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 negative impact on one or more quality attributes, such as comprehensibility and maintainability, we call it a design problem. For instance, the Fat Interface problem indicates that an interface exposes non-cohesive services: clients and implementations of this interface may have to handle with services which they are not interested. Thus, the prevalence of a design problem such as this may hamper 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 the harmfulness of design problems, it is difficult to identify them in source code due to the unavailability or unreliability of design documentation. To alleviated this difficulty, we conducted a qualitative analysis on how developers identify design problems of systems in two different scenarios: (1) one with a system that they are familiar with and (2) the other system they are not. In the former case, the developers applied a diverse set of strategies during the identification of each design problem. Some strategies were more successful in helping developers to find candidate source code locations for inspection, and other strategies were better to support their decisions on whether those locations indeed contained a design problem. In the latter case, the developers unfamiliar with the systems relied only on the use of code smells along the task. These findings suggest that developers need mechanisms less rigid in terms of enforcing a single strategy to guide identification of design problems.

CCS CONCEPTS

•Software and its engineering →Software design engineering;

KEYWORDS

software design, design problem, 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]

XXXI SBES, Fortaleza, Ceará, Brazil

DOI: 10.1145/nnnnnnn.nnnnnnn

. 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.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software design is a fundamental concern during the software development process [6, 11]. No wonder 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 and characteristics 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, 4, 18, 29]. An example of design problem is the *Fat Interface* [16]. This problem occurs when an interface offers a general entry point for several non-cohesive services, complicating the logic of its clients.

Design problems are so harmful that software systems have been discontinued or reengineered due to their prevalence [9, 12, 23, 32]. They also have been related to the rejection of almost 40% of pull requests [24]. Given the harmfulness of design problems, developers need to remove them from software systems as early as possible [8, 23, 37]. However, their identification is not a trivial task [4, 28]. One of the key 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 [10, 29].

In the implementation level, a design problem is reified as a structure (e.g. a set of classes or methods) in which symptoms of the problem can be observed. These problematic structures and their 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. An identification strategy refers a set of actions applied by developers to reveal symptoms of a design problem. Some symptoms can be detected with metrics, code smells, and change history of elements.

As design problems are harmful and difficult to identify, it is no wonder why developers and researchers are concerned with them. Even so, it is surprising that we know so little about how developers actually identify design problems. In particular, no study has observed what identification strategy (or strategies) is used by developers in practice. For instance, we do not know what strategies developers use to identify design problems and whether

they can successfully identify problems using them. Moreover, we do not know if they use the same strategy or different ones to identify design problems regardless of their knowledge on the target systems.

In order to address this gap, we conducted an exploratory study to observe what strategy(ies) developers use to identify design problems in the source code. We observed how developers identify design problem in two scenarios: one with a system that they are familiar with and the other scenario with systems they are not familiar. Our analysis resulted in several findings:

- In general, developers use multiple strategies to identify design problems. In this study, they used six preeminent strategies: smell-based, principle-based, problem-based, element-based, quality attributes-based, and pattern-based. These strategies are described in Section 4.1.
- We observed that the developers used the element-based and problem-based strategies to locate a design problem and then used other strategies to confirm this.
- Developers familiar with the target systems tried to use all available strategies. Developers unfamiliar with the systems used only the smell-based strategy.
- Regardless the familiarity with the target systems, developers searched for multiple symptoms of design problems in the source code. In fact, our results indicate that developers need mechanisms flexible in term of providing multiple strategies and symptoms to guide the identification of design problems.

The remainder of this paper is organized as follow. Section 2 presents the background to understand the paper. 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 process [27]. It is expected that these decisions contribute to creating software systems that are comprehensible, maintainable, robust, secure, and the like. However, along the way, design problems can be injected into software systems. A **design problem** occurs when a part of the software design, i.e., a design fragment, negatively impact the software quality attributes. In fact, as the presence of design problems has negative consequences, they are often targets of significant maintenance efforts [8, 23, 37].

