Code Anomalies Flock Together

Exploring Code Anomaly Agglomerations for Locating Design Problems

William Oizumi
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
woizumi@inf.puc-rio.br

Alessandro Garcia
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Leonardo da Silva Sousa
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
lsousa@inf.puc-rio.br

Bruno Cafeo
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
bcafeo@inf.puc-rio.br

Yixue Zhao
Dept. of Computer Science
USC
Los Angeles, USA
yixuezha@usc.edu

ABSTRACT

Design problems affect every software project. Diverse software systems have been discontinued or reengineered due to design problems. As design documentation is often informal or nonexistent, design problems need to be located in the source code. The main difficulty to identify a design problem in the implementation stems from the fact such a problem is often scattered through several program elements. Previous work assumed that code anomalies – popularly known as code smells – may provide sufficient hints about the location of a design problem. However, each code anomaly alone may represent only a partial embodiment of a design problem. In this paper, we hypothesize that code anomalies tend to "flock together" in order to realize a design problem. We analyzed to what extent groups of inter-related code anomalies, named agglomerations, suffice to locate design problems. Specifically, we analyzed more than 2200 agglomerations found in seven software projects of different sizes and from different domains. Our analysis indicates that certain forms of agglomerations are consistent indicators of both congenital and evolutionary design problems, with accuracy often higher than 80%.

1. INTRODUCTION

Design problems are structures that indicate violations of key design principles or rules [36]. Every software system suffers from design problems, introduced either during original development or during evolution. Examples of design problems are *Fat Interfaces* [10, 36], *Overused Interfaces* [10, 36], and *Scattered Concerns* [10, 36]. These problems may have different degrees of severity, but all of them should be detected and possibly removed from the source code. Software systems have been often discontinued [21] or have had to be fundamentally reengineered [12, 38, 32] when design problems were allowed to persist in a system and to be compounded by other design problems introduced later.

Design problems are introduced and allowed to remain in a system because it is difficult to locate them in the source code. As design documentation is often informal or nonexistent, code anomalies – popularly known as code smells [7] – are used as surface indicators of design problems. Examples of code anomalies are *Long Method*, *Feature Envy*, and *God Class*. Even though each code anomaly can provide some hint to developers, it alone might not suffice to indicate the presence of a design problem. Each design problem is rarely localized in a single anomalous element; instead, it is scattered into different code elements of the implementation [28].

As an example, let us assume a developer is in charge of identifying *Fat Interfaces*. He will need to go through all code anomalies affecting each interface in the program. In particular, he will need to inspect all the classes that implement the interfaces in the source code. Then, he will have to analyze all the clients of such classes. Furthermore, it is hard and time-consuming to identify on which code anomalies he should focus. Even for small software systems, there are hundreds of code anomalies [24] and thousands of possible relationships to examine.

The relation between design problems and their counterpart code anomalies is often complex. Unfortunately, there is no understanding of which relationships between code anomalies are frequent indicators of design problems in an evolving program. There is a recent growing interest in conceptually characterizing interactions between code anomalies [1, 2, 28, 34, 42]. However, the relation of code anomalies and design problems is rarely investigated. Empirical studies only address how individual occurrences of code anomalies emerge during software evolution [37] or affect quality attributes [5, 16, 17, 20, 30, 13]. They do not analyze how individual anomalies and their relationships in the code might help developers to spot design problems.

In this paper, we hypothesize that code anomalies "flock together" in order to embody different design problems. Multiple anomalies may interact in the source code structure, with each code anomaly only representing part of the design problem. We perform a multi-case study in order to investigate how code anomaly relationships can help to locate design problems in the source code. We focus on *agglomerations* of code anomalies that are syntactically or semantically related. We analyze seven systems of different sizes (8 KSLOC to 129 KSLOC) and from different domains. Our analysis involves a total of 5418 code anomalies and 2224 agglomerations. We investigates the circumstances under which agglomerations are related (or not related) to design problems. The analyzed circumstances involve: (i) the statistical significance of the relation between agglomeration types and design problems, (ii) the strength of this relation, and (iii) the extent to which agglomerations manifest themselves at different stages of a system's lifetime, i.e., in early vs. late versions of a system. The aforementioned analysis resulted in several findings:

1. Overall, approximately 50% of the syntactic agglomerations were related to design problems. Their use would help developers to discard almost 4000 non-agglomerated code anomalies that are irrelevant to locate design problems. In general, at least 3 code anomalies of each syntactic agglomeration are related to the same design problem in all the systems.

2. However, syntactic agglomerations do not suffice to assist developers in locating all design problems. Semantic agglomerations are much more consistent indicators of design problems across our subject systems. On average, 80% of the semantic agglomerations were related to design problems. Several design problems in all the analyzed systems can only be revealed with semantic agglomerations.

3. Analysis of history co-changes has been recently used as a means to identify design problems during software evolution [39, 33, 40]. However, we observed this approach is ineffective in locating a significant proportion of design problems. The reason is that those problems were "congenital", i.e., they were already introduced in the system's initial version. Co-changes affecting the corresponding anomalous elements might occur only in later versions of a software project, but it is usually too late to identify and remove congenital design problems. Many clients already depend on the anomalous code elements that realize the design problem.

The paper is organized as follow. Section 2 presents the background required to understand the paper. Section 3 describes the settings of our study, including the research questions and the procedure for data collection and analysis. Section 4 summarizes the main results of our study. Section 5 and Section 6 present the related work and the threats to validity of this work, respectively. Finally, Section 7 concludes the paper.

2. BACKGROUND

This section presents the background required to understand our study settings. Section 2.1 presents key concepts, and Section 2.2 introduces a motivating example.

2.1 Code Anomalies and Design Problems

A code anomaly, popularly known as a "code smell", is a micro structure in the program that represents a surface indication of a design problem [7]. Examples of code anomaly types vary from method-level smells, such as *Long Method* and *Feature Envy*, to class-level smells, such as *God Class*, *Data Class*, *Shotgun Surgery*, and *Divergent Change* [7, 18]. Even in small programs, developers often have to face hundreds or thousands of code anomaly instances. They need to analyze all these anomalies in order to discard, postpone or further consider them. Each code anomaly possibly represents a partial hint of a design problem, but many anomalies may no contribute at all in helping developers to locate any design problem [23, 22, 24].

An important class of code anomalies actually represents design problems. Design problems are structures that indicate violation of intended design rules or fundamental design principles, and negatively impact design quality [8, 36]. Examples of relevant design problems are *Fat Interface* [25] and *Unwanted Dependencies* [31]. The former is a general, ambiguous entry point of a design component that provides non-cohesive services, thereby complicating the logic of its clients. This design problem violates the well-known principles of cohesion, abstraction, and separation of concerns. The latter represents the violation of a design rule, specifying two design components that should not be allowed to communicate.

