippets in order to discard, postpone or further consider them in the identification of design problems.

Let us assume that the developer only reasons about each code smell in isolation to identify the design problem, i.e., without taking into consideration smell relationships in an agglomeration. Thus, she can choose to analyze the *DataSourceWorkflowRepository* class first because the class contains the highest number of smells in the package. Analyzing the 21 instances of code smells in the class, the developer will notice that the class has smells related to high coupling with other classes (*Intensive Coupling* and *Dispersed Coupling*), low cohesion (*Feature Envy*), and overload of responsibilities (*God Class*). However, all these smells may indicate different problems. Thus, she has to extend the analysis to other classes in order to gather more information that can potentially indicate a design problem. Unfortunately, the other classes also have different instances of code smells, and these instances may not be related to any design problem. Therefore, the developer can face difficulties to find the relevant code smells that can help him to identify a design problem. Thus, the analysis of stinky program locations, as revealed by agglomerations, seems to be a better strategy. However, there is limited empirical understanding about this phenomenon.

3 STUDY PLANNING

This section contains the settings of this study. Here, we present the research questions, empirical procedures and other details about our quantitative and qualitative analysis.

3.1 Research Questions

Previous studies suggest code smell agglomerations are consistent indicators of design problems [26]. However, there is a need to investigate whether developers can indeed identify design problems when exploring smell agglomerations. In order to address this matter, we defined two research questions. The first one is presented as follow:

RQ1. Does the use of agglomerations improve the precision of developers in identifying design problems?

Research question RQ1 allows us to analyze whether code smell agglomerations help developers to identify design problems with high precision. To answer this question, we conducted a controlled experiment with 11 professional developers. In this context, precision is measured based on the percentage of true positives indicated by the developers – i.e., the percentage of correctly identified design problems. Precision is an important aspect of the identification task. Through the correct identification of design problems, developers are able to optimize their work by solving problems that really impact design. On the other hand, the lack of precision would lead software development teams to spend time and budget with irrelevant tasks. For example, in companies adept to code review practices [21], the lack of precision can lead developers to waste time on refactoring tasks that do not contribute to system maintainability. The precise identification of design problems is also important in open source projects. For instance, the contributions of eventual collaborators are often rejected by core developers due to the presence of design problems [29]. Therefore, in this case, a lack of precision could lead core developers to reject relevant contributions due to "false design problems".

In this study, we did not measure recall because of the high number of design problems in the analyzed systems. Together with the system's original developers, we created a ground truth of design problems (Section 3.5) with more than 150 instances of design problems. Hence, it would be impracticable for participants to find all the design problems in the system due to the time constraints in the study (45 minutes). Consequently, they were expected to reach a low recall value. Therefore, we focused on the precision.

In order to measure if there was an improvement or not in the precision, we are comparing the participants using agglomerations with a control group. The control group comprises of participants identifying design problems without agglomerations, but only with (non-agglomerated) code smells. Thus, we analyzed the list of design problems identified by the participants firstly. In this analysis, we use a ground truth to confirm or refute each design problem indicated by participants. Then, we compared the number of false positives and true positives produced with the code smell agglomerations against the number of false and true positives produced by the control group.

Someone could assume that developers would often benefit from the use of agglomerations in their quest for finding design problems. However, it is through the analysis of RQ1 that we will be able to verify if developers can correctly identify more design problems using smell agglomerations. Regardless of the result, another question that should be investigated concerns how to better support developers in exploring smell agglomerations. Even though a previous study [26] has shown the strong relation between design problems and code smells within an agglomeration, we do not know whether and how the identification of design problems with agglomerations can be improved. The following question address this matter.

RQ2. How can the identification of design problems with code smell agglomerations be improved?

This question was addressed by conducting a qualitative analysis. This analysis was based on the observation of participants during the experiment and based on a post-experiment survey (Section 3.6). This analysis is necessary because it provides a complementary perspective on the identification of design problems with agglomerations. We could reveal advantages and barriers on the use of smell agglomerations. As reported in Section 4, the combination of quantitative and qualitative analysis helped us to draw more well grounded conclusions.

3.2 Experiment Procedures

We applied a *quasi-experiment* [28] in order to perform our study. A *quasi-experiment* is an experiment in which the units or groups are not assigned to conditions randomly. In our study, we could not select participants randomly because we need to ensure that they meet the requirements described in Section 3.3. The experiment was conducted individually with each participant. They had to perform the experiment in two steps with four tasks in each one. Both steps comprise the same set of tasks; the only difference between the steps was regarding the usage of agglomerations.

As explained before, we need to compare developers using agglomeration with a control group. This comparison allowed us to verify if there was an improvement in the precision when developers use agglomerations. Hence, the control group comprises the developers that had to identify design problems with a list of non-agglomerated code smells. Thus, we divided the participants into two groups. The first group would identify design problems using agglomerations in the first step. After that, they would identify design problems using a list of non-agglomerated code smells in the second step. The second group of participants would make the identification inversely: using the non-agglomerated code smells in the first step and, then, using the agglomerations in the second step.

