Pontifícia Universidade Católica
do Rio de Janeiro

Willian Nalepa Oizumi

Synthesis of Code Anomalies: Revealing Design Problems in the Source Code

Dissertação de Mestrado

Dissertation presented to the Programa de Pós Graduação em Informática of the Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
September 2015

Pontifícia Universidade Católica do Rio de Janeiro

Willian Nalepa Oizumi

Synthesis of Code Anomalies: Revealing Design Problems in the Source Code

Dissertation presented to the Programa de Pós Graduação em Informática of the Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Mestre em Informática.

Rio de Janeiro, September 02nd, 2015

Willian Nalepa Oizumi

He received his Bachelor degree in Informatics from Universidade Estadual de Maringa, Brazil in 2012.

Bibliographic data

Oizumi, Willian Nalepa

Synthesis of Code Anomalies: Revealing Design Problems in the Source Code / Willian Nalepa Oizumi ; advisor: Alessandro Fabricio Garcia. — 2015.
103 f. : il. ; 30 cm

Dissertação (Mestrado em Informática)-Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015.
Inclui bibliografia

1. Informática – Teses. 2. Anomalia de Código. 3. Problema de Projeto. 4. Síntese. 5. Código Fonte. I. Garcia, Alessandro Fabricio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

Firstly, I would like to thank my beloved wife, Julia Antonia Dalbelo do Prado Oizumi, who supported me during the realization of this work. There is no words to describe her love, patience and kindness.

My deepest gratitude goes to my whole family, who always supported me. A special thank goes to my aunt, Leticia Oizumi, who was responsible for raising and educating me. Without her I would not have come this far.

I would like to thank my advisor, Alessandro Garcia, for giving me the opportunity to work with him. His patience, energy and good will are incredible. He contributed significantly to my professional growth, providing the means for me to exceed my limits.

I also thank my colleagues from the OPUS Research Group and from the Software Engineering Laboratory. In particular, I would like to thank Leonardo Sousa, for all your help and fellowship.

I thank my friends Jorge Rodrigo, Jackes Martins, Jean Valera, Renato Brito, Renato Galli, Renan Galli and Carlos Crivello.

I thank all the professors from PUC-Rio for their contribution to my education. My appreciation also goes to my professors from UEM. In particular, I would like to thank Thelma Colanzi.

Finally, I thank CAPES, FAPERJ and PUC-Rio for financial aid, without which this work would not have been possible.

Abstract

Oizumi, Willian Nalepa; Garcia, Alessandro Fabricio (advisor). **Synthesis of Code Anomalies: Revealing Design Problems in the Source Code**. Rio de Janeiro, 2015. 103p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Design problems affect almost all software projects and make their maintenance expensive and impeditive. As design documents are rarely available, programmers often need to identify design problems from the source code. However, the identification of design problems is not a trivial task for several reasons. For instance, the reification of a design problem tends to be scattered through several anomalous code elements in the implementation. Unfortunately, previous work has wrongly assumed that each single code anomaly – popularly known as code smell – can be used as an accurate indicator of a design problem. There is growing empirical evidence showing that several types of design problems are often related to a set of inter-related code anomalies, the so-called code-anomaly agglomerations, rather than individual anomalies only. In this context, this dissertation proposes a new technique for the synthesis of code-anomaly agglomerations. The technique is intended to: (i) search for varied forms of agglomeration in a program, and (ii) summarize different types of information about each agglomeration. The evaluation of the synthesis technique was based on the analysis of several industry-strength software projects and a controlled experiment with professional programmers. Both studies suggest the use of the synthesis technique helped programmers to identify more relevant design problems than the use of conventional techniques.

Keywords

Code Anomaly; Design Problem; Synthesis; Source Code;

Resumo

Oizumi, Willian Nalepa; Garcia, Alessandro Fabricio. **Síntese de Anomalias de Código: Revelando Problemas de Projeto no Código Fonte**. Rio de Janeiro, 2015. 103p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Problemas de projeto afetam quase todo sistema de software, fazendo com que a sua manutenção seja cara e impeditiva. Como documentos de projeto raramente estão disponíveis, desenvolvedores frequentemente precisam identificar problemas de projeto a partir do código fonte. Entretanto, a identificação de problemas de projeto não é uma tarefa trivial por diversas razões. Por exemplo, a materialização de problemas de projeto tende a ser espalhada por diversos elementos de código anômalos na implementação. Infelizmente, trabalhos prévios assumiram erroneamente que cada anomalia de código individual – popularmente conhecida como *code smell* – pode ser usada como um indicador preciso de problema de projeto. Porém, evidências empíricas recentes mostram que diversos tipos de problemas de projeto são frequentemente relacionados a um conjunto de anomalias de código inter-relacionadas, conhecidas como aglomerações de anomalias de código. Neste contexto, esta dissertação propõe uma nova técnica para a síntese de aglomerações de anomalias de código. A técnica tem como objetivo: (i) buscar formas variadas de aglomeração em um programa, e (ii) sumarizar diferentes tipos de informação sobre cada aglomeração. A avaliação da técnica de síntese baseou-se na análise de diversos projetos de software da indústria e em um experimento controlado com desenvolvedores profissionais. Ambos estudos sugerem que o uso da técnica de síntese ajudou desenvolvedores a identificar problemas de projeto mais relevantes do que o uso de técnicas convencionais.

Palavras-chave

Anomalia de Código; Problema de Projeto; Síntese; Código Fonte;

Contents

List of Figures

1
Introduction

A design problem (or a design smell) represents the realization of either: (i) unintended design decisions, which violate the original, intended design of a system, or (ii) violations of well-known design principles (Garcia, Popescu, Edwards & Medvidovic 2009)(Perry & Wolf 1992). Unwanted module dependencies (Perry & Wolf 1992) are a type of design problem falling in the first category. Fat interface (Garcia et al. 2009) is an example of a design problem in the second category. The maintainability and longevity of a system depends directly on its design (MacCormack, Rusnak & Baldwin 2006)(van Gurp & Bosch 2002). Therefore, design problems must be properly identified and removed. However, the identification and understanding of design problems are not trivial tasks, as they often need to rely on source code analysis. Design documentation is rarely available or up-to-date (Macia, Arcoverde, Garcia, Chavez & Staa 2012*b*).

For a long time, some authors simply assumed (Fowler 1999)(Hochstein & Lindvall 2005) most design problems can be observed in a system's implementation through the detection of single code anomalies. However, recent studies showed that this assumption is too simplistic (Macia et al. 2012*b*)(Oizumi, Garcia, Colanzi, Ferreira & Staa 2014*a*). In fact, each design problem, in general, is realized by various anomalous elements scattered in the implementation (Macia et al. 2012*b*). In addition, a high proportion of individual code anomalies may not affect the system's design.

Therefore, the identification of each design problem in a program are far from being trivial. 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, (iii) how these relationships are connected to the design problem, and (iv) what are other code elements potentially being also affected by the design problem. The gathering of all this scattered knowledge is time-consuming and error prone. Ideally, this information should be summarized and provided for programmers, so that they could diagnose design problems in the implementation (Oizumi, Garcia, Sousa, Albuquerque & Cedrim 2014*b*).

However, the summarization of all relevant information for reasoning about design problems in the implementation remains a challenge in software engineering. The reason is that there is still limited knowledge about the relationship of design problems and code anomalies. Therefore, there is a need for conceiving a technique that synthesizes code anomalies. The synthesis should search for groups of inter-related code anomalies using information extracted from different system artifacts. This technique is required to help programmers in diagnosing and characterizing design problems in their source code. Nevertheless, to the extent of our knowledge, there is no technique for the systematic synthesis of code anomalies. State-of-art techniques (Emden & Moonen 2002)(Ratzinger, Fischer & Gall 2005)(Wong, Cai, Kim & Dalton 2011)(Marinescu 2004) do not even reveal any form of relationship between code anomalies. A previous work has tried to partially classify a subset of possible relationships between code anomalies (Macia et al. 2012*b*)(Macia, Arcoverde, Cirilo, Garcia & Staa 2012*a*). However, the proposed technique is not intended to explicitly support the synthetization of all relevant information for the identification and diagnosis of design problems. For instance, it provides very limited information about the relationships and surrounding context of code anomalies.

1.1 Motivating Example and Problem

The relationship between a design problem and its counterpart code anomalies is often complex. Prior work (Macia, Garcia, Popescu, Garcia, Medvidovic & Staa 2012*c*)(Macia et al. 2012*b*)(Macia 2013) has shown that each design problem is often realized by various code anomalies scattered in the program. These anomalies interact across the code structure in order to reify a single design problem in the program. Therefore, the detection of individual code anomalies is only a first simple step to understand major design problems. Many of these single anomalies are not related and are not meaningful to any design problem. Without full knowledge about the relationships between code anomalies, programmers cannot decide whether and how anomalies represent a design problem. It also becomes hard to decide how to refactor out the design problem from the program. Therefore, these tasks can only be realized if programmers synthesize code anomalies into coherent groups of code anomalies.

The cohesion of a group of code anomalies is determined by the relation that exists between them. This relation between anomalies can assume different forms. For instance, code anomalies can be related through structural relationships in the program, such as method calls and inheritance relations. Anomalies can also be related through relationships closer to the software design. For instance, the anomalies can be located in the implementation of the same design component.

A coherent group of code anomalies is called *agglomeration of code anomalies* – or just *agglomeration* for brevity. As already mentioned, the anomalies that compose an agglomeration may be related to each other by different forms of relationship. For example, the code anomaly instances in Figure 1.1 (three occurrences of Feature Envy) occur in different code elements, but in the same hierarchy. That is, all of them implement the same interface (*Versioner*), thereby contributing to form an anomalous hierarchical structure. Given this relation, the three code anomalies can be grouped into an hierarchical agglomeration. This type of agglomeration is useful to reveal design problems that involve classes and interfaces related through hierarchical relationships (i.e., inheritance or interface implementation). An hierarchy is typically designed to segregate functionalities using polymorphism, which usually leads to a better modularization and reuse of functionalities. However, a badly designed hierarchy may cause the inverse effect in the design: reduce modularization and reuse of functionalities.

In the given example, the hierarchical relationships connecting the individual anomalies seem to be easy to spot. However, this is not the case for most agglomerations. The programmer might not be able to notice these anomaly relationships if he/she just browses either the source code or the design. Moreover, the analysis of code anomalies in isolation is not efficient, as isolated code anomalies are weakly related to design problems (Macia 2013).

Therefore, there is a need for a *synthesis technique*. Such a technique could reduce the effort required from programmers to identify design problems. To achieve this objective, two main synthesis steps should be performed: (1) search for instances of different forms of agglomeration, and (2) summarize relevant information about each agglomeration instance.

A synthesis technique should use hierarchical relationships to group the anomalies in the *Versioner* hierarchy (Figure 1.1). Moreover, a synthesis technique should help programmers to understand the context of each agglomeration. For example, the surrounding context of the agglomeration in Figure 1.1 is represented by code elements coupled to the hierarchy. In other words, they represent code elements that are clients or servers of anomalous classes in the hierarchy. Even though they are not anomaly sources, such clients and servers are also being directly affected by the design problem. Therefore, programmers also need to reason about them in order to determine the extent

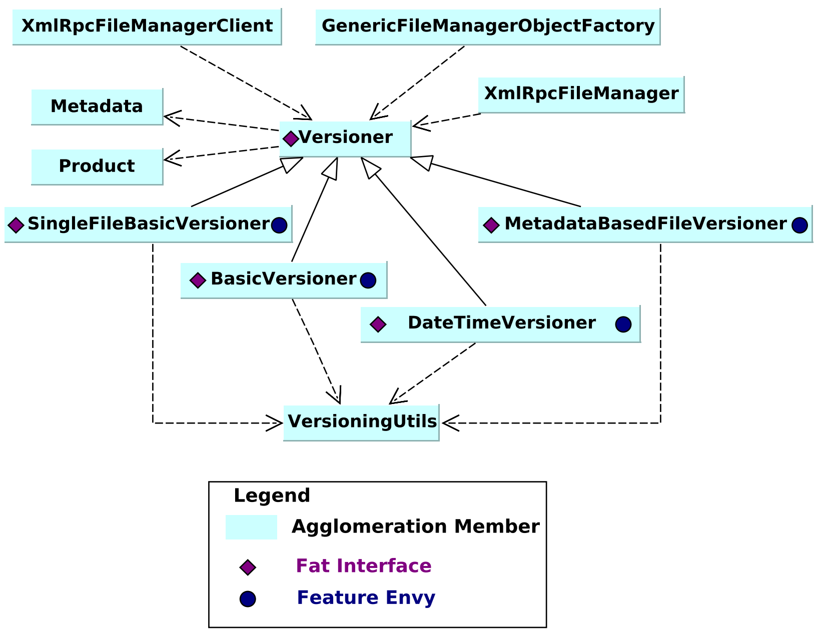


Figure 1.1: Hierarchical agglomeration in the OODT system.

of the design problem in the program. A synthesis technique should explore this information by revealing the surrounding context of each agglomeration. Moreover, other relevant information should be provided by the synthesis technique. For instance, information about each agglomeration's evolution along different versions of the system. This information would help to identify critical or irrelevant design problems, by revealing, for example, design problems that are "expanding" or "shrinking" along different versions of the system.

1.2 Limitations of Existing Studies

Different techniques for code anomaly detection have been proposed and studied (Emden & Moonen 2002)(Lanza & Marinescu 2006)(Wong et al. 2011)(Mara, Honorato, Dantas, Garcia & Lucena 2011). However, to the extent of our knowledge, none of them effectively help programmers to identify design problems. They only present limited information about each code anomaly, such as anomaly type and name of the main affected method or class. Even worse, they are unable to reveal agglomerations of code anomalies that may be realizing design problems in the source code. As a result, using these techniques, programmers have to put a considerable effort in understanding the context of each anomaly and in locating those that altogether contribute to design problems. The synthesis technique proposed in this dissertation overcomes the aforementioned limitations by (i) revealing agglomerations of code anomalies composed by different forms, and (ii) providing a rich set of information about each agglomeration of code anomalies.

A few studies have recently investigated the impact of preliminary forms of agglomeration in the maintainability of systems (Moha, Gueheneuc, Duchien & Meur 2010)(Abbes, Khomh, Gueheneuc & Antoniol 2011)(Sjobert, Yamashita, Anda, Mockus & Dyba 2013)(Yamashita & Moonen 2013). Their studies bring up the notion that code anomalies are more harmful to the system's maintainability when they are related to other anomalies (i.e., when they are agglomerated). However, they investigate only specific types of code anomalies. Moreover, none of them investigated when agglomerations actually represent (or not) design problems. More importantly, none of them proposed a technique that searches for agglomerations and summarizes relevant information about each agglomeration. Hence, this dissertation overcomes such limitations by (i) investigating agglomerations composed by a varied set of anomaly's types, (ii) investigating the relation between agglomerations of code anomalies and design problems, and (iii) proposing a new technique for the synthesis of code anomalies.

