How Do Software Developers Identify Design Problems?

A Qualitative Analysis

Leonardo Sousa[1], Roberto Oliveira[1], Alessandro Garcia[1], Jaejoon Lee[3],
Tayana Conte[2], Willian Oizumi[1], Rafael de Mello[1], Adriana Lopes[2],
Natasha Valentim[2], Edson Oliveira[2], Carlos Lucena[1]
[1]PUC-Rio, Rio de Janeiro, Brazil, [2]UFAM, Manaus, Brazil, [3]Lancaster University, Lancaster, UK
{lsousa, rfelicio, afgarcia, woizumi, rmaiani, lucena}@inf.puc-rio.br, {tayana, adriana, natashavalentim, edson.cesar}@icomp.ufam.edu.br, j.lee3@lancaster.ac.uk

ABSTRACT

When a software design decision has a negative impact on one or more quality attributes, we call it a design problem. For example, the Fat Interface problem indicates that an interface exposes non-cohesive services. Thus, clients and implementations of this interface may have to handle with services that they are not interested. A design problem such as this hampers the extensibility and maintainability of a software system. As illustrated by the example, a single design problem often affects several elements in the program. Despite its harmfulness, it is difficult to identify a design problem in a system. It is even more challenging to identify design problems when the source code is the only available artifact. In particular, no study has observed what strategy(ies) developers use in practice to identify design problems when the design documentation is unavailable. In order to address this gap, we conducted a qualitative analysis on how developers identify design problems in two different scenarios: when they are either **familiar** (Scenario 1) or **unfamiliar** (Scenario 2) with the analyzed systems. Developers familiar with the systems applied a diverse set of strategies during the identification of each design problem. Some strategies were frequently used to locate code elements for analysis, and other strategies were frequently used to confirm design problems in these elements. Developers unfamiliar with the systems relied only on the use of code smells along the task. Despite some differences among the subjects from both scenarios, we noticed that developers often search for multiple indicators during the identification of each design problem.

CCS CONCEPTS

•Software and its engineering →Software design engineering;

KEYWORDS

design problem, software design, strategy, symptoms

ACM Reference format:
Leonardo Sousa[1], Roberto Oliveira[1], Alessandro Garcia[1], Jaejoon Lee[3], Tayana Conte[2], Willian Oizumi[1], Rafael de Mello[1], Adriana Lopes[2], Natasha Valentim[2], Edson Oliveira[2], Carlos Lucena[1]
. 2017. How Do Software Developers Identify Design Problems?. In *Proceedings of 31st Brazilian Symposium on Software Engineering, Fortaleza, Ceará, Brazil, September 2017 (XXXI SBES),* 10 pages.

XXXI SBES, Fortaleza, Ceará, Brazil
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software design is a fundamental concern during the software development process [7, 12]. Indeed, 25% of discussions in commits, issues and pull requests are about design [2]. Such concern is explained by the fact that design decisions made by developers have an influence on many properties of the software systems, such as maintainability, robustness, comprehensibility, performance, and the like. When a software design decision has a negative impact on quality attributes, we call it a design problem [1, 5, 30]. An example of design problem is the *Fat Interface* [17]. This problem occurs when an interface offers a general entry point for several non-cohesive services, complicating the logic of its clients. In addition to affecting the system extensibility, a design problem such as this hampers software comprehensibility since developers have to understand several services rather than one [17].

Design problems might be so harmful that software systems have been discontinued or reengineered due to their prevalence [10, 13, 24, 32]. In fact, a recent study [25] showed that the design is one of the most common categories of technical debt that leads to the rejection of pull requests. Given the harmfulness of design problems, developers should remove them from software systems as early as possible [9, 24, 36]. However, their identification is not a trivial task [5, 29]. One of the reasons is that developers often have to identify design problems in the source code since design documentation is almost always nonexistent, informal or not up-to-date [11, 30].

As developers often have to use the source code to identify design problems, they need to locate a structure (e.g. a set of classes or methods) in which symptoms of the design problem can be observed. A **symptom** is a partial sign or indication of the presence of a design problem. In the *Fat Interface* example, the interface, its clients and their implementing classes represent the structures affected by the design problem. In this example, symptoms include the number of services exposed by the interface as wells as their lack of cohesion.

Such code structures and symptoms are not always straightforward to locate in the program, especially when developers have limited or no familiarity with the source code. Thus, developers need to use a strategy to identify design problems in the implementation. In our context, an **identification strategy** refers to an action that the developer does and that contributes to identify a design problem. For instance, a strategy can consist of the analysis of services exposed by the interface in order to reveal one of the symptoms of a *Fat Interface*.

Unfortunately, we know little about how developers actually identify design problems in the source code. To the best of our knowledge, we did not find any other study that investigated how developers identify design problems. In general, the existing studies [5, 14–16, 20, 21, 29, 33, 34] propose solutions that will help developers to identify design problems. We highlight that before proposing solutions that will help developers to identify design problems, first, we need to understand how they conduct the identification task in practice. By understanding this task, researchers will be able to build mechanisms that are most suitable in helping developers during the identification of design problems.

As no study has observed how developers identify design problems in practice, we conducted a study to observe what strategy(ies) developers use to identify design problems as well as how and when they use these strategies. In particular, we also investigated whether the developers' familiarity with the target systems influences on the use of these strategies. For this purpose, we observed how developers identify design problems in two scenarios. In the first one, developers are familiar with the systems, and in the second one, they are unfamiliar with the systems. Our analysis resulted in some findings:

- In general, developers used multiple strategies to identify design problems. In this study, they used six preeminent strategies. These strategies are described in Section 4.1.
- Some strategies were most frequently used to locate elements that may contain a design problem. Other strategies were most frequently used to confirm whether the fragment contains or not a design problem.
- Developers familiar with the target systems applied and combined the strategies, in which each strategy was likely to provide one or more symptoms for a design problem. Thus, they tend to search for multiple symptoms before confirming the occurrence of a design problem.
- Developers unfamiliar with the systems relied only on one type of strategy, but still they managed to use the strategy to reveal multiple symptoms for each design problem.
- Indeed, we noticed that developers search for multiple symptoms that can indicate a design problem. This behavior happened regardless the familiarity with the target systems.

