Identifying Design Problems in the Source Code

A Grounded Theory

Leonardo Sousa, Anderson Oliveira
PUC-Rio, Rio de Janeiro - RJ
{lsousa,aoliveira}@inf.puc-rio.br

Willian Oizumi, Simone Barbosa, Alessandro Garcia
PUC-Rio, Rio de Janeiro - RJ
{woizumi,simone,afgarcia}@inf.puc-rio.br

Jaejoon Lee
Lancaster University
Lancaster, Lancashire
j.lee3@lancaster.ac.uk

Marcos Kalinowski, Rafael de Mello
PUC-Rio, Rio de Janeiro - RJ
{kalinowski,rmaiani}@inf.puc-rio.br

Baldoino Neto
UFAL, Maceio - AL
baldoino@ic.ufal.br

Roberto Oliveira, Carlos Lucena, Rodrigo Paes
PUC-Rio, Rio de Janeiro - RJ
{roliveira,lucena}@inf.puc-rio.br
rodrigo@ic.ufal.br

ABSTRACT

The prevalence of design problems may cause re-engineering or even discontinuation of the system. Due to missing, informal or outdated design documentation, developers often have to rely on the source code to identify design problems. Therefore, developers have to analyze different symptoms that manifest in several code elements, which may quickly turn into a complex task. Although researchers have been investigating techniques to help developers in identifying design problems, there is little knowledge on how developers actually proceed to identify design problems. In order to tackle this problem, we conducted a multi-trial industrial experiment with professionals from 5 software companies to build a grounded theory. The resulting theory offers explanations on how developers identify design problems in practice. For instance, it reveals the characteristics of symptoms that developers consider helpful. Moreover, developers often combine different types of symptoms to identify a single design problem. This knowledge serves as a basis to further understand the phenomena and advance towards more effective identification techniques.

CCS CONCEPTS

• **Software and its engineering → Software design engineering**;

KEYWORDS

design problem, grounded theory, software design, symptoms

ACM Reference Format:
Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Neto, and Roberto Oliveira, Carlos Lucena, Rodrigo Paes. 2018. Identifying Design Problems in the Source Code: A Grounded Theory. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180239

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180239

1 INTRODUCTION

The development and maintenance of long-lived software systems require special attention to non-functional requirements, such as maintainability, extensibility, availability and performance. Each non-functional requirement may be affected, either positively or negatively, by design decisions [35], such as (mis-)prioritizing an objected-oriented principle over another or (mis)using certain design patterns [11]. A *design problem* [2, 6, 48] is the result of one or more inappropriate design decisions, *i.e.*, decisions that negatively impact non-functional requirements. An example of design problem is Fat Interface [25], which occurs when a developer decides to aggregate multiple non-cohesive functionalities in a single system interface; as a result, this interface becomes highly coupled to other modules. The occurrence of a Fat Interface negatively impacts the maintainability and extensibility of the software system [25].

Design problems are often harmful in several software systems. For instance, an industrial study [7] with 745 software systems, from 160 different organizations, showed that technical debts – primarily associated with design problems – were directly related with a significant increase in software project costs [7]. Another study [39] showed that design problems are one of the most common categories of technical debt that leads to the rejection of code contributions. Thus, the prevalence of design problems may cause the redesign or even the discontinuation of software systems [14, 19, 37, 49]. Given the harmfulness of design problems, developers should identify them as early as possible [12, 37, 54].

However, identifying a single design problem can itself quickly turn into a very complex task [6, 47]. One of the main reasons is that design documentation is often unavailable or outdated [18, 48]. Then, developers are forced to analyze several elements in the source code to identify each design problem. Each single design problem often manifests as multiple *symptoms* scattered in several program elements. For instance, identifying a Fat Interface requires the search for symptoms affecting not only suspicious interfaces, but also classes that either implement or depend on those interfaces.

Unfortunately, there is limited understanding about how developers identify design problems in practice, in particular when the

source code is the only artifact available in a project. Existing studies tend to focus on proposing solutions for assisting developers in identifying design problems [6, 20–22, 28, 32, 47, 50, 52, 53]. However, such proposed solutions may be misaligned with how developers identify design problems in practice. For instance, most of these studies make oversimplified assumptions about the process of identifying design problems. They consider that developers would rely on a single type of symptom (e.g., either code smells [20, 28, 32] or design principle violations [52, 53]) to infer the occurrence of a design problem. However, this assumption might not hold in real project settings. In fact, we know little about how developers identify, in practice, design problems in the source code.

To provide such necessary understanding, in this paper we address the following research question: *how do developers identify design problems in the source code?* To do so, we conducted a multi-trial industrial experiment with professional software developers from five different companies, where they had to identify design problems in their systems under development. In the experiment, we captured data on their behaviour by filming the environment, recording audio and capturing their computer screens on video. These data allowed us to conduct an in-depth qualitative analysis based on Grounded Theory procedures [44]. As a result, we have built a theory of design problem identification.

According to Stol and Fitzgerald [42], nascent research areas typically take the research-then-theory approach, whereas more mature areas rely on (and refine) theories to further advance the field. Aligned with this statement, the theory presented here offers insightful propositions and explanations on how design problems are identified, which can serve as a basis to improve the state-of-art. For example, while most studies address only one type of design problem symptom, the theory reveals that, in practice, developers rely on a heterogeneous set of symptoms. Thus, previous studies are misaligned not only for assuming that developers will use a single, dominant type of symptom, but also for not considering how they use these symptoms. Based on the theory, researchers can build solutions most suitable to help developers. For instance, we noticed when developers consider a symptom useful to identify design problems. Thus, researchers can use this knowledge to build tools that prioritize symptoms that are likely to be helpful for developers.

The remainder of this paper is organized as follows. Section 2 presents basic concepts and an example of design problem diagnosis. Section 3 presents our research design. Section 4 summarizes the results in which our theory is grounded. Section 5 complements the theory with additional propositions concerning the developer. Section 6 presents how the theory can be used to drive research on identifying design problems. Sections 7 and 8 presents related work and threats to validity, respectively. Section 9 concludes the paper.

2 CONCEPTS AND MOTIVATION

This section presents basic concepts about design problems (Section 2.1); an example of how developers may identify them (Section 2.2); and concepts related to Grounded Theory (Section 2.3).

2.1 Design Problems

Software design results from a series of decisions made during the software development [45, 46]. Those decisions directly influence software quality attributes, such as maintainability, robustness, performance, and the like. However, along the way, software design may decay due to the introduction of design problems. A **design problem** arises from one or more design decisions which, when implemented in source code, negatively affect software quality requirements [2, 6, 48]. Although some of them are self-admitted, most are introduced by unintended decisions, which makes them harder to identify in source code. Developers often have only the source code as a resource to identify design problems, because of missing, informal, or outdated design documentation [18, 48].