There are different types of design problem. Some problems affect interfaces, others affect components, hierarchies or even other abstractions that are relevant to the design. Examples of design problems include *Scattered Functionality* [8], *Ambiguous Interface* [8] and *Cyclic Dependency* [5]. *Fat Interface* [16] is another example of design problem. This problem indicates that an interface exposes a general entry point for several non-cohesive services, thereby complicating the logic of its clients. For instance, suppose that a class will implement an interface with this design problem, and suppose that this class is designed to provide only one of the interface's services. However, since this interface exposes more than one service, the class will be forced to handle requests to other services as well, even without providing implementations to all of them. In this case, this problem affects the extensibility as well as the comprehensibility of the system since developers have to understand several functionalities rather than one.

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 process, 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, they need to follow at least two basic steps to perform the identification. First, they need to detect a structure in the implementation that is a *candidate* to embody a design problem. Next, developers need to confirm the existence of the design problems in those reifications of design fragments in the implementation.

Not only are design problems harmful, but their identification is also cumbersome [4, 28]. 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, systems are designed and implemented by multiple people [34]. Not all of them have knowledge of the entire system. Third, each design problem usually pervades the implementation of several elements [8, 20]. Thus, developers need to analyze several elements to identify a single design problem [28].

Design documentation is often nonexistent, informal or not up-to-date. As the source code is the only artifact available for the developers in most cases, developers need to use it to identify design fragments that contain design problems. Thus, they need to locate code elements that comprise the structure of a design fragment. Then, they need to verify if these elements have any symptom that indicate a design problem. An example of symptom was prosed by Martin Fowler [5]. He has proposed a catalog of code smells that can indicate design problems. Other examples include quality metrics [20], change-prone elements [19] and non-functional attributes.

3 STUDY PLANNING

This section presents the study design reported in this paper.

3.1 Research Questions

Identification of design problems is a challenging task. Thus, developers need to use strategies to reveal symptoms that can indicate a design problem. Nevertheless, these strategies are rarely investigated. In order to address this matter, we observed how developers could identify design problems to answer our RQ1:

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

A major contribution of this paper is to provide results that may lead researchers to build better solutions for the identification of design problems. However, before providing solutions, first, we need to understand how developers perform this task. Regarding this matter, familiarity with the systems is a variable that may have an influence on how developers identify design problems. Thus, we address this point through the following research questions:

RQ2 How do developers identify design problems in familiar systems?

RQ3 How do developers identify design problems in unfamiliar systems?

To address RQ1, we conducted a qualitative analysis using a sub-set of Ground Theory (GT) procedures [26]. We addressed RQ2 and RQ3 by investigating two scenarios: when developers are either **familiar** (Scenario 1) or **unfamiliar** (Scenario 2) with the analyzed systems.

3.2 Studied Scenarios and Subjects

In the following, we explain in details the studied scenarios and the procedure to recruit subjects for the study.

1) Developers familiar with the system

In the first scenario, we searched for software companies that could provide us 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 [22]. Based on these criteria, we chose two Brazilian software companies. After that, 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 in terms of amount of modules and involved developers. Thirdly, systems with several commits, which means they were not in initial versions. Lastly, systems developed in Java. As each selected company has to provide software systems, we selected Java because its popularity [3, 25]. 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 [17]. After providing us with the programs, we asked the companies' managers to indicate developers that were familiar with each program and could be subjects of the study.

2) Developers unfamiliar with the system

In the second scenario, we selected two programs from the Apache OODT project to developers identify design problems. We selected Apache OODT systems because they have a well-defined set of design problems identified by developers who actually implemented the systems (Section 3.4). The system programs are:

- Push Pull (P6): it is responsible for downloading remote content (pull) or accepting the delivery of remote content (push) to a local staging area.
- Workflow Manager (P7): This is a subsystem of a client-server system, responsible for describing, executing, and monitoring workflows.