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

2 BACKGROUND

Software design is the result of a series of decisions made during the software development [28]. It is expected that these decisions contribute to creating software systems that are maintainable, robust, secure, and the like. However, along the way, design problems can be injected into software systems. A **design problem** occurs when a design fragment negatively impacts software quality attributes. Design fragment is a part of the software design that represents either a design decision or a relevant concept for the system design. As design problems have negative consequences, they are often targets of significant maintenance effort [9, 24, 36].

In general, design problems affect interfaces, components, hierarchies or even other abstractions that are relevant to the design. An example of design problem is the *Cyclic Dependency* [22]. This design problem happens when two or more elements depend on each other directly or indirectly. When there are long dependency cycles among the elements, the system might end up at a stage where these cycle dependencies compromise the understandability, testability, reusability and maintainability of the software systems [22]. Also, *Cyclic Dependency* can cause deadlock [4], which negatively affect the system performance and availability. Other examples of design problems include *Scattered Concern* [9], *Ambiguous Interface* [9] and *Fat Interface* [17].

We call **problem identification** the task of finding a design fragment that contains a design problem. In the implementation, the structure of a design fragment comprises the code elements that form the fragment. In the identification task, developers identify design problems in the counterpart structure realizing the relevant design fragment in the program. Although little is known about how developers identify design problems, this task requires at least two basic steps, named *location* and *confirmation*. *Location* comprises the process of finding a structure in the implementation that may embody a design problem. In the *confirmation* step, developers confirm or refute the existence of a design problem in those reifications of design fragments in the implementation.

Some characteristics of the problem identification make the task challenging. To start, systems tend to be large in size and complexity, increasing the search space for design problems. Second, each design problem usually pervades the implementation of several elements [9, 20]. Thus, developers need to analyze several elements to identify a single design problem [29]. Third, design documentation is often nonexistent, informal or not up-to-date. Thus, the source code is the only artifact available for the developers identify design problems in most cases.

To perform each one of the identification steps, developers can apply different strategies. In our context, an *identification strategy* refers to an action that an developer does, and it contributes to identifying a design problem. For instance, a strategy can be the prioritization of a specific type of element that could be most likely to have a design problem. Another strategy can be the presence of code smells to confirm a design problem. These strategies can be used to reveal symptoms of design problems. Symptom is a sign that a fragment may contain a design problem. In the case of the presence of code smells, for example, a smell itself can be a symptom of a design problem [6].

3 STUDY PLANNING

This section presents the study design reported in this paper.

3.1 Research Questions

In order to understand how developers identify design problems, we conducted a series of experiments. In these experiments, we observed the actions that developers perform to finding a design fragment that contains a design problem. Before analyzing the actions performed by developers, we first need to identify what are these actions. In other words, we need to identify what are the strategies that developers use to identify design problems. RQ1 addresses this matter:

RQ1. What are the strategies that developers use to identify design problems?

After revealing the strategies, we need to verify how developers applied these strategies. Thus, we investigated how developers use the strategies to identify design problems in two scenarios: when developers are either **familiar** (Scenario 1) or **unfamiliar** (Scenario 2) with the analyzed systems. The following RQs address this investigation:

RQ2. How do developers identify design problems in familiar systems?

RQ3. How do developers identify design problems in unfamiliar systems?

Both scenarios were used to answer RQ1, while each scenarios was used to answer RQ2 and RQ3, respectively. In order to support the answering for these RQs, we applied a qualitative analysis using some Ground Theory (GT) procedures [27] (Section 3.4). Upon this qualitative analysis, we can understand how developers identify design problems, for instance, we can find out the strategies that developers use (RQ1) and how they apply these strategies to identify design problems (RQ2 and RQ3).

3.2 Studied Scenarios and Subjects

In the following, we explain in detail the two scenarios (Scenario 1 and Scenario 2) as well as the procedure to recruit subjects for the study.

1) Developers familiar with the system

In the Scenario 1, we searched for software companies that could provide developers to our study. We defined the following criteria to select the companies: experience of their developers, size in terms of number of developers in a project, application domain of their projects, and development process. We defined these criteria in order to promote some variation while selecting companies from our industrial collaboration network, thereby balancing contextual diversity with convenience [23]. Based on these criteria, we chose two Brazilian software companies.

After selecting the companies, we asked the companies' managers, some of them were software designers, to suggest specific systems that met the following characteristics. Firstly, systems in different stages of design degradation. Secondly, systems from different domains and with different sizes with respect to amount of modules and developers. Thirdly, projects that are not in their initial versions. Lastly, systems developed in Java. As each selected company has to provide software systems, we selected Java projects given the popularity of the Jjava programming language [3, 26]. Thus, it would be easier to keep the consistency among the provided systems: all of them implemented in the same program language. The selected programs are described as follows.

- Company 1: Program 1 (P1) supports the management of registry offices for audit and control from the Justice Court of Brazil. Program 2 (P2) is a computational solution for maintaining information on the patients' health status, and their medical records. Program 3 (P3) is a system developed to trace products from a production line.
- Company 2: Program 4 (P4) is a legacy system to process tax and to control the entrance of products from the state of Amazonas. Program 5 (P5) was developed for standardizing budget in the same state.

Full details about the companies and the programs are available in our online material [18]. After providing us with the programs, we asked the companies' managers to indicate developers that were familiar with each program and could act as subjects of the study.

2) Developers unfamiliar with the system

In the Scenario 2, we selected two programs that represent components of the Apache OODT project. Then, we recruited developers, who were unfamiliar with OODT, to identify design problems in those programs. We selected Apache OODT programs because they have a well-defined set of design problems previously identified by OODT developers who actually implemented the systems. The programs are:

- Push Pull (P6): 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 (P7): it is a component that is part of the OODT client-server system. It is responsible for describing, executing, and monitoring workflows.