A design problem may affect a single or multiple elements in the program. For instance, the Delegating Abstraction [5] problem happens when one class exists only for passing messages from one element to another, while a Fat Interface is a design problem that forces some elements – *i.e.*, the interface itself and related classes – to deal with many functionalities [25]. Moreover, design problems also impact different non-functional requirements. For instance, the Misplaced Concern can impact the understandability, since it happens when an element implements functionality that is not the predominant one in the element [12]. Conversely, the Cyclic Dependency can impact the system performance and availability due to dependency cycles that can lead to deadlocks [34].

2.2 Design Problems Identification

As design problems have negative consequences for software systems, they are often targets of significant maintenance effort [12, 37, 54], increasing the cost related to maintaining the software systems. Unfortunately, identifying design problems is challenging, for a number of reasons [6, 47]. Firstly, software systems tend to be increasingly large in size and complexity, thereby expanding the search space for problems. Secondly, each design problem usually pervades the implementation of several elements [12, 28]. Thus, developers need to analyze several elements to identify a single design problem [47]. Thirdly, design documentation is often unavailable or outdated, making the source code the only artifact available for developers to identify design problems in most cases.

To illustrate the design problem identification, let us consider the example in Figure 1, which uses a UML-like notation to show a partial view of a real university management system we used in our experiments (Section 3.1). During design problem identification, one of the developers started looking for design problem *symptoms* in the classes that extended the `AbstractService` class. A **symptom** is a partial sign or indication of the presence of a design problem. The first one noticed by him was the incidence of Feature Envy [10] in methods of the `InstitutionalEnrollmentService` class. A method affected by Feature Envy is more interested in other classes rather than in its own class [10]. Based on the analysis of Feature Envies, the developer noticed that the class was implementing two distinct responsibilities: query enrollment and manage enrollment. He also noticed that the class had a high dependency with `UserService` and `IncidentService` classes. He then concluded that the class had a design problem because it presented the following symptoms: (i) implementation of two unrelated responsibilities, (ii) strong coupling with other classes, and (iii) methods that are overly complex.

In this example, the most helpful symptoms to identify a design problem were located in the `InstitutionalEnrollmentService`.

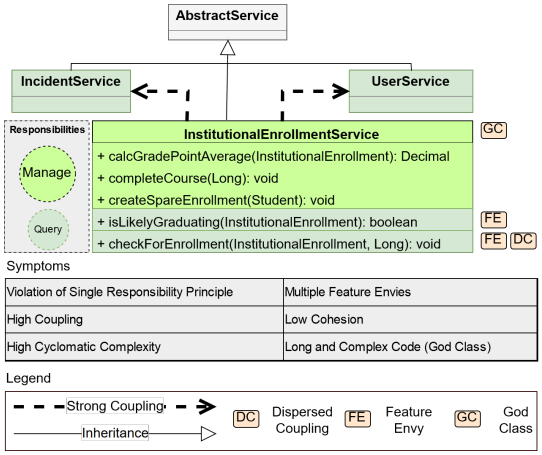

Figure 1: Design problem occurring in the UniM system

However, developers had to analyze at least three other classes to identify a design problem known as Concern Overload [12]: elements overloaded with multiple unrelated responsibilities (*i.e.,* concerns) tend to be harder to understand and to change, resulting in low maintainability and understandability [12]. Also, the developer needed to rely on three types of symptoms (code smells affecting the class, high coupling, and high complexity) to identify the design problem. Unfortunately, a system may contain hundreds of symptoms, where not all are related to design problems. Consequently, the design problem identification becomes even harder.

To alleviate the identification task, several industrial and academic tools have been proposed to support the identification of design problems [9, 28, 29, 31, 51]. Despite the variety of available tools, they assume that developers rely on only a single predefined type of symptom (*e.g.*, code smells, design principle violation, or the like). Thus, they return a single, predefined type of symptom. Consequently, developers need to run several tools in case they need to consider different types of symptoms. Also, these tools still can return a disjoint output, which forces developers to manually combine the symptoms indicated by these tools. Hence, developers still have to analyze the code elements to confirm or reject the presence of a design problem. In summary, even with tools, the identification of design problems remains challenging for developers [12].

2.3 Grounded Theory

The provided example only illustrates a scenario where the developer analyzed some symptoms without explaining how he found the symptoms that helped him to identify the problem. Understanding a phenomenon like the identification of design problems requires more explanation than a couple of examples. Such understanding can be provided by a theory with explanations and understanding of concepts and factors that go beyond of the mere observation of a phenomena [15–17, 40].

We applied the principles of Grounded Theory (GT) to further understand and explain how developers identify design problems in source code. GT is a qualitative research method that uses a systematic set of procedures to inductively develop a theory about a phenomenon [44]. It is used to understand the action in a substantive area from the point of view of the actors involved in a phenomenon [13]. There are different versions of GT [43], and we adopted Strauss's and Corbin's GT [44], since it allows us to ask questions about conditions upon which a phenomenon occurs [43].

GT contains three coding procedures: *open coding*, *axial coding*, and *selective coding*. Coding refers to the task of data analysis [43, 44]. Open coding involves the breakdown, analysis, comparison, conceptualization, and categorization of the data. Axial coding consists in examining the identified categories to establish conceptual relations between them. Finally, in selective coding, we aim to reach the theoretical saturation, at which we further refine the categories and relations, and identify the core category to which all others are related. More details about the use of GT to derive the theory in this study are presented in Section 3.4.

3 RESEARCH DESIGN

Several researchers have proposed solutions to help developers identify design problems [6, 20–22, 28, 32, 47, 50, 53]. However, they do not focus on explaining how developers identify those design problems. Despite their contribution, they do not clarify "the mechanisms through which and the conditions under which [the cause-effect relationship] holds" [38, p. 8]. Therefore, the support they provide for developers to identify design problems may be somewhat misaligned with developers' current practice.

To determine and understand the activities and factors that influence how developers identify design problems, we conducted a multi-trial controlled experiment in different software companies. We then analyzed the collected data and derived a theory using GT. Such theory provides an overview, explanation and understanding on *how developers identify design problems in the source code.*

3.1 Software Systems and Developers' Selection

We searched for software companies that could provide us with software systems and developers to conduct the experiments. 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 achieve some heterogeneity while selecting companies from our industrial collaboration network, thereby balancing contextual diversity and convenience [36]. Based on these criteria, we chose five software companies from the North and Northeast of Brazil. After selecting the companies, we asked their managers, some of whom were software designers, to suggest specific systems that met the following characteristics:

1. Systems in different stages of design degradation;
2. Systems from different domains and with different sizes with respect to the number of modules and developers;
3. Systems that were not in their initial versions;
4. Systems developed in Java.