Table 2: Combinations of groups, projects and steps

	Step 1		Step 2	
Arrange	**Group**	**Project**	**Group**	**Project**
1	Agglomeration	Project 1	Control	Project 2
2	Agglomeration	Project 2	Control	Project 1
3	Control	Project 1	Agglomeration	Project 2
4	Control	Project 2	Agglomeration	Project 1

Thus, in each step, we have two groups of participants: a group using agglomerations and a control group.

As each participant identifies design problems twice (first and second step), we had to select two software projects. Thus, each participant could identify design problems using a different project in both steps. Another reason for providing two software projects is to avoid bias with the learning curve. For example, supposing that the participant uses the same project in both steps. She could find more problems in the second step than in the first step. That could happen because she can identify in the second step the same problems that she identified in the first step, plus other design problems identified only in the second step. This increase in the number of design problems found in the second step would not be due to the use of agglomerations, but rather due to the knowledge acquired by the participant.

There are four possible combinations with the participants based on the distribution between steps and software projects. Therefore, all participants were divided into four groups equally to promote a fair comparison. Table 2 presents the cross design for the four arranges. The agglomeration group represents the group of participants that identified design problems using the agglomerations, and the control group comprises the participants that identified design problems using the list of non-agglomerated code smells.

The study was composed by a set of six activities distributed into three phases, as represented in Figure 2 described as follows.

Activity 1: Apply the questionnaire for subjects' characterization. The subjects' characterization questionnaire is composed of questions to characterize each participant, including academic degree, professional experience with Java programming, background on code smells and Eclipse IDE.

Activity 2: Training Session. After defining the order of execution of each steps, the next step was to provide a training session for the participants. The main objective of the training session was to level the participant at the same background required to understand and properly execute the experimental tasks. Thus, they received training about basic concepts and terminologies. This training was given only once for each participant before the first steps of the experiment. The training consisted of a 15-minute presentation that covered the following topics: software design, code smells, and design problems. The training session took approximately 15 minutes, and the participants could make any question throughout it.

After the training, subjects received some artifacts that could be used during the experiment. They received a list with a brief description of the types of design problems presented in the training session. They also received a list with the description of basic

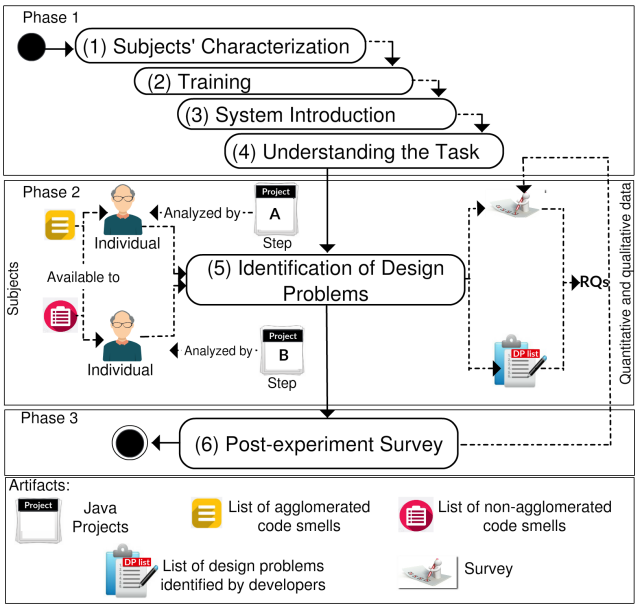

Figure 2: The experimental design

principles of object-oriented programming and design. They received a document containing: (i) a brief description of both project systems (Section 3.3), and (ii) a very high-level description of their design blueprint. We gave these documents because when they have to conduct perfective maintenance tasks, they need to have some minimal information about the systems to be maintained. The design blueprint represented the high-level design in the view of the project managers, but it was not detailed enough to support the identification of design problems. As it often occurs in practice, the analysis of the source code is inevitably required to identify a design problem.

Activity 3: System Introduction. We asked the participants to read the document containing the description of the project in which they would identify design problems. They had 20 minutes to read the description and the design blueprint of the system. Thus, they could start the identification with a certain level of familiarity with the software project.

Activity 4: Understanding the Task. In this activity, we explained how the participant could use the Organic tool (Section 2.1) to collect either the agglomerations or the list of (non-agglomerated) code smells. As the Organic tool was developed as an Eclipse plug-in, we explained each one of the sections displayed in the Eclipse IDE and that was related to the Organic tool. This activity lasted approximately 10 minutes.