After selecting the programs, we had to select developers to participate as subjects in the study. We could have used the five subjects from the first scenario. However, they already have to identify design problems in the system that they are familiar (Scenario 1). Thus, we have to selected other subjects to comprise the Scenario 2. Then, we sent a questionnaire to several developers from our network in order to select developers who could be eligible for the study. We selected subjects who had at least four years of experience with software development and maintenance. We have chosen four years because this is the average time that companies like Yahoo [35] and Twitter [31] consider to developers as experienced, and four years was the least experience time of one of the subjects in the first scenario. Also, we selected subjects who were unfamiliar with the OODT project. Further details about the recruitment process are available on our online material [18].

Table 1 presents the characterization of all our subjects. First column indicates the identification number of the subject, second column has the experience on software development in years, third column has education level and fourth column indicates the system that the subject had to identify design problems. The first five subjects in the Table 1 were assigned to identify design problems in their own systems, while the other subjects had to identify design problems in the system with which they were no familiar.

3.3 Study Activities

The study was composed by three activities: Training, Problem identification and Follow-up questionnaire.

Activity 1: Training. In this activity, we conducted a training for all the subjects regarding software design and design problems. We also presented some examples of design problems pertaining to different categories (Section 2). The following design problems

Table 1: Characterization of the subjects

ID	Experience(years)	Education	System	Scenario
S1	13	Graduate	P1	1
S2	4	Graduate	P2	
S3	10	Master	P3	
S4	9	Graduate	P4	
S5	12	Graduate	P5	
S6	5	PhD	P6	2
S7	6	Graduate	P7	
S8	8	Master	P7	
S9	4	Graduate	P7	
S10	5	Master	P6	
S11	5	Graduate	P6	
S12	12	Graduate	P7	
S13	5	Graduate	P6	
S14	10	Graduate	P7	
S15	4	PhD	P7	
S16	5	PhD	P6	

were included in the training session: *Ambiguous Interface*, *Unwanted Dependency*, *Component Overload*, *Cyclic Dependency*, *Scattered Concern*, *Fat Interface*, and *Unused Abstraction*. We selected these design problems together with the project managers, who suspected that these represented common cases of design problems in their projects. However, we let clear to the subjects that they were allowed to identify other types of design problems. In fact, they identified a wide range of other design problems that were relevant to their projects (Section 4.1). The training was organized in two parts: the first one (approx. 25 minutes long) was used for a Powerpoint-based presentation; the second one (approx. 15 minutes long) was devoted to discussion and questions, if necessary.

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 principles of object-oriented programming and design. Subjects unfamiliar with the systems received a document containing: (i) a brief description of P6 and P7 systems, and (ii) a very high level description of their design blueprint. We gave these documents because when they have to maintain unfamiliar systems, they need to have some minimal information about the systems to be maintained. We used the same document provided in the OODT project. 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.

Subjects had access to mechanism to reveal a wide range of symptoms. For instance, they received a list of code smells affecting the systems. We provided the list of code smells because previous studies suggest that code smells can be used as indicators of design problems [14–16, 21]. We used well-known metrics-based strategies to identify 15 types of code smells from Fowler's Catalog [6]. We highlight that subjects were free to use or not these code smells. In the same way, they were free to use information from other artifacts too. The provided artifacts are available in our online material [18].

Activity 2: Problem identification. In this task, we asked subjects to identify design problems, and to report their findings in an online form [18]. They had 45 minutes to perform this task. At the beginning of this activity, we asked them to explain aloud what they were doing while we video recorded the task. In this way, we could combine the form answers with the video recording to complement the qualitative analysis. Camtasia[1] tool was used to record audio and screenshots of each computer used by the subjects. A video camera was installed in the room to also record them.

Activity 3: Follow-up questionnaire. Developers were asked to answer a questionnaire about their general perception on the identification of design problems. The questionnaire used in the activity is available in our online material [18]. The answers were also used to complement the qualitative analysis.

3.4 Data Analysis and Oracle Creation

Data Analysis. The qualitative data analysis was based on the procedures of Grounded Theory (GT) suggested by Strauss and Corbin [27]. The procedures comprise three phases: *open coding* (1st phase), *axial coding* (2nd phase) and selective coding (3rd phase). Open coding involves the breakdown, analysis, comparison, conceptualization, and the categorization of the data. Axial coding examines the relations between the identified categories. Finally, selective coding performs all the process refinements by identifying the core category to which all others are related. When analyzing the data, we created codes for the developers' speeches (1st phase). After, these codes were related to each other through axial coding (2nd phase). We did not apply the selective coding since we were not aiming to reach a theoretical saturation, as expected in GT method [27]. Therefore, we decided to postpone the selective coding phase. For this reason, we do not claim that we applied the GT method, only some specific procedures.

We did the open coding to associate codes with quotations of transcripts, and we did the axial coding, at which the codes were merged and grouped into more abstract categories. For each transcript, the codes and identified networks (memos showing the relationships in the categories) were reviewed, analyzed and changed upon agreement with the others researchers.

Oracle Creation. For each one of the analyzed systems, we had to validate the identified design problems as true positives or false positive. 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 oracle list were not subjects of the experiment.

For the systems used in the first scenario (P1 to P5), we analyzed each subject's answer, and we asked developers and designers (not the same developers that participated in the experiment) to validate the answers. If developers or designers have agreed with the subject's answer we marked the identified problem as true positive, then we added the design problem to the oracle. Otherwise, we put the design problem to re-validation. In the re-validation process,

[1]Camtasia is available at www.techsmith.com/camtasia.html

we invited the subjects to discuss each design problem in the re-validation in order to establish the final list of true positives and false positives. The validation process was conducted by both the first and second authors to avoid bias in the validation.

For the systems used in the second scenario (P6 and P7), we asked original designers and developers of these systems to provide us a list of design problems affecting the systems. Then, we identified some design problems using a suite of design recovery tools [8]. 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 detection strategies presented in [14], (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 oracle of design problems validated by the original designers and developers.