In this study, we focus on such design problems that affect a system's design decomposition into major sub-systems (components) and their interfaces. Such design problems are often targets of major maintenance efforts [10, 32, 41], and are very often associated with design degradation that leads to partial or full discontinuation of a software project [12, 38, 32, 21]. Therefore, they should be removed as early as possible from a system.

Design problems often affect multiple program elements and are hard to identify in the source code. Then, code anomalies analyzed individually may not be sufficient to help developers in diagnosing a design problem. Additionally, certain anomaly types convey micro-structures that are the natural implementation solutions given the role of the program element. For instance, certain types of classes (e.g., lexical-analyzer classes) naturally address several concerns, while certain methods (e.g. command processing and dispatching) are naturally long [18]. Therefore, certain code anomalies neither cause nor contribute to the location of a design problem.

2.2 Motivating Example

Let us consider the example shown in Figure 1 extracted from the Apache OODT system [26]. OODT is a data-grid system aimed at supporting the management and storage of scientific data. OODT's *Versioning* component, which exports the *Versioner* interface (see Figure 1) is responsible for managing and storing versions of different *Product* types using different storage strategies. All classes that implement the *Versioner* interface have to implement a method, called *createDataStoreReferences*. One of the parameters of this method is a *Product* instance. The *Product* class, in its turn, can represent several types of structures, such as flat and hierarchical. As there are no sub-classes for each type of *Product*, each *createDataStoreReferences* implementation has to decide if it is handling with the correct *Product* type. For example, the *MetadataBasedFileVersioner* only deals with "flat" products.

Over time, OODT developers realized that there may be a design problem located in the *Versioning* component. Scattered changes involving this sub-system have been a major source of maintenance effort. They ran a static analysis in order to identify code anomalies that point to the design problem. The analysis output consists of several dozens of code anomalies scattered across several classes and methods of the *Versioning* component. They analyze several anomalies individually and decide to discard them from the list, as they did not represent a threat to the program structure.

Finally, developers zero in on a frequently changed method of *BasicVersioner*, called *createDataStoreReferences*. In fact, this method is simultaneously affected by four anomalies: *Long Method*, *Feature Envy*, *Shotgun Surgery* and *Divergent Change*. However, such local analysis does not suffice to conclude whether this is part of a major design problem. OODT developers begin looking for other anomalies at neighboring methods and classes, i.e. those methods and classes that have syntactic relationships with *createDataStoreReferences* and *BasicVersioner*. They work their way "outward" by navigating in the hierarchy structure, and move up to analyzing all clients of the *Versioner* interface, such as *GenericFileMngObjFactory* and *XmlRpcFileMngrClient*.

After observing all the anomalies affecting the hierarchical structure of *Versioning* and its direct clients, the developers are able to infer that the component is being affected by the *Fat Interface* problem: the *Versioner* interface is a single entry point of the component, but only the anomalies (e.g., *Feature Envies*) scattered in its four implementations reveal the non-cohesive nature of the interface. Those classes need to change every time each of the different product types are changed (i.e., *Divergence Changes*). The scattered occurrences of *Shotgun Surgeries* in the *Versioning* component's clients reinforce the interface is providing several non-cohesive services, which should be segregated into separate interfaces. Each client also needs to change every time the list of products and their details are changed. All the components that are related to *Versioner* and have both anomalies (*Feature Envy* and *Shotgun Surgery*) are inside of the dashed line in the Figure 1. These are the components affected by *Fat Interface*.

The presence of *Fat Interface* in the *Versioning* component is causing other harms. While a *Fat Interface* decouples components, it makes the component less understandable and analyzable. Determining the actual services exposed by such a component requires inspecting its implementation details. Furthermore, the generality of the interface also makes it easier to misuse, since different functionalities are exposed by the same interface. Yet the only way this potentially critical design flaw could be discovered was by reflecting upon multiple related code anomalies that were located in syntactically related modules: the interface, its subclasses, and the interface clients.

3. STUDY DEFINITION

Our study investigates whether inter-related code anomalies, referred to *anomaly agglomerations*, are related to design problems. Section 3.1 describes and motivates our research questions. Sections 3.2 and 3.3 present the types of agglomerations we have investigated. Section 3.4 describes the target systems of our study. Finally, 3.5 addresses the procedure for data collection and analysis.

3.1 Research Questions

Existing techniques [1, 28, 34] usually assume that individual anomalies suffice for assisting developers in locating a design problem in the program. The previous section has shown that each design problem may be realized by various inter-related code anomalies scattered in the program. However, there is no understanding of whether certain anomaly relationships can help developers in better locating design problems than individual code anomalies. We are primarily interested in groups of syntactically related code anomalies, i.e. *syntactic agglomerations* for short (Section 3.2). Two anomalies are syntactically related if their host program elements are connected through method calls and inheritance relationships. We first investigate whether syntactic agglomerations often embody design problems in the source code. If this hypothesis holds, the inspection of such agglomerations can help developers in locating design problems in the source code. This reasoning leads to our first research question:

RQ1 Are syntactic anomaly agglomerations sufficient indicators of design problems?

Syntactic agglomerations may not suffice to locate several design problems. Thus, other forms of code anomaly agglomerations may be required to locate design problems in the source code (Section 3.3). Therefore, we address RQ1 by analyzing the relation between diverse forms of agglomerations and design problems in several evolving systems. We also analyze this relation in early versions of a program. Recent studies [2, 37] revealed that most of times programs are affected by code anomalies since their creation. In addition, it might be that some design problems are "congenital", i.e. they manifest in the first versions of a program (Section 2.2). However, there is no understanding about the relation between "early" code anomalies and congenital design problems; and more importantly, between anomaly

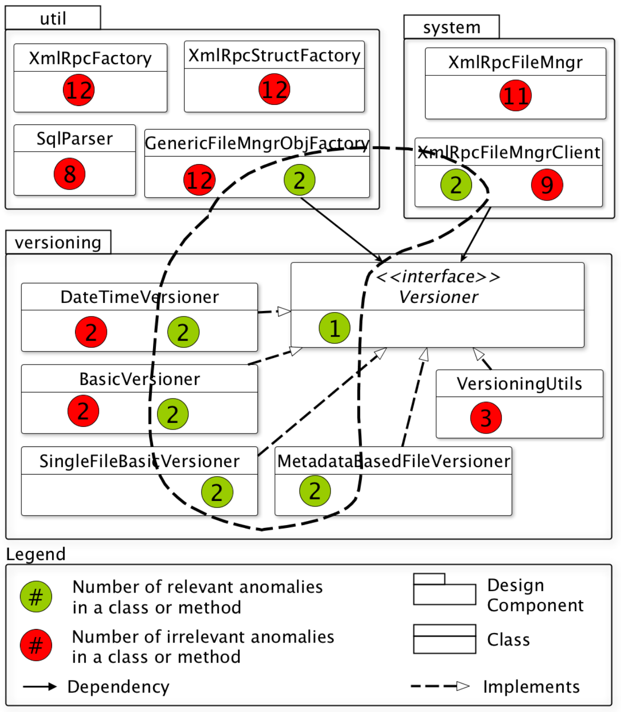

Figure 1: Fat interface affecting Versioner implementation

agglomerations and design problems. This gap motivates our second research question:

RQ2 What proportion of design problems manifest as anomaly agglomerations in early versions of a program?

