How Do Software Developers Identify Design Problems? A Qualitative Analysis

Anonymous 1
University 1
City 1, Country 1
anonymous1@university1.country1

Anonymous 2
University 2
City 2, Country 2
anonymous2@university2.country2

Anonymous 3
University 3
City 3, Country 3
anonymous3@university3.country3

***Abstract*—Design problems represent parts of the software design that negatively impact one or more quality attributes, such as comprehensibility, maintainability and the like. Their harmfulness can make software systems to be restructured or even discontinued. However, many design problems prevail in the source code, as their identification is often cumbersome. Unfortunately, there is limited knowledge about how developers identify design problems when the source code is the only artifact available. There is an apparent popularity of smells as a strategy to find symptoms of design problems. However, we do not know if developers can actually identify design problems through their use. Moreover, little is known if developers can use code smells as a successful strategy regardless their familiarity with the analyzed systems. In this context, we conducted a qualitative analysis of how developers identify design problems in systems they are familiar or unfamiliar with. Our results indicate developers applied diverse strategies to identify design problems. Some strategies were mainly used to locate candidate fragments to inspect, while other strategies were used to confirm the existence of a design problem in the located fragment. Developers also tend to combine various strategies to identify a single design problem in systems they had familiarity. Our results also show that developers use multiple code smells as complementary symptoms to identify a single design problem in systems they did not have familiarity.**

I. Introduction

Several software systems are discontinued or reengineered due to the prevalence of design problems in the source code [1], [2], [3], [4]. A design problem is a part of the software design (a design fragment) that negatively impacts one or more quality attributes, such as comprehensibility, maintainability and performance [5], [6], [7], [8]. For instance, *Fat Interface* is an example of design problem that simultaneously hinders all these three quality attributes [9]. Given the harmfulness of design problems, developers need to remove them from the system as early as possible [3], [9], [10]. However, design problems can be removed only after they are identified.

The identification of design problems is often a cumbersome task [5], [11]. One of the key reasons is that developers often have to identify design problems in the source code rather than in the design documentation. Unfortunately, design documentation is almost always nonexistent, informal or not up-to-date [6], [12]. In the implementation, 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, even more where developers have limited or no familiarity with the source code. Thus, developers need to use a **strategy** to identify occurrences of design problems in the implementation. An identification strategy is a set of actions applied by developers to reveal symptoms in their quest for identifying a design problem.

The use of code smells [13] is considered to be a preeminent strategy to identify symptoms of a design problem. A code smell is a microstructure in the program that represents a surface indication of a design problem [13]. Several studies [11], [14], [15], [16], [17], [18] seem to assume that code smells are the main strategy to identify design problems in the source code. This is indeed a sensible assumption as each code smell represents a symptom potentially associated with a design problem [11], [17], [18]. Moreover, smells can be identified directly in the source code, which is often the only tangible artifact available for developers. Code smells also can be somehow objectively identified in software systems [18], [19] where developers have limited familiarity.

Unfortunately, little is known about how developers actually identify design problems. In particular, no study has observed what identification strategy (or strategies) is used by developers in practice. Despite apparent popularity of code smells, we do not know if developers use code smells and if they are able to successfully identify design problems using them. Moreover, we do not know if developers can identify design problems regardless their knowledge of the analyzed systems. For example, developers might identify design problems using code smells only in systems they are familiar with.

In this context, we conducted a qualitative study to observe how developers identify design problems in the source code. We used Grounded Theory procedures [20] to conduct our investigation. We observed developers in two scenarios, i.e. when they are either familiar or unfamiliar with the analyzed systems. The second scenario represents situations where developers need to identify design problems in legacy systems or in parts of a system where the designer and developers are no longer available. We provided developers with an initial list of code smells affecting the systems in both scenarios. The developers had the freedom to use or not the initial list of code smells. Thus, we could investigate if developers rely on code smells as a dominant strategy to identify design problems and to what extent they can identify design problem using code smells. We could also observe if developers have used other strategies to identify design problems.

The study results indicate that developers applied diverse strategies to identify design problems. We characterized five strategies used by developers. Some strategies were mainly used to locate candidates of design fragments containing design problems, while other strategies were mostly used to confirm the existence of a design problem. Certain strategies were employed with both purposes. We also observed that developers often combine multiple strategies to locate a single design problem in systems that are familiar to them. We also noticed that developers were able to identify design problems even in systems with which they were unfamiliar. In this scenario, code smells was indeed the dominant strategy. However, regardless the scenarios, we noticed developers search various complementary symptoms, which may indicate together the manifestation of a single design problem.

The paper is organized as follow. Section II presents the background to understand the paper. Section III describes the settings of our study, including the research questions and the experimental procedure followed to conduct the study. Section IV summarizes the main results of our study. Sections V and Section VI present the related work and the threats to validity, respectively. Finally, Section VII concludes the paper.

II. Background

Software design is the product of a series of decisions made during the software development process [21]. It is expected that these decisions contribute to creating software systems that are comprehensible, maintainable, robust, secure, and the like. However, designers do not have the ability to predict all possible changes [22] and certainly cannot predict if the software design will meet the quality attributes. Therefore, as the system is maintained and evolved, they need to identify design problems that are negatively impacting quality attributes in the system [6], [8]. In fact, as the presence of design problems has consequences on quality attributes, they are often targets of significant maintenance effort [3], [9], [10].