Table 1: Characterization of the subjects

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

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, that could introduce a bias as they already knew about the experiment. Thus, in order to avoid the learning curve bias, we decided to recruit another set of developers. We decided to select subjects who have similar characteristics of the five subjects from the first scenario. Thus, we sent a questionnaire to several developers from our network in order to select developers that 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 [36] and Twitter [30] consider to developers as experienced, and four years was the least experience time of one of the subjects in the first scenario. Besides, we also selected subjects with at least intermediary knowledge on Java programming. We included in the questionnaire a description of our definition of knowledge level, thereby allowing developers to have a similar interpretation of the possible answers. Details about level's definition and recruitment process are available on our online material [17].

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, 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 were assigned to identify design problems in their own systems, while the other subjects had to identify design problems in the system they were no familiar with.

3.3 Study Activities

The study was composed by three activities:

Activity 1: Training. In this activity, first and second authors conducted the 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 were included in the training session: *Ambiguous Interface*, *Unwanted Dependency*, *Component Overload*, *Cyclic Dependency*, *Scattered Concern*, *Fat Interface*, and *Unused Abstraction*. We opted by selecting these design problems since they are often considered as critical, representing common reason of major refactorings [13–15]. 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, such as violations of design pattern rules (see 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.

Subjects also 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 [13–15, 21]. We used well-known metrics-based strategies to identify 15 types of code smells from Fowler's Catalog [5]. 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 and the description of code smells can be found in our online material [17].

Activity 2: Problems identification. In this task, we asked subjects to identify design problems, and to report their findings in an online form [17]. 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.

Activity 3: Follow-up questionnaire. Developers were asked to answer a questionnaire about their general perception on the identification of design problems [17]. The answers were also used to complement the qualitative analysis.

3.4 Data Analysis and Oracle Creation

We used the data collected from the questionnaire. In addition, we video recorded the task. Camtasia[1] tool was used to record audio and screenshots of each computer used by developers. A video camera was installed in the room to also record the subjects.

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

Data Analysis. The qualitative data analysis was based on the procedures of Grounded Theory (GT) suggested by Strauss and Corbin [26]. The procedure has 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 [26]. 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. Such analysis was useful to answer our research questions (Section 4).

Oracle Creation. For each one of the analyzed systems, we had to validate the identified design problems. 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 checked subjects' answers using the video records. We analyzed each subject's answer, and we asked developers and designers 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, 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 [7]. 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 [13], (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 identified 70 instances of design problems. Those familiar with the systems (Scenario 1) identified 39 design, in which 31 were validated as true positives according 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. This section discusses these results.

4.1 Strategies to Identify Design Problems

Here, we answer RQ1: *What are the strategies that developers use to identify design problems?* After analyzing codes that emerged from GT procedures (Section 3.4), we noticed that the subjects used a diverse range of strategies. Our results revealed six preeminent strategies: **smell-based**, **problem-based**, **principle-based element-based**, **quality attributes-based**, and **pattern-based**. They are described as follow. Due to the confidentiality with the companies, we changed the code element name 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 [13–15], developers can use smells as an indicator 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 it 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 affecting 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 [16] (e.g. open-closed principle and information hiding) to identify design problems. This strategy was only used by the subjects of the first scenario. The subjects tried to use this strategy to locate candidate fragments, but they did not succeed to find any candidate fragment. However, they succeed in using this strategy to confirm if an analyzed fragment 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 the problem-based 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* [16] 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 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 symptom (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 this 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 who are familiar with the system. 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 attributes-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, developers also often 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 attributes-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 (overuse of *Adapters*). 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 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 subjects. 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"

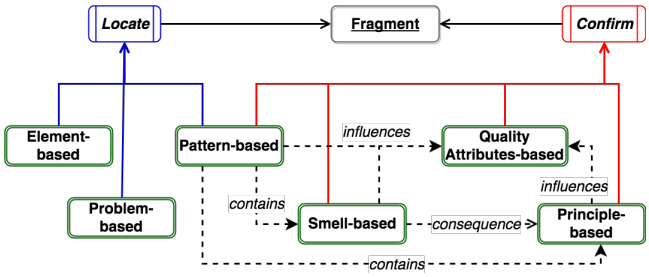

Figure 1: Relation among the strategies

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?* As presented in the previous subsection, the subjects of Scenario 1 used all six strategies.