4 RESULTS AND ANALYSIS

The subjects familiar with the systems (Scenario 1) identified 39 design problems, in which 31 were validated as true positives according to the oracle (Section 3.4). The subjects unfamiliar with the systems (Scenario 2) identified 31 design problems, in which 17 was validated as true positives. In the total, the subjects identified 70 instances of design problems. This section discusses these results.

4.1 Strategies to Identify Design Problems

In this subsection, we address the RQ1: *What are the strategies that developers use to identify design problems?* After the qualitative analysis (Section 3.4), we noticed that the subjects used six preeminent strategies: **smell-based**, **problem-based**, **principle-based element-based**, **quality attribute-based**, and **pattern-based**. They are described as follow. Due to the confidentiality with the companies, we changed each name of code elements to a letter.

Smell-based strategy is the strategy in which the subjects use the code smells to identify design problem. As mentioned by other studies [14–16], developers can use smells as an symptom of design problems. However, it was interesting to observe that this strategy was used differently in each scenario. Its degree of success on identifying design problems also varied (Section 4.2).

The subjects familiar with the systems mainly used the smell-based strategy to confirm the existence of a design problem. They marked a candidate fragment as having a design problem whenever they were analyzing a candidate fragment, and they noticed that the fragment had a code smell. We also observed that developers familiar with the systems explored some types of code smells that were not in the provided initial list. For example, they explored the number of switch statements in methods (*Switch Statements* smell) when they were analyzing certain classes. Similarly, they mentioned that some classes have similar code snippets (*Duplicate Code* smell). As an example, S4 subject identified a *God Class* smell even though the instance of the smell was not in the provided list. After finding the smell, the S4 subject confirmed the occurrence of a design problem in the class:

"It is not its responsibility to print on screen (...) It accesses the database, and it shows (the data) on the screen (...) This class deviates from its function that turns the class in a God Class"

In addition to using code smells to confirm the existence of design problems, the subjects unfamiliar with systems also used the smell-based strategy to search for candidate fragments. As they were unfamiliar with analyzed systems, they used the presence of code smells to guide them towards a candidate fragment. After finding a fragment, they used other code smells affecting the fragment to confirm the design problem.

Principle-based strategy is the strategy that the subjects used design principles [17] (e.g. open-closed principle and information hiding) to identify design problems. This strategy was only used by the subjects of the Scenario 1, and it was mainly used to confirm if an fragment under analysis has a design problem. In this case, they marked a candidate fragment as having a design problem when they noticed that a class under analysis was violating a design principle. This case happened with S5 subject:

"The affected element is the A class, this class has a problem because it accesses the database and attributes of other class directly (...) besides, this class violates the interface segregation principle"

Problem-based strategy is the strategy in which the subjects searched for occurrences of a specific type of design problem they already had in their mind in the source code. We classify that a subject used the problem-based strategy when he explicitly mentions he is looking for a specific type of design problem across the system. Only the subjects of Scenario 1 used this strategy. We observed that they tended to focus on searching for design problems related to interfaces and components (realized as packages in the source code). For instance, *Fat Interface* [17] and *Component Overload* [21] were problems that developers identified with high frequency. On the other hand, we did not observe developers looking for problems related to abstract concepts, such as *Delegating Abstraction* [21] and *Unused Abstraction* [21]. The following example illustrates the case that the S2 subject sought for *Cyclic Dependency* design problem:

"Well, I am now thinking about a particular candidate of cyclic dependency... I suspect this is located in this package"

Element-based strategy is the strategy in which the subjects selected specific code elements to investigate if it is affected by a design problem. They do not necessarily reason about specific types of design problems, but they look for any sort of indication (e.g. frequent modifications) in those elements that may signal the manifestation of a design problem. In this strategy, the subjects focused their reasoning on code elements – such as core classes, interfaces, and hierarchies – that represent key design abstractions in the program. Given the relevance of such elements to the system, developers knew these elements could form structures realizing a design problem in the implementation. Thus, they directly started inspecting these code elements and reasoning about symptoms in those elements. Only the subjects of Scenario 1 used this strategy, as it probably requires familiarity with the system design.

Developers often knew already which code elements they should analyze first. Interestingly, most of these cases were classes: we expected developers would also analyze often interfaces and packages given their relative importance to the design. However, such elements were rarely analyzed. Moreover, there were a few cases in which they had to determine a criterion to choose such elements explicitly. For example, one of the subjects chose a class based on the number and nature of variables and methods located in the class. Another subject decided to limit the search to classes within specific subsystems. He picked a subsystem that was visibly large regarding the number of classes. The same subject also suggested restricting the search to a generic subsystem. All class that did not belong to any other specific subsystem were created in or moved to this subsystem. In the following quotation, we illustrate an example of how the element-based strategy was used by subjects of Scenario 1. At the beginning of the task, the S1 subject was trying to determine which elements he should analyze, and then he decided to prioritize the analysis of classes in large subsystems. After that, he browsed a few classes until selecting one:

"Let's open the source code. We should start analyzing big subsystems... I know which one, let's start by the X subsystem. We already fixed a design problem in X subsystem, but it still might have more problems (...) I suspect this class contributes to a design problem"

Quality attribute-based strategy is the strategy where the subjects reasoned about quality attributes that are negatively affected by certain design fragments. They reasoned how a program structure, which realizes a design fragment, explicitly hinders one or more quality attributes. Again, they did not necessarily reason about specific types of code smells or design problems. The subjects used this strategy when they were analyzing a candidate design fragment, and they noticed that the counterpart implementation of the fragment impacted one or more quality attributes. Thus, the subjects reasoned about quality attributes as consequences of a design problem affecting a fragment. The most cited quality attribute was maintainability. However, subjects also mentioned flexibility, readability, adaptability, performance, security, and robustness.