Macia (Macia 2013) studied nine forms of relationship between code anomalies. She observed that groups of code anomalies composed by these relationship forms are better indicators of design problems than individual code anomalies (Macia 2013). The results of her study revealed a statistically significant relationship between some of these inter-related code anomalies and design problems. However, Macia did not investigate how each form of relationship is related to design problems. In addition, Macia did not analyze the impact of inter-related code anomalies in the evolution of systems. We overcome the limitations of her study by investigating (i) the relation of different forms of agglomeration with design problems, and (ii) the impact of agglomerations across the evolution of systems.

To conduct her study, Macia proposed and implemented a technique to detect the nine types of inter-related code anomalies (Macia 2013). Such technique overcomes limitations of conventional techniques for code anomaly detection, since it exploits relationships between code anomalies. However, her technique still has some noteworthy limitations. First and foremost, it provides very little information about each group of code anomalies. To identify and understand design problems using Macia's technique, programmers still need to manually grasp from the source code information about the surrounding context of each agglomeration. Second, it does not provide specific visual aids for each agglomeration topology. Finally, it does not allow programmers to specify custom forms of relationship (i.e., the relationships used to group inter-related code anomalies). Therefore, her technique does not meet the minimum requirements for a full-fledged synthesis technique. It is not able to summarize information about anomalies that a programmer would use to identify design problems. This dissertation addresses this gap in the literature, proposing and evaluating a synthesis technique. This technique searches for different forms of agglomerations, summarizes relevant information about each agglomeration and allow programmers to specify custom forms of agglomeration.

1.3 Objective

There is growing evidence that code anomalies may be helpful to identify and understand design problems (Macia et al. 2012*c*). Therefore, programmers and researches should be equipped with proper means to reason about code anomalies and their relationships. In this context, the main objective of this dissertation is to propose and evaluate a new technique to synthesize code anomalies. It is expected from the proposed technique to overcome key limitations of state-of-art techniques (Section 1.2). To achieve this objective, four goals were established. These goals provided guidance to the planning and execution of this dissertation's research. Next, details about each goal are provided:

1. **Propose a Synthesis Technique.** As explained in the previous section, current techniques are unable to effectively aid programmers in the identification of design problems. Therefore, the first goal of this dissertation is to propose a new technique for the identification of design problems using code anomalies. This technique is called *synthesis of code anomalies.* The proposal of a synthesis technique encompasses the following steps: (1) gather the minimum set of requirements that a synthesis technique must satisfy, and (2) develop tool assistance for programmers to use the proposed technique.

2. **Study the Relation of Design Problems and Agglomerations.** This goal consists of studying agglomerations to understand when and how they represent (or not) design problems. This is essential to ensure that the synthesis technique provides useful information to the identification of design problems.

3. **Conduct a Controlled Experiment.** After collecting evidence on the relevance of agglomerations, it is essential to gather feedback about the use of a synthesis technique. Therefore, this goal consists in evaluating the proposed technique in the context of a controlled experiment. This

experiment compares the synthesis technique with a state-of-art technique, which we name as *conventional technique*. With this experiment it is possible to answer the following questions: (1) which was the most useful technique, (2) how the proposed technique could be improved, and (3) which are the most and least useful forms of agglomerations from the point of view of programmers.

1.4 Steps Towards the Synthesis of Code Anomalies

In order to achieve the objective of this dissertation, complementary studies were conducted. In order to achieve the first goal (i.e., "Propose a Synthesis Technique"), the first step consisted in identifying the requirements for devising a synthesis technique, i.e., the requirements for a technique that searches for agglomerations and extracts relevant information about them. In this dissertation, we analyzed characteristics of agglomerations and design problems in the context of several systems with different characteristics. This varied set of target systems allowed us to assume the requirements can be generalized for most common types of software system. Using these requirements, we proposed a synthesis technique (Chapter 3). In order to provide full support for our synthesis technique, a tool was designed and implemented. The implemented tool is aimed at supporting the identification of design problems in Java systems (Oracle 2015). This tool was implemented as a plug-in for the Eclipse IDE (Ecl 2015).Besides detecting different forms of agglomeration, the tool provides useful information about each agglomeration.

Achieving the second goal – "Study the Relation of Design Problems and Agglomerations", we conducted an exploratory study in the context of seven systems (Chapter 4). In this study, using different perspectives, we analyzed when agglomerations represent design problems. This study provided positive evidence about the relation of agglomerations and design problems. Moreover, this study revealed that anomalies related through concern-based relationships are more likely to represent design problems than those related through only syntactic relationships.

Finally the last goal ("Conduct a Controlled Experiment") was achieved by conducting a controlled experiment with experienced programmers (Chapter 5). This experiment allowed us to see from the programmers perspective whether the proposed technique is better (or not) than state-of-art techniques. In the experiment, two different components were individually evaluated by programmers. For the analysis of each component, subjects used either the *synthesis* or *conventional* technique. While the first is proposed in this dissertation, the second is the technique used by most state-of-art tools for code anomaly detection. In order to promote a fair comparison, both techniques were used by all subjects. Overall, this evaluation revealed the use of a synthesis technique leads to less false positives. After the evaluation, subjects also provided feedback on how the synthesis technique can be improved. For example, they said the technique should provide a prioritization algorithm to help them in deciding which agglomerations should be analyzed first.

1.5 Dissertation Structure

The remainder of this dissertation is organized as follow. Chapter 2 introduces background and outlines related work on inter-related code anomalies and design problems. Chapter 3 presents a detailed description about our technique for the synthesis of code anomalies. Chapter 4 reports results for the evaluations we conducted to understand and characterize code-anomaly agglomerations. Chapter 5 describes results for the controlled experiment conducted to evaluate the synthesis technique. Finally, Chapter 6 summarizes the main contributions of this dissertation and presents our future work.

2
Background and Related Work

This chapter contains the background and related work of this dissertation. Section 2.1 outlines the aspects of software design considered in this dissertation. Next, Section 2.2 presents the concept of software design problem as well as provides illustrative examples. Finally, a brief overview of the literature about code anomalies is presented in Section 2.3.

2.1
Software Design

Software design is concerned with various activities governing the conception of the solution required to address the software stakeholders' concerns (Freeman & David 2004)(Booch 2004). In other words, software design is the process of defining the organization of a system's structure so that the resulting functionality satisfies the stakeholders' concerns (Freeman & David 2004)(Booch 2004). Software design includes all the decisions before actually programming. However, in certain software projects, decisions are often made in a way that is intertwined with programming. Software design can be classified in two main stages (Booch 2004): (i) early software design (or software architecture design) – this stage is focused on defining the overall organization of a software system into components (or sub-systems), interfaces and their relationships, and (ii) detailed design – this stage is focused on achieving more specific decisions governing the design of each software component.

In this dissertation, we mainly focus on early software design as the most important design decisions are made in this stage. Thus, in this work, software design represents the overall organization of the system into design components, interfaces and relationships among them (Bass, Clements & Kazman 2003)(Gorton 2006). Design components are elements which address one or more stakeholders' concerns (Taylor, Medvidovic & Dashofy 2009). Each component has one or more provided interfaces. Services provided by a component are exposed in the component's interfaces. Interactions between components are defined by *connectors*. A connector is a design element that represents interactions among components and the rules that governs those interactions (Mehta, Medvidovic & Phadke 2000). The following types of interaction are provided by connectors: communication, coordination, conversion and facilitation (Mehta et al. 2000).

2.2 Software Design Problem

A design problem occurs due to the addition of unintended design decisions that either violate (1) the original, intended design of a system or (2) general software modularity principles (Perry & Wolf 1992). A design problem can be introduced when certain decisions are made before programming. In addition, when software changes are made, the system's design can degrade due to new design problems being introduced (Hochstein & Lindvall 2005). We refer to an individual manifestation of a design problem as a design problem instance.

To illustrate the violation of an intended design, consider the Health Watcher design in Figure 2.1. For each figure in this chapter, we rely on a UML-like notation (Booch, Rumbaugh & Jacobson 2005). Elements that are not part of UML are explained in each figure's legend. We partially represent the system implementation in two different views. At the top of Figure 2.1, we represent the components of Health Watcher and interactions between them. Solid arrows represent expected relationships between components; dotted arrows represent unexpected relationships. We represent the concerns of the *Business* component by characters within circles. At the bottom of Figure 2.1, we represent some (but not all) classes of the *Business* component that have instances of code anomalies (initials within circles). In this example, we have three instances of design problems: (1) a dependency from *Data* to *Business*, (2) a dependency from *Data* to *GUI*, and (3) a dependency from *Business* to *GUI*. These dependencies were not part of the intended design of Health Watcher. However, they were implemented in the actual design reified in the source code, thus, violating the intended design.

Design problems are not restricted to violations of intended design rules. Design problems also occur when elements of a design violate modularity principles. Table 2.1 summarizes all the design problems considered in this dissertation. Detailed information about design problems can be found in (Martin 2002) and (Garcia et al. 2009). We decided to focus on this catalog of design problems as they cover violations to a wide range of modularity principles, such as abstraction, information hiding and separations of concerns. In addition, these design problems capture all the problems we have found in the systems used in the empirical studies of this dissertation (Chapters 4 and 5).

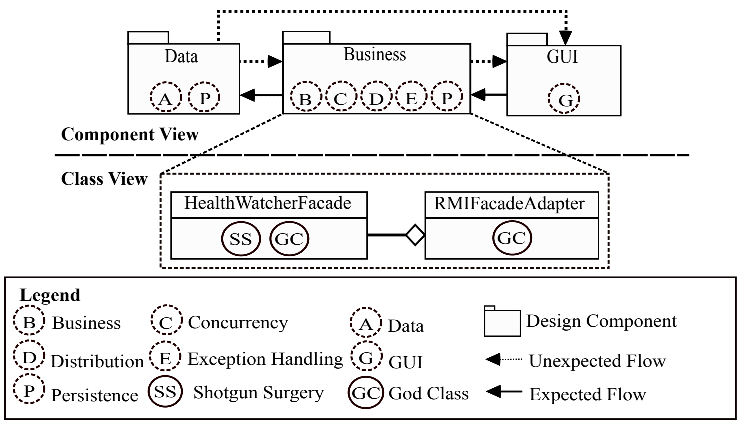
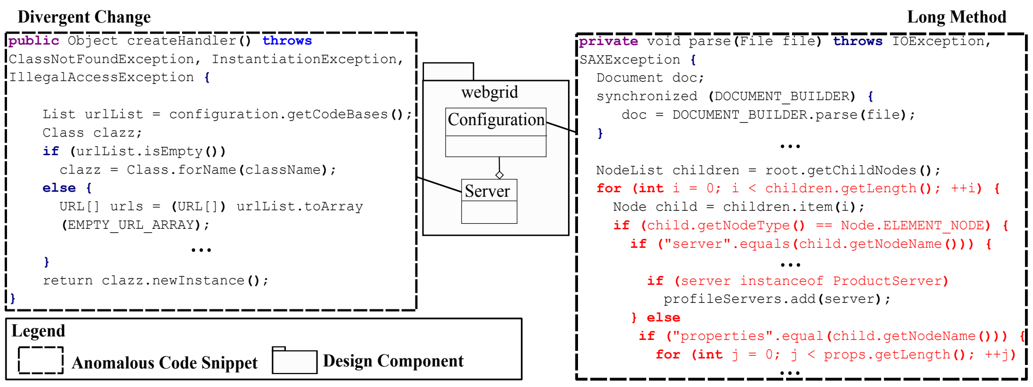


Figure 2.1: Partial View of Health Watcher's Design



```
public Object createHandler() throws
ClassNotFoundException, InstantiationException,
IllegalAccessException {

    List urlList = configuration.getCodeBases();
    Class clazz;
    if (urlList.isEmpty())
      clazz = Class.forName(className);
    else {
      URL[] urls = (URL[]) urlList.toArray
      (EMPTY_URL_ARRAY);
                        ...
    }
    return clazz.newInstance();
}
```

```
private void parse(File file) throws IOException,
SAXException {
  Document doc;
  synchronized (DOCUMENT_BUILDER) {
     doc = DOCUMENT_BUILDER.parse(file);
  }
                        ...

  NodeList children = root.getChildNodes();
  for (int i = 0; i < children.getLength(); ++i) {
    Node child = children.item(i);
    if (child.getNodeType() == Node.ELEMENT_NODE) {
      if ("server".equals(child.getNodeName())) {
                        ...
        if (server instanceof ProductServer)
          profileServers.add(server);
      } else
       if ("properties".equal(child.getNodeName())) {
         for (int j = 0; j < props.getLength(); ++j)
                        ...
```

Figure 2.2: Concern Mixing in the Webgrid Component

As an example, consider the *webgrid* component from the Apache OODT (Object Oriented Data Technology) system in Figure 2.2. The *webgrid* component (1) retrieves resources (e.g., scientific datasets, images, and documents) in platform-neutral formats and (2) describes and discovers resources using extensible metadata. This component uses HTTP to transmit resources. The *Configuration* class (which is encompassed by *webgrid*) holds the complete runtime configuration of resource servers, metadata servers, properties, and other settings for the *webgrid* component. The *webgrid* component is infected by an instance of the Concern Mixing problem, which is caused by including the implementation of interaction-related concerns along with system-specific concerns. For example, besides implementing its main concern, the *Configuration* class also performs conversion services (from and to XML files) through the *parse* method. These interaction services are best delegated to a specific connector detaching the conversion services from the main functionality of *Configuration*. Like the *Configuration* class, other classes in *webgrid* also contribute to the Concern Mixing problem.

Table 2.1: Design Problems

Name	Description
Fat Interface (FI)	Interface of a design component that offers only a single, general entry-point, but provides two or more services.
Unwanted Dependency (UD)	Dependency that violates a intended design rule.
Concern Mixing (CM)	Component that mixes extensive interaction-related concerns (e.g., conversion) with a high-level concern.
Cyclic Dependency (CD)	Two or more design components that directly or indirectly depend on each other.
Multiple Interaction Types (MI)	Two interaction types that are used to link the same pair of components.
Scattered Concern (SC)	Multiple components that are responsible for realizing the same design concern.
Overused Interface (OI)	Interface that is overloaded with many clients accessing it. That is, an interface with “too many clients”.
Unused Interface (UI)	Interface that is never used by external components.