Two-stage strategies. The strategies can be divided into two groups based on how the subjects had success to use them across the problem identification steps: (i) some strategies were more successful in helping the subjects to find candidate program locations for inspection, and (ii) other strategies are better to support their decisions on whether those locations indeed contain a design problem. We noticed that in most cases, the subjects of Scenario 1 had more success in using element-based and problem-based strategies to detect candidate fragments, while smell-based, principle-based and quality attributes-based strategies were used to confirm if fragments have a design problem indeed. Pattern-based strategy was used both to locate a candidate fragment and to confirm a design problem. Figure 1 shows the both stages (blue and red squares) that the subjects most successfully used each strategy (green squares).

Strategy blends. It was often the case that the subjects used a strategy to locate a design fragment and then used another strategy to confirm if the fragment actually contains a design problem. Indeed, we noticed that the subjects tended to combine the six strategies. For instance, whenever the subjects used the pattern-based strategy, they were looking for violations of a design or architecture pattern in order to detect candidate fragments. However, 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 violating structure of the candidate fragment was either explicitly affecting a quality attribute or hosting one or more code smells. These combinations happen because each strategy leads to a symptom, and these symptoms are related to each other. For instance, if a fragment violates a pattern, it is likely that the fragment contains smells and violations of design principles. Consequently, these symptoms can influence quality attributes negatively. Figure 1 also shows the relation (dotted arrows) among the strategies.

Single vs. combined strategies. In many cases, the subjects combined multiple strategies. We noticed that they combined these strategies to find complementary symptoms of a design problem in the candidate fragment. The symptoms are complementary in the sense of each symptom adds a piece of information that will help subjects to decide if there is a design problem in the fragment. As an example, in the quotation that illustrated the quality attributes-based strategy (Section 4.1), we could see that the S1 subject combined the pattern-based and quality attributes-based strategies. Before discussing the impact on reusability, they used first the pattern-based strategy to identify classes that could be violating the *Adapter* pattern. After that, they used the quality attributes-based strategy to reason through the consequences that the design problem caused on the reusability attribute. Some subjects also combined more than two strategies in order to identify a single design problem.

Table 2 illustrates the number of design problems (true positives are inside of the parentheses) found by subjects in familiar systems. The first column indicates the strategy or combination of strategies that one or more subjects used to identify design problems. The second column indicates how many times the strategy or combinations were applied and led to a design problem. The third column indicates the design problems found when the subject used the the strategies. For clarity, we abbreviated the types of design problems as follow: 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*. 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 [17]. Finally, the last column indicates which subjects used the strategies.

The second column also indicates the total number of design problems found when the subjects used the strategies. We obtained these values by applying the procedures of the GT method (Section 3.4). We applied the method to the online form and the video transcription to count each strategy that the subjects used and led to the identified design problem. For example, whenever subjects identified a design problem, we asked them to write in the online form the description of the design problem and the elements affected by the problem. In addition to that, we used the video transcription to detect the first time that the subject started a new round of problem identification. Then, we analyzed all the actions that he did since the first time that he mentions a symptom (usually associated with a particular strategy) until the moment he confirms the design problem. If one of these actions was the 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). On the other hand, when subjects combined multiple strategies, they identified 27 design problems (21 true positives). 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

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 attributes	1 (1)	ICA	S1
Problem, Quality attributes	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 attributes, Pattern	2 (2)	UWD, CCO	S1, S3
Element, Quality attributes, 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 attributes, Pattern	1 (0)	UWD	S3
Principle, Quality attributes, Pattern, Smell	2 (1)	UWD, CCO	S4

of design problems might be more complex than one can expect. The subjects of Scenario 1 combined all six strategies at least once. 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

Most Successfully Strategy. It was not always the case that a strategy contributed to identifying design problems directly. For instance, a subject can use the element-based strategy to locate a candidate fragment to analyze; however, he realizes that the fragment does not have a design problem. Then, he used the strategy again and found a fragment that contains a design problem. In this example, the element-based strategy was applied twice, in which one led to a design problem successfully. Upon the analyzed data, we were able to verify the cases of each strategy led to a design problem successfully. Table 3 presents these values. The first column indicates the name of the strategy, the second column the number of times that the strategy was applied. Third column indicates the number of times that strategy contributed to identifying a problem. Finally, the last column indicates the percentage of success. We calculated this value dividing the third column by the second column. As we can see, the pattern-based was the most successful strategy.