We address RQ2 by investigating the relation of design problems and code anomaly agglomerations in early versions of a system. If this relation holds, early detection of congenital design problems can be improved by the synthesis of code anomaly agglomerations. We address RQ2 by analyzing the available initial versions of the analyzed software systems.

3.2 Syntactical Agglomerations

A *syntactic agglomeration* is a group of at least two anomalous code elements explicitly related in a program. A program element is *anomalous* when it is affected by one or more code anomalies. In our study, we consider the following program elements: classes, interfaces, methods, constructors, and fields. An explicit relationship is established between two anomalous elements whenever they have at least one of the following dependencies: a shared attribute, a method call, class extension and method overload. We classify syntactic agglomerations according to their scope in the program: (i) *intra-component agglomerations*, i.e. those entirely located within a single component, and (ii) *inter-component agglomerations*, i.e. those located in two or more components. In our study, we consider each component is realized by a package in the analyzed Java programs (Section 3.4), unless the system developers specify otherwise. Developers may state a component is realized by a set of classes, which are not necessarily located in the same package (Section 3.4).

Inter-Component Agglomeration is a syntactic agglomeration that involves two or more design components, with at least one anomalous program element located within each of them. The example in (Section 2.2) illustrates an inter-component agglomeration in the OODT system. The agglomeration is formed by anomalous code elements located in three OODT components: *Versioning*, *Util* and *System*. In Figure 1, green circles represent the code anomalies located in the syntactically related program elements that compose the inter-component agglomeration. In the figure, the inter-component agglomeration is surrounded by the dashed black line.

Algorithm in Listing 1 illustrates the pseudo-code for computing inter-component agglomerations. The algorithm has two parameters: (i) *dc*: a set of design components, and (ii) *agglomThreshold*: a threshold value for the minimum agglomeration size. First, the algorithm identifies all anomalous elements of a design component by using the function *getAnomalousElemsOfComponent()* (line 6). The anomalous elements of the program are recorded in the attribute *anomalousElems*. The algorithm generates a graph (*graphInterElems*) with the syntactic relationships (edges) between the anomalous elements (vertices) located in different design components (line 10 and lines 12–19). To find the relationships between elements, the algorithm uses the function *isRelated()* (line 15). In order to detect the agglomerations, the algorithm finds the subgraphs within the graph in which any two vertices are connected (i.e. the subgraph formed by the relationships among the anomalous elements). Finally, only the agglomerations with the size greater than *agglomThreshold* are included in the resulting set of inter-component agglomerations, i.e. *interAgglomerations* (lines 25–30).

Intra-Component Agglomeration is a syntactic agglomeration composed of anomalous code elements located in the same design component. Given space constraints, the algorithm is available in our on-line supplementary material [29]. However, the algorithm for computing intra-component agglomerations is similar to the previous algorithm for identifying intra-component ones. The algorithm inputs are the same. The only difference is that the algorithm generates a graph with the relationships (edges) between the anomalous elements (vertices) within each design component.

Listing 1: Inter-component Agglomeration Algorithm

```
1. public List{} getInterAgglomeration(Set dc, int
    agglomThreshold){
2. InterAgglomerations = {}
3. anomalousElems = {}
4.
5. for(each design component dc in the program){
6.  anomalousElems.addAll(
    getAnomalousElemsOfComponent(dc))
7. }
8.
9. //Adding anomalous code elements as vertices in
    the graph
10. graphInterElems = new graphInterElems().addVertex
    .addAll(anomalousElems)
11.
12. for(each e1 in anomalousElems){
13.   for(each e2 in anomalousElems){
14.   //Creating the Inter-component graph
15.     if(isRelated(e1,e2) && (e1.getDesignComponent
    () != e2.getDesignComponent())){
16.       graphInterElems.addEdge(e1,e2)
17.     }
18.   }
19. }
20.
21. // Adding the connected components as
    agglomerations
22. InterAgg = new InterAgg().addAll(
    getConnectedComponents(graphInterElems))
23.
24. // Selecting only agglomerations above the
    predefined threshold
25. for(each a in InterAgg){
26.   if(size(a.getAnomalies()) > agglomThreshold){
27.     InterAgglomerations.add(a)
28.   }
29. }
30. return InterAgglomerations
31. }
```

3.3 Semantic Agglomerations

Based on the example of Section 2.2, we hypothesized that code anomalies might somehow interact through the program structure because the presence of a single design problem. In that example, the syntactic relationships amongst the anomalous elements would be sufficient to help developers in revealing the design problem. However, this might not be the case for all occurrences of design problems. In certain cases, design problems might be related to two or more anomalies semantically connected. For instance, such anomalies might be closely related because their host program elements are addressing the same purpose or concern. In such cases, their closely related semantics may better help to locate certain design problems.

Let us consider an example extracted from the Mobile Media system [43]. Mobile Media is a software product line to derive applications that manipulate photos, videos and music on mobile devices [43]. Figure 2 depicts a partial view of three components in Mobile Media design: *Controller* and *Screens* and *Sms*. Classes of the *Screens* component are affected by the *Divergent Change* anomaly, while classes of the *Controller* and *Sms* components were infected by the anomalies: *Divergent Change* and *Shotgun Surgery*. These code anomalies did not represent isolate problems. Many of these classes are not syntactically connected, but their methods are in charge of partially addressing the same concern, called *Photo Label Management*. Therefore, the anomalous code elements are altogether realizing the design problem *Scattered Concern*, i.e. multiple components that are responsible for realizing a crosscutting concern [23, 9]. The realization of *Photo Label Management* should be modularized in the *Controller* component. This problem was the cause of major design refactorings along Mobile Media evolution. However, the design problem is better spotted if the scattered anomalies (i.e. *Divergent Changes* and *Shotgun Surgeries*) realizing the same concern are considered altogether. In our study, we chose concern-based agglomeration as a representative type of semantic agglomerations.