Activity 5: Identification of Design Problems. In this activity, the participant had 45 minutes to identify design problems in the project. We emphasized to the participant the importance of achieving the key goal of finding design problems. For each identified design problem, the participant was asked to provide the following information: (i) short description of the problem, (ii) possible consequences caused by the problem, (iii) classes, methods or packages realizing the design problem in the source code, and (iv) the category(s) of agglomerations (Section 2.1) that helped him to identify the design problems. If the participant was identifying design problems as part of the control group, she needed to provide almost the same information; the difference was that instead of providing the agglomeration (and its category), she needed to provide the code smells that she used to identify the design problem.

Activity 6: Post-experiment Survey. In this activity, the participant received a feedback form. This form provides a list of questions, which enables the participant to expose her opinion on the identification of design problems. More details about this activity are provided in Section 3.6.

After the sixth activity had been completed, we asked the same participant to repeat all tasks in the second phase.

3.3 Software Projects and Participant Selection

In order to conduct the experiment as explained in the previous section, we selected two software systems in which developers had to identify design problems. We selected two programs that represent components of the Apache OODT project [20]: *Push Pull* and *Workflow Manager*. We selected subsystems of the OODT project since it is a large heterogeneous system; then, we could choose subsystems based on their diversity. Also, the Apache OODT project has a well-defined set of design problems previously identified by OODT developers who actually implemented the systems (Section 3.5) [26]; thus, avoiding the introduction of false positive design problems in the ground truth. In addition, the OODT project was developed for NASA, used in other studies [15–18, 26] and with a global community involved in its development. A brief description of the project systems are presented as follow:

- Push Pull: it is the OODT component responsible for downloading remote content (pull) or accepting the delivery of remote content (push) to a local staging area.
- Workflow Manager: it is a component that is part of the OODT client-server system. It is responsible for describing, executing, and monitoring workflows.

After choosing the projects, our next step was to recruit developers for the experiment. Thus, we sent a characterization questionnaire for a group of developers of our network. Their answers were analyzed to determine which of them were eligible to participate in the study based on the following requirements:

R1. Four years or more of experience with software development and maintenance. We have chosen four years because this is the average time used by companies such as Yahoo [37] and Twitter [32] to classify a developer as experienced.
R2. No previous knowledge about Push Pull and Workflow Manager.
R3. At least basic knowledge about code smells.
R4. At least intermediary knowledge on Java programming and Eclipse IDE.

We defined the knowledge in each topic based on a scale composed of five levels: *none*, *minimum*, *basic*, *intermediary*, *advanced* and *expert*. We included in the questionnaire a description of each level, allowing the subjects to have a similar interpretation of the answers. The description of such classification can be found in the complementary material [7]. Table 3 summarizes the characteristics of each developer selected for the experiment.

Table 3: Characterization of the Participants

ID	Experience in years	Education Level	Knowledge		
			Java	Code Smells	Eclipse
P1	5	PhD	Advanced	Advanced	Advanced
P2	6	Graduate	Advanced	Basic	Advanced
P3	8	Master	Advanced	Intermediary	Advanced
P4	4	Graduate	Intermediary	Basic	Basic
P5	5	Master	Advanced	Intermediary	Intermediary
P6	5	Graduate	Intermediary	Intermediary	Intermediary
P7	12	Graduate	Expert	Advanced	Expert
P8	5	Graduate	Advanced	Advanced	Advanced
P9	10	Graduate	Intermediary	Intermediary	Intermediary
P10	4	PhD	Advanced	Intermediary	Advanced
P11	5	PhD	Advanced	Intermediary	Advanced

3.4 Quantitative Analysis Procedures

In order to answer research question RQ1, we asked the experiment participants to analyze two systems with the aim of identifying design problems as described above. For each system, we analyzed the precision of participants regarding the identification of design problems. The precision of participants was measured based on *true positives* (TP) and *false positives* (FP). In this context, a true positive is a candidate of design problem, as indicated by the participant, that was confirmed by a ground truth analysis. On the other hand, a false positive is a candidate of design problem that was not confirmed in the ground truth analysis. Thus, the precision is calculated using the following formula:

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

3.5 Ground Truth Analysis

We had to validate the identified design problems as true positives or false positive for each one of the analyzed systems. However, we could not argue that a design problem was correct or not since we were not involved with the design of each system. Thus, we relied on the knowledge of the systems' original designers and developers to help us in validating the design problems. We certified they were the people who had the deepest knowledge of the design of the investigated projects. We highlight that designers and developers used to validate the ground truth were not subjects of the experiment.