2.3 Code Anomalies

A code anomaly, popularly known as a “code smell”, is a symptom of a bad decision observed in a program’s low-level structure. Examples of code anomalies are Long Method, God Class, Shotgun Surgery, Feature Envy, Divergent Change and Data Class (Fowler 1999)(Lanza & Marinescu 2006). There are several techniques and tools in the literature that aim at detecting code anomalies (Emden & Moonen 2002)(Ratzinger et al. 2005)(Wong et al. 2011)(Marinescu 2004). Table 2.2 presents the complete list of code anomalies considered in this work.

Detection Strategies (Marinescu 2004) is the most widely-used technique for the detection of code anomalies. This technique exploits information that is extracted from the source code structure, relying on the combination of static code metrics and thresholds into logical expressions. Each detection strategy is a heuristic that identifies code elements that possibly suffer from a particular code anomaly (Marinescu 2004). Therefore, as we will describe latter, the synthesis technique (Chapter 3) uses detection strategies to extract code anomalies from a program.

Code anomalies have been largely studied by several researchers and practitioners. Therefore, the remainder of this section is dedicated to present an overview of the code anomaly’s literature. Section 2.3.1 outlines studies on the relation of code anomalies and design problems. Finally, Section 2.3.2 presents

Table 2.2: Code anomalies

Type	Description
Brain Class/God Class	Long and complex class that centralizes the intelligence of the system
Brain Method	Long and complex method that centralizes the intelligence of a class
Data Class	Class that contains data but not behavior related to the data
Disperse Coupling	The case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes
Feature Envy	Method that calls more methods of a single external class than the internal methods of its own inner class
Intensive Coupling	When a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes
Refused Parent Bequest	Subclass that does not use the protected methods of its superclass
Shotgun Surgery	This anomaly is evident when you must change lots of pieces of code in different places simply to add a new or extended piece of behavior

a brief analysis of studies about visualization techniques for anomaly detection. Even though our research does not focus on software visualization, we also discuss why existing visualization techniques are not sufficient to properly support identification of design problems.

2.3.1 Studies on the Relation of Code Anomalies and Design Problems

Design problems have caused the discontinuation or reengineering of several software projects (Hochstein & Lindvall 2005). As previously mentioned, programmers frequently need to detect such problems in the source code due to the lack of formal design documentation. In other words, it is expected that a wide range of design problems are reflected in a system's implementation through code anomalies (Fowler 1999)(Hochstein & Lindvall 2005). A method infected by Long Method (Fowler 1999), for example, is highly complex and contains excessive functionality. As an example, consider the code snippet of the method *parse* on the right side of Figure 2.2. This method parses a serialized configuration file and stores the result to an instance of the *Configuration* class. The *parse* method is complex and overloaded in terms of responsibilities, as manifested as nested conditions and loops that handle different details of the configuration file. The method's complexity reduces its understandability and maintainability. To reduce this complexity, the method can be refactored into smaller methods. However, this refactoring in isolation may not be enough to remove the Concern Mixing problem from *webgrid*. As already mentioned, a group of inter-related classes of this component contributes to the same problem. Therefore, the full removal of Concern Mixing depends on the refactoring of multiple classes.

The impact of code anomalies has been largely studied. Khomh *et al.* (Khomh, Penta & Gueheneuc 2009), Kim *et al.* (Kim, Sazawal, Notkin & Murphy 2005), Lozano *et al.* (Lozano & Wermelinger 2008) and Olbrich *et al.* (Olbrich, Cruzes & Sjoberg 2010) investigated the impact of code anomalies throughout the system's evolution. Specifically, the authors analyzed whether the number of code anomalies tended to increase over time, and how often they resulted in code changes. D'Ambros *et al.* (D'Ambros, Bacchelli & Lanza 2010) observed that code anomalies are often related to software defects. Sjoberg *et al.* (Sjobert et al. 2013) showed that single code anomalies were not related to maintenance effort. Macia *et al.* (Macia et al. 2012*b*) analyzed the relevance of code anomalies to identify design problems. This research revealed that none of the studied code anomalies was consistently a strong indicator of design problems. The results also revealed that a higher proportion of individual code anomalies did not impact the system design.

Moha *et al.* (Moha et al. 2010) documented relationships among code anomalies that are recurrently related to four design anomalies. According to them (Moha et al. 2010), relationships among Long Method and God Classes are usually indicators of Spaghetti Code design anomaly. The study by Abbes *et al.* (Abbes et al. 2011) brings up the notion of interaction effects across code anomalies. 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. Yamashita and Moonen (Yamashita & Moonen 2013) observed that inter-related code anomalies negatively affect systems maintenance. None of the aforementioned authors investigated the relation of design problems and code-anomaly agglomerations. In this context, Macia (Macia 2013) observed that a specific set of nine inter-related code anomalies are better indicators of design problems than individual code anomalies. The results of her study revealed a statistically-significant relationship between some of these inter-related anomalies and design problems.

None of the aforementioned studies investigated when agglomerations actually represent (or not) design problems. More importantly, none of them proposed a technique that searches for agglomerations and summarizes relevant information about each agglomeration. Hence, this dissertation overcomes such limitations by (i) investigating agglomerations composed by a varied set of anomaly's types, (ii) investigating the relation of different forms of agglomeration with design problems, (iii) evaluating the impact of agglomerations across the evolution of systems, and (iv) proposing a new technique for the synthesis of code anomalies.

2.3.2 Visualization Techniques for Anomalies Detection

Some researchers have proposed the use of software visualization techniques specifically for the context of code anomaly detection (Murphy-Hill & Black 2010)(Wettel & Lanza 2008)(Carneiro, Silva, Mara, Figueiredo, Sant'Anna, Garcia & Mendonça 2010)(D'Ambros et al. 2010). Murphy-Hill and Black (Murphy-Hill & Black 2010) proposed an anomaly detector, called Stench Blossom. Their technique is based on an interactive visualization environment. It provides a programmer with a quick, high-level overview of the anomalies in the code. According to the programmer's needs, Stench Blossom provides additional information to help him in understanding the sources of those code anomalies. Wettel and Lanza (Wettel & Lanza 2008) proposed disharmony maps, a visualization technique to locate code anomalies in large systems. A disharmony map displays the whole system using a 3D visualization approach based on a city metaphor. Besides showing classes and methods of the system, this map also shows results from conventional anomaly detection strategies. The work of Carneiro *et al.* (Carneiro et al. 2010) presents a multi-view approach that enriches four categories of source code views in order to support the visualization of stakeholders' concerns. The enriched views enable programmers to visualize how certain high-level concerns (e.g., *persistence*, *error handling*, *caching*, *security* and the like) are realized in the program elements. The proposed approach is not intended to support the detection of groups or instances of code anomalies. Instead, their approach consists in combining different forms of concern's view to spot anomalous classes and methods. It does not provide specific feedback about the type of the code anomaly infecting classes and methods.

All techniques described above are somehow helpful to the identification of design problems. However, none of them directly explores relationships to reveal and synthesize code-anomaly agglomerations. The programmer still needs to visually try to recognize which visual metaphors and hints might represent individual anomalies or agglomerations. In addition, existing program visualization techniques do not explicitly summarize all the information about elements composing an agglomeration. Hence, they need underlying algorithms to perform this activity. In this context, the synthesis technique proposed in this dissertation is complementary to existing software visualization techniques.

3
Synthesis of Code Anomalies

Code anomalies are often not isolated from each other in a program (Macia et al. 2012*a*)(Macia et al. 2012*b*)(Macia et al. 2012*c*)(Macia 2013)(Oizumi et al. 2014*a*)(Oizumi, Garcia, Colanzi, Staa & Ferreira 2015). They are related to each other in different forms, thereby composing the so called *code-anomaly agglomerations*. The information concerning code-anomaly agglomerations may be useful for programmers to reveal more relevant problems in a software system, such as *design problems*. However, to the extent of our knowledge, there is no state-of-art technique that enables someone to explore this information. Moreover, most techniques for design problem identification depend on formal design documentation, which is usually not available. The only option left is to resort to source code analysis. However, existing techniques for source code analysis do not explore and identify relationships between code anomalies (Fowler 1999)(Lanza & Marinescu 2006). Instead, they provide only individual code anomalies information.

Nevertheless, design problems are usually scattered in the implementation of multiple code elements of a program. The reason is that a single design decision is usually reified in multiple code elements. These program elements may be in “distant” locations of the source code. Therefore, it becomes hard to reason about their relationships and infer how they relate to a design problem. In addition, code elements affected by a design problem are not always explicitly related to each other. As a result, this task tends to be even more error prone and time consuming, even for experienced programmers. As a consequence, many programmers neglect design problems until it is impossible to maintain the program without removing them. Therefore, a new technique to assist programmers in the identification of design problems is required.

To overcome the limitation of existing techniques and properly assist programmers, a technique for the Synthesis of Code Anomalies (SCA) is proposed and evaluated in this work. SCA is the systematic search and summarization of information about code-anomaly agglomerations (Oizumi et al. 2014*b*). SCA considers different forms of relationship to search for occurrences of code-anomaly agglomerations in a program (Oizumi et al. 2014*b*). This is important because each form of agglomeration provides information that other forms may not provide. Moreover, each of them presents a distinct perspective to analyze code anomalies (Oizumi et al. 2014*a*)(Oizumi, Garcia, Sousa, Cafeo & Zhao 2016). The main steps of SCA are presented below:

- **Detect Code Anomalies.** Using a conventional technique for code anomaly detection, SCA analyzes the source code of the program aiming at detecting different types of code anomaly.
- **Search for Agglomerations.** After the detection of code anomalies is completed, SCA explores different forms of relationship between anomalies in order to search for code-anomaly agglomerations.
- **Summarize Contextual Information.** To provide valuable information about the agglomerations found, SCA also summarizes contextual information about each agglomeration, including the list of code elements surrounding each agglomeration, such as information about the agglomeration's surrounding code elements.
- **Summarize History Information.** Finally, SCA uses information about past versions of the program to provide historical information about each agglomeration.

To better understand the information provided by SCA, consider the hierarchy of classes presented in Figure 3.1. This figure shows three snapshots of the *Versioner* hierarchy. Each snapshot refers to a specific version of the Apache OODT system (Apa 2015)(Mattmann, Crichton, Medvidovic & Hughes 2006). In OODT, the *Versioner* hierarchy is responsible for managing and storing versions of different *Product* types using alternative storage strategies. All classes in the *Versioner* hierarchy have to implement the *createDataStoreReferences* method. This method has two parameters: a *Product* instance and a *Metadata* instance. As there are no sub-classes for each type of *Product*, each *createDataStoreReferences* implementation has to decide if it is handling the correct *Product* type (e.g., the *MetadataBasedFileVersioner* must only process "flat" products). Consequently, the *Product* type handled by each *Versioner* implementation cannot be discovered from the *createDataStoreReferences* interface. Instead, this can only be discovered by analyzing details of each *createDataStoreReferences* implementation. Hence, the programmer can conclude the *Versioner* implementations are affected by the Fat Interface design problem. This design problem occurs in interfaces that expose multiple functionalities through a general interface. This problem could only be identified through a careful analysis of OODT source code. However, as we already mentioned, this task is not trivial. In this context, SCA would be helpful as described below.

All implementations of *Versioner* (*SingleFileBasicVersioner*, *BasicVersioner*, *DateTimeVersioner* and *MetadataBasedFileVersioner*) are affected by instances of the Feature Envy anomaly. SCA detects these instances of code anomaly using metrics based strategies (Marinescu 2004). All these anomaly instances occur in classes that implement the same interface – which is the *Versioner* interface. This means the anomalous classes are inter-related through hierarchical relationships. SCA could, for example, explore such relationships to detect an agglomeration.

Besides searching for this agglomeration, SCA would also provide contextual information about it. Contextual is defined as information about the relationships between the agglomeration and its surrounding code elements. In the example of Figure 3.1, the code elements *Metadata*, *Product*, *VersioningUtils*, *XmlRpcFileManager*, *XmlRpcFileManagerClient* and *GenericFileManagerObjectFactory* are not anomalous. However, they are part of the agglomeration's context, as they are related to anomalous classes in the agglomeration. This information is important because a design problem affects not only the anomalous code elements in the agglomeration, but also the agglomeration's surrounding code elements. Moreover, if a programmer eventually removes the design problem, some surrounding classes would surely be affected.

Finally, SCA would provide history information about the agglomeration. In the left side of Figure 3.1, there is an example of history information for the *Versioner* agglomeration. Using this information, the programmer is able to see how the agglomeration changed along the evolution of OODT, for instance. The *Versioner* agglomeration, for example, was "born" with only two anomalous code elements comprising it. Then, the agglomeration size expanded in the two subsequent versions. This agglomeration growing behavior along the project history may indicate the presence of an even more harmful design problem in OODT.

The remainder of this Chapter is organized as follows. Section 3.1 provides a detailed discussion about code-anomaly agglomerations. Section 3.2 presents details about the summarization of contextual and history information. Section 3.3 outlines different topologies of agglomerations. The implementation of a tool for the synthesis of code anomalies is presented in Section 3.4. Finally, Section 3.5 summarizes the current limitations of SCA.

3.1 Code-anomaly Agglomerations

A code-anomaly agglomeration is a coherent group of inter-related code anomalies that contributes to the realization of a bigger design problem.

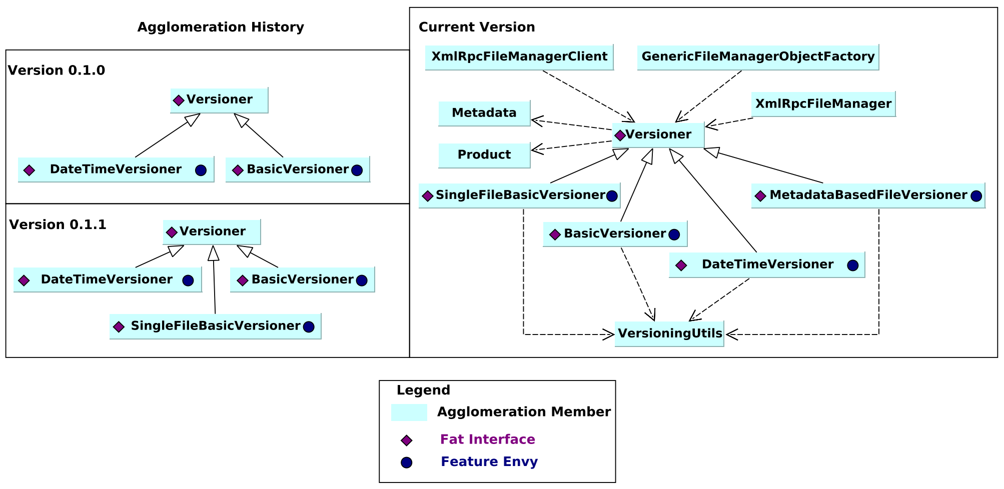


Figure 3.1: Hierarchical agglomeration in the OODT system.