A **design problem** occurs when a part of the software design (a design fragment) negatively impact quality attributes, such as comprehensibility, maintainability, reusability and performance [23], [24], [25]. For instance, *Fat Interface* is an example of design problem [9]. This problem is a general entry point of a design component that provides several non-cohesive services, thereby complicating the logic of its clients. There are different categories of design problems. Some problems affect interfaces, others affect components, hierarchies or even other abstractions that are relevant to the design. Other examples of design problems include *Scattered Functionality* [9], *Ambiguous Interface* [9] and *Cyclic Dependency* [13].

Developers should identify and remove design problems in order to combat design degradation. 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. From now on, let us assume that developers are in charge of conducting identification of design problems in the source code. In this process, developers identify design problems in the counterpart structure realizing the relevant design fragment in the program. Developers follow at least two basic steps to perform design problem 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 [5], [11]. 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 [26]. Not all of them have knowledge of the entire system. Third, each design problem usually pervades the implementation of several code elements [9], [18]. Thus, developers need to analyze several code elements to identify a single design problem [11]. Also, often developers cannot rely on the design documentation.

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 symptoms that indicate a design problem. Code smells are examples of symptoms that may (partially) indicate a design problem [11], [17], [18].

A **code smell** is a structure in the implementation of the program that possibly indicates a design problem [13]. An example of a code smell is *Feature Envy*. This smell refers to a method that is more interested in other classes than the one to which it belongs [13]. Other examples of code smells include *Long Method*, *Long Parameter List*, *God Class*, *Shotgun Surgery*, and *Divergent Change* [13], [27]. Code smells have been used as a preeminent strategy to identify design problems [14],[28],[29]. First, they can be seen as a objective means to find symptoms associated with a design problem and quality attributes [30], [31], [32]. This is one of the reasons why some studies use code smells as indicators of design problems [11], [18]. Second, code smells are directly observable in the source code. Thus, developers can reason about smelly elements to identify structures realizing a design problem. Code smells might represent partial symptoms of any of the aforementioned design problems [14], [15].

III. Study Planning

This section presents the study design reported in this paper.

A. Research Questions

Identification of design problems is a challenging task (Section II). Developers need to locate candidate fragments and analyze their implementation structure before deciding if they have or not a design problem. In order to do that, they use a strategy to conduct the problem identification. However, the strategies that developers use to identify design problems are rarely investigated. There is often an assumption that developers use code smells as a strategy to identify a design problem in the implementation. According to some studies [11], [14], [15], [17], [18], code smells often represent symptoms of design problems. Therefore, one would expect that code smells represent the dominant strategy followed by developers to identify design problems. In order to address this matter, we are observing how developers identify design problems with the purpose of answering the following question:

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

As previous studies suggest that code smells can be used as indicators of design problems [14], [15], [16], [33], [34], we provided developers with a list of code smells affecting each one of the software systems used in our study. We highlight that despite providing code smells, developers were free to use them or not. In case they use code smells, we could investigate if smells can indeed support the problem identification or not.

We addressed this research question in two scenarios, i.e. when they are either **familiar** or **unfamiliar** with the analyzed systems. In the first scenario, we asked developers to identify design problems in software systems that they implemented. In the second scenario, we asked a second group of developers to identify design problems in software systems developed by others. Thus, we can answer the following research question:

RQ2 How do developers identify design problems in familiar and unfamiliar systems?

Investigating both scenarios is worthwhile because it allows us to find out similar and varying behaviors of developers in such different settings. We could also investigate if the use of code smells is effective in both or any of these two scenarios.

B. Studied Scenarios and Participants

The following we explain in detail the studied scenarios and the procedure to recruit developers for the study.

1) Developers familiar with the system

In the first scenario, we searched for software companies that could provide us with: (i) developers, and (ii) systems where identification of design problems was currently seen as critical. Thus, we could naturally ask developers to identify design problems in their systems. These developers would also be motivated to perform tasks for identifying design problems given the criticality of such tasks in their own project contexts. We defined several criteria to select the companies, including the level of 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 [35].

We chose two software companies to conduct the study after the search. After that, we asked the companies' managers, some of them were designers, to suggest specific systems that met the following criteria. First, systems in different stages of design degradation. Second, systems with a comprehensive set of code smells (Section III-E). Third, systems from different domains, and size in terms of amount of modules and involved developers. Fourth, systems comprising several commits, which means they were not in initial versions. Last, systems developed in Java. The five selected systems that met the criteria are described as follows. Full details about them are in our online material [36]. We omitted some details, such as the companies' country due to the double-blind reviewing.