We performed two steps to incrementally develop the ground truth. First, we asked original OODT designers and developers to provide us a list of design problems affecting the systems. They listed the problems and explained the relevance of each one through a questionnaire [7]. They also described which code elements were contributing to the realization of each design problem. Second, we identified some design problems using a suite of design recovery tools [11]. We asked developers of the systems to validate and combine our additional design problems with their list. The procedure for the additional identification was the following: (i) an initial list of design problems was identified using a method presented in [16], (ii) the developers had to confirm, refute or expand the list, (iii) the developers provided a brief explanation of the relevance of each design problem, and (iv) when we suspected there was still inaccuracies in the list of design problems, we discussed with them. In the end, we had the ground truth of design problems validated by the original designers and developers.

3.6 Qualitative Analysis Procedures

The experiment with professional developers helped us to assess the precision of developers in the identification of design problems with agglomerations. The results observed in the experiment revealed that agglomerations can, in fact, help to improve the precision of some developers in the identification of design problems (Section 4.1). Nevertheless, we also observed that there is room for improvements. Therefore, we conducted a qualitative analysis to investigate what should be improved from the perspective of professional software developers. Besides identifying possible improvements, this analysis also helped us to understand what are the main strengths of exploring agglomerations for design problem identification.

As described in Section 3.2, we asked the participants to provide us feedback about the identification of design problems. They answered a post-experiment survey, and we use their answers to conduct a qualitative analysis. The objective of the survey was to gather participant's opinion regarding (i) the (dis)advantages of using the agglomerations or code smells to identify design problems, (ii) whether the provided information could be easily understood, (iii) which types of information were fundamental to identify design problems, (iv) what she believes that should be done to improve the identification of design problems, (v) what she thought about the use of the code smells for the identification of design problems, (vi) how the visualization mechanism provided by the Organic tool affected her performance, and (vii) which types of code smell and categories of agglomeration were the most useful for identifying design problems. The results of this survey helped us to answer research question RQ2.

By conducting the survey, we were able to gather the opinion of developers regarding the use of code smell agglomerations. However, as reported by [8], what is reported in the survey may not be what actually happens in practice. Therefore, to obtain more reliable results, we also observed the participants of our experiment during the identification of design problems. This observation was performed during the experiment and also in analyzes after the experiment, through video and audio recorded during the experiment. This analysis allowed us to look at code smell agglomerations from the standpoint of professional software developers. It is important to note that the observation of participants during the experiment does not replace nor invalidate the survey. In fact, the combination of observations and survey helped us to obtain a deeper understanding and interpretation on the results observed in the experiment.

4 RESULTS AND ANALYSIS

The results of this study are organized in two sub-sections. Section 4.1 presents the results of our quantitative analysis regarding research question RQ1. Section 4.2 provides the results of our qualitative analysis to answer research question RQ2.

4.1 Do Agglomerations Improve Precision?

As described in Section 3.4, we conducted a quantitative analysis to answer our first research question: *Does the use of agglomerations*

Table 4: Precision

ID	Agglomeration Group			Control Group		
	TP	FP	Precision	TP	FP	Precision
1	2	1	66.67%	1	1	50%
2	0	3	0%	1	4	20%
3	3	2	60%	1	4	20%
4	2	0	100%	1	3	25%
5	4	0	100%	3	1	75%
6	1	0	100%	1	0	100%
7	1	1	50%	1	1	50%
8	3	0	100%	3	0	100%
9	0	1	0%	0	6	0%
10	0	0	-	1	1	50%
11	0	1	0%	0	0	-
All	**16**	**9**	**64%**	**13**	**21**	**38.24%**

improve the precision of developers in identifying design problems?. Table 4 presents the precision results for each participant (rows). The first column (*ID*) shows the identification number of each participant. The second column (*Agglomeration Group*) presents the true positives (*TP*), false positives (*FP*) and precision for the participants when they were provided with agglomerations to identify design problems. Similarly, the third column (*Control Group*) presents the true positives (*TP*), false positives (*FP*) and precision for the participants in the control group, i.e., when they were provided with a flat list of single smells.

Developers identified a few more true positives using agglomerations. We can see in Table 4 that the developers identified a few more design problems (TPs) when they were in the agglomeration group (16 TP design problems) than when they were in the control group (13 TP design problems). As far as the per-subject analysis is concerned, 4 developers (light gray rows) identified more true positives when they used agglomerations than when they used the list of code smells in the control group. The use of agglomerations clearly outperformed the use of smells in these 4 cases. On the other hand, 2 participants (2, 10) did not identify any true positive using the agglomerations, but they identified a true positive each in the control group. The rest of the participants (6, 7, 8, 9 and 11) identified the same number of true positives (5 TP design problems) regardless the group.

Upon data analysis, we were able to reveal the main reason why the 4 developers in the light gray rows identified more true positive design problems in the agglomeration group than in the control group. As illustrated in the example in the Figure 1 (Section 2.2), these 4 participants systematically used each agglomeration's smell as an indicator of the presence of a design problem. They analyzed each one of the code smells as a complementary symptom of the presence of a design problem, which gave them increasing confidence to confirm the occurrence of the design problem. Surprisingly, we noticed the same behavior for the participant 8 even when she was in the control group. She was capable of agglomerating the code smells on her own, starting from the individual smells given in the flat list. Then, she used such agglomerations to identify design problems in the control group. This is the reason why she reached a precision value of 100% in both groups.