Code anomalies in an agglomeration are inter-related through syntactic or semantic relationships. Examples of syntactic relationships include method calls, inheritance relations, and the same enclosing component. Semantic relationships between two or more code anomalies occur when the anomalous code elements are intended to realize a single design's purpose or concern. 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 module in the source code. The subsection below presents and describes a meta model for agglomerations. This meta model is helpful to understand how different forms of relationship are explored to form agglomerations.

3.1.1
Agglomeration Meta Model

Figure 3.2 uses UML (Booch et al. 2005) to represent the agglomeration meta model. The first meta model element that should be understood is *code element*. In this work, a *code element* is the most basic unit of description in the system implementation. Two *code element types* are considered, which are (1) *class* and (2) *method*. We do not support fine-grained program elements (e.g., code blocks or statements) as they are less relevant to the system's design. In addition, the consideration of code blocks and statements would lead to an intractable number of agglomerations. *Code-anomaly agglomerations* are related to *code elements* in the following way: each *code anomaly* affects a single *code element*. On the other hand, a *code element* may be affected by zero or multiple *code anomalies*. A *code-anomaly agglomeration* groups a set of one or more *code anomalies*. Conversely, each *code anomaly* may be a member of multiple *agglomerations*. In addition, each *code anomaly* is an instance of an

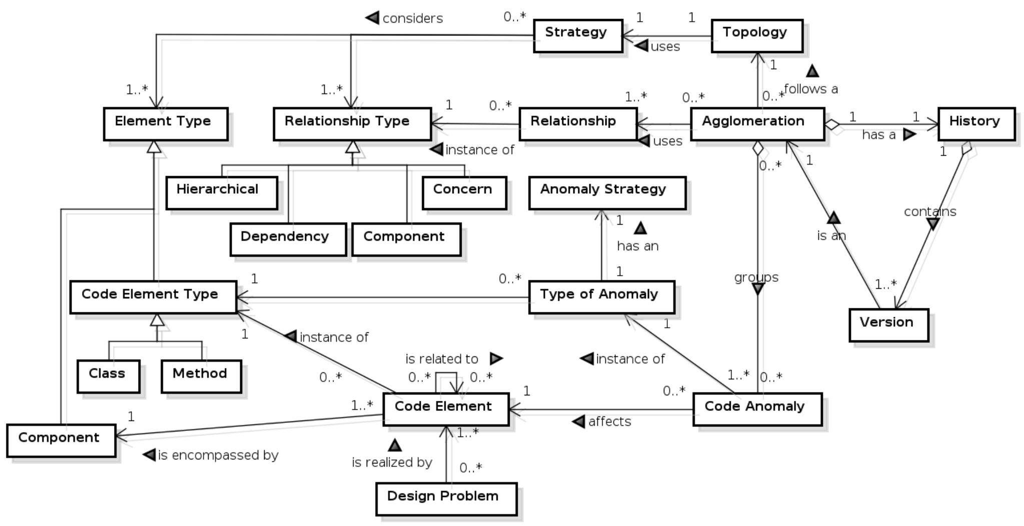


Figure 3.2: Meta model for Code-anomaly Agglomerations.

anomaly type. An *anomaly type* has an *anomaly strategy* in charge of detecting code anomaly instances. Moreover, a *type of code anomaly* is associated with a specific *type of code element* that may be affected by the anomaly type instances.

Code anomalies are grouped into *agglomerations* based on one or more *relationships*. The relationships used for each *agglomeration* are defined by specific *topologies*. Moreover, a *topology* provides a specific *strategy* in charge of characterizing occurrences of *agglomerations*. This *strategy* defines the algorithm used to search for *agglomerations*. In this work, we defined *strategies* for six *topologies*. These *strategies* are described in Section 3.3. Each *strategy* may explore different relationship types. We present below a list of the types of relationship that can be explored by a *strategy*:

– **Hierarchical Relationship.** Code elements that implement the same interface or inherit from the same code element.

– **Dependency Relationship.** Code elements related by method calls or type references.

– **Component Relationship.** Code elements enclosed by the same component.

– **Concern Relationship.** Code elements implementing the same concern. In this context, a concern is a system's functionality that may be (or not) scattered in the implementation of diverse code elements.

Each *strategy* considers different *types of element*. The following types of elements can be used in a *strategy*: *class*, *method* and *component*. Differently from other meta model elements, a *component* represents a sub-system, which may be associated with either an implementation package or a cohesive group of classes. Since each *component* is not necessarily mapped directly to a single code element, this mapping needs to be obtained from some tacit knowledge (i.e., from programmers) or some formal documentation. This mapping can be performed manually or using some design recovery technique (Garcia, Ivkovic & Medvidovic 2013). In case none of these options are feasible, we consider that a component is mapped to a single package in the implementation.

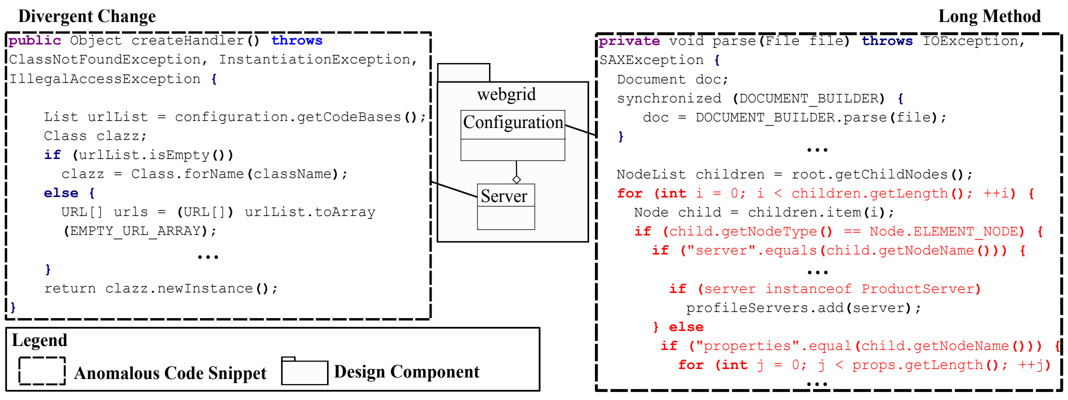
The combination of *element types* and *relationship types* defines the scope considered in the *agglomerations* of a specific *topology*. Finally, a *design problem* may be reified in the source by one or multiple (anomalous) code elements, which can be *classes* or *methods*. To make clear how design problems are related to agglomerated anomalies in the program, Section 3.1.2 presents motivational examples.

3.1.2 Code Anomaly Relationships: Motivating Examples

The first motivating example is the *webgrid* component (Figure 2.2) which was extracted from the Apache OODT system (Apa 2015). The *webgrid* component (1) retrieves resources (e.g., scientific datasets, images, and documents) in platform-neutral formats and (2) describes and discovers resources using extensible metadata. This component uses HTTP to transmit resources. The *Configuration* class (which is encompassed by *webgrid*) holds the complete runtime configuration of resource servers, metadata servers, properties, and other settings for the *webgrid* component.

The *webgrid* component has an instance of the Concern Mixing design problem (Garcia et al. 2009), which is caused by including the implementation of interaction-related functionality along with system-specific functionality. For example, besides implementing its main concern, the *Configuration* class also performs conversion services (from and to XML files) through the *parse* method. These interaction services are best delegated to a specific connector — detaching the conversion services from the main functionality of *Configuration*. Like the *Configuration* class, other classes in *webgrid* also contribute to the Concern Mixing problem.

Figure 3.3 contains a partial view of *webgrid*. This figure shows two instances of code anomaly. First, it was observed that the *createHandler* method in the *Server* class is commonly changed in different ways for diverse reasons. Therefore, this method is infected by the code anomaly called Divergent Change. In addition, the *parse* method in the *Configuration* class is highly complex and fulfills several responsibilities. This method parses a serialized



```
public Object createHandler() throws
ClassNotFoundException, InstantiationException,
IllegalAccessException {

    List urlList = configuration.getCodeBases();
    Class clazz;
    if (urlList.isEmpty())
      clazz = Class.forName(className);
    else {
      URL[] urls = (URL[]) urlList.toArray
      (EMPTY_URL_ARRAY);
                       ...
    }
    return clazz.newInstance();
}
```

```
private void parse(File file) throws IOException,
SAXException {
  Document doc;
  synchronized (DOCUMENT_BUILDER) {
     doc = DOCUMENT_BUILDER.parse(file);
  }
                       ...
  NodeList children = root.getChildNodes();
  for (int i = 0; i < children.getLength(); ++i) {
    Node child = children.item(i);
    if (child.getNodeType() == Node.ELEMENT_NODE) {
      if ("server".equals(child.getNodeName())) {
                       ...
        if (server instanceof ProductServer)
          profileServers.add(server);
      } else
       if ("properties".equal(child.getNodeName())) {
         for (int j = 0; j < props.getLength(); ++j)
                       ...
```

Figure 3.3: Inter-related anomalies in OODT

configuration file and stores the result to an instance of the *Configuration* class. The *parse* method is complex and overloaded in terms of responsibilities. Such complexity and responsibility overload manifest as nested conditions and loops that handle different details of the configuration file. Thus, the *parse* method is infected by the Brain Method anomaly.

These anomalies in isolation may not represent severe design problems. However, the *Configuration* and *Server* classes are inter-related through a *component relationship*, since they are explicitly coupled to each other and both are enclosed by the *webgrid* component. Therefore, *Configuration* and *Server* form a code-anomaly agglomeration, which should be throughly analyzed to identify the Concern Mixing problem.

The individual analysis of an anomaly instance – of Brain Method or Divergent Change – could eventually help a programmer to identify the Concern Mixing problem in specific cases. In exceptional circumstances, a Concern Mixing problem may be confined to a single method or even to an inner block of code. In such cases, the programmer would have to concentrate his/her effort mostly in the affected method. However, in the *webgrid* example the design problem is reified by at least two classes, as illustrated above. Therefore, a programmer would have to inspect all classes in the *webgrid* component in order to fully understand the problem itself and its extent. Otherwise, a partial analysis of individual code anomalies in those classes can lead to an even worse situation. The programmer assumes the problem is confined to a single method and neglects the inspection and refactoring of other methods. Thus, he/she would remove only a single part of the problem. As a consequence, the Concern Mixing problem would still remain in the source code.

Therefore, in the *webgrid* example, the analysis of an agglomeration would be much more beneficial. The programmer would be provided with a coherent group of anomalous classes, which are strong candidates of classes affected by a design problem. This agglomeration would spare the programmer's effort, as he/she would not need to manually search for inter-related anomalous classes. Moreover, an agglomeration, identified with the SCA technique, contains information about its surrounding context. This would help the programmer to understand how the design problem may be affecting other non-anomalous classes. In the *webgrid* example, the intra-method agglomeration is surrounded by the following *java servlets* (Oracle 2015): *ConfigServlet*, *LoginServlet*, *ProductQueryServlet* and *ProfileQueryServlet*. Each of them is responsible for handling specific HTTP requests. For example, *ProductQueryServlet* is responsible for receiving, handling and replying *product* queries.

In the second example, consider again the *Versioner* hierarchy in Figure 3.1. As explained in the beginning of this chapter, the *Versioner* hierarchy is responsible for managing and storing versions of different *Product* types using alternative storage strategies. All classes in the *Versioner* hierarchy implement the *createDataStoreReferences* method. This method has two parameters: a *Product* instance and a *Metadata* instance. As there are no sub-classes for each type of *Product*, each *createDataStoreReferences* implementation has to decide if it is handling the correct *Product* type (e.g., the *MetadataBasedFileVersioner* only deals with "flat" products). Consequently, the *Product* type handled by each *Versioner* implementation cannot be discovered from the *createDataStoreReferences* interface. Hence, the *Versioner* implementations are infected by the Fat Interface problem, which is an interface that exposes multiple functionalities through a general interface.

While a Fat Interface decouples components and simplifies the use of it, such components are also less understandable and analyzable. Determining the actual services exposed by such a component requires inspecting its implementation. Furthermore, the generality of the interface, which simplifies its use, also makes it easier to misuse, since different functionalities are exposed by the same interface.

The *Versioner* hierarchy contains a hierarchical code-anomaly agglomeration. More specifically, the method *createDataStoreReferences* in several classes is affected by the Feature Envy code anomaly. This anomaly in isolation does not represent severe design problems. Nevertheless, it might indicate a deeper design problem in the system when it is inter-related with other anomalies. In the given example, the Feature Envy instances are inter-related through *hierarchical relationships*. Thus, these hierarchically-related anomalies could be analyzed together to spot the Fat Interface problem. As already mentioned in the first example, it would be much more difficult to identify a design problem like this by observing only a single code anomaly. The programmer would have an extra effort to spot the whole extension of the Fat Interface problem.

The last example was extacted from the Mobile Media system (Young 2005). Mobile Media is an academic software product line to derive applications that manipulate photos, videos and music on mobile devices (Young 2005). During the evolution of Mobile Media, its *Controller* component implementation had to be refactored into three components. This "wide" refactoring had to be made as it was no longer possible to maintain a single component over time; both the interface and the class realizing it in the code had to artificially change every time other non-related requirements were being implemented or modified. The artificial changes were not only related to the *Controller* component, but also its client classes, which were realizing other requirements. In other words, there were increasingly-critical ripple effects being observed as the system evolved. The overload of adjacent responsibilities in the *Controller* component reflected in the design as the Concern Overload and Scattered Concern problems (Macia et al. 2012*b*). Figure 3.4 depicts a partial view of the Mobile Media design. This figure shows two components: *Controller* and *View*. Using SCA, we observed a certain case of anomaly agglomeration already in the first version of Mobile Media. This agglomeration was composed by 11 classes, which were located in the implementation of *Controller* and *View*. Each of the agglomeration's classes was affected by multiple anomaly instances. The type of each anomaly instance is described in the legend of Figure 3.4.

The agglomeration of Figure 3.4 was not formed with explicit relationships, such as hierarchical and dependency relationship. Instead, the code anomalies realizing Concern Overload and Scattered Concern in the source code were inter-related through *concern relationships*. This means that each of these classes was not necessarily directly related, but there was a cross-cutting concern, called *Photo Label*, establishing a coherent relation between them. This concern relationship was observed already in the first version of Mobile Media. Thus, the instance of Concern Overload and Scattered Concern problems were cases of congenital design problem, as they were "born" with Mobile Media's design. The detected agglomeration provided useful information about how different anomalies were contributing to such critical design problems.