TABLE I
CHARACTERIZATION OF DEVELOPERS FAMILIAR WITH THE SYSTEMS

ID	Experience as developer (years)	Education	System
D1	12	Graduate	S1
D2	13	Graduate	
D3	4	Graduate	S2
D4	8	Certificate Program	
D5	4	Master	S3
D6	10	Graduate	
D7	7	Certificate Program	S4
D8	7	Graduate	
D9	9	Graduate	
D10	12	Graduate	S5
D11	9	Certificate Program	

- Company 1: System 1 (S1) supports the management of registry offices for audit and control from the Justice Court of the country to which the company belongs. System 2 (S2) is a computational solution for maintaining information on the patients' health status, and their medical records. System 3 (S3) is a system developed to trace products from a production line.
- Company 2: System 4 (S4) is a legacy system to process tax and to controll the entrance of products from the state that the company belongs. System 5 (S5) was developed for standardizing budget in the same state.

After providing us with the systems, we asked the companies' managers to indicate at least two developers that were familiar with each system. Managers mentioned that developers often perform perfective maintenance tasks in pairs. Thus, we decided to allow developers of each system to identify design problems together. We asked them to explain aloud what they were doing while we video recorded the task. Table I presents the characterization of the developers assigned to identify design problems in their systems. Developers are divided based on the systems that they work. Thus, if two developers work together in the same system, they form a team. Notice that one of the teams is composed of three developers because the company provided an additional developer to this team.

2) Developers unfamiliar with the system

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

TABLE II
CHARACTERIZATION OF DEVELOPERS UNFAMILIAR WITH THE SYSTEMS

ID	Experience as developer (years)	Education Level	Knowledge	
			Java	Code Smells
D12	5	PhD	Advanced	Advanced
D13	6	Graduate	Advanced	Basic
D14	8	Master	Advanced	Intermediary
D15	4	Graduate	Intermediary	Basic
D16	5	Master	Advanced	Intermediary
D17	5	Graduate	Intermediary	Intermediary
D18	12	Graduate	Expert	Advanced
D19	5	Graduate	Advanced	Advanced
D20	10	Graduate	Intermediary	Intermediary
D21	4	PhD	Advanced	Intermediary
D22	5	PhD	Advanced	Intermediary

- Push Pull (S6): it is responsible for downloading remote content (pull) or accepting the delivery of remote content (push) to a local staging area.
- Workflow Manager (S7): This is a subsystem of a client-server system, responsible for describing, executing, and monitoring workflows.

After selecting the systems, we had to select developers to participate in the study. We recruited a set of developers, and we asked them to answer a questionnaire. The goal was to determine which of them were eligible for the study. We selected developers who had at least four years or more of experience with software development and maintenance. These developers only had basic knowledge of code smells and 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 [36]. Table II summarizes the characteristics of each developer.

As previously mentioned, all the developers – independently from their familiarity with the source code – received a list of code smells identified in the systems. We intend to observe if developers used code smells during the problem identification and if code smells can support accurate identification of design problems. Thus, for sake of simplicity, let us call the developers familiar with the system of **Group 1** and developers unfamiliar with the system of **Group 2**. Figure 1 shows the organization of developers according to their familiarity with the analyzed systems. Five systems were used in Group 1, while 2 systems were used in Group 2.

C. Study Activities

The study was composed of three activities. They were applied for all developers with very few variations in agreement with each group (Section III-B).

Activity 1: Training. In this activity, first and second authors conducted the training of all the developers regarding the main concepts about design problems, such as software design, code smells, and design problems. We also presented some examples of design problems pertaining to different categories (Section II), however making it clear that they do

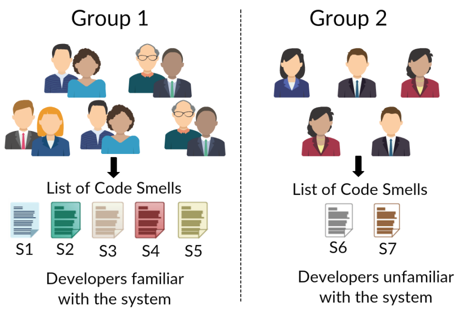
Group 1
Group 2
List of Code Smells
S1
S2
S3
S4
S5
Developers familiar with the system
List of Code Smells
S6
S7
Developers unfamiliar with the system

Fig. 1. Experimental Design

not necessarily represent the most relevant cases of design problems in their projects. 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. We gave developers of Group 2 a document containing: (i) a brief description of S6 and S7, and (ii) a very high level description of their design blueprint. We gave them 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 document can be found in our online material [36].

Activity 2: Problems identification. In this task, we asked developers to identify design problems. They had 45 minutes to perform this task, and we asked them to report their findings in an online form [36]. At the beginning of this activity, we gave them a guide characterizing different types of design problems (Section III-E). This guide is available in our complementary material [36].

Activity 3: Follow-up questionnaire. Developers were asked to answer a questionnaire about their general perception on the identification of design problems and the challenges related to accomplishing the task [36]. The answers were used to complement the qualitative analysis (Section III-D).

D. Data Collection

We used the data collected from the forms and the recorded content to conduct a qualitative analysis. In addition to the online forms filled by the developers, different resources were used to support the study. Developers used the Eclipse IDE to import and analyze the source code. Camtasia[1] tool was used to record audio and screenshots during Activity 2. A video camera was installed in the room to record the performance of the participants. The two first resources were installed on each computer used by the developers, while the last one was installed in the study environment (room).