Agglomerations help developers to avoid false positives. In general, developers identified less false positives when they used agglomerations (9 FP design problems) than when they used the list of code smells (21 FP design problems). With the exception of participant 11, all others identified either less or equal number of false positives when they were in the agglomeration group than when they were in the control group. When we analyze the control group, we can notice that more than half of the identified design problems are false positives (61,76%) while the agglomeration group identified only 36% of false positives.

After observing how developers identify design problems in the control group, we noticed that they did not go further with the analysis of the elements. Usually, a developer needs to analyze other classes in order to gather more information that can potentially indicate a design problem as discussed in Section 2.2. When the participants used the agglomerations, they analyzed multiple elements because they analyzed each code smell within the agglomeration even when the smells were in different elements. This behavior did not happen when participants were in the control group. In most of the cases, the participants in the control group analyzed only one code smell, which increased the likelihood of reporting false positives. Then, they reported a design problem in the class due to the presence of the code smell. However, some code smells are not related to any design problem; thus, the developer can report a false positive if she mistakenly considers a code smell that is not related to a design problem. That explains why developers in the control group found so many false positives. As developers tend to look at all agglomeration' smells before reporting a design problem, the likelihood of reporting a false positive decreases, even when there is a code smell that is a false positive by itself.

Agglomerations improve the precision. Even though we cannot claim a statistical significance in our results due to the sample size of this study, we can notice that developers achieve a higher precision (64%) when they use agglomerations than when they use code smells (38,24%). Therefore, this result suggests that agglomerations may improve the precision of developers in identifying design problems, answering our first research question. However, someone could expect that *all* developers using agglomerations would significantly outperform the control group. As a matter of fact, we noticed some factors that explain, at least partially, why developers did not find much more design problems when they were in the agglomeration group than when they were in the control group. These factors are presented in the next subsection, and they are useful to discover improvements for the identification of design problem with the analysis of stinky program locations.

4.2 How to Improve Design Problem Identification?

This section presents the answer for our second research question: *How can the identification of design problems with code smell agglomerations be improved?* We conducted a qualitative analysis to answer this question. As described in Section 3.6, this analysis was based on the observation of participants during the identification of design problems as well as the analysis of the post-experiment survey.

Where to start from? As discussed in the previous section, the participants identified few more true positives using agglomerations. Someone could expect that *all* developers using agglomerations would significantly outperform the control group. However, we observed that participants spent much more time analyzing the agglomerations than analyzing the smells in the control group. That happened because they analyzed each code smell in the stinky program location as previously explained Section 4.1. Furthermore, sometimes the participants analyzed agglomerations that were not related to any design problem. That is another factor that explains the almost same number of true positives between both groups.

Unfortunately, almost all the participants analyzed irrelevant agglomerations. Participants 6, 9, 10 and 11 were the ones that suffered the most from the analysis of irrelevant agglomerations. Since these four participants faced such issue, they suggested in our post-experiment survey that the Organic tool (Section 2.1) should provide means to prioritize relevant agglomerations. Hence, they would not spend time with the analysis of irrelevant stinky code. This issue helps us to explain why they fell short in identifying design problems through the analysis of agglomerations.

Need for prioritizing agglomerations. The aforementioned need for prioritization shows that the time and effort required to identify design problems is a key factor for developers; thus, prioritization should be taking into consideration. As a matter of fact, the prioritization of smelly code has been the focus of recent research [2, 33, 34]. In [33], for example, we proposed and assessed prioritization criteria for smell agglomerations. As we have observed, the prioritized list of agglomerations would help a developer to progressively analyze the agglomerations that have more chance to represent design problems, discarding the irrelevant ones. This would be especially useful in large legacy systems, in which thousands of agglomerations may be detected. Nevertheless, there is no prioritization criterion that is effective for any system [33].

Based on our qualitative analysis, we noticed that existing criteria for prioritization should select agglomerations that are cohesive. A cohesive agglomeration in our context is an agglomeration in which all the code smells are related to the same design problem. If there is one code smell that is not related to the design problem, such smell may direct the developer away from the design problem in the worst case. In the best case, the developer will spend time analyzing a code smell useless to identify the design problem. This fact suggests that developers need accurate algorithms to find cohesive agglomerations and to discard the less cohesive ones. However, prioritization algorithms based on existing criteria are unable to do this as far we are concerned. Consequently, the prioritization of stinky program locations still poses as a challenging research topic.