For example, the multiple combinations of Divergent Change and Shotgun Surgery, together with information about the *Photo Label* concern, were very strong indicators of the Scattered Concern problem. Divergent Change indicates that a class/method has different reasons to change. Shotgun Surgery is an evidence that multiple small changes would be performed in order to change a single concern. Finally, the information about *Photo Label* shows that it is scattered in the implementation of multiple anomalous code anomalies. If we had analyzed each of these data in isolation, the identification of Scattered Concern would be much more difficult. The reason is that, each of these data is only a small evidence of a design problem. Therefore, when individually analyzed, they are not able to reveal the fully extension of the problem. On the other hand, when we analyzed a combination of them, which was provided by the agglomeration, we identified how the anomalous classes were realizing the Scattered Concern problem.

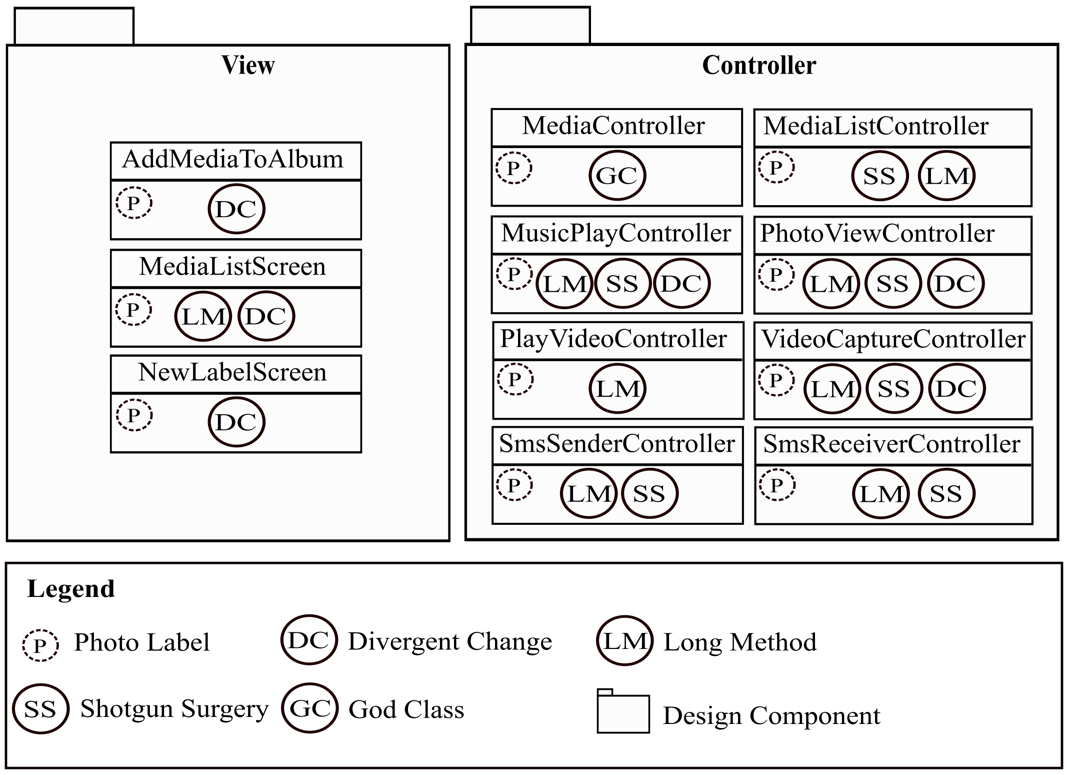


Figure 3.4: Scattered Concern in Mobile Media

3.2 Summarization of Contextual and History Information

Each agglomeration carries a varied set of information that may help programmers to identify design problems. However, besides being unable to reveal the relationships between code anomalies, most of the techniques for code anomaly detection (Emden & Moonen 2002)(Lanza & Marinescu 2006)(Wong et al. 2011)(Mara et al. 2011) provide very little information about the anomalous elements. In general, they provide only the anomaly type and the name of the affected code element. Therefore, in order to overcome this limitation, the summarization of information about code-anomaly agglomerations was proposed.

This summarization should include two types of information about each agglomeration: (1) contextual information and (2) history information. Next, a detailed description about these two types of information is provided.

3.2.1 Contextual Information

For an agglomeration, the context is composed by its surrounding code elements – i.e., external code elements related to one or more code elements of the agglomeration. This information includes both (i) external elements that "use" services provided by agglomeration's elements, and (ii) external elements that provide services to agglomeration's internal elements. In the meta model (Figure 3.2), this information is represented as a self relationship involving *Code Element* meta model element. This relationship establishes that a *Code Element* can be related to zero or many (external) *Code Elements*.

Figure 3.1 illustrates an agglomeration related to external code elements. In this example, there is an agglomeration in the *Versioner* hierarchy: *BasicVersioner*, *DateTimeVersioner*, *MetadataBasicVersioner* and *SingleFile-BasicVersioner* are all affected by the Feature Envy code anomaly. This agglomeration is not isolated from other classes in the system. The *Versioner* agglomeration contains relationships with surrounding classes. *Versioner* is an interface, which provides services to three classes: *XmlRpcFileManager*, *XmlRp-cFileManagerClient* and *GenericFileManagerObjectFactory*. In addition, the *Versioner* hierarchy depends on the following surrounding classes: *Metadata*, *Product* and *VersioningUtils*. When diagnosing design problems, it is important for a programmer to know how each agglomeration relates to other elements in the system. These external relationships provide programmers with concrete information to: (i) identify non-anomalous classes that may be indirectly contributing to the design problem, (ii) plan how to remove a design problem with a minimum impact on the system, and (iii) identify non-anomalous classes that may also be changed to remove a design problem.

3.2.2 History Information

History information consists of the change history of the agglomeration as extracted from previous versions (if any) of the program under analysis. The agglomeration meta model represents history information through two elements: *History* and *Version*. According to Figure 3.2, each *agglomeration* has one and only one *history*. The *history* of an agglomeration is composed by one or multiple *versions*. Finally, each *version* represents the status of an *agglomeration* in that *version* of the program. The minimum number of *versions* for an *agglomeration history* is one, which is the current *version* of the *agglomeration*. Otherwise, if there is no previous version of the program under analysis, the agglomeration does not have a history yet.

As an example of history information, let us consider the source code of version 10 from system S, where it is possible to analyze the history information of various agglomerations that emerged and evolved in previous versions (i.e., 9, 8, 7... 1) of S. This information can be extracted from a control version system, or some other system that stores all versions of the system. Whenever information about multiple versions is available, the synthesis process is able to combine information collected from different versions to help programmers to reason about design problems. Using history information, a programmer is able to identify different patterns of change in an agglomeration. We currently identify four general history patterns, which are described below.

Growing Agglomeration. Using history information a programmer may be able to identify agglomerations that are expanding along the system evolution. That is, agglomerations that accumulate more anomalies as the system evolves. This kind of agglomeration is named as *Growing Agglomeration*. The identification of an expanding agglomeration may be beneficial to the diagnosis of design problems. For example, consider a hypothetical system S that persists information using different mechanisms (database, raw file, memory, etc). Consider also that an abstract class A was created for representing the persistence mechanisms. That is, A defines the interface and the common operations that each persistence mechanism should implement. Finally, consider that different mechanisms were introduced in different versions of system S. The existence of code anomalies in A could cause (i.e., propagate) the insertion of anomalies in subclasses implementing new persistence mechanisms. In this scenario, the analysis of history information would help programmers in identifying a growing agglomeration around the abstract class A, which in turn indicates a possible design problem.

Shrinking Agglomeration. History information also allows programmers to identify agglomerations that are shrinking along a system's evolution; that is, agglomerations that progressively have less anomalies at each new version. We name this kind of agglomeration as *Shrinking Agglomeration*. A shrinking agglomeration may indicate, for instance, that a design problem was identified but not fully removed. A programmer without proper assistance may miss some elements that contribute to the design problem. Therefore, using history information about shrinking agglomerations a programmer may be able to identify and refactor these missing elements, thereby fully removing the associated design problem.

Idle Agglomeration. History information is also useful to identify those agglomerations that almost never change during the evolution of a system. This kind of agglomeration is named as *Idle Agglomeration*. Although this information may look like irrelevant, this may also be helpful to programmers. First, this may indicate that the classes in the agglomeration are so hard to change that very few changes were made during the system's evolution. This would reinforce the need of applying refactorings in the anomalous elements, aiming to improve their maintainability. Last, an idle agglomeration may also contain anomalous elements that hardly ever must be changed. This type of agglomeration may also be the case of stable implementations, whose requirements never change. In this situation, unless the anomalous elements are propagating anomalies to their dependents, it would not be recommended to refactor the agglomeration. A refactoring would represent an useless effort, since that code probably will never change.

Waving Agglomeration. The three previous kinds of agglomeration history represent persistent changing behaviors. In other words, they represent behaviors (grow, shrink or idle) that repeat along the analyzed versions of a system. However, during the evolution of a system, the various patterns may occur at different versions of the same agglomeration. In a situation like this, we name it a *Waving Agglomeration*. This kind of agglomeration may lead to varied conclusions, depending on the combination of patterns manifesting along the agglomeration history. For example, in a system with 20 versions, this may be the case that in the first 10 versions an agglomeration is idle. However, in the subsequent versions this agglomeration starts to grow. Using this information, a programmer may infer that a design problem was exacerbated after version 10 (albeit introduced earlier), which is making the agglomeration to grow.

As described by the changing patterns above, the history of an agglomeration may be the source of valuable information. A programmer might benefit in different ways from analyzing th changing patterns of agglomerations. For instance, history information also helps to reveal the moment in which a design problem arose. A design problem may be either congenital or evolutionary, depending on the moment it was "born". A congenital design problem arises when the system was originally designed, i.e., it is present in the "first" design of the system. On the other hand, an evolutionary design problem arises through changes made along the system evolution. The introduction of such evolutionary design problems may occur (i) gradually, caused by small changes or (ii) abruptly, caused by a big change.

3.3 Topologies of Code-anomaly Agglomerations

As described in the sections above, SCA may use different strategies to search for agglomerations. Each strategy is defined by an *agglomeration topology*. Since a topology defines which relationships should be used to group code anomalies, it is natural to classify agglomerations according to their corresponding topologies. Therefore, this section presents a list of supported topologies. This list was defined based on characteristics of well-known design problems (Garcia et al. 2009)(Martin 2002). For example, the cross-component topology (presented below) was defined based on design problems that occur in the communication between components (e.g., Fat Interface). This set of topologies is not exhaustive since other topologies may be defined. In fact, one of the desirable characteristics of a synthesis technique is to allow the use of flexible strategies, according to the user's need. Therefore, other topologies will certainly be defined in upcoming studies.

3.3.1 Intra-component Topology

Intra-component topology defines agglomerations of code anomalies that occur inside a single component. Agglomerations that fall into this topology comprise code elements that are located within a single design component and are composed of at least two inter-related anomalous code elements. For example, supposing a package has two classes C1 and C2, and each class has one method M1 and M2, respectively. Classes C1 and C2 are related if C1 is referenced within the code of C2, or vice versa. The methods M1 and M2 are related if M1 calls M2, or vice versa. Finally, two code elements, C1 and C2, or M1 and M2, form a intra-component agglomeration if they are related and are affected by the same type of anomaly. Please note that the inheritance relationship is not considered by this topology, because it characterizes the occurrence of hierarchical agglomerations (Section 3.3.4).

Intra-component agglomerations are useful to identify poorly designed components. The reason is that a number of code elements suffering from code anomalies may suggest a problem in the component's design. This usually occurs when the system's design causes the recurring introduction of anomalies

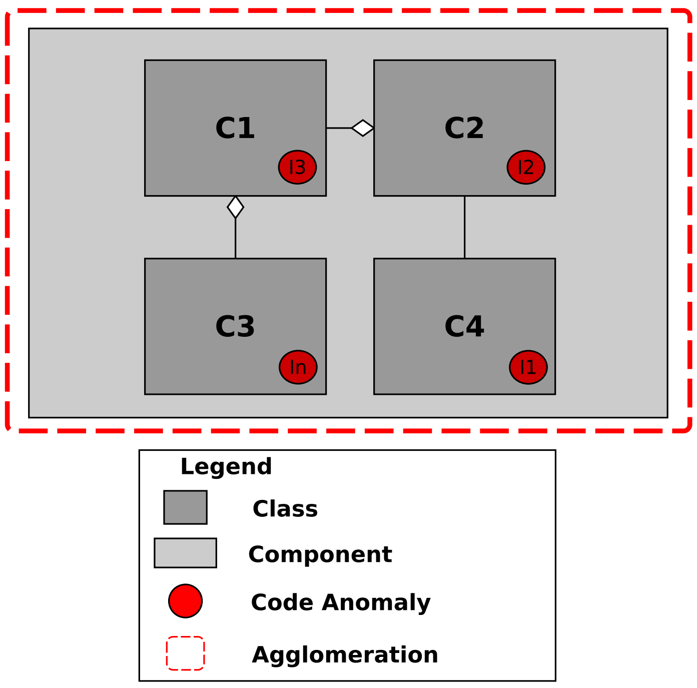


Figure 3.5: Abstract Representation of Intra-component Agglomerations

in a component's implementation. Therefore, anomalies in the individual elements can only be fixed after fixing the system's design.

The formal definition for intra-component agglomerations is the following: given a set of code anomaly instances S composed by anomalies *I1 to In*, a component O, which contains classes *C1* to *Cn*, and a threshold T. We consider there is a code-anomaly agglomeration in component O if at least $T + 1$ code anomaly instances (from S) affect inter-related classes inside O.

To better understand this topology, consider Figure 3.5. This figure depicts an abstract representation of an intra-component agglomeration. As described in the figure's legend, a dashed line represents the agglomeration. The dashed line is around the agglomeration's elements: component, class and code anomalies. The component is represented by a rectangle, the classes are represented by squares, and the code anomalies are represented by ellipses. As illustrated by the figure, in agglomerations of this topology all anomalies must occur inside a single component.

Concrete Example. As an example of intra-component agglomeration, consider again the diagram of OODT in Figure 3.3. This diagram presents a partial view of the *webgrid* component. This figure shows two anomaly instances, Divergent Change in the *Server.createHandler* method and Long Method in the *Configuration.parse* method. These two anomalous methods form an intra-component agglomeration because (i) they are enclosed by the same component, and (ii) they are inter-related through a composition relationship. As already discussed, this agglomeration represents a Concern Mixing problem.