We applied the two first phases of the coding process suggested by Strauss and Corbin [20] to conduct the qualitative analysis of the collected data (audio, video, and forms): *open coding* (1st phase) and the *axial coding* (2nd phase). We

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

did not apply the *selective code* (3rd phase) since we were not aiming to reach a theoretical saturation, as expected in Grounded Theory (GT method) [20]. The authors did the open coding, associating codes with quotations of transcripts, and 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 other researchers. Such analysis was useful to answer our research questions (Section IV).

E. Oracle Creation

Occurrences of design problems are often subjective. Therefore, we can not argue that a design problem is correct or not. Nevertheless, we can confirm the occurrence of a design problem with respect to the point of view of developers who actually know the system. Thus, we contacted either the developers that implemented the systems or the designers to help us to validate the design problems. We certified they were the people who had the deepest knowledge of the design of the investigated projects. We established an oracle of design problems for each target system according to each group of developers. For the systems used in the Group 1 (S1 to S5), we checked developers' answers using the audiovisual records. We analyzed each answer reported during the problem identification tasks to create the oracle. If the most experience developers (and/or designers) have agreed with the developer's answer (true positive), then we added the design problem to the oracle. Otherwise, we put it to re-validation. After that, we invited developers 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 Group 2, we asked the developers of the systems S6 and S7 to provide us a list of design problems affecting the systems. Then we run an automatic detector of design problems using a suite of design recovery tools [37] for additional identification of design problems. We asked developers of the systems to validate and combine the additional design problems with their own list. The procedure for the additional identification was the following: (i) an initial list of design problems was identified using detection strategies presented in [34], (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 developers.

The following design problems were included in the training session: *Ambiguous Interface*, *Unwanted Dependency*, *Component Overload*, *Cyclic Dependency*, *Delegating Abstraction*, *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 [15], [16], [33], [34]. However, we let clear to developers 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 IV-A).

As previously mentioned, all developers received a list of code smells. The detection of the code smells affecting the systems was performed using well-known metrics-based strategies [27]. These detection strategies are similar to those typically used in previous empirical studies (e.g. [15], [16], [33], [38], [39]). Such strategies have been showed effective for detecting code smells in other systems, reaching precisions higher than 80% [16], [34]. For this study, we considered 15 types of code smells: *Data Class*, *Divergent Change*, *God Class*, *Shotgun Surgery*, *Feature Envy*, *Long Method*, *Long Parameter List*, *Class Data Should Be Private*, *Complex Class*, *Blob Class*, *Lazy Class*, *Message Chain*, *Refused Bequest*, *Spaghetti Code*, *Speculative Generality*. They were selected because they represent different types of symptoms at class and method levels. However, again, developers were free to reason about other code smells, as they actually did (see Section IV-A). Additional details about the code smells and design problems are available on our online material [36].

IV. Results and Analysis

This section discusses our results and main findings.

A. Strategies to Identify Design Problems

We conducted a qualitative analysis of the collected data to answer RQ1: *What are the strategies that developers use to identify design problems?* Differently from one could expect, the use of smells was not the only preeminent strategy to identify design problems in a program. Our analyses indicate developers used a diverse range of strategies in addition to relying on smells. After analyzing codes that emerged from GT method (Section III-D), we revealed five preeminent strategies: **smell-based**, **problem-based**, **element-based**, **consequence-based**, and **pattern-based**. They are described as follow. Due to the confidentiality, we changed the element name whenever developers mentioned one.

Smell-based strategy is the strategy in which developers use the code smells as an indicator of a design problem. As mentioned by other studies, developers can use code smells as a strategy to identify design problem. However, it was interesting to observe that this strategy was used differently in each scenario. Their degree of success on identifying design problems with this strategy also varied (Section IV.C). Developers of Group 1, who were familiar with the source code, used it to confirm the existence of a design problem. They marked a candidate fragment as having a design problem whenever they have found a code smell. Developers of Group 1 identified only 3 design problems using this strategy. They also tried to use the strategy to locate candidate fragments, but they did not mark in the form any fragment as having a design problem. We also observed that, curiously, developers of Group 1 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).

In addition to using code smells to confirm the existence of design problems, developers of Group 2 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 candidate fragment, they used the presence of code smells to confirm the design problem as usual. All the seventeen design problems identified by developers of Group 2 were identified with this strategy. According to them, the smell-based strategy was indeed fundamental to support design problem identification. As an example, D7, D8, and D9 developers identified a *God Class* smell while confirming the occurrence of a design problem affecting the class:

"It is not its responsibility to show what it is on the screen" – D9
"Yeah... Exactly" – D8
"It accesses the database, and it shows what it is on the screen (...)" – D7
"This class deviates from its function" - D9
"That turns the class in a God Class" – D8

Problem-based strategy is the strategy in which developers search for occurrences of a specific type of design problem in the source code. We state a developer used the problem-based strategy when he explicitly mentions he is looking for a specific type of design problem across the system. Surprisingly, only developers of Group 1 used the problem-based strategy. They marked in the online form eleven fragments that could have a design problem. Out of these fragments, only one case represented a false positive, i.e. it did not have a actual design problem according to the view of the designer.