Stinky code analysis is challenging. Besides the prioritization issue, participants also suffered to analyze the smelly source code. As reported in Section 4.1, this problem was even worse for agglomerations affecting larger program scopes, i.e., agglomerations crosscutting implementation packages or class hierarchies. We noticed that a large agglomeration requires that developers reason about a wide range of scattered code smells. As they tend to use each code smell as a symptom of design problem, they have difficulties to correlate the multiple symptoms of an agglomeration. This is a challenging task because the higher the number of code elements involved in an agglomeration, the greater is the volume of code that must be analyzed. Consequently, developers will have more code to analyze, which increases the complexity of the analysis.

Need for proper visualization mechanisms. In order to alleviate the analysis of stinky code, some participants suggested the adoption of visualization mechanisms. For instance, participant number 8 suggested the visualization of agglomerations through a graph-based representation [13]. She mentioned that such visualization would provide an abstract and general view of agglomerations. The main advantage of this form of visualization is that the more abstract a representation is, the less details will be displayed for analysis. Consequently, the developers would not be overloaded with details. At the same time, an abstract representation like the graph-based visualization would help developers to see the full extent of an agglomeration. After providing an abstract view, a visualization mechanism could allow developers to progressively explore the agglomeration details such as the types of smells, location of stinky code and relationships among smells. Such details could be displayed in the graph itself, in the source code, or in complementary views.

Identification of the design problem type. The difficulty in analyzing agglomerations also raised the need for recommendations on which types of design problem each smell agglomeration is more likely to indicate. These recommendations would reduce the effort required to decide whether the elements are affected by design problems or not. For example, the agglomeration of Figure 1 occurs in classes of the same hierarchy that are implementing the *WorkflowRepository* interface. All smelly code of this stinky program location present the same type of smell, which is the *Feature Envy*. The occurrence of multiple Feature Envies in a unique hierarchy, suggests that there is a problem in the interface, which is spreading through all classes of the hierarchy. Therefore, to help developers to decide whether there is a problem or not, the Organic tool could suggest the analysis of this hierarchical agglomeration trying to identify problems like Ambiguous Interface [12] and Fat Interface [19], for example.

Suggestions of design problem types can help developers to focus their attention is specific characteristics of the suggested design problems. However, this kind of recommendation algorithm requires multiple case studies to understand how and when each form of agglomeration may represent specific types of design problem. As reported in our previous study [25], this is a challenging research topic.

5 RELATED WORK

Previous studies have not investigated if developers can indeed find more design problems when they focus on inspecting stinky program locations. Thus, we do not know whether the analysis of multiple smells actually provides better precision for the identification of design problems. In fact, related works propose techniques for supporting the detection and visualization of both single smelly code and inter-related smells. There are several studies that investigated detection and visualization of single smelly code [9, 23, 27, 35]. However, we found few studies that investigated the detection of inter-related smells affecting a program location [24, 34]. In this

context, Vidal et. al. [34] present a tool for detecting code smells and agglomerations of a (Java-based) system and ranking them according to different criteria [34]. The main benefit of using this tool is that developers can configure and extend the tool by providing different strategies to identify and rank the smells and groups of smells (i.e., agglomerations). However, a disadvantage of this tool is that it represents agglomeration without show the relation that could exist between the code smells.

Regarding detection and visualization of single smelly code, Van Emden and Moonen [9] present a tool that detects and visualizes code smells in source code, displays the code structure as a graph and maps code smells onto the attributes of that graph. This tool can be problematic for several reasons. The visualization is built assuming that code smells are concentrated in a particular region of the code and that metrics will point reviewers there. This assumption does not always hold; many code smells require understanding the relationships between many interacting components and thus are spread throughout the program. These relationships cannot be represented by a simple mapping between code structure and color.

Other studies [17, 39] have investigated the effects of code smells on the software design. For instance, Yamashita et al. [39] studied collocated smells – code smells that interact in the same source code file –, and coupled smells – code smells that interact across different source code files. Regarding software design, they observed that limiting the analysis to collocated smells would reduce their capability to reveal design problems, as coupled smells may reveal critical design problems.

We also found studies that have investigated the use of information other than code smells to identify design problems [22, 36]. In this case, Mo et al. [22] proposed and evaluated the combination of structural, history and design information to identify potential design problems. Xiao et al. [36] introduced an approach that uses a history coupling probability matrix to identify and quantify design problems. However, one disadvantage of such studies is they rely on design information, which may not exist for many software systems. In addition, they have not evaluated from the perspective of software developers.

Based on these related studies, we observed that they did not present whether developers can indeed find more design problems when they focus on inspecting stinky program location. Therefore, our research covers this gap by investigating whether the analysis of stinky program locations help developers in revealing more design problems than the analysis of single smells.

6 THREATS TO VALIDITY

This section presents some threats that could limit the validity of our main findings. For each threat, we present the actions taken to mitigate their impact on the research results.