Only the subjects of Scenario 1 used this strategy. We present below an example of a subject (S1) who reasoned about the complexity and reusability of a structure to confirm the occurrence of a design problem. In this case, he was investigating the use of *Adapters* in the system, but without confirming if the class has a design problem or not. Then he used the quality attribute-based strategy to reveal additional symptoms related to a possible design problem. Thus, he used the consequence that the design problem causes on the reusability to confirm the design problem. We present part of the quotation as follows.

"It (the implementation structure) increases the complexity and reduces the reuse (...) It has not been reused at all, and that is the problem. Look at the number of adapters that are associated with this class as compared to the number of adapters in the other parts of the system"

Pattern-based strategy is the strategy that subjects searched for instances of a design or architectural pattern in the source code and verify if their implementation violates the pattern rules. This strategy was frequently used both to locate candidate design fragments and to confirm the existence of design problems. In this strategy, developers analyzed code structures potentially violating a pattern rule. Whenever developers could confirm the violation, they marked the fragment as having a design problem. Subjects discussed a wide range of patterns, including *Adapter*, *Builder*, *Facade*, *SOA* and *MVC*.

Even though only the subjects of Scenario 1 used the pattern-based strategy, it was the most successful one (Section 4.2). Maybe the reason of using this strategy has to do with their familiarity with the systems. Even though the aforementioned patterns are well known and used across different system domains, developers had to know how these patterns were particularly instantiated in different contexts of their systems. As an example, in the previous quotation, the S1 subject was investigating classes realizing the *Adapter* pattern. Before confirming the design problem, he used first the pattern-based strategy to identify classes that could be violating the *Adapter* pattern. Another example happened with S5 subject. He was analyzing a group of classes when he noticed that a class in the MVC pattern was illegally accessing the database:

"Class A is the affected structure. This class is problematic because it accesses the database directly. In other words, it has a design problem because it is not following the MVC pattern"

As aforementioned, we did not know which strategies developers use to identify design problems. The observation and characterization of all the six frequently used strategies enable us to answer RQ1. This answer leads us to our first finding:

Finding 1. Developers often use multiple strategies to identify design problems

4.2 Problem Identification in Familiar Systems

In this subsection, we address RQ2: *How do developers identify design problems in familiar systems?*

Different strategies for each identification step. As mentioned in Section 2, the identification of design problems can be composed by two steps at least. Figure 1 shows each step and the strategies most frequently used in the step. The *location* step is represented by a blue square, the *confirmation* step is represented by a red square, and the strategies are represented by green squares (the dotted arrows are explained latter). We noticed that in most cases, the subjects of Scenario 1 used element-based and problem-based strategies to locate candidate fragments for further analysis. On the other hand, smell-based, principle-based and quality attribute-based strategies were used to confirm if the candidate fragments were indeed affected by design problems. Finally, the pattern-based strategy was used both to locate candidate fragments and to confirm design problems.

We also noticed that the subjects tended to combine the strategies. For instance, whenever the subjects were looking for violations of a design or architectural pattern (pattern-based strategy) to detect candidate fragments, they did not only rely on the violation itself to support the confirmation of a design problem. They often confirmed the existence of a design problem when they noticed other symptoms, e.g., the structure of the candidate fragment was either explicitly affecting a quality attribute (quality attribute-based strategy) or hosting one or more code smells (smell-based strategy). These combinations happened because a fragment may contain several symptoms that indicate design problems. For instance, if a fragment violates a pattern, it is likely that the fragment also contains smells and violations of design principles. Consequently, these symptoms may influence the quality attributes negatively.

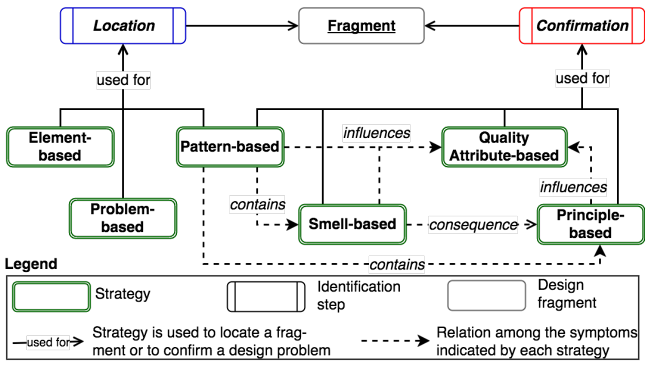

Figure 1: Strategies frequently used in each identification step

Figure 1 shows how the symptoms, revealed by each strategy, relate to other symptoms. The dotted arrows indicate the type of influence among the strategies.

Single vs. combined strategies. We noticed that developers combined these strategies to find complementary symptoms of a design problem in the candidate fragment. The symptoms can be complementary in the sense that each symptom adds a piece of information that will help subjects to decide if there is a design problem in the fragment. For example, in the quotation that illustrated the quality attribute-based strategy (Section 4.1), the S1 subject combined the pattern-based and quality attribute-based strategies. First, he used the pattern-based strategy to identify classes that could be violating the *Adapter* pattern. After that, he used the quality attribute-based strategy to reason through the consequences that the design problem caused on reusability. In this case, the both symptoms (overuse of *Adapters* and the negative effect on reusability) complemented each other to indicate the design problem.

The subjects also combined more than two strategies to identify a single design problem. Table 2 provides an overview on the use of the strategies to identify design problems. The first column indicates the strategy or combination of strategies that the subjects used to identify design problems. The second column indicates how many times the strategy or combinations of strategies led to a design problem. We used the oracle (Section 3.4) to validate the design problems as true positives (inside of the parentheses in the second column). The third column indicates the design problems found when the subject used the strategies. We obtained the values of the second column after applying some GT procedures (Section 3.4). During the data analysis, we count each strategy that the subjects used and led to the identified design problem. For instance, whenever a subject identified a design problem, we analyzed all the actions that he did since the first time that he mentions a symptom (associated with a particular strategy) until the moment he confirms the design problem. If one of these actions consisted of a strategy usage (e.g., the subjects reasoned about a design pattern), then we counted that the strategy contributed to identifying a design problem.