Details about the companies and the systems are presented in the online material [26]. After the companies' managers provided us with the systems, we asked them to indicate developers familiar with each one and who could act as subjects in the study. For conducting our study the subjects were divided into teams. Table 1 presents the subject characterization and the corresponding teams.

Table 1: Characterization of the Developers

Team	ID	Experience (years)	System	Company
T1	D1	3	S1	1
	D2	5		
T2	D3	13	S1	1
	D4	14		
T3	D5	14	S1	1
	D6	6		
T4	D7	7	S2	2
	D8	2		
T5	D9	4	S2	2
	D10	4		
T6	D11	10	S3	3
	D12	8		
T7	D13	12	S4	4
	D14	13		
T8	D15	4	S5	4
	D16	8		
T9	D17	4	S6	4
	D18	10		
T10	D19	7	S7	5
	D20	7		
	D21	9		
T11	D22	12	S8	5
	D23	9		

All teams are composed of two developers, except for T10, whose company asked us to involve three developers.

3.2 Experimental Tasks

The experiment comprises the following four activities:

Activity 1: Subjects characterization. We asked the developers to fill out a questionnaire to gather their information, including educational level, professional experience with software development, Java programming, and knowledge about design problems.

Activity 2: Training. We conducted a training session for all the developers about software design and design problems, with examples of problems pertaining to different categories. 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 selected these design problems together with the project managers, who suspected that these represented common cases of design problems in the selected projects. However, we made it clear to the developers that they were also allowed to identify other types of design problems with which they were already familiar. The 40-minute training session was organized in two parts: a Powerpoint-based presentation; and discussions and questions.

Activity 3: Problem identification. We asked developers to identify design problems in their software systems. They had 90 minutes to analyze the source code to identify all the design problems they could find. At the beginning of this activity, we asked them to explain aloud what they were doing while we recorded the task on video. Thus, we could triangulate the results from the questionnaire and the video recording to improve the data analysis. In total, the problem identification sessions lasted over 13 hours.

Activity 4: Follow-up. Developers filled out a questionnaire about their perception of the task. We also asked them to indicate whether each symptom was useful to identify a design problem.

3.3 Provided Data

Most design problems manifest themselves in source code through different symptoms, which many developers use tools to help them identify these symptoms. Thus, to make Activity 3 more realistic, we provided our subjects with a set of symptoms we had detected in the code of the analyzed systems after running and manually combining the output of some tools [4, 28, 31]; simulating the use of tools, but not limiting the developers to the output of a specific one. For each module, we provided the following types of symptoms:

1. **Violation of Non-functional Requirements**: Information of non-functional requirements (*e.g.,* readability, testability, robustness, security), which were possibly being violated;
2. **Code Smells**: We provided the list of code smells because previous studies suggest that code smells can be used as indicators of design problems [20–22, 32]. We used well-known metrics-based strategies to identify 15 types of code smells from Fowler's Catalog [10];
3. **Visual Representation of Modules**: A visual representation of the module and relationships between modules;
4. **Design Pattern Violation**: Information on the use of architectural and design patterns [11], to help identify misused patterns;
5. **Quality Requirements**: Information about quality requirements (*e.g.* cohesion, coupling, complexity)
6. **Violation of Object-Oriented Principles**: Information about object-oriented principles [24] that were possibly being violated, which may indicate a problem. These principles have been pointed out as guides to avoid design problems.

We summarized and presented these symptoms to developers through a web page based on SonarQube [4]. We provided a visualization similar to SonarQube because it is a well-known platform for inspection of software systems, and which was familiar to most subjects. Thus, we could reduce the learning curve or aversion related to how the symptoms are presented. The main difference of our mechanism is that it presents all symptoms in a single page. It is noteworthy that, while SonarQube provides several pieces of information unrelated to design problems, our mechanism provides only symptoms that may help developers to identify design problems.

3.4 Data Collection and Analysis

We used different instruments to collect data. Developers had to answer characterization and follow-up questionnaires. They also had to write any observation in a specific field at the web page in which we presented the symptoms. Developers could write anything in the observation field, such as: the name of a design problem affecting the elements; whether he agreed with the suggested symptoms; or even comments about the code. They used either the Eclipse or IntelliJ IDEs to analyze the source code, and they used the browser to access the web page with the symptoms. We used the think-aloud

method [8], asking the developers to verbalize their thoughts during the experiment. All their procedures were recorded on audio and video. We used Techsmith's Camtasia[1] to record audio and screenshots of their computer. In addition, a video camera was installed in the room to record the developers during the study.

After the data collection, we employed GT procedures to analyze the data. We first transcribed all the video and audio recordings. We then performed *open coding* to associate codes with quotations of developers' utterances, as shown in the example below:

Raw Transcript. *"D6: The readability here is awful, but there is no way to escape from this (implementation). That is the standard (implementation). (...) indeed, it (the class) is not easy to ready"*
Code 1. developer mentions that the class readability is awful
Code 2. developer mentions that there is no way to escape from the analyzed implementation
Code 3. developer mentions that the analyzed implementation is the standard implementation
Code 4. developer accepts that the class is hard to read

We related the codes through *axial coding*. In this procedure, the codes were merged and grouped into more abstract categories, and the type of relation [44] was established. For instance, the previous codes were grouped into the following two categories:

Category 1. analysis of a non-functional requirement
Category 2. explanation for the existence of the symptom

For each transcript, the codes, memos, and networks showing the relations in the categories and codes were peer reviewed and changed upon agreement with some of the paper's authors. Then, we used *selective coding* to identify core categories that best explain how developers identify design problems. Next, we used the Sjøberg's framework [40] to represent and describe the theory constructs, propositions, explanations and scope. A *construct* is a basic particle that composes a theory; thus, the categories identified in the axial and selective coding are candidate constructs for the theory. A *proposition* is an interaction among constructs, which comprise the relations established among the categories. An *explanation* comprises the factors behind the propositions. The explanations are grounded in the categories, codes, relations, and in the transcripts. The *scope* is the universe to which the theory is applicable.