We observed that developers tended to focus on searching for design problems related to interfaces and components (realized as packages in the source code). For instance, *Overused Interface* [14] and *Component Overload* [14] 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* [14] and *Unused Abstraction* [14]. Moreover, curiously, one team was responsible for identifying most of the design problems using this strategy. They identified 4 instances of *Overused Interface*, 2 instances of *Component Overload*, 1 instance of *Scattered Functionality* [9], 1 instance of *Ambiguous Interface* [9] and 1 instance of *Cyclic Dependency* [13]. When we analyzed the videos, we noticed this team became continuously more confident with the strategy each time they identified a new design problem. However, this team was also the one that identified a false positive (*Ambiguous Interface*). The following example illustrates the case that they 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" – D3

Element-based strategy is the strategy in which developers select specific code elements to investigate if it is affected by a design problem. They do not necessarily reason about specific types of code smells or design problems. 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, developers focus 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 start inspecting these code elements and reasoning about symptoms in those elements. Only developers of Group 1 used this strategy as it requires familiarity with the system design. In fact, it does not seem a reasonable strategy to guess which elements to search first while identifying design problems in Group 2. When developers of Group 1 used this strategy, they marked in the online form seventeen design fragments that could have design problems. Only two of them were false positives, according to the view of the system's designer.

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 explicitly determine a criterion to choose such elements. For example, one of the teams chose a class based on the number and nature of variables and methods located in this class. Another team decided to limit the search to classes within specific subsystems. They picked a subsystem that was visibly large regarding the number of classes. The same team 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 conversation, we illustrate an example of how the element-based strategy was used by developers who are familiar with the system. At the beginning of the task, D1 and D2 developers were trying to determine which elements they should analyze. D1 suggested prioritizing the analysis of classes in large subsystems. After that, they browsed a few classes until they find one to analyze:

"Let's open the source code" – D2
"We should start analyzing big subsystems... I know which one, let's start by the X subsystem" – D1
"We already fixed a design problem in X subsystem" – D2
"It still might have more problems (...) I suspect this class contributes to a design problem" - D1

Consequence-based strategy is the strategy where developers reason about quality attributes that are negatively affected by certain design fragments. They discuss about how a program structure, which realizes a design fragment, explicitly hinders one or more quality attributes as negative *consequences*. Again, they do not necessarily reason about specific types of code smells or design problems. Developers used this strategy when they were analyzing a candidate design fragment, and they noticed that the counterpart implementation of the fragment impacts one or more quality attributes. Thus, developers 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. Moreover, they also used violations of design principles [40] (e.g. open-closed principle and information hiding) to illustrate other types of consequences and infer occurrences of design problems with this strategy.

Only developers of Group 1 used this strategy. They identified 12 design problems using it. We present below an example of developers who reasoned about the complexity and reusability of an implementation structure to confirm the occurrence of a design problem. In this case, developers were investigating the use of *Adapters* in the system, but without confirming if the class has a design problem or not. Then D1 used the consequence-based strategy to reveal additional symptoms related to a possible design problem. However, there was a disagreement between the developers. D2 believed that the class has been reused. D1 tried to argue that the class was not reused when compared with other *Adapter* classes. The discussion extended for additional minutes up to the point where D2 agreed that the class was not reused after all. Thus, they used the consequence that the design problem causes on the reusability to confirm the design problem (overuse of *Adapters*). We present part of the discussion as follows.

"It (the implementation structure) increases the complexity and reduces the reuse." – D1
"This class has been reused a lot" – D2
"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" – D1

Pattern-based strategy is the strategy in which developers search 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. Developers tend to discuss a wide range of patterns, including *Adapter*, *Builder*, *Facade*, *SOA* and *MVC*.

Even though only developers of Group 1 used the pattern-based strategy, it was the most frequent and successful one in this group. They correctly identified 18 design problems using this strategy. Maybe the reason of frequently 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. This might explain why developers of Group 2 did not rely on this strategy. As an example, in the previous quotation, D1 and D2 were investigating classes realizing the *Adapter* pattern. Before they get into that discussion, they used first the pattern-based strategy to identify classes that could be violating the *Adapter* pattern. Thus, they combined both strategies (consequence and pattern) to identify a design problem (Section IV-B). Other example of this strategy happened with D10 and D11 developers. They were analyzing a group of classes when they noticed that a class in the MVC pattern was illegally accessing the database:

"ClassA 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" – D10

As aforementioned, we were expecting that developers would use the code smells as the dominant strategy. One could not expect they often use a range of other strategies to identify design problems. The observation and characterization of all the five frequently used strategies enable us to answer RQ1. This answer leads us to our first finding:

Finding 1. Developers use multiple strategies to identify design problems

B. Problem Identification in Familiar Systems

In this subsection and the next one, we address RQ2: *How do developers identify design problems in familiar and unfamiliar systems?* We will focus here on how developers identify design problems in systems with which they had familiarity beforehand. As presented in the previous subsection, developers of Group 1 used all five strategies. In fact, the strategies can be divided into two groups based on how developers used them across the problem identification steps (Section II): (i) some developers use a strategy to locate a candidate design fragment for inspection; and (ii) others use a strategy to confirm the existence or not of the design problem in the candidate fragment.