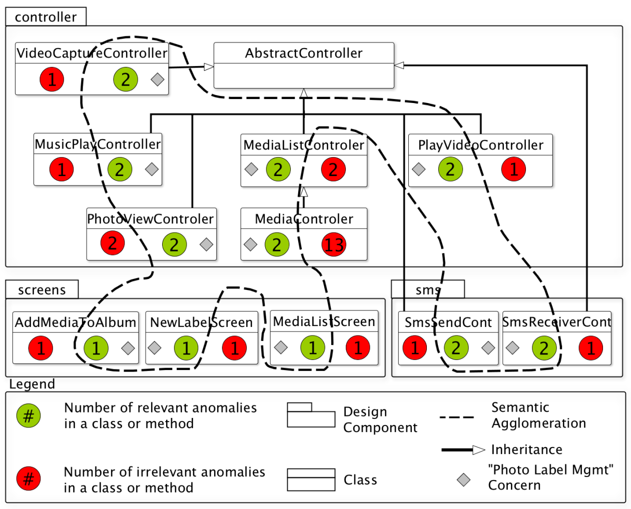

Figure 2: Scattered Concern in Mobile Media

Semantic Agglomeration is composed of anomalous code elements realizing a single concern, which is not modularized by design components. A concern is a property or functionality of interest to the designers of a system, but its realization is not necessarily modularized in a single component. Examples of design-relevant concerns can be classified in domain-specific concerns, such as Photo Label Management, or general-purpose concerns, such as persistence, error handling, security and the like. Semantic relationships between two or more code anomalies occur when their host code elements are intended to (partially) realize a single design's purpose or concern.

Algorithm in the Listing 2 presents the high-level pseudocode for detecting semantic agglomerations. The full algorithm is available at the website with supplementary material for the study [29]. The basic idea of the algorithm is searching for agglomerations for each concern *con* that satisfies the following conditions: (i) *con* is located in one or more components (line 7), and (ii) at least one of these components is not mainly dedicated to realize *con* (line 5). The components in (ii) are named *weakly-dedicated components*. The identification of weakly-dedicated components is implemented by the function *weakDedicatedComponents()* (line 5). This function computes, for each component realizing a concern, if program elements in the component are mostly dedicated to realize other concerns. In other words, a component *dc* is weakly dedicated to a concern *con* if *con* is not the main concern of the program elements within *dc*. Although *dc* partially implements *con*, the predominant concern of *dc* is not *con*. The minimum degree of dedication is captured based on a percentage threshold, named *weakThreshold*.

Therefore, the main algorithm uses four inputs: a set of design components *dc*, the mappings for each concern *con*, a threshold for the minimum agglomeration's size *agglomThreshold*, and *weakThreshold*. In our case study, the system developers provided the mapping of each design concern (Section 3.5). The anomalous code elements contributing to weakly-dedicated components of a crosscutting concern are grouped into an agglomeration candidate (lines 8 to 9). If the agglomeration candidate has a number of anomalies higher than the *agglomThreshold* (line 11), then this agglomeration is included in the results (line 12) and confirmed as an actual agglomeration.

Listing 2: Semantic Agglomeration Algorithm

```
1. public List[] getSemanticAgglomeration(Set dc, Set
     con, int agglomThreshold, int weakThreshold){
2. semanticAgglomerations = []
3.
4. for(each design concern con in the program){
5.   W = weakDedicatedComponents(con, weakThreshold)
6.   agglomeration = []
7.   if(size(W) > 0){
8.     ae = getAnomalousElemsPerConcern(W, con)
9.     agglomeration.addAll(ae)
10.   }
11.   if(size(agglomeration) > agglomThreshold){
12.     semanticAgglomerations.add(agglomeration)
13.   }
15. }
15.
16. return semanticAgglomerations
17. }
```

3.4 Target Systems

In order to address the two research questions, we analyzed systems with a wide range of characteristics. We selected 7 systems of different sizes (8 KSLOC to 129 KSLOC), leveraging different design styles, and spanning different domains. Table 1 summarizes the general characteristics of each target system. We focused on such seven systems because: (i) they undergone design degradation, (ii) there present a wide range of design problems, and (iii) their developers were available to provide us with a reliable list of design problems (which are causes of major maintenance effort) and the mappings of their design's concerns.

The first system in our list is Health Watcher (HW), a web framework system that allows citizens to register complaints regarding health issues in public institutions [35]. The second is Mobile Media (MM), an academic software product line to derive applications that manipulate photos, videos, and music on mobile devices [43]. The next four systems are proprietary and, due to intellectual-property constraints, we will refer to them as S1, S2, S3 and S4. The goal of S1 and S2 is to manage activities related to the production and distribution of oil. S3 manages the trading stock

Table 1: Characteristics of the Target Systems

	HW	**MM**	**S1**	**S2**	**S3**	**S4**	**OODT**
Aplication Type	Web Framework	Software Product Line	Desktop Application	Desktop Application	Desktop Application	Web Application	Middleware
Arquitectural Design	Layers	MVC	Client-Server	Client-Server	Client-Server	MVC	Components
KSLOC	8	10	122	118	93	116	129

of oil, while S4 is intended to support the financial market analysis. Finally, the last system is Apache OODT (Object Oriented Data Technology), whose goal is to develop and promote the management and storage of scientific data [26]. For all the target systems, several classes implement each component. In OODT, for example, each design component is implemented by an average of 24 classes. There was always a 1-to-1 mapping between components and packages in three projects: MM, HW, OODT, S3 and S4. In the other cases, the developers provided us with the set of classes implementing each component.

Table 2: Analyzed Design Problems

Name	**Description**	**Instances**
Fat Interface	Interface of a design component that offers only a general, ambiguous entry-point that provides non-cohesive services, thereby complicating the clients' logic.	114
Unwanted Dependency	Dependency that violates an intended design rule.	1947
Component Overload	Design components that fulfill too many responsibilities.	141
Cyclic Dependency	Two or more design components that directly or indirectly depend on each other.	351
Delegating Abstraction	An abstraction that exists only for passing messages from one abstraction to another.	35
Scattered Concern	Multiple components that are responsible for realizing a crosscutting concern.	216
Overused Interface	Interface that is overloaded with many clients accessing it. That is, an interface with "too many clients".	39
Unused Abstraction	Design abstraction that is either unreachable or never used in the system.	59

3.5 Data Collection Procedure

For all the systems, the data collection process comprised the following activities: (i) identifying design problems with the support of systems developers, (ii) detecting code anomalies, concern mappings and agglomerations, and (iii) correlating agglomerations and design problems. Next, we describe each activity.