4 A THEORY ON HOW DEVELOPERS IDENTIFY DESIGN PROBLEMS

As aforementioned, we used Sjøberg's framework [40] to describe the theory, which is summarized in Figure 2. According to his framework, our theory fits in the *Explanation type* since it describes and explains how the identification of design problems is conducted (Section 4.1), the symptoms and their characteristics (Section 4.2), and how the symptoms are used to diagnose design problems (Section 4.3). In his framework, Sjøberg also describes criteria to evaluate theories. Testability is one criterion, which indicates "the degree to which a theory is constructed such that empirical refutation is possible." Regarding such criterion, our theory has high testability since empirical refutation of its propositions is possible by replicating the study. In fact, the replication is practicable as we first ran the experiment in three companies and then we replicated it with more companies until reaching theoretical saturation (Section 2.3).

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

Table 2: Helpfulness according to developers

Symptom	Applied times	No. of contributions	Percentage of success
Design Pattern Violation	43	34	79.07%
Quality Requirements	43	31	72.09%
Violation of Non-functional Requirements	62	46	74.19%
Code Smells	37	17	45.95%
Violation of Object-oriented Principles	38	20	52.63%

When describing the theory, we introduce the constructs and propositions, identifying them in the text with **C** and **P**, respectively. We discuss the propositions and their constructs next. We also present explanations for propositions that are aligned with findings of previous studies and explanations that comprise findings that have not been presented elsewhere. Complete description of the constructs and propositions is available in the online material [26].

4.1 Identification of Design Problems

A *design problem* (**C1**) arises in code elements due to one or more *design decisions* (**C2**), made intentionally or not. In fact, a design problem may affect one or more elements in such a way that these elements manifest symptoms of its presence. A *symptom* (**C3**) is an indication of the presence of a design problem.

Three Steps to Identify Design Problems Using Symptoms. A code element may contain several design problem symptoms. Thus, we define a *syndrome* (**C4**) as a set of symptoms affecting the same code element. In this context, we refer to *diagnosis* (**C5**) as the process of identifying a design problem through the analysis of symptoms that manifest themselves in the source code (**P1**). From the data collected during the subjects' diagnostic activities, we noticed that the identification of design problems was often divided into three steps: (i) locating code elements, (ii) analyzing the elements, and (iii) confirming or rejecting the presence of a design problem. In all these three steps, developers rely on design problem symptoms in the source code (**P2**).

4.2 Design Problem Symptoms

Symptom Helpfulness. We noticed that developers do not always consider all the symptoms of a syndrome when identifying design problems. Instead, they only consider those symptom instances that they judge helpful during the identification. We could identify when developers judge a symptom helpful because we asked them to evaluate the symptom based on how helpful it was to identify the problem (Section 3.2). Table 2 presents the percentage of helpfulness of each type of symptom. The first column indicates the name of the symptom, while the second column shows the number of times that the symptom was used by developers. The third column shows the number of occasions that the developers mentioned the symptoms were helpful to identify a design problem. Finally, the last column indicates the percentage of helpfulness, *i.e.*, the percentage that developers used the symptom and evaluated it as helpful.

Symptom attributes that drive developers to select what to analyze. Based on the helpfulness mentioned by developers, we performed a qualitative analysis to investigate which symptom

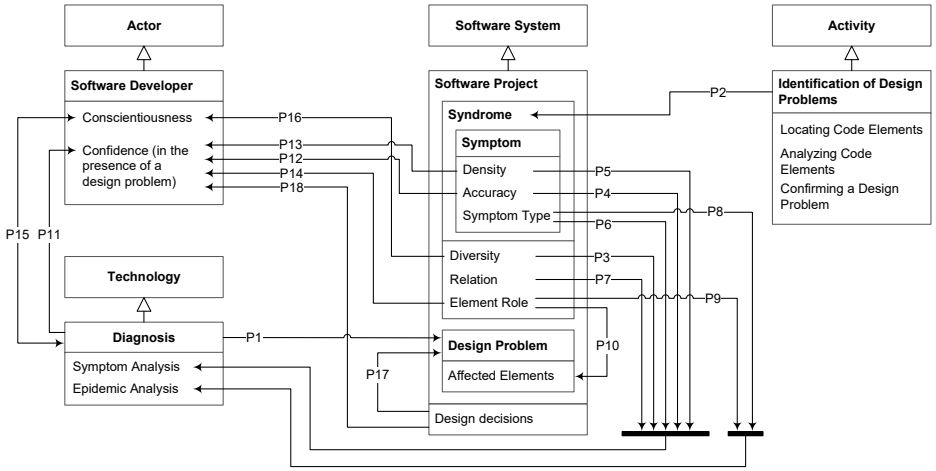

Figure 2: Theory representation of how developers identify design problems through the analysis of symptoms

attributes *i.e.*, characteristics of the symptom (such as its accuracy or type), developers take into consideration when they choose the symptoms most likely to help them. We observed that the following symptom attributes are most helpful for developers to identify design problems: symptom type, accuracy, density, relation (among the symptoms), and diversity. Symptom *type* (**C6**) indicates a category to which a set of symptoms with common characteristics belongs (*e.g.*, code smell). *Accuracy* (**C7**) is the degree to which a symptom is correct in indicating a design problem, while *density* (**C8**) is the number of symptom instances in a syndrome. Regarding these attributes, we were already expecting that accuracy and density would be attributes that developers take into account to consider a symptom helpful. However, we had not expected that they would take into consideration the relation among symptoms and the diversity of symptoms in the syndrome.

Diversity of a syndrome. *Relation* (**C9**) is how two or more symptoms are connected. For instance, both Intensive Coupling smell and violation of the layered pattern [3] measure the degree to which elements are undesirably coupled with others. Since they measure similar (albeit complementary) properties of an element, they are related to each other. We noticed that developers use the relation among the symptoms to discover other types of symptoms that can indicate a design problem (Section 4.3.1). We also noticed that developers frequently located elements that manifested several different types of symptoms (**P3**). In fact, we observed that *diversity* (**C10**) is another attribute that developers consider. Diversity is the degree to which a syndrome contains a variety of symptom types. Upon data analysis, we found that the more different types of symptoms a syndrome has, the greater the chance the developer will identify at least one design problem in the element (Section 5.2).

Indeed, we noticed that the diversity of a syndrome has a strong influence on the diagnosis. As this finding had not been observed before, studies that assume that developers rely on only one type of symptom [27, 32, 52, 53] may be misaligned with how diagnosis is conducted in practice, in two ways: (i) they may assume that developers will use a predefined, dominant type of symptom; and (ii) they may not consider that diversity of symptoms is other indicator that developers use when identifying a design problem. We will discuss later how the diversity influences human aspects (Section 5.2).