Two-stage strategies. We noticed that in most cases, developers of Group 1 used smell-based, element-based and problem-based strategies to detect candidate fragments, while smell-based and consequence-based strategies were used to confirm if fragments have indeed a design problem. Pattern-based strategy was used both to locate a candidate fragment as well as to confirm that a fragment has a design problem. Thus, it was often the case that developers often used a strategy to locate a design fragment, and then used another strategy to confirm if the fragment actually contains a design problem.

Strategy blends. Indeed, we noticed developers tended to combine the five strategies. For instance, whenever developers 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 strictly rely on the violation itself to support the confirmation of a design problem. They often only confirmed the existence of a design problem when they noticed the violating structure of the candidate fragment was either explicitly affecting a quality attribute or hosting one or more code smells. In other words, they were combining multiple strategies in order to find complementary symptoms of a design problem in the candidate fragment.

As an example, in the quotation that illustrated the consequence-based strategy (Section IV-A), we could see

TABLE III
DESIGN PROBLEMS IDENTIFIED BY THE COMBINATION OF STRATEGIES

Strategy	Element	Problem	Smell	Consequence	Pattern
Element	5	0	1	4	5
Problem	0	4	0	4	2
Smell	1	0	0	1	2
Consequence	4	4	1	0	4
Pattern	5	2	2	5	5
Design Problems	*15*	*10*	*3*	*12*	*18*

that D1 and D2 developers combined the pattern-based and consequence-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 consequence-based strategy to reason through the consequences that the design problem caused on the reusability attribute. Thus, they combined two strategies to identify a design problem. Some developers also combined more than two strategies in order to identify a single design problem. As an example, developers can use the element-based strategy to locate a candidate fragment, and then they can use pattern-based, smell-based and consequence-based strategies to confirm that the fragment indeed contains a design problem.

Table III illustrates the number of design problems found by developers in familiar systems. Each cell represents the number of design problems correctly found when the strategy at the row was combined with the strategy at the column. Cells noted as 0 indicate that strategies at that particular row and column were not combined. We highlighted with grey-tone the cells that represent the number of design problems found when developers used only one strategy. Last row shows the total number of design problems found when developers used the strategies. We obtained the values presented in Table III by applying the procedures of the GT method (Section III-D). We applied the method to the online form and the video transcription to count each strategy that developers used and led to the identified design problem. For example, whenever developers 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 developer started a new round of problem identification. Then, we analyzed all the actions that he did since the fist 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., developers reasoned about a design pattern violation), then we counted that the strategy contributed to identify a design problem.

Single vs. combined strategies. We can see in Table III that developers using only one strategy identified 14 design problems. On the other hand, when developers blended multiple strategies, they identified 44 design problems. This result demonstrates developers, who are familiar with the systems under analysis, can identify more design problems when they combine multiple strategies than when they use only one strategy. For instance, the pattern-based strategy was the strategy that led to the identification of the highest number of design problems, and it was combined with all strategies. Developers used the pattern-based strategy to confirm occurrences of design problems when they combined this strategy with the element-based and problem-based strategies. Developers also used the pattern-based strategy to locate candidate fragments. In these cases, they often combined it with consequence-based and smell-based strategies to confirm design problems.

The prevailing behavior of combining different strategies in Group 1 also suggests that the identification of design problems might be more complex than one can expect. Developers of Group 1 combined all five strategies at least once. In fact, developers 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

C. Problem Identification in Unfamiliar Systems

Here we answer the RQ2 taking into account only developers of Group 2. They did not use all the five strategies. Actually, it was quite the opposite. They only explicitly used the smell-based strategy.

Smells as predominant strategy. It seems developers of Group 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 tangible evidence the participants used another strategy. In principle, it is somehow surprising that developers of Group 2 did not use other strategies. For example, there are some design problems, such as *Cyclic Dependency* [13] and *Fat Interface* [9], we thought developers did not have to be familiar with the systems to identify them. Thus, they could have applied either the problem-based or element-based strategy for the identification of these design problems. Developers of Group 2 actually used the smell-based strategy to identify problems such as *Fat Interfaces*. For instance, they reasoned about the concomitant occurrence of *God Classes* and *Feature Envies* to locate and confirm instances of *Fat Interfaces*.

Different usages of code smells. We observe a difference on how developers of Groups 1 and 2 used code smells to identify design problems. Developers of Group 1 tend to use the smell-based strategy to confirm design problems in candidate fragments. In addition, developers of Group 2 also used the smell-based strategy to search for candidate fragments. They used the presence of code smells to guide them towards a candidate fragment. After finding one, they used the presence of code smells to confirm the design problem.

Reasoning about single smells may not suffice. Actually, developers of Group 1 tried to use code smells to locate candidate fragments. However, they felt short to conduct this step. We noticed that when these developers found the first code smell in the fragment, they did not keep analyzing other smells affecting the same fragment. Instead of reasoning other code smells, they dropped the analysis and either gave up or moved on to other strategies. Whenever this behavior happened, we noticed they did not succeed to use code smells to identify design problems in the target fragments. We observed that developers of Group 2 tended to follow on searching for other smells in the same fragment, and were successful to find the design problem in the fragment. If developers of Group 1 have not dropped the such for other smells, they could have been successful. Upon data analysis of Group 1, we noticed that at least two candidate fragments, which they located using single code smells, had design problems. Unfortunately, developers missed those design problems.