We can see in Table 2 that the subjects using only one strategy identified 12 design problems (9 true positives). However, when they combined multiple strategies, they identified 27 design problems (21 true positives). Some of the design problems were not presented in the training session (Section 3.3), but developers were able to identify them. The description of each one of the identified design problems is available in our online material [18]. This result demonstrates that subjects, who are familiar with the systems, can identify more design problems when they combine multiple strategies than when they use only one strategy. The prevailing behavior of combining different strategies in Scenario 1 also suggests that the identification of design problems might be more complex than one can expect. In fact, the subjects often need to rely on various strategies to locate (and confirm) a single fragment that contains a design problem. This behavior leads us to the second finding:

Finding 2. Developers familiar with the systems often combine strategies to identify a single design problem

Table 2: Design problems identified by the strategies

Strategies	Instances	Design Problems	Subjects
Element	6 (4)	STC, UWD, CPO, CCO, ICA	S1, S2, S5
Pattern	2 (2)	UWD, DLA	S1, S5
Problem	4 (3)	AMI, FTI, CCD	S2
Element, Pattern	6 (5)	UWD, CPO	S1, S5
Element, Principle	3 (1)	UWD, AMI, CPO	S4, S5
Element, Problem	1 (1)	FTI	S2
Element, Quality attribute	1 (1)	ICA	S1
Problem, Quality attribute	3 (3)	FTI, CCO	S3
Problem, Smell	1 (1)	FTI	S2
Element, Problem, Smell	1 (1)	FTI	S2
Element, Pattern, Smell	2 (2)	UWD	S5
Element, Quality attribute, Pattern	2 (2)	UWD, CCO	S1, S3
Element, Quality attribute, Smell	1 (1)	CCO	S4
Element, Principle, Pattern, Smell	2 (2)	UWD, MPC	S1, S5
Element, Principle, Quality attributes, Pattern	1 (1)	CCO	S5
Element, Problem, Quality attribute, Pattern	1 (0)	UWD	S3
Principle, Quality attribute, Pattern, Smell	2 (1)	UWD, CCO	S4

AMI = Ambiguous Interface, CCD = Cyclic Dependency, CCO = Concern Overload, CCP = Component Overload, DLA = Delegating Abstraction, FTI = Fat Interface, ICA = Incomplete Abstraction, MPC = Misplaced Concern, UWD = Unwanted Dependency, STC = Scattered Concern

Most Successful Strategy. We noticed that sometimes the subjects used a strategy which did not lead to any design problem. For instance, the S4 subject used the problem-based strategy to locate a candidate fragment to analyze; however, he realized that the fragment did not have a design problem. Then, he decided to analyze a specific interface by following the element-based strategy. During the analysis, he found instances of the smell *Long Parameter List* in the interface methods (smell-based strategy). Thus, he confirmed a design problem in the interface. In this example, the problem-based strategy did not lead to the design problem because the problem was in a different element. Based on this situation, we

Table 3: Percentage of success of each strategy

Strategy	No. of times applied	No. of contributions	Percentage of success
Pattern-based	23	19	82.61%
Quality attribute-based	19	15	78.95%
Element-based	40	31	77.50%
Smell-based	11	6	54.55%
Problem-based	22	8	36.36%
Principle-based	14	4	28.57%

counted how many times a strategy was applied, and how many times the strategy leaded to a design problem. We used these values to compute the percentage of success of each strategy.

Table 3 presents the percentage of success of each strategy. The first column indicates the name of the strategy, while the second column the number of times that the strategy was applied. The third column shows the number of occasions that a strategy led to the problem identification. Finally, the last column indicates the percentage of success. We calculated this value dividing the third column by the second column. Even though the element-based strategy was the most frequently used (40 times), the pattern-based strategy was the most successful. However, element-based and quality attribute-based strategies were quite close in terms of success. Surprisingly, the smell-based strategy was the less frequently used (11 times), and amongst the ones that was less successful in Scenario 1.

4.3 Problem Identification in Unfamiliar Systems

In this subsection, we answer the RQ3: *How do developers identify design problems in unfamiliar systems?*

Smells as predominant strategy. Different from the subjects of Scenario 1, the subjects of Scenario 2 did not use all the six strategies. Actually, it was quite the opposite. They only explicitly used the smell-based strategy. We observed that the subjects of Scenario 2 had strictly to rely only on code smells given their lack of familiarity with the systems. There were some design problems, such as *Cyclic Dependency* and *Fat Interface* [17], that we thought the subjects did not have to be familiar with the systems to identify them. For theses design problems, they could have applied the problem-based strategy or element-based strategy for their identification. They could have used other strategies in conjunction with the smell-based strategy. However, we did not find any tangible evidence that the subjects used another strategy. Their lack of deep knowledge about the criticality of the code elements was mentioned as the reason for not using other strategies.

Same strategy for each identification step. We noticed that the subjects of Scenario 2 used code smells to locate candidate fragments and also to confirm a design problem. After using the smells to guide them towards a candidate fragment, they used the presence of code smells to confirm the design problem. We also noticed that the subjects of Scenario 2 had to reason about more than one code smell to confirm a design problem. The subjects of Scenario 1 usually confirmed a design problem after finding a single code smell. On the other hand, the subjects of Scenario 2 tended to follow on searching for other smells in the same fragment. If they found a code smell in the source code, they searched for more code smells instead of confirming a design problem in the fragment as the subjects of Scenario 1 did. In fact, most the subjects of Scenario 2 analyzed almost all the code smells within a fragment.