Considering the attributes that influence the developers. As mentioned before, developers do not consider all symptom instances to identify a design problem. They take into account only those symptoms that they consider helpful to identify a problem. We showed the attributes developers consider to assume that a symptom is helpful. For instance, if a syndrome has several types of symptoms, developers consider the density and diversity to select the symptom. That is, developers select a symptom when they are satisfied with these attributes. Conversely, when attributes do not satisfy the developers (*e.g.*, the syndrome does not contain diverse types of symptoms), the symptom would be ignored and not considered helpful, possibly leading to missing a design problem.

Knowing about how developers consider a symptom helpful is useful for researchers since they can propose solutions that emphasize symptoms helpful for the developers. For instance, some studies propose solutions to prioritize smells that can help developers to identify design problems [1, 32, 50]. As code smells are a type of symptom, they also present some of the attributes discussed above. However, some studies on code smells may not consider attributes as the density of smells or diversity. Therefore, developers may neglect some code smells for not considering them helpful for the identification. These studies could use the attributes that developers take into account as a mechanism to prioritize smells (Section 6).

4.3 Design Problem Diagnosis

As aforementioned, diagnosis is the process of identifying a design problem through the analysis of symptoms. We noticed that developers diagnose a design problem based on two types of analyses: a *symptom analysis* (**C11**), and an *epidemic analysis* (**C12**).

4.3.1 Symptom Analysis. In symptom analysis, developers choose and analyze a set of symptoms affecting a single element, *i.e.*, they

Table 3: Combining Symptoms

Symptoms	Instances	Design Problems	Teams
1	16	11	T7, T8, T9
2	13	10	T1, T3, T7, T8, T9
3	14	11	T3, T4, T5, T6, T7, T8, T9
4	10	6	T2, T3, T5, T8, T9
5	3	1	T4

do not analyze multiple elements. This happens because they usually rely on the aforementioned symptom attributes: type, accuracy, density, relation (among symptoms), and diversity. In this analysis, developers verify, based on these attributes, whether the symptoms affecting the analyzed element indicate a design problem. If so, then they do not proceed to analyze other elements.

Incorrectly ignoring symptoms. Someone can expect that the accuracy (**P4**), the density (**P5**), and the type (**P6**) of the symptoms influence the problem identification. For instance, let us consider code smells. Palomba *et al.* [33] investigated to what extent code smells are perceived as design problems. They noticed that developers take into account the type of the code smell to decide whether it is a problem. Surprisingly, developers tend to incorrectly associate the type of symptom with its accuracy or density, and that does not happen only with code smells. Thus, if they rely on the accuracy or density to disagree that a symptom indicates a design problem, they tend to not consider the same type of symptom in the other elements, even when they indicate a design problem. For instance, if a developer analyzes the violation of a design pattern such as Data Access Object and concludes that this type of symptom is irrelevant for identifying a design problem, then he is less likely to consider a violation of a design pattern in the elements he analyzes next. This happened, for instance, with the T3 developers.

Combining multiple related symptoms. Someone can argue that analyzing a single element is not enough to identify a design problem. Nevertheless, we noticed that they combine symptoms in a single element in order to confirm the presence of a design problem. Table 3 shows the frequency with which developers either used only one symptom or combined multiple symptoms. Its first column indicates the number of symptoms that developers combined to identify design problems. Its second column indicates how many times the symptom or combination of symptoms happened. Its third column indicates the number of design problems found when the subject used a symptom or a combination of symptoms. Its last column indicates the teams that used or combined the symptoms. We obtained these data after applying the GT. The online material [26] has a full version of the table containing (i) which symptoms were combined and (ii) which design problems the team found.

We can see in Table 3 that most developers tend to combine symptoms to identify a design problem. Also, we noticed that developers identify more design problems when they combine symptoms. Based on this result, we investigated how the combination takes place. We noticed that developers use symptom relations to identify the symptoms to combine. Thereby, the relation helps developers to identify other helpful symptoms in the syndrome. We observed, therefore, that the more related to others a symptom is, the greater the likelihood of a developer selecting it for combination (**P7**).

As an example of how developers use the symptom relation to find other helpful symptoms, let us consider the developers of the T2 team. They were analyzing the code smells, and they noticed that the class had the Dispersed Coupling smell. Due to the presence of this smell, they analyzed the coupling quality requirement. When analyzing this type of symptom, they noticed it was indicating a high coupling with other classes. This finding increased their confidence that the class contained a design problem. These developers also noticed that the coupling was related to a third type of symptom: violation of non-functional requirements. When they analyzed this symptom, they noticed that the high coupling was making the class harder to read. In this example, the developers used the relation among the three symptoms (code smells, quality requirements, and violation of non-functional requirements). Then they combined these symptoms to identify that the element was involved in the Concern Overload design problem [12].

Sousa *et al.* have shown that developers often combine multiple symptoms to identify design problems [41]. However, they did not observe how that combination took place. In our case, we noticed that the relation among the symptoms is what drives the combination, by helping to identify other related symptoms. The combination of the symptoms is another evidence that previous studies [27, 32, 52, 53] may have proposed solutions for the problem identification that do not fit the developers' needs. In other words, developers consider multiple symptoms (Section 4.2) and they also combine these symptoms to increase their confidence in the presence of a design problem. Therefore, forcing the developers to use only a reduced set of symptoms is likely to go against the way in which developers identify design problems in practice.

4.3.2 Epidemic Analysis. When developers analyze an element, they do not consider only the symptoms affecting that element; sometimes they also consider whether other elements are affected by the same set of symptoms. We name this process epidemic analysis. Analogously to the way in which attributes influence the selection of symptoms in a single element, there are attributes that developers consider before choosing elements for an epidemic analysis. In addition to considering the types of symptoms (**P8**), developers also take into account the *element role* (**C13**) to choose the epidemic elements most likely to help them to identify a design problem (**P9**). Element role is the function that an element plays in the software system, *e.g.*, the role of `Service`.

Complementary analysis. The reason why developers use the element role to identify epidemic elements is that each design problem may be scattered over several elements. Since those elements share the same symptoms, developers assume that they may help them to identify the design problem, which justifies the epidemic analysis. However, what is surprising is that developers analyzed epidemic elements only when they had used the symptom analysis but had not succeeded in identifying a design problem. We noticed that, since they are not confident about the presence of a design problem during the symptom analysis, they proceed to an epidemic analysis of other elements in order to decide whether there is a design problem in the elements under analysis.