Smell groups as complementary symptoms of a design problem. Most developers of Group 2, on the other hand, analyzed all the code smells within a fragment. They also used the amount of code smells in fragments to prioritize possible candidates. For example, D20 developer searched for elements with multiple code smells to locate candidate fragments. He also used each code smell affecting the fragment as a symptom of a design problem. When we asked him about the strategy that he used, 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 wrongly identify design problems. 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." – D20

D20 was not the only developer that considered multiple code smells affecting the same fragment. Figure 2 splits developers that grouped the code smells from those that did not group. In addition, it presents the results of precision and in parentheses the number of design problems correctly identified by each developer. As we can see in Figure 2, developers that grouped code smells identified more code smells than developers that did not group the code smells. Moreover, most developers that grouped smells tend to achieve a higher precision than the other developers. More than half of the developers grouped the code smells in order to conduct the problem identification. These developers 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 to the third finding:

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

If developers cannot find most of the symptoms that indicate a design problem, they might not be able to identify the problem. That is one reason why developers of Group 1 fell short when they tried to use the smell-based strategy to locate candidate fragments, and why some developers in Figure 2 have identified 0 design problems. As a code smell represents only a partial hint, they could not identify the design problem. If developers want to use the code smells to locate candidate

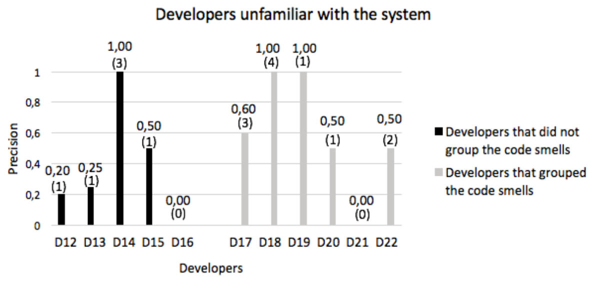

Fig. 2. Number of identified design problems by developers of Group 2

fragments, they may need to consider more than one smell.

Even when developers used multiple strategies in Group 1, in the end, they are considering multiple symptoms that indicate a design problem. Each strategy is likely to provide a symptom. Thus, when developers combine multiple strategies, they have multiples indicators that the design fragment contains a design problem. Similarly, when developers analyze multiple code smells, they are using each smell as a symptom to indicate a design problem. Therefore, developers search for multiples symptoms in the source code, regardless if they are familiar or not 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

V. Related Work

There is recently a growing interest in studying the relevance of code smells to support the identification of design problems [14],[28],[29]. Palomba et al. [28] reported an empirical study aimed at analyzing to what extent code smells are perceived as design problems. In their study, they showed developers code snippets affected and not affected by code smells. In an affirmative case, they asked developers to explain what problem they perceive. They reported that some code smells are, in general, not perceived by developers as design problems. We go beyond Palomba et al. [28]. We investigated various strategies followed by developers to identify a design problem. We found that there are other strategies than a smell-based strategy to identify design problems. Moreover, we noticed that when developers use multiple instances of smells, they succeed to identify design problems.

Vidal et al. [29] presented and evaluated a suite of 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. They have assessed the prioritization criteria in the context of more than 23 versions of 4 systems, analyzing their effectiveness for spotting the locations of design problems in the source code. The results provide evidence that one of the proposed criteria helped to prioritize the location of more than 80 design problems correctly. In particular, the findings of Vidal et al. [29] 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. [14] 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.

VI. Threats to Validity

Construct validity. Threats to construct validity are related to possible errors introduced in the study planning. We considered as the most relevant threats: (i) the set of analyzed systems, and (ii) the oracle creation. We have tried to mitigate the first threat by (i) using systems with different stages of design degradation, and (ii) choosing two systems with a large set of design problems validated by multiple developers and researchers. In principle, our training session was focused on discussing only eight types of design problems. However, we made clear for the developers that there are more types of design problems, and they are free to explore them during the task. They actually identified several other types of design problems. We recruited developers that implemented the systems to mitigate the second threat. Moreover, we solicited an explanation about the severity of each design problem. We discarded those that were not adequately explained. Regarding code smells, we selected detection strategies and thresholds that presented satisfactory results in other studies. In any case, if these strategies led to false positives or false negatives, they would have a similar effect on all the systems.

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

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 and developers 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 (first and second phases of GT).

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

VII. Concluding Remarks

The incidence of design problems is harmful to software systems due to their negative impact on quality attributes. Developers need to identify fragments to be analyzed and then confirm whether these code elements contain or not design problems. Little is known about the strategies actually followed by developers to identify design problems. Related works typically restrict the identification of design problems to the incidence of code smells. Moreover, they did not explore how developers use code smells and if there are other strategies to identify design problems. The study presented here addresses these gaps through observing developers behaviors while identifying design problems.