3.3.2 Cross-component Topology

Cross-component topology defines agglomerations of code anomalies that cross the boundaries between different components. An agglomeration of this topology groups code anomalies that occur in inter-related code elements of different components. Agglomerations of this topology are useful to identify design problems that occur in the frontiers between components. That is, design problems related to the communication between components. An example of this situation is the Overused Interface design problem. This problem occurs when the public interface of a component is used by several external components. This problem often represents an undesired situation because an overused interface usually provides diverse unrelated services, which make the interface difficult to use and expensive to maintain.

The formal definition of cross-component agglomeration is the following: given a set of components *S*, where the components contain classes *C1* to *Cn*, and a threshold *T*. We consider there is a cross-component agglomeration involving components of *S* if at least an anomalous class *Cx* of a Component *O* from *S* is related to at least *T + 1* anomalous classes of other components from *S*. The relationships can be either dependencies of *C* to other classes, or dependencies from other classes to *C*.

Figure 3.6 depicts an abstract representation of a cross-component agglomeration. Class *C1* is affected by code anomaly instance *I3*. Moreover, *C1* is related to other three anomalous classes *C2*, *C3* and *C4*, which are affected by anomaly instances *I2*, *I1* and *I4*, respectively. Note that classes *C2* to *C4* are not located in the same *C1's* component, thus characterizing a cross-component agglomeration.

Concrete Example. As an example of cross-component agglomeration, consider the OODT diagram in Figure 3.7. This diagram presents a partial view of four components from OODT: *metadata*, *crawler*, *filemgr* and *pushpull*. The code anomalies identified in the program elements are represented by initials within circles and described in the legend. The *metadata* component encloses, among others, an anomalous code element (*Metadata*), which is "used" by three external anomalous code elements (*ProductCrawler*, *RemoteFile* and *XmlRpcFileManager*). As the anomalous code elements are enclosed by different

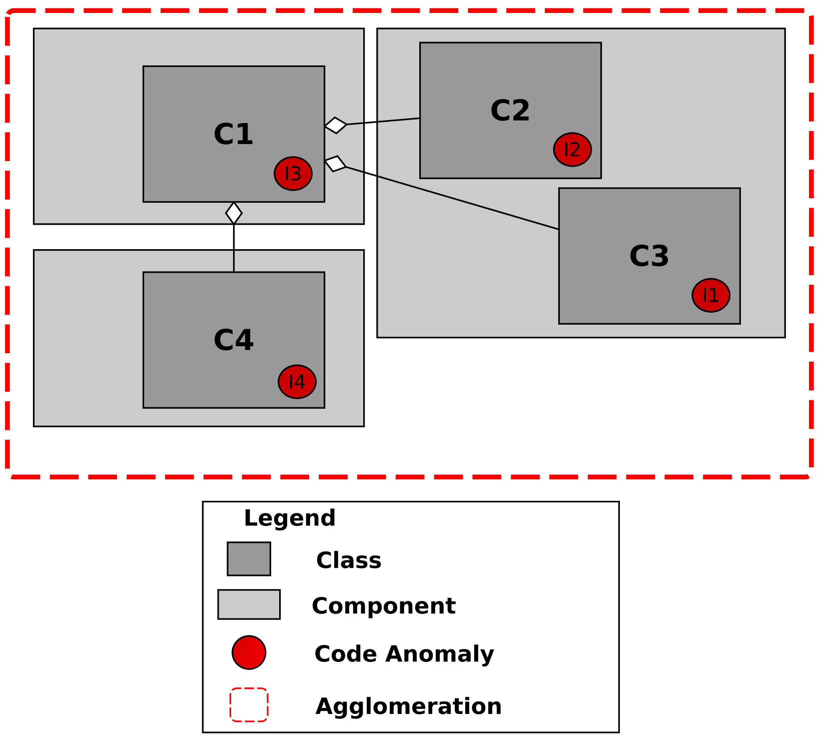


Figure 3.6: Abstract model for Cross-component Agglomerations

components, they form an agglomeration that crosses the component boundaries. In this example there is only one agglomeration, which has four anomalous code elements (i.e., *ProductCrawler*, *Metadata*, *RemoteFile* and *XmlRpcFileManager*).

3.3.3 Concern-based Topology

This topology defines agglomerations composed by anomalous code elements located in the same component that implements diverse concerns. The main advantage of concern-based agglomerations is that they reveal anomalous code elements that are overloaded with concerns. In most of the cases, this is caused by poor modularization of those concerns, which is a design problem by itself. Nevertheless, the overload of concerns also contributes to the introduction of various other types of design problem. As a result, concern-based agglomerations are helpful to the identification of varied types of design problem.

Searching Algorithm. As concern-based topology is the most complex one, we will define it in terms of its searching algorithm. Let us consider the high-level pseudo-code of the concern-based topology, presented in Algorithm 1. This algorithm uses five inputs: (1) a set of components *co*, (2)

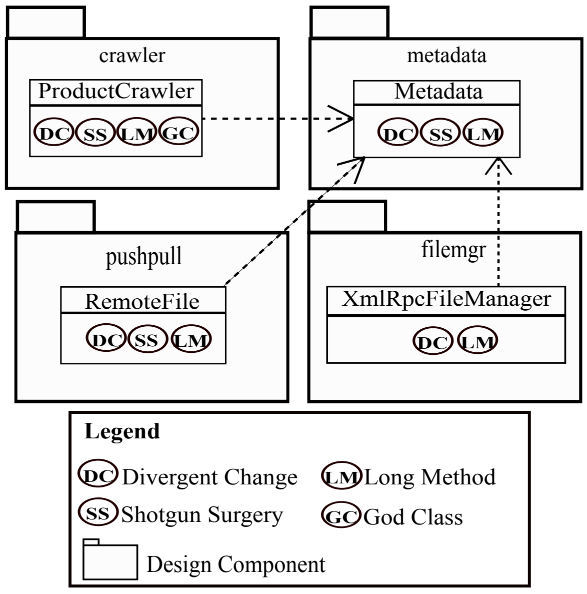


Figure 3.7: Example of cross-component agglomeration

a threshold for the minimum number of concerns *concernThreshold*, (3) a threshold for the minimum agglomeration's size *agglomerationThreshold*, (4) a threshold for identifying weakly-dedicated components, named *weakThreshold*, and (5) a threshold for identifying strongly-dedicated components, named *strongThreshold*. In this context, a component *Cp* is weakly-dedicated to a concern *Cn* if *Cn* is not the main concern of *Cp*. That is, although *Cp* implements *Cn*, the predominant concern of *Cp* is not *Cn*. Conversely, a component *Cp* is strongly-dedicated to a concern *Cn* if *Cn* is the main concern of *Cp*.

As exposed by Algorithm 1, concern-based topology searches for two types of agglomeration. The first type is related to anomalous components overloaded with the assignment of multiple concerns (lines 3 to 15). In order to identify this type of agglomeration, the algorithm inspects all components in the system (lines 4 to 15), searching for anomalous elements that have a number of concerns greater than the *concernThreshold* (lines 7 to 11). For each component, all the anomalous elements that satisfy the aforementioned condition are included in an list of agglomeration candidates. After inspecting the component, if the agglomeration candidate has a number of anomalies higher than the *agglomerationThreshold*, then this agglomeration is included in the final results, i.e., in the list of actual agglomerations. That is, the agglomeration candidate becomes an actual agglomeration.

The second type of concern-based agglomeration is related to scattered anomalous elements related to the same cross-cutting concern (lines 17 to 29). In this case, the algorithm uses two functions to verify if a concern: (i) is modularized in one or more components (strongly-dedicated components), and

(ii) is partially scattered in the implementation of other components (weakly-dedicated components). The combination of these two conditions indicate that a concern cross-cuts the implementation of other concerns. The anomalous code elements contributing to a cross-cutting concern are grouped into an agglomeration candidate (lines 22 to 25). If the agglomeration candidate has a number of anomalies higher than the *agglomerationThreshold*, then this agglomeration is included in the results and confirmed as an actual agglomeration.

Algorithm 1 Searching strategy for concern-based agglomerations

```
 1: let results = {}
 2:
 3: //Searching for overload of concerns
 4: **for** each design component co in the program **do**
 5:    anomalous = getAnomalousElementsOfCompoent(co)
 6:    let agglomeration = {}
 7:    **for** for each ae in anomalous **do**
 8:       **if** $size(ae.getConcerns()) > concernThreshold$ **then**
 9:          agglomeration.add(ae)
10:       **end if**
11:    **end for**
12:    **if** $size(agglomeration.getAnomalies()) > agglomerationThreshold$
       **then**
13:       results.add(agglomeration)
14:    **end if**
15: **end for**
16:
17: //Searching for cross-cutting concerns
18: **for** each design concern con in the program **do**
19:    W = weakDedicatedComponents(con, weakThreshold)
20:    S = strongDedicatedComponents(con, strongThreshold)
21:    let agglomeration = {}
22:    **if** $size(W) > 0$ and $size(S) > 0$ **then**
23:       ae = getAnomalousElementsImplementingConcern(W, con)
24:       agglomeration.addAll(ae)
25:    **end if**
26:    **if** $size(agglomeration) > agglomerationThreshold$ **then**
27:       results.add(agglomeration)
28:    **end if**
29: **end for**
30: **return** $results$
```

In order to gather a complementary explanation, consider Figure 3.8. This figure provides an abstract representation of a concern-based agglomeration. Each class in the component implements a number of concerns. All classes implement *C01*, which is the main concern of the component. However,

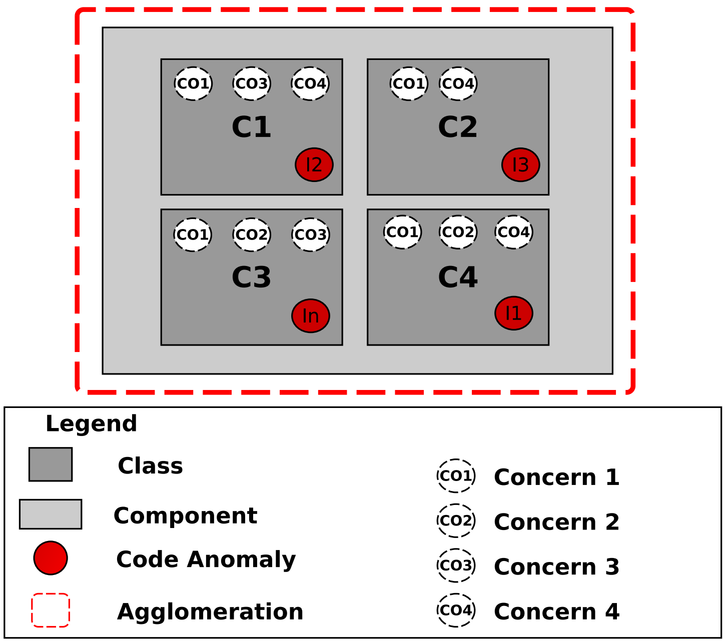


Figure 3.8: Abstract representation for Concern-based Agglomerations

each of them implements at least one additional concern. For example, class C1 implements two additional concerns, which are *C03* and *C04*. In a nutshell, the agglomeration of Figure 3.8 is composed by 4 code anomalies that affect 4 classes. Each anomalous class implements at least two concerns, and all of them are in the same component.

Concrete Example. The Mobile Media example, described in Section 3.1.2, is an example of concern-based agglomeration. As already explained, the agglomeration involving classes from the *Controller* and *View* components provided fundamental information for the identification of a Scattered Concern problem.

Figure 3.9 depicts another example of concern-based agglomeration. This figure shows a partial view of the Health Watcher design. Characters represent the concerns addressed by the code elements of each component. Their respective names are described in the legend. We consider that the anomalous code elements of *Business* (*HealthWatcherFacade* and *RMIFacadeAdapter*) form a concern-based agglomeration. The explanation is that the *Business* component is responsible for realizing several concerns, such as *Persistence* and *Exception Handling*, which are not related to the main concern *Business*. Hence, the anomalous code elements of *Business* form an agglomeration overloaded with multiple concerns.

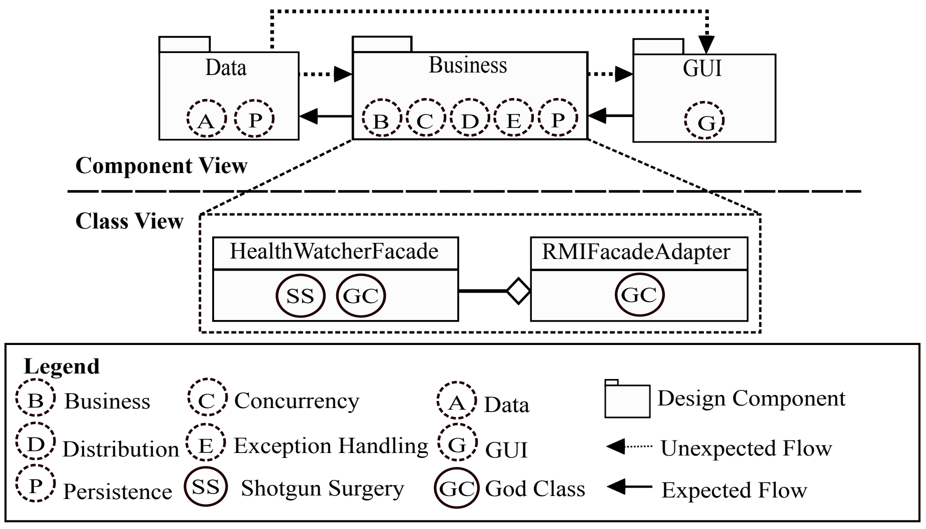


Figure 3.9: Concern-based agglomeration in the Health Watcher system

3.3.4 Hierarchical Topology

Hierarchical topology defines agglomerations of code anomaly that are located in classes of the same hierarchy. This topology is more useful in systems with intensive use of inheritance relationships and interface implementations, such as software frameworks and libraries. Hierarchical agglomerations often help programmers to identify design problems introduced in the root of hierarchies. For example, a problematic interface may induce all of its implementations to have the same type of anomaly. Using hierarchical agglomerations, this type of problem can be more easily spotted as long as there is a first implementation of the interface.

In the definition of this topology, an hierarchy may be composed by two types of relationship: (1) class inheritance or (2) interface implementation. Every agglomeration of this topology has a single root and its descendants. To be considered an hierarchical agglomeration, multiple elements of the hierarchy must be affected by the same type of anomaly. The rationale behind this constraint is that those anomalies should be part of the same design problem. Therefore, if they are of the same type, they are more likely to be coherently part of the same design problem. There is also a higher likelihood of the anomalies in the descendants being propagated by the anomaly in the hierarchy root.