Prioritization of key elements. We noticed that developers tend to prioritize epidemic elements that provide a central functionality in the system. Such behavior happens because they associate the role played by the element with the probability of the element containing a design problem (**P10**). As an example, the T2 developers were analyzing the symptoms of an element. During the analysis, they noticed that the element was playing the `Service` role in the system. At this point, the developers included other `Service` classes in the analysis. When they focused the analysis on all the classes that play a service role, this change of focus led them to identify a design problem. They mentioned that all the service classes in the systems are affected by the Scattered Concern design problem [12]. Curiously, these developers had already analyzed other `Service` classes before, without identifying any design problem. In that case, the T2 developers were not applying the epidemic analysis; thus, they did not take into account the element role to select elements with similar set of symptoms.

Although developers are more likely to accept that elements playing an important role have a design problem, we found cases that subjective factors influenced their decision, as discussed next.

5 PROPOSITIONS CONCERNING THE DEVELOPER

In this section, we provide some additional propositions concerning the developer, observed through the think-aloud method [8] with the support of video and audio recordings.

5.1 Confidence in the Presence of a Design Problem

The confirmation or rejection of a design problem in a group of elements is mainly influenced by the developers' *confidence* (**C14**), which is the degree to which they are convinced about the presence of a design problem. The most confident the developer is, the greater the likelihood of confirming a design problem.

Attributes that increase developers' confidence. The attributes that influence a design problem diagnosis also affect the developers' confidence (**P11**). According to our study, the attributes that influence the developers' confidence the most are: accuracy, density, element role, and diversity. It is not a surprise that the more accurate (**P12**) and denser (**P13**) the developer believes that the symptom is, the more confident he will be in the presence of a design problem. Nevertheless, we noticed that the element role plays an even greater influence on the developers' confidence (**P14**).

Developers' divergence regarding element role. At first glance, we noticed that when most developers analyze an element that plays an important role in the system, they tend to assume that the element contains a design problem. Examining further, we observed two behaviors. When developers analyzed element role together with other attributes, they tended to confirm the corresponding design problem. Conversely, whey they only considered the element role (ignoring other attributes), they tended to reject the design problem, arguing it is acceptable to have design problem symptoms in elements that play an important role in the system.

These two behaviors happened with T2 and T4, respectively. Developers of the T2 team confirmed the design problem in the element because, among other attributes, the element played an important role in the system. On the other hand, T4 developers said that, due to the element role, it is acceptable that the element contains the design problems symptoms. According to them, if the element were not an important class for the system, it would not be acceptable to have a design problem or its symptoms in the class.

Pondering about the number of symptoms. When a developer analyzes individual symptoms, the number of symptoms with which he agrees or disagrees influences his confidence in the design problem identification. When analyzing each symptom, the developer decides whether it indicates a design problem. In the end, he counts the number of symptoms he judged as indicating a problem and the number of symptoms he judged as irrelevant. If the former is greater than the latter, then he confirms that the element has a design problem. T3 developers used this strategy to increase their confidence in the presence of a design problem in some elements.

5.2 Conscientiousness

Conscientiousness (**C15**) is a personality trait related to being careful, responsible, and persevering [30]. The more conscientious the developer is, the greater the likelihood of identifying a design problem. Likewise, when developers diagnose more design problems, they become more conscientious. As these attributes have a circular effect between them (**P15**), it would be interesting to find ways to increase the developers' conscientiousness.

Diversity as an attribute to increase conscientiousness. The diversity of symptoms is the attribute that most influences the conscientiousness of the developers (**P16**). The higher the diversity of a syndrome, the greater the chance the developer will identify a design problem in the element. That happens because the diversity not only increases the confidence of the developers, but it can also help the developers to decide whether the element contains a design problem. In fact, we noticed that the diversity had a great influence on developers of the T7, T9, T10 and T11 teams, because they tended to assume diversity as a strong indicator of a design problem (Section 4.2). Therefore, this finding is another evidence that studies that assume that developers rely on only one type of symptom may be misaligned with the developers' practice [27, 32, 52, 53]. Even worse, these studies are not taking advantage of the impact that the diversity attribute has on developers' conscientiousness.

Side effect of only considering the diversity attribute. We noticed a side effect when developers overestimate the importance of the diversity of a syndrome without further analyzing other attributes. For instance, after the T4 developers had analyzed a set of elements with diverse symptoms, they later judged an element as free of a design problem because it did not have the same diversity of symptoms as the ones analyzed previously. Although this behavior was not very frequent, it brings out another issue that studies that rely on only one type of symptom do not take into account.

5.3 Incapability of Providing an Alternative

Justifying the presence of a design problem with design decisions. Sometimes the developers are convinced that an element contains several symptoms that indicate a design problem, even though they do not confirm the presence of a design problem. Although such behavior seems contradictory, they argue that they do not consider the element as containing a design problem because

they see no other way to implement the element. In these cases, developers use the concept of design decision to justify why they do not consider the presence of a design problem (**P17**). Consequently, the decision design that developers use as an argument influences their confidence in the presence of a design problem (**P18**).

Developers justified the presence of a design problem mostly when they could not provide an alternative implementation. This behavior is aligned with the theory discussed by March and Simon [23], who theorized that developers typically do not choose an optimal solution because such solution would require that all alternatives to a problem be perceived. However, they argue that in practice it is unlikely for developers to know all alternatives. Hence, the known alternatives represent the boundaries that developers face before making a decision. Therefore, developers stop searching for further solutions when one that satisfies their needs is found.

Justifying the presence of a design problem with the lack of an alternative implementation. March and Simon's [23] theory also manifests in the context of identifying design problems, as we observed in our study. The developers used the limited known alternatives to justify why a specific implementation does not present a design problem. In these cases, they mentioned that they could not find any other alternative solution (optimal or not) to implement the element. According to them, the element should not be considered as an element involved in a design problem. That is, the known alternatives are not only boundaries that developers face, but also used to justify the presence or absence of a design problem.

6 TOWARDS IMPROVING DESIGN PROBLEM DIAGNOSIS

Researchers can use the discussions presented in this paper as an underlying mechanism to drive solutions for supporting developers during design problem identification. For instance, in this section, we present some of these solutions that emerged from the theory.

6.1 Supporting Multiple Symptoms

Providing multiple design problem symptoms. Most studies rely on a single, predefined, dominant type of symptom [27, 32, 52, 53], which may be limiting how developers identify design problems in practice. Thus, there is a need for solutions that provide developers with multiple symptoms, and then help them to navigate among these symptoms and to combine them. In fact, we noticed that developers would benefit from mechanisms to automatically provide symptoms for combination. For instance, a solution in this sense is to provide other symptoms that are complementary to the one being analyzed. Such tool, for instance, could have helped the T2 developers to identify a design problem (Section 4.3.1). They used the Dispersed Coupling to choose the coupling attribute to analyze next. Later, they chose the readability non-functional requirement to complement their analysis. In this example, a tool could provide the coupling attribute and the readability requirement as soon as the developers indicate the Dispersed Coupling code smell as helpful.