We investigated how developers identify design problems in two scenarios. In the first one, developers identified design problems in systems they had familiarity. In the second scenario, developers identified design problems in systems they are not familiar with. We found developers use four other strategies to identify design problems that are not based in code smells, namely: **element-based**, **problem-based**, **consequence-based**, and **pattern-based**. Some of the strategies are mostly used to locate fragment candidates, while other strategies tend to support the confirmation that some candidate fragments have design problems. There are also strategies that are used both to locate design fragments as well as to confirm the existence of design problems.

We were expecting that developers would use code smells as the main strategy to identify design problems, but that was always not true among the study subjects. Moreover, we noticed that developers search in the source code various complementary symptoms that indicate a design problem. In general, we found that developers familiar with the systems combined multiple strategies to identify a single design problem. Developers unfamiliar with the systems used multiple instances of code smells to identify a single design problem.

As future work, we intend to further investigate how to support the automation of combined strategies. Developers that were unfamiliar with the systems were able to identify design problems only with code smells. However, they could potentially find other problems if tool support could help them: (i) to reveal and reason about other types of symptoms they were not aware of, and (ii) summarize how such heterogeneous symptoms were related to the same fragment.

References

[1] A. MacCormack, J. Rusnak, and C. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Manage. Sci.*, vol. 52, no. 7, pp. 1015–1030, 2006.

[2] M. Godfrey and E. Lee, "Secrets from the monster: Extracting Mozilla's software architecture," in *Proc. of the Second Intl. Symposium on Constructing Software Engineering Tools (CoSET-00); Limerick, Ireland*, 2000, pp. 15–23.

[3] S. Schach, B. Jin, D. Wright, G. Heller, and A. Offutt, "Maintainability of the linux kernel," *Software, IEE Proceedings -*, vol. 149, no. 1, pp. 18–23, 2002.

[4] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, no. 2, pp. 105 – 119, 2002.

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

[6] A. Trifu and U. Reupke, "Towards automated restructuring of object oriented systems," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, ser. CSMR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 39–48.

[7] H. Bär and O. Ciupke, "Exploiting design heuristics for automatic problem detection," in *Workshop Ion on Object-Oriented Technology*, ser. ECOOP '98. London, UK, UK: Springer-Verlag, 1998, pp. 73–74.

[8] P. F. Mihancea and R. Marinescu, "Towards the optimization of automatic detection of design flaws in object-oriented software systems," in *Ninth European Conference on Software Maintenance and Reengineering*, March 2005, pp. 92–101.

[9] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering; Kaiserslautern, Germany*. IEEE Computer Society, 2009, pp. 255–258.

[10] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 306–315.

[11] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," in *12th Working Conference on Reverse Engineering (WCRE'05)*, Nov 2005, pp. 10 pp.–.

[12] P. Kaminski, "Reforming software design documentation," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, Oct 2007, pp. 277–280.

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

[14] W. Oizumi, A. Garcia, L. Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *The 38th International Conference on Software Engineering; Austin, USA*, 2016.

[15] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, Sept 2012, pp. 662–665.

[16] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, March 2012, pp. 277–286.

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

[18] N. Moha, Y. Gueheneuc, L. Duchien, and A. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transaction on Software Engineering*, vol. 36, pp. 20–36, 2010.

[19] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *2011 33rd International Conference on Software Engineering (ICSE)*, May 2011, pp. 1037–1039.

[20] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998. [Online]. Available: https://books.google.com.br/books?id=wTwYUnHYsmMC

[21] A. Tang, A. Aleti, J. Burge, and H. van Vliet, "What makes software design effective?" *Design Studies*, vol. 31, no. 6, pp. 614 – 640, 2010, special Issue Studying Professional Software Design.

[22] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 279–287.

[23] R. Marinescu, "Detecting design flaws via metrics in object-oriented systems," in *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*, 2001, pp. 173–182.

[24] ——, "Measurement and quality in object-oriented design," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Sept 2005, pp. 701–704.

[25] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do crosscutting concerns cause defects?" *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, July 2008.

[26] J. Wu, T. C. N. Graham, and P. W. Smith, "A study of collaboration in software design," in *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, Sept 2003, pp. 304–313.

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

[28] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 101–110.

[29] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos, "Identifying architectural problems through prioritization of code smells," in *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, Sept 2016, pp. 41–50.

[30] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Proceedings of the 10th International Conference on Quality Software; Zhangjiajie, China*, 2010, pp. 23–31.

[31] K. Khomh, M. D. Penta, and Y. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the 16th Working Conference on Reverse Engineering; Lille, France*, 2009, pp. 75–84.

[32] D. Sjobert, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Transaction on Software Engineering*, vol. 39, pp. 1144–1156, 2013.

[33] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems," in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, ser. AOSD '12. New York, NY, USA: ACM, 2012, pp. 167–178. [Online]. Available: http://doi.acm.org/10.1145/2162049.2162069

[34] I. Macia, "On the detection of architecturally-relevant code anomalies in software systems," Ph.D. dissertation, Pontifical Catholic University of Rio de Janeiro, Informatics Department, 2013.

[35] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Wiley Publishing, 2012.

[36] I. Submission, "ICPC-2017: Complementar Material," https://icpcsubmission.github.io/2017/.

[37] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering; Palo Alto, USA*, 2013, pp. 486–496.

[38] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15. New York, NY, USA: ACM, 2015.

[39] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1 – 14, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001053

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