Identifying Design Problems. We produced a "ground truth" of design problems for each target system. We performed two steps to incrementally develop the ground truth. First, original developers of the target systems provided us with an initial list of design problems. They answered a questionnaire [29] where they listed the problems and explained the relevance of each design problem. They reported what was the maintenance effort caused by the presence of each design problem. They also described which program elements were contributing to the realization of each design problem. Second, we performed other steps in order to validate the initial list for correctness and completeness. An additional identification of design problems was performed using the source code and the system design. For systems without design documentation, we relied on a suite of design recovery tools [9]. The procedure for deriving the list of design problems with developers was the following: (i) an initial list of design problems was identified using detection strategies presented in [2], (ii) the developers had to confirm, refute or expand the design problems identified, (iii) the developers provided a brief explanation to the researcher on the (ir)relevance of the design problem, and (iv) when we suspected there was still inaccuracies in the confirmed list of design problems, we asked the developers for further feedback. Table 2 describes the types of design problems and number of instances identified in our sample of systems.

Concern Mappings. The initial lists of concerns and their mappings in the source code were provided with the assistance of systems' developers. For each concern in their initial lists, they provided a list of methods or classes realizing those concerns. Given the large size of certain systems, developers could eventually not produce complete concern mappings. Therefore, we also relied on the use of an available tool for automatic concern location, named Mallet [27]. Mallet explores topic modeling in order to identify a list of concerns and, for each concern, the list of program elements realizing it in a program. We have chosen Mallet as the tool proved to be robust and scalable for large systems. Then, we computed and compared two sets of semantic agglomerations: one based on the concern mapping produced by developers, and one based on the concern mapping generated with Mallet. We discuss the comparison of such agglomerations in Section 4.2.

Code Anomalies and Agglomerations. We considered 7 types of code anomalies: *Data Class*, *Divergent Change*, *God Class*, *Shotgun Surgery*, *Feature Envy*, *Long Method* and *Long Parameter List*. They were selected because they represent different types of symptoms at both class and method levels. The detection of code anomalies was performed using well-known metrics-based strategies [18]. These detection strategies are similar to those typically used in previous empirical studies (e.g. [23, 22, 24, 37]). Such strategies have proven to be effective for detecting code anomalies in other systems, with precision higher than 80% [23, 2]. Once the anomalies were detected, we computed the syntactic and semantic agglomerations. The detection of both anomalies and agglomerations were carried out with a tool, called Organic [22]. Organic implements: (i) the 7 detection strategies for code anomalies, and (ii) the algorithms for computing agglomerations (Section 3.2 and Section 3.3). The results of all these steps are available at the supplementary material [29].

Correlating Agglomerations and Design Problems. As aforementioned, the developers determined the location of each design problem. In order to address our research questions, we defined a criterion for correlating design problems with anomalies and agglomerations. A design problem is related to an individual code anomaly if the former is either partially or fully realized by the code element affected by the anomaly. We consider a design problem and an agglomeration are related if they co-occur in at least two program elements. Even though both agglomeration and design problem may be located in more than two anomalous elements in the program, our criteria considers sufficient if they co-occur in two elements. Therefore, an agglomeration fails to indicate design problems when either none or only one of its code anomalies is related to a design problem. Finally, we have also computed the strength of the relation between

Table 3: Contingency Table and Fisher's Test Results

Agglomeration	**AG-DP**	**NoAG-NoDP**	**p-value**	**ORs**
Intra-component	247	3996	<0.001	5,669636
Inter-component	167	4207	<0.001	4,878982
Syntactic	296	3759	<0.001	5,513531
Semantic	97	4463	<0.001	7,392637
All	**312**	**4596**	**<0.001**	**2,999686**

agglomerations and design problems, i.e. we computed the number of anomalies in agglomeration that contributed (or not) to a design problem (Section 4.1). To check the statistical significance of our results, we used Fisher's exact test [6] and computed the Odds Ratio [4] through the use of the R tool [3].

4. RESULTS AND ANALYSIS

This section presents the results and analysis of our study. Section 4.1 discusses whether syntactic agglomerations assist the location of design problems. Then, Section 4.2 analyzes to what extent syntactic agglomerations suffice to locate design problems and, if not, whether semantic agglomerations can further improve this task. Section 4.3 discusses if agglomerations can also be effective to assist the location of design problems in early system versions.

4.1 Exploring Syntactic Agglomerations

Before answering our first research question, we reflect upon the relation of syntactic agglomerations and design problems. First, we check whether syntactic agglomerations are more often related to design problems than non-agglomerated code anomalies. By non-agglomerated anomalies, we mean those "detached" anomalies that do not take part of any agglomeration.

The results are presented in the first three columns of our contingency table (Table 3). The first column (named Agglomeration) lists the agglomeration types analyzed in our study. The second column (named AG-DP) shows the amount of agglomerations (for each category) related to design problems. The third column (named NoAG-NoDP) presents the number of "detached" anomalies that are irrelevant to locate design problems, i.e. they are not related to any design problem. We discuss the data of the last two columns later in this section. The data of the row "semantic" is discussed in Section 4.2.

Reducing the Search Space for Design Problems. Table 3 confirms that syntactic agglomerations are good indicators of design problems. Almost 300 syntactic agglomerations are related to design problems. Each row also shows how many "detached" anomalies could be discarded when developers are exploring a specific type of syntactic agglomeration (i.e. intra-component or inter-component agglomerations) in order to look for design problems. For instance, if developers are inspecting inter-component agglomerations, they will be able to: (i) find 167 design problems in their systems, and (ii) discard more than 4000 code anomalies returned by their static analysis tools. This result suggests that syntactic agglomerations can assist developers in locating design problems. When identifying design problems in the source code, the scope of analysis can be reduced to the list of code anomalies taking part in the agglomeration, rather than an unmanageable list of individual code anomalies.

Syntactic Agglomerations as Design Problems: Strength of the Relation. Even though the previous analysis is interesting, it does not consider the fact that anomaly agglomerations have higher probability of being related to design problems. This increase of probability stems from the fact that each agglomeration involves more code elements than a single code anomaly. The latter affects a single code element (e.g. a method or a class), while the former involves at least two program elements. The size of an agglomeration ranged from 2 to 9 in all systems. Therefore, we investigate the strength of the relation between agglomerations and design problems. That is, we perform further analyses to check to what extent various code elements of the agglomeration are, in fact, contributing to the realization of a design problem.

First, we calculate the Odds Ratio [4], which shows how much higher is the chance of code elements taking part in agglomerations to be related to design problems than individual anomalous elements. The chance of each anomaly within a syntactic agglomeration being related to a design problem is more than five times (5,513) higher than each "detached" anomaly. This result is indicated by the 5th column (ORs) of Table 3, which reports the Odds Ratio [4] for each type of agglomeration. Similar results are observed for both intra-component and inter-component syntactic agglomerations. Second, we also observe that almost all syntactic agglomerations had at least 3 anomalous code elements simultaneously related to the same single design problem in most of the systems. Therefore, when fixing a particular design problem, developers will likely to reveal several anomalous code elements involved in the refactoring strategy.