Smell groups as complementary symptoms of a design problem. We observed that the subjects had to compensate the inability of using other strategies by exploring, even more, the code smells. For instance, some subjects of the Scenario 2 used the number of code smells in fragments to prioritize possible candidates. Thus, instead of using a problem-based or element-based strategy to locate a candidate fragment, they chose to inspect elements that contained several code smells. That was the approach followed by the S14 subject. When we asked him why he used the number of code smells to prioritize the analysis, he gave us the following answer:

> *"In my opinion, the main challenge to identify design problems was to filter out what fragments are most important to analyze. There is a lot of information to consider, thereby inducing you to identify design problems wrongly. I considered useful to analyze the (design) problems that seemed to me graver or more important. I considered as being grave those design fragments that concentrated more code smells."*

The S14 subject was not the only one that considered multiple code smells affecting the same fragment. As previously mentioned, some subjects reasoned about more than one code smell to identify a design problem. We noticed that some subjects analyzed code smells as a group instead of a unit. For instance, the S12 subject reasoned about the concomitant occurrence of *God Classes* and *Shotgun Surgery* to locate and confirm instances of *Scattered Concern* design problem. Figure 2 divides the subjects unfamiliar with the systems in two categories: subjects who grouped the code smells and subjects who did not group. In addition, it presents the results of precision, and in parentheses the number of design problems correctly identified by each subject. We noticed that the subjects who grouped code smells had a better result than the subjects who considered the code smells as units. As we can see in Figure 2, the subjects who grouped code smells identified more code smells than the subjects that did not group the code smells.

The subjects of the Scenario 1 used multiple strategies to get different symptoms of a single design problem. We noticed that developers of Scenario 2, especially those who grouped the smells, followed a similar behavior. However, instead of using multiple strategies, they used multiple code smells. In the case of subjects who grouped the smells, they used each instance of the code smell as a symptom of a single design problem. Thus, if an element had several code smells, they assumed that each smell was a symptom of a design problem. This result leads us to the third finding:

Finding 3. Developers unfamiliar with the systems use multiple code smells as symptoms of a single design problem

4.4 Identifying Complementary Symptoms

If developers cannot find enough symptoms that indicate a design problem, they might not be able to identify the problem. That was one of the reasons why the subjects of Scenario 1 fell short when they tried to use the smell-based strategy in the same way that subjects of Scenario 2 used it. In other words, they were unable to use the code smells to both locate candidate fragments and confirm

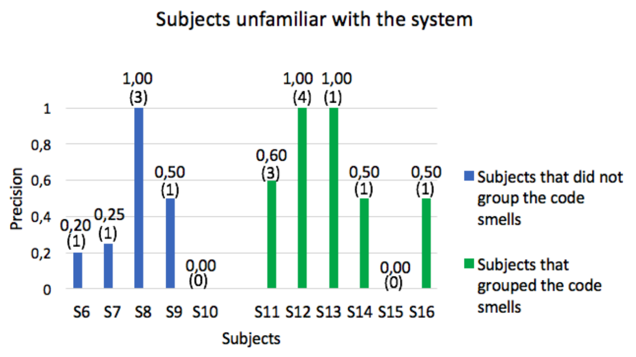

Figure 2: Precision of the subjects from the Scenario 1

the design problem. We noticed that after finding a candidate fragment using the smell-based strategy, the subjects of Scenario 1 dropped the analysis instead of further reasoning about other smells in the same fragment. Whenever this behavior happened, they did not succeed to use code smells to identify design problems in the target fragments.

When the subjects of Scenario 2 did not find enough symptoms, they also could not identify design problems. That was one of the reasons why some subjects in Figure 2 did not identify any design problem. They could not identify design problems because a code smell often represented only a partial hint of the design problem. These subjects were not able to use the code smells to both locate candidate fragments and confirm the problem.

These cases show that developers usually need to find multiple symptoms in the source code in order to identify a design problem. Indeed, we noticed that they often search for multiple symptoms in the source code, and such behavior is regardless their familiarity with the systems. For example, when the subjects used multiple strategies in Scenario 1, they are considering multiple symptoms that indicate a design problem. Also, when they combined the strategies, they had multiple indicators that the design fragment contained a design problem. Similarly, when the subjects of Scenario 2 analyzed multiple code smells, they were searching for multiple symptoms, in which each smell is a symptom to indicate a design problem. This result leads us to our last finding, which is a generalization of findings 2 and 3:

Finding 4. Developers search complementary symptoms in the source code that indicate a single design problem

5 RELATED WORK

To the best of our knowledge, we did not find any other investigation on how developers identify design. In general, studies assess if some technique or tool can indicate a design problem. For instance, Mo et al. [19] proposed the detection of recurring design problems by the combination of structural, history and design information. Xiao et al. [34] introduced a solution – based on a history coupling probability matrix - to identify and quantify design problems. The proposed solution uses 4 patterns of design flaws that show the correlation between design problems and reduced software quality. The aforementioned techniques depend on design information, which may not exist for many software systems.

Vidal et al. [33] presented and evaluated criteria for prioritizing groups of code smells that are likely to indicate design problems in evolving systems. In the context of their work, they focused on architectural design problems. [33] and our findings indicate the importance of investigating the concomitant use of multiple instances of code smells as stronger indicators of design problems. However, as already mentioned, we go beyond by presenting other strategies not based in code smells to identify design problems.

Oizumi et al. [21] investigated to what extent code smells could "flock together" to realize a design problem. These code smells that flock together and are related to each other composed what they authors called agglomeration. After analyzing more than 2200 agglomerations of code smells from seven software systems with different sizes and from different domains, the researchers concluded that certain forms of agglomerations are consistent indicators of design problems. Although we also have investigated multiple instances of code smells as indicators of design problems, our findings are more grounded on the in-depth observation of the developers' behavior than in quantitative results of retrospective studies. Moreover, similar results found on both studies helps to strength evidence that developers often reason about multiple symptoms to identify design problems in the implementation.