The first threat to validity is related to the number of participants in the study. We have selected a sample of 11 participants, which may not be enough to achieve conclusive results. However, instead of drawing conclusions based on the quantitative results, we conducted a qualitative analysis. In addition to conduct a qualitative analysis, we defined a set of requirements to selecting developers suitable for the study. Also, we conducted training sessions with all participants. Such sections aimed to resolve any gaps in the participants' knowledge and any terminology conflicts, allowing us to increase our confidence in the results.

The second threat is related to possible misunderstandings during the study. As we asked developers to conduct a specific software engineering task and to answer a survey, they could have conducted the study different from what we asked. To mitigate this threat, we assisted the participants during the entire study, and we make sure of helping them to understand the experiment tasks and survey questions. We highlighted that our help was limited to only clarify the study in order to avoid some bias on our results.

Finally, there are two threats concerning the selected projects. The first one is about the difficulty of the participants in understanding the source code used in the experimental tasks. This difficulty appears due to the complexity of the source code and time constraints to complete each task. The second threat is related to one software project could be easier to identify design problem than the other. We minimized the first threat by running a pilot-experiment to define a experimental time reasonable to perform the tasks. To minimize the second threat, we selected projects with similar size, complexity, and number of known design problems. We also have trained all participants about each project. In addition, our results suggest no variation in difficulty for identifying design problems in the two projects.

7 CONCLUDING REMARKS

In this paper, we assessed if developers are effective in revealing design problems when they reason about multiple code smells. We conducted such investigation because recent studies have shown that design problems are likely to be located in elements affected by two or more smells. However, these studies do not evaluate if developers can reason about multiple smells to reveal design problems. Thus, we conducted a multi-method study with 11 developers. In the study, we asked them to identify design problems using code smell agglomerations. After that, we compared their results with the results of when they used the flat list of code smells to identify design problems.

The data analysis showed that developers find most design problems when they use code smell agglomerations to identify design problems, i.e., when they reason about stinkier program elements. In addition, we noticed that agglomerations help developers to avoid false positives. Therefore, our results suggest that agglomerations may improve the precision of developers in identifying design problems. When we analyze the survey's answers and how the developers identified design problems, we noticed that developers tended to have higher confidence to identify the occurrence of non-trivial design problems when using agglomerations. That happens because the developers tend to analyze each agglomeration's smell before reporting a design problem. Consequently, the likelihood of reporting a false positive decreases. In addition, we noticed that agglomerations help them to understand complex stinky structures.

Our results also indicate that developers need better tool support to analyze stinky code. For instance, the developers need to prioritize agglomerations that are most likely to indicate a design problem. The prioritization algorithms are required because the analysis of stinky code is difficult and time-consuming. Thus, developers should focus on those agglomerations that more likely

indicate design problems. In addition, we also noticed that developers need proper visualization mechanisms to support the analyses of stinky code scattered across wider program locations, such as hierarchies or packages. Some agglomerations are widely spread in the source code; a single agglomeration may contain code smells located in multiple class hierarchies. Thus, the developers have a large program scope to analyze. They may have difficulty to visualize how the code smells are related in the agglomeration. A graph-based visualization can help developers to figure out how the code smells are related in the agglomeration.

The results of our study encourage the use of smell agglomerations to identify design problems. However, there are some issues that should be addressed before developers can explore smell agglomerations in a time-effective manner. As discussed above, there is a need to provide mechanisms for better prioritizing and visualizing smell agglomerations. In the future, we plan to implement these mechanisms in Organic (Section 2.1) and evaluate their effectiveness.

REFERENCES

[1] M Abbes, F Khomh, Y Gueheneuc, and G Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proceedings of the 15th European Software Engineering Conference; Oldenburg, Germany*. 181–190.

[2] R. Arcoverde, E. Guimarães, I. Macía, A. Garcia, and Y. Cai. 2013. Prioritization of Code Anomalies Based on Architecture Sensitiveness. In *2013 27th Brazilian Symposium on Software Engineering*. 69–78. https://doi.org/10.1109/SBES.2013.14

[3] L Bass, P Clements, and R Kazman. 2003. *Software Architecture in Practice*. Addison-Wesley Professional.

[4] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. Mello, B. Fonseca, M. Ribeiro, and A. Chávez. 2017. Understanding the Impact of Refactoring on Smells. In *11th Joint Meeting of the European Software Engineering Conference and the ACM Sigsoft Symposium on the Foundations of Software (ESEC/FSE'17)*. Paderborn, Germany.

[5] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. 2016. Does Refactoring Improve Software Structural Quality? A Longitudinal Study of 25 Projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering (SBES '16)*. ACM, New York, NY, USA, 73–82.

[6] O. Ciupke. 1999. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30 (Cat. No.PR00278)*. 18–32.

[7] Online Companion. 2017. https://wnoizumi.github.io/SBCARS2017/. (2017).