Syntactic Agglomerations as Design Problems: Statistical Significance. Finally, we analyze the statistical significance of the relation between syntactic agglomerations and design problems. We apply the Fisher's exact test in order to check the statistical significance. The fourth column (p-value) in Table 3 shows the correlation between each agglomeration type and design problems. We assume a confidence level higher than 99% (that is, p-value threshold = 0.001) as the threshold value to the significant level of the test. Applying the test, we observe that for all agglomeration type the p-value was lower than 0.001. That is, there is a very high correlation between both forms of syntactic agglomerations and design problems in the target systems. However, it does not necessarily mean that syntactic agglomerations should not be complemented with other forms of agglomerations in order to further improve the location of design problems. Therefore, we explicitly address our first research question in the next section.

4.2 Do Syntactic Agglomerations Suffice?

Even though syntactic agglomerations are statistically related to design problems, it is important to understand if syntactic agglomerations alone suffice to locate design problems. Thus, this section addresses the question RQ1: "Are syntactic anomaly agglomerations sufficient indicators of design problems?" In order to answer RQ1, we first analyze the proportion of syntactic and semantic agglomerations (un)related to design problems for each target system. We compared how often syntactic and semantic agglomerations relate to design problems. For each agglomeration type (represented in the columns), Table 4 shows: (i) the percentage of agglomerations related to design problems (Related), (ii) the percentage of agglomerations unrelated to design prob-

Table 4: Agglomerations (Un)Related to Design Problems

Agglomeration	Intra-component			Inter-component			Syntactic			Semantic		
System	Related	Not	DP	Related	Not	DP	Related	Not	DP (*)	Related	Not	DP (**)
OODT	62%	38%	196	73%	27%	549	70%	30%	640 (535)	91%	9%	431 (201)
MM	42%	58%	12	82%	18%	9	57%	43%	21 (21)	100%	0%	5 (3)
HW	39%	61%	22	45%	55%	163	44%	56%	163 (141)	75%	25%	187 (24)
S1	43%	57%	64	51%	49%	58	47%	53%	104 (86)	81%	19%	34 (18)
S2	38%	62%	77	40%	60%	60	39%	61%	110 (83)	82%	18%	47 (21)
S3	38%	62%	76	39%	61%	58	38%	62%	109 (84)	75%	25%	50 (25)
S4	66%	34%	66	34%	66%	45	49%	51%	69 (27)	100%	0%	10 (6)
Median	**42%**	**58%**		**45%**	**55%**		**47%**	**53%**		**82%**	**18%**	
SD	**0.1186876228**		-	**0.1825730543**		-	**0.1093058436**		-	**0.1074808442**		-

* Number of design problems exclusively related either to intra-component or to inter-component agglomerations.
** Number of design problems exclusively related to semantic agglomerations.

lems (Not), and (iii) the absolute number of design problems related to agglomerations (DP). Moreover, for each agglomeration type, Table 4 shows the median and the standard deviation for the columns Related and Not. The additional numbers between parentheses in the DP columns are discussed later in this section.

The fourth column (named Syntactic) indicates the aggregate results for both types of syntactic agglomerations, presented in the second and third columns. The reader should also notice that the values of the Syntactic column cannot be directly obtained by using the data in the previous columns. For example, the total number of design problems in the fourth column is not the result of summing the numbers of design problems for both types of syntactic agglomerations. The reason is that there are intersections between instances of intra- and inter-component agglomerations. That is, an inter-component agglomeration might be composed of anomalies that also are members of one or more intra-component agglomerations. As a consequence, a design problem might be related to more than one agglomeration.

Syntactic Agglomerations Suffice to Indicate Several Design Problems. Table 4 indicates that inter-component agglomerations help to locate more instances of design problems than intra-component agglomerations. In four (out of seven) systems, at least almost half (45%) of the inter-component agglomerations were related to design problems. However, Table 4 also reveals that both types of syntactic agglomerations are useful to locate different design problems in most systems. As we expected, these two types of syntactic agglomerations are complementary. This finding can be observed by comparing the numbers of the sub-columns DP from the second to the fourth columns. In the column Syntactic, there are additional numbers between parentheses close to the numbers of design problems (sub-column DP). These numbers represent the amount of design problems that were exclusively related to either intra-component or inter-component agglomerations. Therefore, the subsets of design problems related to intra- and inter-component agglomerations in six systems are significantly different. The only exception was the Health Watcher system. The reason is that all the design problems found in this system were related to the communication between two components.

Semantic Agglomerations are Consistent Indicators. Semantic agglomerations were, by far, the most consistent indicators of design problems. This finding is sustained by the fact that semantic agglomerations were strong indicator of design problems across all the systems. The 5th column (ORs) of Table 3 shows that the chance of each anomaly within a semantic agglomeration being related to a design problem is more than seven times (7,392) higher than each "detached" anomaly. This likelihood is higher than for anomalies within syntactic agglomerations. In addition, semantic agglomerations presented, proportionally, the highest correlation with design problems, when compared to all other syntactic agglomerations. For example, 91% of the semantic agglomerations in OODT were related to 253 instances of design problems.

Semantic agglomerations are also very often related to design problems even for the other systems where the absolute number of related design problems is much lower: (i) the percentage of true positives was very high (from 75% to 100%), and (ii) the percentage of false positives was very low (from 0% to 25%). Considering the data from all the systems, we observed a median of 82% semantic agglomerations related to design problems. The standard deviation of 10.74% is very low as well. That is, the percentage of semantic agglomerations related to design problems was very high in all the target systems. Regarding the absolute number of design problems (DP), the semantic agglomerations were related to a high number of design problems in all the systems.

Syntactic and Semantic Agglomerations are Complementary. In the last column, there are additional numbers between parentheses close to the numbers of design problems related to semantic agglomerations. These numbers represent the amount of design problems exclusively related to semantic agglomerations, i.e. those design problems that could not be located only with syntactic agglomerations. This is the case of the semantic agglomeration presented in Figure 2.

Approximately 50% of the design problems related to semantic agglomerations have no relationship to syntactic agglomerations in most of the systems. These design problems were often cases of *Scattered Concerns*, *Component Overload*, *Overused Interface* and *Delegating Abstraction*. This result is interesting in the sense that it suggests semantic agglomerations are useful for improving the location of a several design problems. These design problems are also much harder to be located by developers as there is no syntactic relationship amongst the anomalous program elements forming the agglomerations. However, existing work only focuses on characterizing interacting code anomalies based on their syntactic relationships [1, 2, 28, 34]. Moreover, these design problems cannot be simply detected through the sole use of concern location techniques. Several crosscutting concerns in the analyzed systems are not harmful, i.e. their implementations did not contain code anomalies and they were not actual sources of design problems.