The formal definition of hierarchical agglomerations is the following: given a set of code anomaly instances S composed by anomalies *I1 to In*, a hierarchy H, which contains classes *C1* to *Cn*, and a threshold T. We consider there is a code-anomaly agglomeration in hierarchy H if at least $T + 1$ code anomaly instances (from S) are of the same type and affect classes of H.

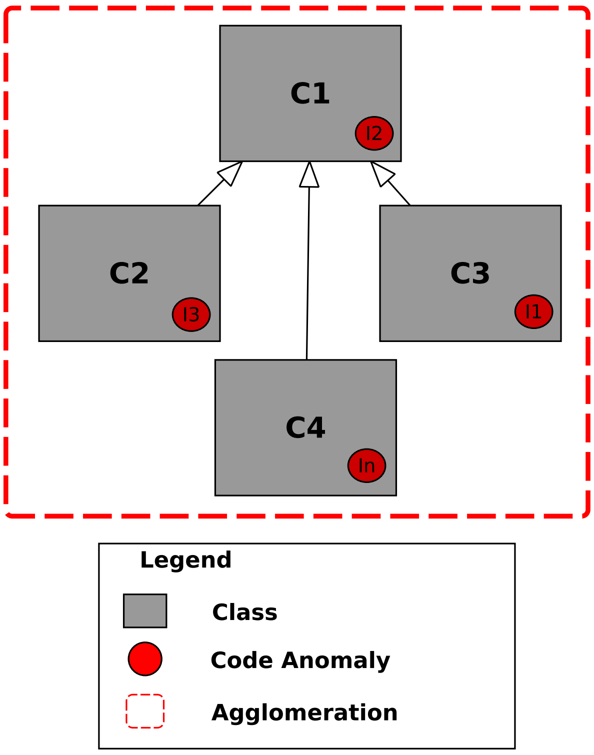


Figure 3.10: Abstract model for Hierarchical Agglomerations

To better understand this topology, consider Figure 3.10. This figure depicts an abstract representation of a hierarchical agglomeration. The hierarchy is composed by four classes (*C1* to *C4*), where (i) *C1* is the hierarchy's root and (ii) classes *C2*, *C3* and *C4* are subclasses of *C1*. As illustrated by the figure, in agglomerations of this topology all anomalies must occur in the same hierarchy and all of them must pertain to the same type.

Concrete Example. An example of this agglomeration topology can be found in the *Versioner* hierarchy (Figure 3.1). As all the classes of *Versioner* are affected by the same anomaly – which is Feature Envy, they are part of a hierarchical agglomeration.

3.3.5 Intra-method Topology

Even though a method is a fine-grained program element, it might be the source of multiple code anomalies. Therefore, the *intra-method* topology defines agglomerations of code anomalies that are located in single methods. Although intra-method agglomerations are very simple and confined to a tiny program behavior, they may be useful for the identification of design problems. Some design problems, such as Ambiguous Interface, can only be identified

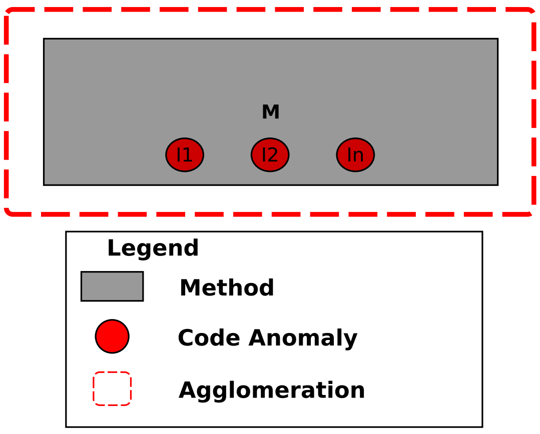


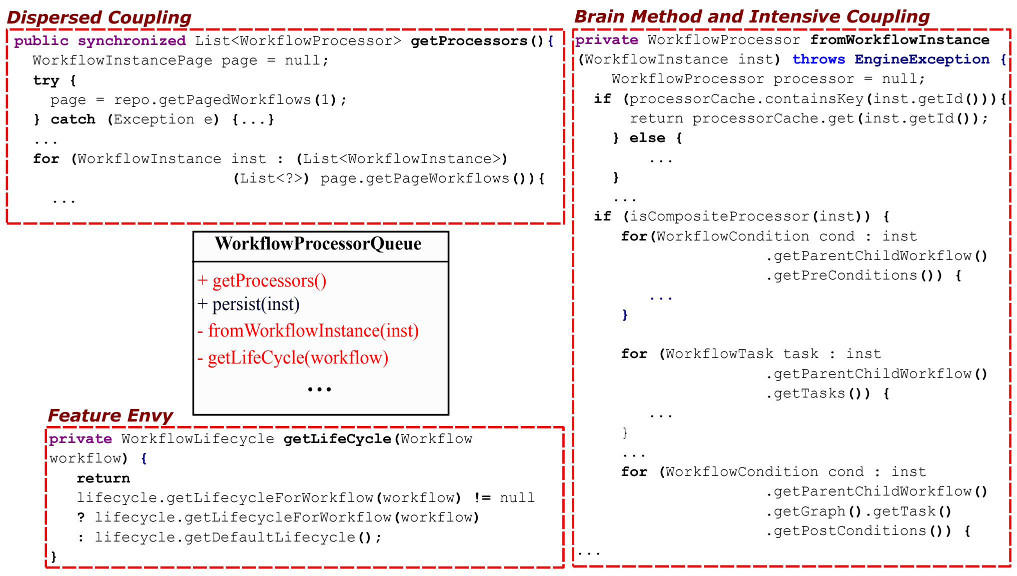
Figure 3.11: Abstract model for Intra-method Agglomerations

by analyzing the interface implementation. The Ambiguous Interface problem is usually reflected in the source code as nested conditions and multiple dependencies. These symptoms can only be identified by analyzing specific implementations of the interface. Therefore, for problems like that, intra-method agglomerations can be good indicators.

This topology is formally defined as follow. Given a set of code anomaly instances S composed by anomalies *I1 to In*, a method M and a threshold T. We consider there is a code-anomaly agglomeration in M if (i) at least $T + 1$ anomaly instances from S affect method M, and (ii) all the anomaly instances affecting method M are singular. That is, method M is affected by at least $T + 1$ different types of anomaly.

To better understand the nature of this topology, consider Figure 3.11. This figure provides an abstract representation of an intra-method agglomeration. This agglomeration affects method M and is composed by three anomaly instances – i.e., *I1*, *I2* and *IN*.

Concrete Example. Figure 3.12 shows an example of intra-method agglomeration extracted from the OODT system. This agglomeration occurs in the *fromWorkflowInstance* method that is implemented in the *WorkflowProcessorQueue* class. This method is affected by two code anomaly instances: Brain Method and Intensive Coupling. This program element is the source of a Brain Method because *fromWorkflowInstance* performs several operations related to pre-conditions, tasks and pos-conditions. All these operations makes the method difficult to read and consequently, difficult to maintain. Moreover, this suffers from Intensive Coupling because it is tightly coupled with few classes, namely *WorkflowInstance*, *WorkflowProcessor*, *WorkflowCondition* and *WorkflowTask*.



```
public synchronized List<WorkflowProcessor> getProcessors(){
  WorkflowInstancePage page = null;
  try {
    page = repo.getPagedWorkflows(1);
  } catch (Exception e) {...}
  ...
  for (WorkflowInstance inst : (List<WorkflowInstance>)
                       (List<?>) page.getPageWorkflows()){
    ...
```

```
private WorkflowLifecycle getLifeCycle(Workflow
workflow) {
   return
   lifecycle.getLifecycleForWorkflow(workflow) != null
   ? lifecycle.getLifecycleForWorkflow(workflow)
   : lifecycle.getDefaultLifecycle();
}
```

```
private WorkflowProcessor fromWorkflowInstance
(WorkflowInstance inst) throws EngineException {
    WorkflowProcessor processor = null;
  if (processorCache.containsKey(inst.getId())){
     return processorCache.get(inst.getId());
   } else {
       ...
   }
   ...
  if (isCompositeProcessor(inst)) {
     for(WorkflowCondition cond : inst
                      .getParentChildWorkflow()
                      .getPreConditions()) {
       ...
     }

     for (WorkflowTask task : inst
                      .getParentChildWorkflow()
                      .getTasks()) {
       ...
     }
     ...
     for (WorkflowCondition cond : inst
                      .getParentChildWorkflow()
                      .getGraph().getTask()
                      .getPostConditions()) {
...
```

Figure 3.12: Intra-class and Intra-method agglomerations in the WorkflowProcessorQueue class

3.3.6 Intra-class Topology

Just like methods, classes may also be affected by diverse code anomalies. Therefore, a topology that characterizes this type of agglomeration has been defined. This topology is named as *intra-class* topology. This topology is useful to identify classes with problematic designs. Moreover, this topology allows programmers to perform bottom-up analysis, going from the class' design until the system's design.

Intra-class is formally defined as follow. Given a set of code anomaly instances *S* composed by anomalies *I1 to In*, a class *C*, which contains methods *M1* to *Mn*, and a threshold *T*. We consider there is a code-anomaly agglomeration in *C* if at least *T + 1* anomaly instances (i) directly affect class *C* or (ii) affect two or more methods of *C*. In this case, an anomaly instance Ix directly affects a class *Cx* if the anomaly occurs due to the internal structure of *Cx*, which is characterized by its attributes and methods. In other words, a class contains an intra-class agglomeration when the sum of its anomalies and its methods' anomalies is higher than the threshold.

To better understand this topology, consider Figure 3.13. This figure provides an abstract representation of an intra-class agglomeration. This agglomeration contains three anomaly instances, which affect a class with two methods. As it is possible to observe, not every anomaly instance in the agglomeration directly affects the class. Instead, two of them (*I2* and *IN*) affect methods in the class.

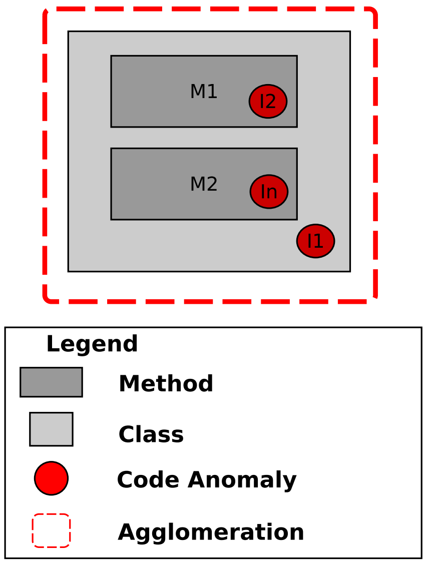


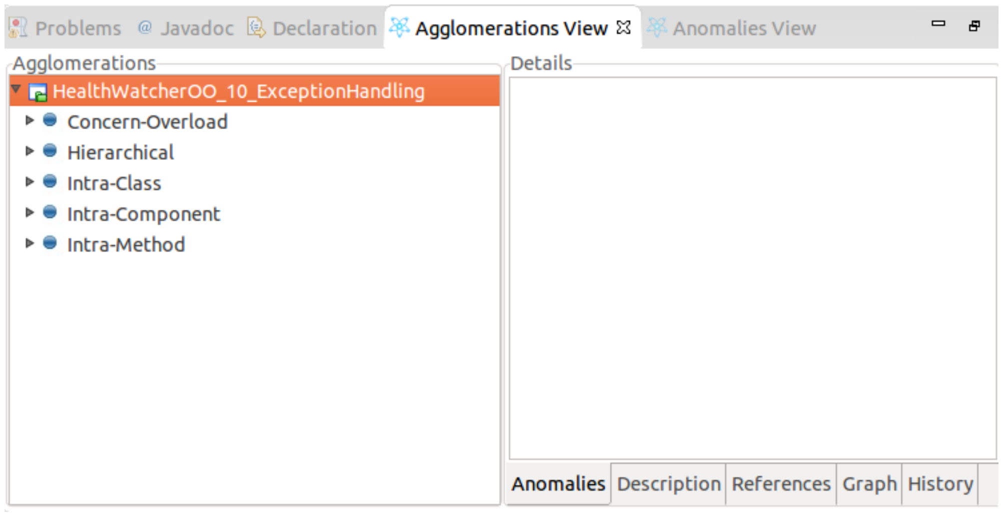
Figure 3.13: Abstract model for Intra-class Agglomerations

Concrete Example. Figure 3.12 shows an example of intra-class agglomeration extracted from the OODT system. This agglomeration occurs in the *WorkflowProcessorQueue* class and is composed by four code anomaly instances: Feature Envy in the *getLifeCycle* method, Dispersed Coupling in the *getProcessors* method, Brain Method and Intensive Coupling in the *fromWorkflowInstance* method.

3.4 Implementation

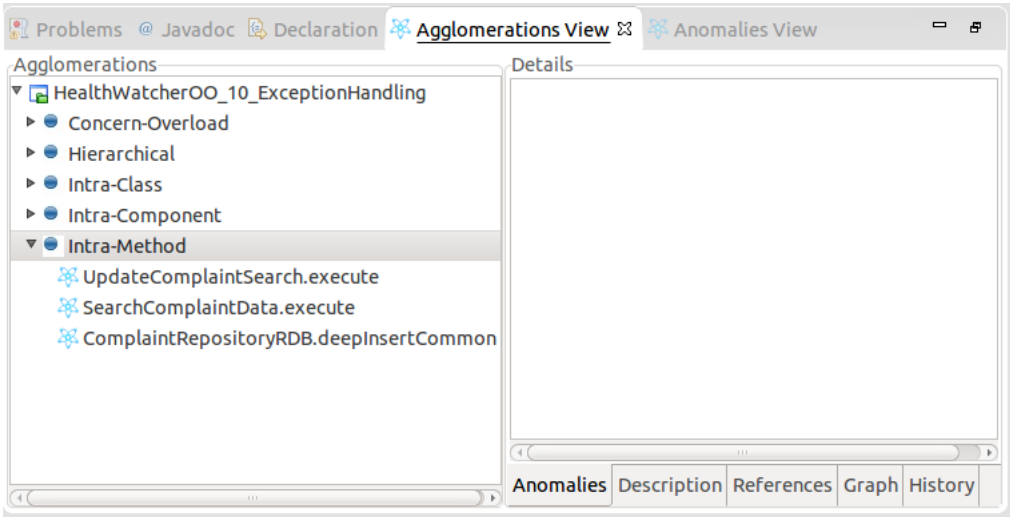
This section provides an overview of Organic: a prototype tool intended to support the systematic synthesis of code anomalies (Oizumi & Garcia 2015). Organic was designed to help programmers in identifying design problems in the source code of Java (Oracle 2015) systems, where design documentation is scare, incomplete or obsolete. Moreover, this tool can also be used by researchers to study the relationships of code-anomaly agglomerations and design degradation. Chapter 4 will describe one of these studies performed with this intent. The tool is implemented as a plug-in for the Eclipse platform (Ecl 2015). In this tool, a number of features collaborate with each other to satisfy the requirements – which are presented in the beginning of this chapter – for a synthesis technique. Next, we present an overview of the features provided by the tool.