[8] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. *Selecting Empirical Methods for Software Engineering Research*. Springer London, London, 285–311. https://doi.org/10.1007/978-1-84800-044-5_11

[9] E Emden and L Moonen. 2002. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering; Richmond, USA*. 97.

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

[11] J Garcia, I Ivkovic, and N Medvidovic. 2013. A Comparative Analysis of Software Architecture Recovery Techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering; Palo Alto, USA*.

[12] J Garcia, D Popescu, G Edwards, and N Medvidovic. 2009. Identifying Architectural Bad Smells. In *CSMR09; Kaiserslautern, Germany*. IEEE.

[13] I. Herman, G. Melancon, and M. S. Marshall. 2000. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics* 6, 1 (Jan 2000), 24–43.

[14] M Lanza and R Marinescu. 2006. *Object-Oriented Metrics in Practice*. Springer, Heidelberg.

[15] I Macia. 2013. *On the Detection of Architecturally-Relevant Code Anomalies in Software Systems*. Ph.D. Dissertation. Pontifical Catholic University of Rio de Janeiro, Informatics Department.

[16] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa. 2012. Supporting the identification of architecturally-relevant code anomalies. In *ICSM12*. 662–665.

[17] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. 2012. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In *CSMR12*. 277–286.

[18] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. 2012. Are Automatically-detected Code Anomalies Relevant to Architectural Modularity?: An Exploratory Analysis of Evolving Systems. In *AOSD '12*. ACM, New York, NY, USA, 167–178.

[19] Robert C. Martin and Micah Martin. 2006. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[20] C Mattmann, D Crichton, N Medvidovic, and S Hughes. 2006. A Software Architecture-Based Framework for Highly Distributed and Data Intensive Scientific Applications. In *Proceedings of the 28th International Conference on Software Engineering: Software Engineering Achievements Track; Shanghai, China*. 721–730.

[21] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. Hyderabad, India, 192–201.

[22] Ran Mo, Yuanfang Cai, R. Kazman, and Lu Xiao. 2015. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. 51–60.

[23] Emerson Murphy-Hill and Andrew P Black. 2010. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization; Salt Lake City, USA*. ACM, 5–14.

[24] W Oizumi and A Garcia. 2015. Organic: A Prototype Tool for the Synthesis of Code Anomalies. (2015). http://wnoizumi.github.io/organic/

[25] W Oizumi, A Garcia, T Colanzi, A Staa, and M Ferreira. 2015. On the Relationship of Code-Anomaly Agglomerations and Architectural Problems. *Journal of Software Engineering Research and Development* 3, 1 (2015), 1–22.

[26] W Oizumi, A Garcia, L Sousa, B Cafeo, and Y Zhao. 2016. Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems. In *The 38th International Conference on Software Engineering; USA*.

[27] Jacek Ratzinger, Michael Fischer, and Harald Gall. 2005. *Improving evolvability through refactoring*. Vol. 30. ACM.

[28] W. R. Shadish, T. D. Cook, and Donald T. Campbell. 2001. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference* (2 ed.). Houghton Mifflin.

[29] Marcelino Campos Oliveira Silva, Marco Tulio Valente, and Ricardo Terra. 2016. Does Technical Debt Lead to the Rejection of Pull Requests?. In *Proceedings of the 12th Brazilian Symposium on Information Systems (SBSI '16)*. 248–254.

[30] G Suryanarayana, G Samarthyam, and T Sharmar. 2014. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann.

[31] A. Trifu and R. Marinescu. 2005. Diagnosing design problems in object oriented systems. In *WCRE'05*. 10 pp.

[32] Twitter. 2017. Working at Twitter. (April 2017). Available at https://about.twitter.com/careers.

[33] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos. 2016. Identifying Architectural Problems through Prioritization of Code Smells. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. 41–50. https://doi.org/10.1109/SBCARS.2016.11

[34] Santiago A. Vidal, Claudia Marcos, and J. Andrés Díaz-Pace. 2016. An Approach to Prioritize Code Smells for Refactoring. *Automated Software Engg.* 23, 3 (Sept. 2016), 501–532. https://doi.org/10.1007/s10515-014-0175-x

[35] Richard Wettel and Michele Lanza. 2008. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM symposium on Software visualization*. ACM, 155–164.

[36] Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. 2016. Identifying and Quantifying Architectural Debt. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 11.

[37] Yahoo! 2017. Explore Career Opportunities. (April 2017). Available at https://careers.yahoo.com/us/buildyourcareer.

[38] A Yamashita and L Moonen. 2013. Exploring the impact of inter-smell relations on software maintainability: an empirical study. In *Proceedings of the 35th International Conference on Software Engineering; San Francisco, USA*. 682–691.

[39] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter. 2015. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. 121–130.