Reducing False Positives of Syntactic Agglomerations. Even though syntactic agglomerations are often related to design problems, developers would still have to inspect and discard several irrelevant agglomerations. Table 4 shows that the number of false positives is higher than 60% in two systems. However, we observed that the combined use of semantic and syntactic agglomerations would significantly reduce the number of false positives yielded by the sole use of syntactic agglomerations. If we only consider syntactic agglomerations that intersect with semantic agglomerations, the number of false positives for syntactic agglomerations would be reduced to 21% or less in all the systems. There is an intersection between a syntactic agglomeration X and a semantic agglomeration Y when at least one of its anomalous elements take part of both X and Y. In summary, the use of syntactic and semantic agglomerations significantly enhances the location of design problems.

Full Automation of Semantic Agglomerations?. The computation of semantic agglomerations relies on mapping of systems concerns to identify design problems. Tables 3 and 4 show the results for semantic agglomerations computed with concern mappings produced by the systems' developers. However, it is important to check whether semantic agglomerations can also improve the location of design problems when input concern mappings are automatically generated. As mentioned in Section 3.5, we also relied on concern mappings provided by Mallet in order to generate a second set of semantic agglomerations. When compared to the developers' mapping, we notice Mallet generates much longer lists of concerns and concern mappings. As a consequence, the use of Mallet results in a higher number of semantic agglomerations, which was, in turn, related to more design problems than our original computation (Table 4). Semantic agglomerations generated with Mallet's output lead to an average increase of 39,58 % in the identification of design problems when compared with the semantic agglomeration generated with developers' mappings. The parameter configurations used for Mallet and all the detailed results are available in our supplementary material [29].

4.3 Agglomerations as Congenital Problems?

This section addresses the research question RQ2: "What proportion of design problems early manifest themselves as agglomerations?" Early manifestation means the design problems were present in the first versions of a system. Therefore, they are likely to represent "congenital" design problems, i.e. they were introduced at design time. In order to answer this question, we compute for all the systems: (i) the number of design problems in the first versions of our target systems, and (ii) the proportion of design problems related to agglomerations.

The analysis of all the initial versions reveals that a considerable number of design problems was introduced in the first version of each target system. As opposed to our expectation, all the initial versions presented a high correlation between agglomerations and design problems. We expected most of the design problems would be "evolutionary", i.e. they would be introduced by software maintainers as the systems evolved. However, the proportion of design problems related to agglomerations in early versions was approximately 75% for MM, S1, S2 and S4. For the other systems, the proportion was close to 65%. We also observed that a considerable proportion of design problems are exclusively related to either syntactic or semantic agglomerations already in the first version.

These results show that change history analysis [11, 39, 33, 40] would not be an effective solution to reveal many instances of design problems. Many design problems are congenital and they are already "born" as agglomerations. It would much harder to remove such design problems in later versions, when eventually the agglomeration's anomalous elements start to suffer co-changes through the later versions. A representative example of this case is the *Fat Interface* problem in the *Versioning* component in OODT (Section 2.2). It would be too cumbersome to restructure the *Versioner* interface in later releases of OODT. The number of client classes for this interface increased from one (in the first release) to more than fifteen (last release).

Comparing Early and Late Versions. We also explicitly compare the nature of design problems and agglomerations in "early" and "late" versions. The comparison of the different versions in two of the analyzed systems – MM and HW – serve to illustrate most of our findings. For instance, let us consider the versions 1 and 8 of the MM system. Version 1 of MM (0.8 KSLOC) is much smaller than version 8 (10 KSLOC). Therefore, the number of code anomalies and agglomerations is proportionally smaller in version 1. Even with fewer code elements, the first version already contained 23% of the agglomerations present in version 8. One of them, for example, is related to a class called *BaseController*. In the first version, this class was identified as being part of an intra-component agglomeration. That is, this anomalous class was related to another anomalous code element of the same component. This agglomeration was related to the *Component Overload* problem. In the subsequent versions, this agglomeration "expanded" to several code elements, including those located in other client components. More specifically, in version 8 we identified the same agglomeration became an inter-component agglomeration, involving classes that use the *BaseController* class. The inter-component agglomeration was related to emerging instances of *Fat Interface* and *Overused Interface* problems.

In the case of the HW system, we compared versions 1 and 10. In this case, version 1 (6 KSLOC) was only somewhat smaller than version 10 (8 KSLOC). However, as in the case of the MM system, we observed again that some agglomerations found in a late version had already started to form in the first version. The first version already contained 39% of syntactic and semantic agglomerations found in version 10. All these observations suggest that the agglomerations are effective to assisting the identification and removal of design problems in the early versions of a system can prevent the introduction of more severe problems in subsequent versions.

5. RELATED WORK

Do Individual Anomalies Suffice? Tufano et al. [37] investigated when and why code smells are introduced in the context of evolving programs. Kim et al. [17], Lozano et al. [20] and Olbrich et al. [30] investigated the effect of individual code anomalies throughout the system's evolution. These studies analyzed whether the number of code anomalies tend to increase over time, and how often they result in code changes. There is also a large body of work aiming at assessing the impact of individual occurrences of code anomalies on certain software quality attributes. For instance, Khomh et al. [15, 16] studied the relation between code anomalies and software change proneness. They concluded that anomalous classes tend to be more change-prone. In addition, corroborating with the work of Li and Shatnawi [19] and D'Ambros et al. [5], they also observe that classes infected by code smells tend to be more fault-prone than other classes [16]. Jürgens et al. [13] and Kapser et al. [14] found that one specific code anomaly adversely impacts on software maintenance. Sjoberg et al. [34] showed that single code anomalies were not related to maintenance effort. Therefore, some of the results of these studies are controversial regarding the usefulness of individual code anomalies as indicators of software maintenance effort. However, none of the aforementioned studies has investigated the relation between anomaly agglomerations and design problems.

Agglomerations and Design Problems. While many authors investigated the impact of individual code anomalies, a few studies have focused on the study of inter-related code anomalies. The study of Abbes et al. [1] brings up the notion of interacting code anomalies and studies its effects. They concluded that classes and methods identified as *God Classes* and *God Methods* in isolation had no effect on effort, but when appearing together, they led to a statistically significant increase in maintenance effort. Additionally, Yamashita and Moonen [42] observed that inter-smell relationships negatively affect a system's maintenance. Macia [2] cataloged a specific set of recurring patterns of inter-related code anomalies. However, none of these authors studied the impact of inter-related code anomalies on design problems. In addition, none of them characterize or explore semantic relationships between code anomalies.