4.3 Problem Identification in Unfamiliar Systems

Here we answer the RQ3 (*How do developers identify design problems in unfamiliar systems?*) taking into account only the subjects of Scenario 2. 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.

Table 3: Number of times that a strategy contributed to identify a design problem

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

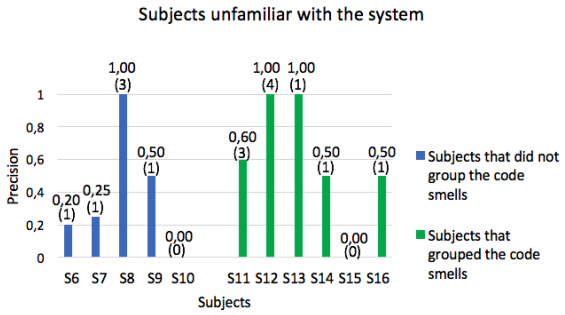

Figure 2: Precision of the subjects from the Scenario 1

Smells as predominant strategy. We observed that the subjects of Scenario 2 had to strictly rely only on code smells given their lack of familiarity with the systems. They could have eventually and implicitly used other strategies in conjunction with the smell-based strategy. However, we did not find any tangible evidence that the subjects used another strategy. For example, there are some design problems, such as *Cyclic Dependency* and *Fat Interface* [16] and we thought the subjects did not have to be familiar with the systems to identify them. Thus, they could have applied the problem-based or element-based strategies for the identification of these design problems. To our surprise, the subjects of Scenario 2 used only the smell-based strategy to identify design problems.

Reasoning about single smells may not suffice. 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. 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. That is, the subjects had to reason about more than one code smell to identify a design problem. In fact, most the subjects of Scenario 2, on the other hand, analyzed 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 absence of other strategies by using more code smells as indicators of design problems. For instance, 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 to locate a candidate fragment, they chose to inspect elements that contained several code smells. That was the approach followed by 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 these subjects analyzed code smells as a group instead of a unit. They used group of code smells to locate candidate fragments, and group of code smells to confirm the design problem. The subjects that grouped code smells had a better result than the subjects that considered the code smells as units. Figure 2 divides the subjects unfamiliar with the systems in two categories: subjects that grouped the code smells and subjects that did not group. In addition, it presents the results of precision, and in parentheses the number of design problems correctly identified by each subject. More than half of the subjects grouped the smells. As we can see in Figure 2, the subjects that grouped code smells identified more code smells than the subjects that did not group the code smells. The subjects that grouped the smells used each instance of the code smell as a symptom. 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 Identification of complementary symptoms

If developers cannot find enough symptoms that indicate a design problem, they might not be able to identify the problem. That is one of the reasons why the subjects of Scenario 1 fell short of when they tried to use the smell-based strategy to locate candidate fragments. After finding a candidate fragment using the smell-based strategy, they dropped the analysis instead of reasoning about other smells or other symptoms in the fragment. Whenever this behavior happened, we noticed they did not succeed to use code smells to identify design problems in the target fragments.

The subjects unfamiliar with the systems had similar issues. That may be the reason why some subjects in Figure 2 did not identify any design problem. They could not identify a design problem because a code smell represents only a partial hint. If subjects want to use the code smells to locate candidate fragments, they may need to consider more than one smell.