6 THREATS TO VALIDITY

Construct validity. We highlight the provided artifacts as threat to construct validity. For instance, the list of code smells may have influence subjects unfamiliar with systems to rely only on the smells. However, they were free to use or not each one the artifacts. We provide these artifacts because when they have to maintain systems, they need to have some minimal support to conduct the task. Even being a threat, the artifacts were useful to identify other characteristics that would not be noticed if these artifacts have not been provided. For instance, we notice that subjects of Scenario 2 grouped code smells to have a stronger indicator of design problems. Moreover, they combined the instances of code smells with the similar goal that subjects of Scenario 1 did when they combined the strategies. The goal was to identifying more symptoms. Another threat regarding this matter was the provided smells came from Fowler's catalog. Those are smells related to maintainability. However, this threat did not have much effect on the results since some subjects identified design problems related to other quality attributes as performance and robustness. In fact, the subjects even used a strategy that is based on quality attributes.

Internal Validity. We considered as threats to the internal validity: (i) different knowledge levels of subjects, and (ii) total time used for the experiment. To mitigate the first threat, all subjects underwent the training sessions. This procedure aimed to resolve any gaps in knowledge or conflicts about main concepts used in the study. Regarding the second threat, we conducted a pilot phase to adjust the time required to perform the identification tasks.

External Validity. The number of companies and developers represent threats to external validity. In order to mitigate this threat, we selected systems from different domains, different stage of degradation, and subjects that met a set of requirements. A second threat is related to the first author to introduce bias during the data analysis. First author's beliefs may cause some distortions when he interpreted the data. Data analysis was performed along with the other paper's authors to mitigate this threat. The other authors reviewed and analyzed all the intermediate results.

Conclusion Validity. This threat concerns the relation between treatment and outcome. We tried to mitigate it by combining data from different resources: quantitative and qualitative data obtained with videos, and questionnaires. We believe data collection and analysis were properly built to answer our questions

7 CONCLUDING REMARKS

We investigated how developers identify design problems, and we noticed six preeminent strategies that they used. The subjects of Scenario 1 combined multiple strategies to have different symptoms of design problems. Although the subjects of Scenario 2 had used only one strategy, they managed to use it in a similar way to those subjects who were familiar with the analyzed systems. That is, instead of using multiple strategies, the subjects of Scenario 2 used multiple code smells, in which each smell was likely to be a relevant symptom towards the identification of a design problem.

Our results indicate that developers search for several symptoms (e.g., several smells) to identify design problems regardless their familiarity with the systems. This finding may suggest that researchers should investigate how to offer flexible mechanisms for developers who wish to combine multiple strategies for identifying a design problem. For instance, a developer could select a subset of strategies, and the underlying mechanisms could: (i) automatically apply these multiple strategies, and (ii) rank the code elements that most likely to contain design problems. The ranking algorithm could be based on the number of symptoms detected with the selected strategies. Thus, as future work, we intend to investigate how to automate the combination of the six strategies. A summary of symptoms detected by the hybrid strategies could be presented to developers. Then, we can evaluate if developers can indeed identify design problems based on the symptoms' summary.

REFERENCES

[1] Holger Bär and Oliver Ciupke. 1998. Exploiting Design Heuristics for Automatic Problem Detection. In *Workshop Ion on Object-Oriented Technology (ECOOP '98)*. Springer-Verlag, London, UK, UK, 73–74.

[2] João Brunet, Gail C. Murphy, Ricardo Terra, Jorge Figueiredo, and Dalton Serey. 2014. Do Developers Discuss Design?. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. New York, NY, USA, 340–343.

[3] Stephen Cass. 2016. The 2016 Top Programming Language. (July 2016). http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

[4] Xi Chen, Abhijit Davare, Harry Hsieh, Alberto Sangiovanni-Vincentelli, and Yosinori Watanabe. 2005. Simulation Based Deadlock Analysis for System Level Designs. In *Proceedings of the 42Nd Annual Design Automation Conference (DAC '05)*. ACM, New York, NY, USA, 260–265. https://doi.org/10.1145/1065579.1065647

[5] 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.

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

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[8] 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.*

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

[10] M Godfrey and E Lee. 2000. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *CoSET-00; Limerick, Ireland.* 15–23.

[11] P. Kaminski. 2007. Reforming Software Design Documentation. In *14th Working Conference on Reverse Engineering (WCRE 2007)*. 277–280.

[12] Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall PTR, Upper Saddle River, NJ, USA.

[13] A MacCormack, J Rusnak, and C Baldwin. 2006. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Manage. Sci.* 52, 7 (2006), 1015–1030.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Complementar Material. 2017. https://ssousaleo.github.io/SBES2017/. (2017).

[19] 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.

[20] N Moha, Y Gueheneuc, L Duchien, and A Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transaction on Software Engineering* 36 (2010), 20–36.

[21] 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.*

[22] David L. Parnas. 1978. Designing Software for Ease of Extension and Contraction. In *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78).* IEEE Press, Piscataway, NJ, USA, 264–277.

[23] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *Case Study Research in Software Engineering: Guidelines and Examples.* Wiley Publishing.

[24] S Schach, B Jin, D Wright, G Heller, and A Offutt. 2002. Maintainability of the Linux kernel. *Software, IEE Proceedings* - 149, 1 (2002), 18–23.

[25] 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.

[26] TIOBE software. 2017. The Java Programming Language. (April 2017). https://www.tiobe.com/tiobe-index/java/

[27] A. Strauss and J.M. Corbin. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* SAGE Publications.

[28] Antony Tang, Aldeida Aleti, Janet Burge, and Hans van Vliet. 2010. What makes software design effective? *Design Studies* 31, 6 (2010), 614 – 640. Special Issue Studying Professional Software Design.

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

[30] Adrian Trifu and Urs Reupke. 2007. Towards Automated Restructuring of Object Oriented Systems. In *CSMR '07.* IEEE, Washington, DC, USA, 39–48.

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

[32] J van Gurp and J Bosch. 2002. Design erosion: problems and causes. *Journal of Systems and Software* 61, 2 (2002), 105 – 119.

[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 *SBCARS16.* 41–50.

[34] 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, 488–498. https://doi.org/10.1145/2884781.2884822

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

[36] A Yamashita and L Moonen. 2012. Do code smells reflect important maintainability aspects?. In *ICSM12.* 306–315.