6. THREATS TO VALIDITY

Construct Validity. The major risk here is related to possible errors in the identification of both design problems and code anomalies. As far as design problems are concerned, original developers reported the lists of design problems they found to be relevant in their systems. Therefore, these lists could be inaccurate and include irrelevant design problems. To mitigate this threat, we recruited developers, who had extensive knowledge of their system's design. They also had previous experience on design reviews and source code inspections. Moreover, we solicited from developers a brief explanation about the severity of the reported design problems. Whenever we could not understand the nature and relevance of a design problem, we asked for further clarification. We discarded the design problems that were not properly explained by developers. As far as code anomalies are concerned, we selected detection strategies and thresholds that presented satisfactory results in a previous study (Section 3.5). In any case, if these strategies resulted in false positives or false negatives, they would have a similar effect on the computation of all types of agglomerations.

Conclusion Validity. The main threat to the conclusion validity is the number of evaluated versions of each system. A study involving several versions of each system is always desired. Nevertheless, it was impracticable in our study due to the number of systems (7) and the limited availability that original developers and architects had to participate in the study. A higher number of versions would demand more time for developers to identify design problems. Therefore, we tried to mitigate this threat by selecting, for each different system, versions in a different lifecycle stage.

Internal and External Validity. Threats associated with internal and external validity concern the degree to which the findings can be generalized to the wider classes of subjects. In the experiment reported upon here, these threats to validity are somewhat mitigated by the fact that we used systems with different sizes (ranging from 8 to 129 KSLOC), with different purposes (academic, commercial and open-source), with different domains, that were implemented using different design styles and that suffered from a different set of design problems. Furthermore, the analyzed systems were developed by teams of different sizes and with different levels of software development skills.

7. CONCLUDING REMARKS

Recent empirical studies [23, 22, 24] suggest that individual code anomalies do not suffice to identify design problems. This finding represents a challenge for developers because they often need to spot design problems exclusively based on source code analysis, especially because of the lack of up-to-date design documents [23]. In order to decide whether and where a relevant design problem is prevailing in the program, programmers first need to know: (i) which are the code anomalies realizing the design problem, (ii) how these code anomalies are related to each other, and (iii) how these anomaly relationships are connected to the design problem. The gathering of all this scattered knowledge is time-consuming and error prone.

In this work, we analyzed to what extent code anomaly agglomerations can help developers to better locate design problems than individual analysis of code anomalies. We confirmed that code anomalies often "flock together" in order to embody a design problem. We studied the relation of design problems with both syntactic and semantic agglomerations in a longitudinal multi-case study involving 7 systems. Syntactic agglomerations are relevant indicators of design problems and help to discard an average of 600 irrelevant anomalies per system. We also found the combined use of syntactic and semantic agglomerations represents a more effective approach for locating design problems. Many congenital and evolutionary design problems were only related to semantic agglomerations. Moreover, a considerable proportion of design problems are congenital and manifest as agglomerations in the implementation. This result means that state-of-the-art solutions based on change history analysis are not adequate to promptly detect such problems. We plan to perform further investigations with the twofold goal of: (i) identifying efficient criteria for ranking code anomaly agglomerations in order to prioritize critical design problems, and (ii) conceiving a technique to synthesize relationships between anomaly agglomerations emerging across software project histories.

8. REFERENCES

[1] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program

comprehension. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 181–190, March 2011.

[2] I. M. Bertrán. *On the Detection of Architecturally-Relevant Code Anomalies in Software Systems*. PhD thesis, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil, 2013.

[3] V. Bloomfield. *Using R for Numerical Analysis in Science and Engineering*. Chapman & Hall/CRC The R Series. Taylor & Francis, 2014.

[4] J. Cornfield. A method of estimating comparative rates from clinical data; applications to cancer of the lung, breast, and cervix. *Journal of the National Cancer Institute*, 11(6):1269–1275, June 1951.

[5] M. D'Ambros, A. Bacchelli, and M. Lanza. On the impact of design flaws on software defects. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 23–31, July 2010.

[6] R. A. Fisher. On the interpretation of $\chi 2$ from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85(1):pp. 87–94, 1922.

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

[8] S. G. Ganesh, T. Sharma, and G. Suryanarayana. Towards a principle-based classification of structural design smells. *Journal of Object Technology*, pages 1–29, 2013.

[9] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 486–496, Nov 2013.

[10] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 255–258, March 2009.

[11] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel. Using concept analysis to detect co-change patterns. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 83–89, New York, NY, USA, 2007. ACM.

[12] M. W. Godfrey and E. H. S. Lee. Secrets from the monster: Extracting Mozilla's software architecture. In *Proc. of the Second Intl. Symposium on Constructing Software Engineering Tools (CoSET-00)*, June 2000.

[13] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 485–495, May 2009.

[14] C. J. Kapser and M. W. Godfrey. Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, Dec. 2008.

[15] F. Khomh, M. Di Penta, and Y. GuÃľhÃľneuc. An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 75–84, Oct 2009.

[16] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

[18] M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[19] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, 2007.

[20] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 227–236, Sept 2008.

[21] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Manage. Sci.*, 52(7):1015–1030, July 2006.

[22] 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*, pages 662–665, Sept 2012.

[23] 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*, pages 277–286, March 2012.

[24] 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*, AOSD '12, pages 167–178, New York, NY, USA, 2012. ACM.

[25] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[26] C. A. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes. A software architecture-based framework for highly distributed and data intensive scientific applications. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 721–730, New York, NY, USA, 2006. ACM.

[27] A. K. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[28] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, Jan 2010.

[29] W. Oizumi. Opus research group: Supplementary

material, 2015. Availabel at
`http://www.inf.puc-rio.br/~woizumi/icse2016`.

[30] S. Olbrich, D. Cruzes, and D. I. Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010.

[31] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct. 1992.

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

[33] R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 891–900, Piscataway, NJ, USA, 2013. IEEE Press.

[34] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba. Quantifying the effect of code smells on maintenance effort. *Software Engineering, IEEE Transactions on*, 39(8):1144–1156, Aug 2013.

[35] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with aspectj. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 174–190, New York, NY, USA, 2002. ACM.

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

[37] 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*, ICSE '15, New York, NY, USA, 2015. ACM.

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

[39] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 411–420, New York, NY, USA, 2011. ACM.

[40] L. Xiao, Y. Cai, and R. Kazman. Titan: A toolset that connects software architecture with quality analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 763–766, New York, NY, USA, 2014. ACM.

[41] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 306–315, Sept 2012.

[42] A. Yamashita and L. Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 682–691, May 2013.

[43] T. J. Young. Using aspectj to build a software product line for mobile devices. Master's thesis, University of British Columbia, 2015.