Filtering relevant symptoms. Developers take into consideration the diversity of symptoms. However, if an element manifests several symptoms, the developers could have a hard time to choose the most helpful one. For instance, D16 (T8 team) mentioned the difficulty that he had to choose helpful symptoms:

> D16: *"Since I was not familiar with each type of symptom and design problem, it was hard for me to match them. Even with the provided symptoms, I could not figure out which one was actually related to the design problems."*

To address this issue, an automatic tool could help them to filter those symptoms that are most likely to indicate a design problem. In the same way that a tool could propose complementary symptoms to the one being analyzed, it could hide symptoms that are least similar to the one under analysis. Such tool could make the analysis of multiple symptoms less cumbersome.

Visualization support. Another solution to help developers to deal with multiple symptoms is to provide visualization mechanisms to help them to analyze the symptoms. For instance, the Scattered Concern [12] problem occurs when multiple code elements implement a functionality that should have been implemented by only a few elements. In this case, developers have to analyze multiple elements that may have the scattered functionality. These elements are likely to share some symptoms. Perhaps if developers could visualize how the multiple symptoms interact in the system, they could identify these elements more easily. In fact, D14 (T7 team) mentioned in the follow-up questionnaire that a visualization mechanism would help him to identify some design problems:

> D14: *"For some design problems e.g. Cyclic Dependency, Scattered Concern, it's hard to find by looking at the source code manually, which is too low level when we don't have a higher level architecture view."*

6.2 Prioritization of Similar Elements

Prioritizing epidemic elements. We noticed that developers tend to prioritize elements that play an important role in the system. In addition, if these elements have diverse symptoms, then they should be the first elements to be analyzed by the developers. Researchers could therefore use the attributes presented here to build tools that prioritize elements. For instance, developers of the T2 team used the element role during the epidemic analysis (Section 4.3.2). In two cases they relied on the element role to select epidemic elements. However, in one case they could identify a design problem, whereas in the other case, they could not. The difference between these two cases was related to the number of epidemic elements playing the same role. While in the first case all the epidemic elements played the `Service` role, in the second case only few epidemic elements played the `Controller` role. The following quotation illustrates this.

> D4: *"I think that all the service classes will have (the design problem)"*
> D3: *"Indeed, the service (classes)"*
> D4: *"I guess that (they) are similar to each other. In fact, I believe that the next service (class) will be similar"*

6.3 Additional Support for the Developer

Based on the propositions concerning the developers (Section 5) we suggest providing the following additional support.

Providing an alternative implementation. It is often difficult for developers to provide an alternative implementation for an element that may contain a design problem (Section 5.3). In this context, a tool could indicate an alternative implementation that could remove the design problem symptoms. Hence, a developer would not be able to use the lack of an alternative implementation as justification for not confirming a design problem.

Personalizing the detection of symptoms. The accuracy of the symptom is also influenced by the developers' subjectivity. Developers mentioned that a certain type of symptom was accurate in indicating a design problem in some elements, but not in others. Thus, most developers mentioned that they need tools that allow them to personalize the detection of symptoms according to their software systems. Allowing developers to adjust thresholds and detection rules would minimize how the (low) accuracy influences their confidence in the presence of a design problem. D11 (T6 team) mentioned in the follow-up questionnaire the need for such feature:

> D11: *"The symptoms suggest a possible design problem. However, none of them should be rigid rules. Often, it makes sense to have long methods, message chains or many parameters (in the method). In some cases, we could replace a long string of conditional (statements), but it would make it difficult to understand. A method was considered long, but its readability was very clear, which did not justify a refactoring."*

7 RELATED WORK

Along the paper, we presented some studies about the identification of design problems. We have not found studies that present the diagnosis of design problems as a theory. Instead, we found studies that focus on presenting the phenomenon rather than explaining it [6, 20–22, 27, 28, 32, 47, 50, 53]. For instance, several researchers proposed techniques to identify design problems [20–22, 32, 53]. Although these studies had encouraging results, they did not conduct experiments with software developers or they have not taken into account the attributes that affect design problem identification.

As far as we are concerned, Sousa *et al.* is the only study that has proposed to explain how developers identify design problems [41]. However, they fell short of framing their results as a theory. Similar to the other studies, they only present the phenomenon, rather than explain it. For instance, the authors only provide the symptoms that developers take into account, but they could neither explain how developers find these symptoms nor describe the attributes that developers take into account during the diagnosis. Conversely, we highlight that our goal was not to provide the most preeminent symptoms nor the symptoms that lead to the identification of design problems. Instead, we focused on revealing the attributes that contribute to diagnosing a design problem the most, which allows us to explain how developers identify design problems in practice.

8 THREATS TO VALIDITY

This section presents and discusses threats to validity.

Construct Validity. We provided some symptoms for developers to use during design problems diagnosis. These data could have biased the experiments. However, we provided these data considering the literature [11, 20–22, 24, 32] and considering the companies' managers. They mentioned that some of the developers not only were familiar with some symptoms but also had the culture of using them. Furthermore, our goal was to explain the diagnosis of design problems, and not delve too deeply into the symptoms. The time allocated for the tasks could be considered another threat to validity. However, we conducted a pilot study to adjust the time required to perform the tasks and thus reduce the threat.

Internal Validity. The difference between the developers' background knowledge can be a threat. However, in the context of applying an analysis through GT, we saw this diversity as an opportunity to strengthen the evidence supporting the depicted propositions. Moreover, we provided training to mitigate this threat.

External Validity. The number of subject represents a threat. All of them worked for companies located in Brazil. However, it is important to note that this is a multi-company study involving five different working environments and eight different systems. Finally, the presented study covered only systems developed in Java. Using other programming languages with different core characteristics may influence developers in identifying design problems.

Conclusion Validity. The participation of the author who followed the GT procedures poses another threat. His beliefs might have caused some distortions when interpreting the data. To mitigate this threat, the GT coding activities were shared with other researchers. Moreover, the identification of the constructs and the depicting of propositions were performed separately by the first author and other researchers. In fact, three authors conducted the Grounded Theory procedures independently; then we merged their results to shape the theory. Thus, the contents were compared and discussed by the researchers until reaching a consensus.