When the subjects used multiple strategies in Scenario 1, they are considering multiple symptoms that indicate a design problem. Each strategy is likely to provide a symptom. Thus, when subjects combine multiple strategies, they have multiples indicators that the design fragment contains a design problem. Similarly, when the subjects of Scenario 2 analyzed multiple code smells, they are

using each smell as a symptom to indicate a design problem. That is, subjects search for multiples symptoms in the source code, regardless their familiarity with the systems. 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

4.5 Implications

We must first understand how developers identify design problems, after that we provide solutions that will help them to locate and to remove design problem. By understanding how developers identify design problems, researchers will be better able to build tools to help developers identify design problems effectively. One of the major purposes of this paper is to show results that will facilitate researchers in their quest to build better design problem identification solutions.

Some techniques have been proposed in the literature to identify few symptoms of design problems. In general, these techniques only provide one strategy for the developers. Some of them use code smells to identify design problems [21], while other strategies rely on the system design's history information [35], or even in quality metrics [20]. Unfortunately, these techniques are rigid in that they do not allow developers to explore the identified symptom or symptoms. Similarly, there are also many commercial and open source tools for helping developers. DECOR [20], SonarQube[2] and Understand [31] are examples of tools that use collections of source code metrics, and the detection of code smells to help developers identify design problems. However, these tools face the same problem of the current techniques; they are too rigid to support developers applying and combining their own strategies. SonarQube is the closest one that allows developers to collect multiple symptoms (not all of them related to design problems). However, it does not direct developers towards design problems since identifying them is not exactly its goal.

Current solutions may not be able to help developers with the identification of design problems because of a possible mismatching between what these solutions offer and what developers actually need. Our results indicate that developers do indeed combine different strategies to get multiple symptoms of design problems. As our study shows, a software developer may use many different strategies when looking for design problems. However, none of the existing techniques provide flexible strategies, which would allow developers to analyze suspicious source code elements from different perspectives. Therefore, researchers should either adapt solutions to be flexible or develop new ones that allow developers to combine multiple strategies in the same way they naturally do.

5 RELATED WORK

As far as we are concerned, none study has investigated how developers identify design problems. In general, studies assess if some technique 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. [35] 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. 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' behaviors 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 were even able to use 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

[2]https://www.sonarqube.org/

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 multiples strategies to have different symptoms of design problems. Although the subjects of Scenario 2 had used only one strategy, they managed to use the strategy in a similar way that the subjects familiar with the analyzed systems. In other words, instead of using multiple strategies, the subjects of Scenario 2 used multiple code smells, in which each smell was likely to a be a symptom.

Our results indicate that developers search for several symptoms – or several smells – to identify design problems regardless their familiarity with the systems. Therefore, the current solutions to help developers to identify design problems need to be more flexible regarding the provided symptoms and strategies. Developers do not only need multiple symptoms but also need to combine them in their own way to identify design problems. This information is useful for researchers that will create solutions to help developers to identify design problems. Furthermore, our results also showed which strategies that developers seem to apply during the identification of design problems naturally. It is worthwhile to put an effort in automate the application of these strategies.

As future work, we intend to investigate the six strategies. For example, how to support the automation of combined strategies. We intend to verify if the developers can identify design problems when they receive the symptoms provided by strategies. Our goal is to apply these strategies automatically, collect the symptoms indicated by them, and provide the symptoms summarized. Then, we will assess if developers unfamiliar with systems are able to identify design problems using our summary of symptoms.

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] 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.

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

[6] 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.

[7] 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*.

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Complementar Material. 2017. https://icpcsubmission.github.io/2017/. (2017).

[18] P. F. Mihancea and R. Marinescu. 2005. Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems. In *Ninth European Conference on Software Maintenance and Reengineering*. 92–101.

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

[23] 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.

[24] 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.

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

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

[27] 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.

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

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

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

[31] Understand. Understand: User Guide and Reference Manual. (????). https://scitools.com/documents/manuals/pdf/understand.pdf

[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] J. Wu, T. C. N. Graham, and P. W. Smith. 2003. A study of collaboration in software design. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.* 304–313. https://doi.org/10.1109/ISESE.2003.1237991

[35] 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

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

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