9 CONCLUDING REMARKS

A design problem is the result of one or more inappropriate decisions that negatively impact non-functional requirements. Despite their harmfulness, the identification of each design problem is not trivial. One of the main reasons is that design documentation is often unavailable or outdated. Thus, developers often have to rely on the source code to identify design problems, which may quickly turn into a complex task. Although researchers have investigated techniques to help developers, there is little knowledge on how developers actually proceed to identify design problems in practice.

In order to address this limitation, we conducted a multi-trial industrial experiment with developers from different companies, where they had to identify design problems in their systems. As a result, we derived a theory describing the activities and factors that influence on how developers identify design problems, which can serve to further understand the identification of design problems. For example, the theory reveals that developers rely on a heterogeneous set of symptoms, and they tend to combine symptoms. The theory also presents the characteristics of symptoms that developers consider helpful. Then, we discussed how the knowledge revealed by our theory can be used to advance the state-of-art.

Future steps in this work involve the execution of new empirical studies to assess in more depth the theory's propositions and explanations. For instance, we intend to address some findings described at Section 6 and verify if they have positive effects on design problem identification. The goal of these studies is to use the theory to implement a novel family of solutions that are more effective than the current ones in helping developers identify design problems.

10 ACKNOWLEDGMENT

The authors would like to thank the reviewers for their valuable comments and suggestions. This work is funded by CAPES/Procad (grant # 175956), CAPES/Procad (grant # 175956), CNPq (grants # 309884/2012-8, 483425/2013-3 and 477943/2013-6), FAPERJ (E26-102.166/2013), FAPERJ (grant # 102166/2013 and 22520 7/2016) and FAPEAL (grant #60030 1201/2016 FAPEAL PPGs 14/2016).

REFERENCES

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

[2] 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.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing.

[4] G Campbell and Patroklos P Papapetrou. 2013. *SonarQube in action*. Manning Publications Co.

[5] Munkhnasan Choinzon and Yoshikazu Ueda. 2006. Detecting Defects in Object Oriented Designs Using Design Metrics. In *Proceedings of the 2006 Conference on Knowledge-Based Software Engineering: Proceedings of the Seventh Joint Conference on Knowledge-Based Software Engineering*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 61–72. http://dl.acm.org/citation.cfm?id=1565098.1565107

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

[7] Bill Curtis, Jay Sappidi, and Alexandra Szynkarski. 2012. Estimating the Size, Cost, and Types of Technical Debt. In *Proceedings of the Third International Workshop on Managing Technical Debt (MTD '12)*. IEEE Press, Piscataway, NJ, USA, 49–53. http://dl.acm.org/citation.cfm?id=2666036.2666045

[8] K. Anders Ericsson and Herbert A. Simon. 1993. *Protocol Analysis: Verbal Reports as Data* (2 ed.). A Bradford Book.

[9] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. 2007. JDeodorant: Identification and Removal of Feature Envy Bad Smells. In *2007 IEEE International Conference on Software Maintenance*. 519–520. https://doi.org/10.1109/ICSM.2007.4362679

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

[11] 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.

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

[13] B.G. Glaser. 1998. *Doing Grounded Theory: Issues and Discussions*. Sociology Press. https://books.google.com.br/books?id=XStmQgAACAAJ

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

[15] J. E. Hannay, D. I. K. Sjoberg, and T. Dyba. 2007. A Systematic Review of Theory Use in Software Engineering Experiments. *IEEE Transactions on Software Engineering* 33, 2 (Feb 2007), 87–107. https://doi.org/10.1109/TSE.2007.12

[16] Ross Jeffery. 2013. *Paths to Software Engineering Evidence*. Springer Berlin Heidelberg, Berlin, Heidelberg, 133–144. https://doi.org/10.1007/978-3-642-37395-4_9

[17] P. Johnson, M. Ekstedt, and I. Jacobson. 2012. Where's the Theory for Software Engineering? *IEEE Software* 29, 5 (Sept 2012), 96–96. https://doi.org/10.1109/MS.2012.127

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] J.G. March and H.A. Simon. 1958. *Organizations*. Wiley. https://books.google.com.br/books?id=fx1HAAAAMAAJ

[24] R Martin. 2002. *Agile Principles, Patterns, and Practices*. Prentice Hall, New Jersey.

[25] 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.

[26] Complementar Material. 2017. https://ssousaleo.github.io/ICSE2018/. (2017).

[27] 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.

[28] 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.

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

[30] W. T. Norman. 1963. Toward an adequate taxonomy of personality attributes: replicated factors structure in peer nomination personality ratings. *Journal of abnormal and social psychology* 66 (June 1963), 574–583.

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

[32] 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*.

[33] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 101–110. https://doi.org/10.1109/ICSME.2014.32

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

[35] Dewayne E. Perry and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes* 17, 4 (Oct. 1992), 40–52. https://doi.org/10.1145/141874.141884

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

[37] 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.

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

[39] 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.

[40] Dag I. K. Sjøberg, Tore Dybå, Bente C. D. Anda, and Jo E. Hannay. 2008. *Building Theories in Software Engineering*. Springer London, London, 312–336. https://doi.org/10.1007/978-1-84800-044-5_12

[41] Leonardo Sousa, Roberto Oliveira, Alessandro Garcia, Jaejoon Lee, Tayana Conte, Willian Oizumi, Rafael de Mello, Adriana Lopes, Natasha Valentim, Edson Oliveira, and Carlos Lucena. 2017. How Do Software Developers Identify Design Problems?: A Qualitative Analysis. In *Proceedings of 31st Brazilian Symposium on Software Engineering (SBES'17)*. 12.

[42] Klaas-Jan Stol and Brian Fitzgerald. 2015. Theory-oriented software engineering. *Science of Computer Programming* 101 (2015), 79 – 98. https://doi.org/10.1016/j.scico.2014.11.010 Towards general theories of software engineering.

[43] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 120–131. https://doi.org/10.1145/2884781.2884833

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

[45] 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.

[46] Richard N. Taylor and Andre van der Hoek. 2007. Software Design and Architecture The Once and Future Focus of Software Engineering. In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 226–243. https://doi.org/10.1109/FOSE.2007.21

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

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

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

[50] 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.

[51] Santiago A. Vidal, Hernán Ceferino Vázquez, Jorge Andrés Díaz Pace, Claudia Marcos, Alessandro F. Garcia, and Willian Nalepa Oizumi. 2015. JSpIRIT: a flexible tool for the analysis of code smells. In *34th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, Santiago, Chile, 1–6.

[52] S. Wong, Y. Cai, M. Kim, and M. Dalton. 2011. Detecting software modularity violations. In *Software Engineering (ICSE), 2011 33rd International Conference on*. 411–420. https://doi.org/10.1145/1985793.1985850

[53] 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

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