Agglomerations View. The Agglomerations View provides the features of the synthesis technique required to support the identification of design prob-



HealthWatcherOO_10_ExceptionHandling
- Concern-Overload
- Hierarchical
- Intra-Class
- Intra-Component
- Intra-Method

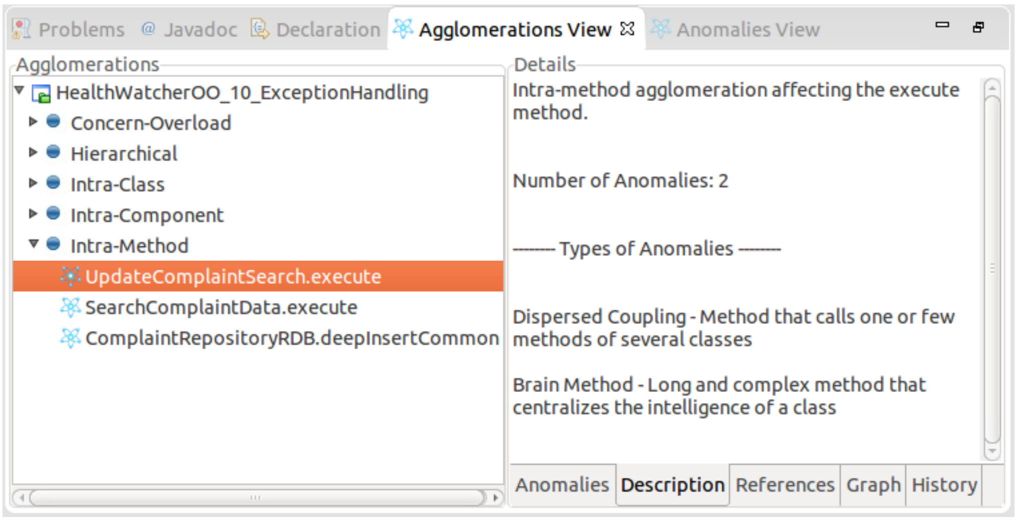
Figure 3.14: Agglomerations View



HealthWatcherOO_10_ExceptionHandling
- Concern-Overload
- Hierarchical
- Intra-Class
- Intra-Component
- Intra-Method
  - UpdateComplaintSearch.execute
  - SearchComplaintData.execute
  - ComplaintRepositoryRDB.deepInsertCommon

Figure 3.15: Expanded Tree into the Agglomerations View

lems. Figure 3.14 shows a snapshot of the Agglomerations View. As it can be observed, this view is separated in two parts: the first part is called Agglomerations and it is shown on the left side; the second part is called Details and it is shown on the right side. The Agglomerations part shows the agglomerations found in one or more projects according to their category. In Figure 3.14, for instance, this view is showing all the agglomerations found in the *Health Watcher* project. When clicking on one of the agglomeration categories, all the detected agglomerations of that category are displayed. Each agglomeration is displayed with a meaningful identifier. For example, there are three intra-method agglomerations in Figure 3.15. The identifier of each intra-method agglomeration is the name of the affected method.



Intra-method agglomeration affecting the execute method.

Number of Anomalies: 2

-------- Types of Anomalies --------

Dispersed Coupling - Method that calls one or few methods of several classes

Brain Method - Long and complex method that centralizes the intelligence of a class

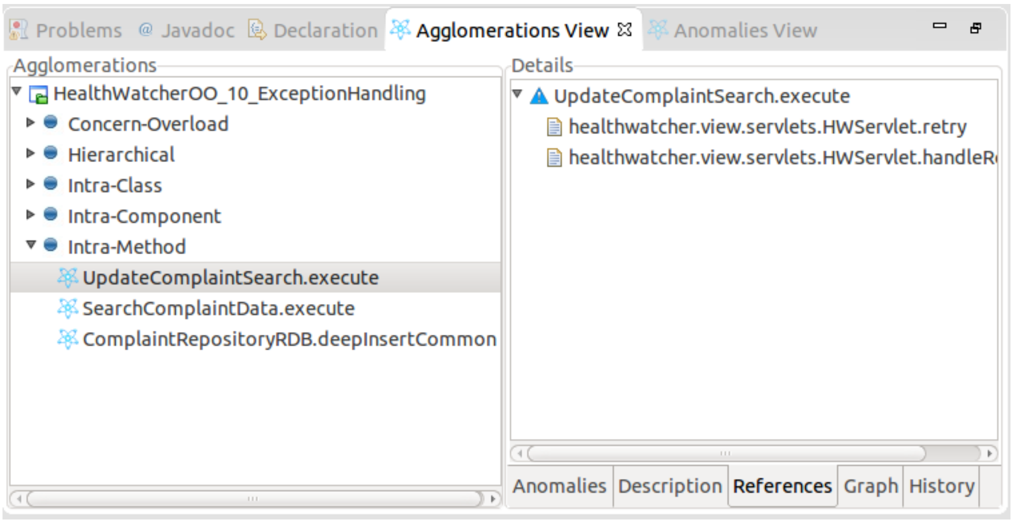
Figure 3.16: Description of an agglomeration

Description. The Description tab shows a summary of all relevant information for the selected agglomeration. The description is composed by the following information:

- **Agglomeration description.** A brief textual description of the agglomeration. This description usually includes the type of agglomeration and the main elements affected by the agglomeration. However, additional information may be provided depending on the type of agglomeration. For example, concern-based agglomerations also provide information about their concerns.
- **Number of Anomalies.** The number of code anomalies that are members of the selected agglomeration.
- **Types of Anomalies.** A brief description of the types of anomalies that are members of the selected agglomeration. Note that, depending on the agglomeration category, an agglomeration may have more than one anomaly of the same type.

Figure 3.16 shows an example of description for a given agglomeration. The selected agglomeration pertains to the intra-method type and affects the *UpdateComplaintSearch.execute* method. This agglomeration contains 2 anomalies, which are Dispersed Coupling and Brain Method.

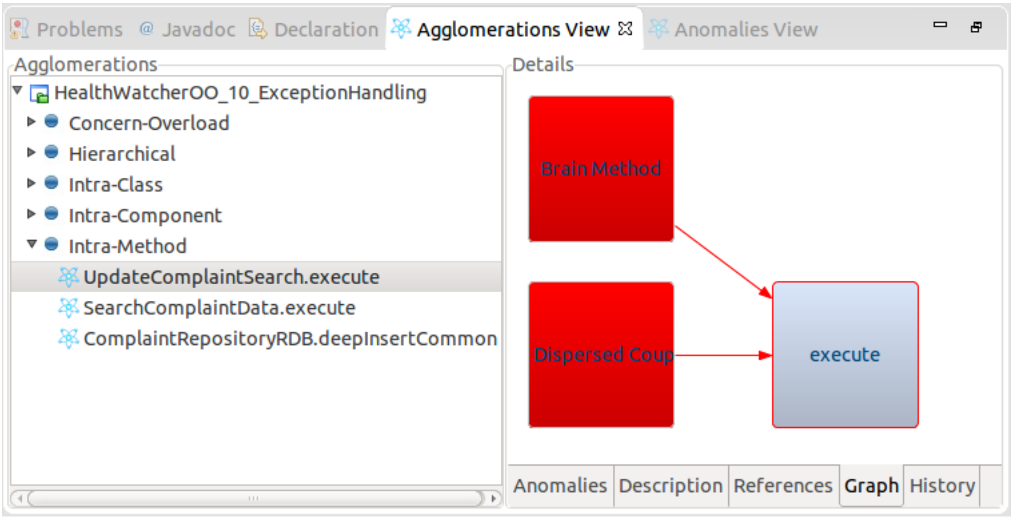
References. The References tab shows the references to each code element affected by the selected agglomeration. In the example of Figure 3.17 the selected agglomeration contains only one code element, which is *UpdateComplaint-Search.execute*. This code element is referenced by two code elements: *HWServlet.retry* and *HWServlet.handleRequest*. These two references are shown in the



HealthWatcherOO_10_ExceptionHandling
Concern-Overload
Hierarchical
Intra-Class
Intra-Component
Intra-Method
UpdateComplaintSearch.execute
SearchComplaintData.execute
ComplaintRepositoryRDB.deepInsertCommon

UpdateComplaintSearch.execute
healthwatcher.view.servlets.HWServlet.retry
healthwatcher.view.servlets.HWServlet.handleR

Figure 3.17: References of an agglomeration

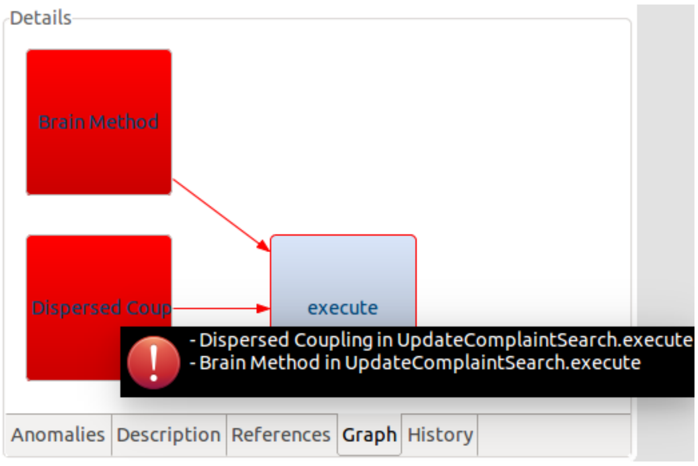


HealthWatcherOO_10_ExceptionHandling
Concern-Overload
Hierarchical
Intra-Class
Intra-Component
Intra-Method
UpdateComplaintSearch.execute
SearchComplaintData.execute
ComplaintRepositoryRDB.deepInsertCommon

Figure 3.18: Graph of an agglomeration

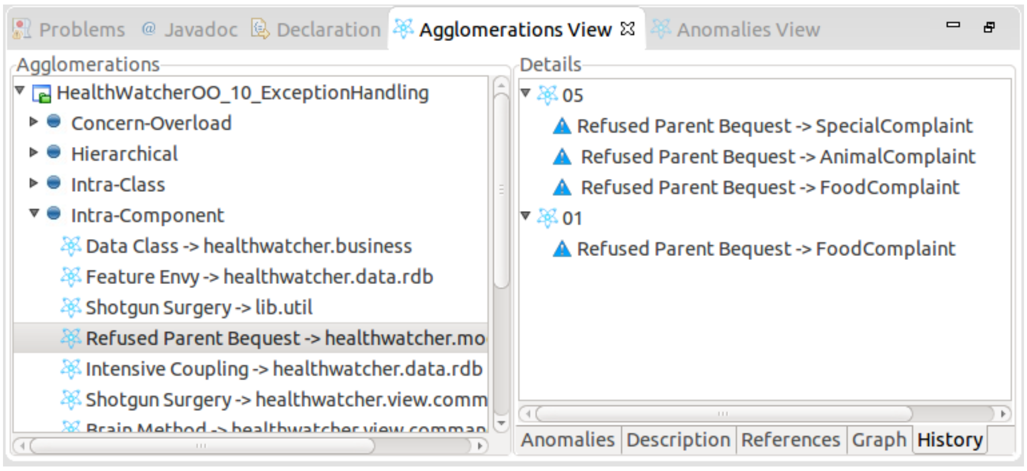
References tab. If more code elements were affected by the agglomeration, this tab would show the references to all of them.

Graph. The Graph tab shows a graphical representation of the selected agglomeration. This representation is not intended to provide a dependency graph of the agglomeration's code elements. Instead, it is intended to provide an abstract representation of the agglomeration, aiming to help the analysis and understanding of the agglomeration. Figure 3.18 shows an example of graph for an intra-method agglomeration. In this graph, code anomalies are represented by nodes labeled with the anomaly type and the affected method is represented by a node labeled with the method's name. As shown in Figure 3.19, additional information appears when the mouse pointer is on one of the nodes.



- Dispersed Coupling in UpdateComplaintSearch.execute
- Brain Method in UpdateComplaintSearch.execute

Figure 3.19: Information displayed in an agglomeration's graph



HealthWatcherOO_10_ExceptionHandling
Concern-Overload
Hierarchical
Intra-Class
Intra-Component
Data Class -> healthwatcher.business
Feature Envy -> healthwatcher.data.rdb
Shotgun Surgery -> lib.util
Refused Parent Bequest -> healthwatcher.mo
Intensive Coupling -> healthwatcher.data.rdb
Shotgun Surgery -> healthwatcher.view.comm
Brain Method -> healthwatcher.view.comman

05
Refused Parent Bequest -> SpecialComplaint
Refused Parent Bequest -> AnimalComplaint
Refused Parent Bequest -> FoodComplaint
01
Refused Parent Bequest -> FoodComplaint

Figure 3.20: History information about an agglomeration

History. The History tab shows the history of the selected agglomeration (see Section 3.1 for details). The history of a given agglomeration may have one or more versions. Each version shows the anomalies that were member of the agglomeration. In Figure 3.20, for example, there are two versions: version 1 and version 5. In version 1, the agglomeration was composed by only one anomaly (Refused Parent Bequest in the *FoodComplaint* class). On the other hand, the same agglomeration was composed by three anomalies (Refused Parent Bequest in *FoodComplaint*, *AnimalComplaint* and *SpecialComplaint*) in the fifth version. Using this resource, it is possible to identify agglomerations that changed along the system's evolution. The programmer can navigate through the agglomeration history in order to identify the four history patterns (Section 3.2.2).

3.5 Limitations

The effective use of this technique depends on the experience and knowledge of the programmer. Different topologies may be used to identify the same design problem. Moreover, each topology may indicate different types of design problem. Therefore, a programmer that uses this technique must analyze the provided information and judge if there is (or not) a design problem. Nevertheless, to the extent of our knowledge, there is no technique capable of revealing a significant number of design problems without an updated design documentation. Existing techniques reveal only local problems, such as individual code anomalies, or require much more information to detect design problems. However, in order to mitigate the aforementioned limitations, we have performed two empirical studies (Chapters 4 and 5) in order to observe the usefulness of specific agglomeration topologies. The results of our studies shed light on whether and how our synthesis technique could be effectively used by programmers.

Another limitation of this technique is related to the detection of code anomalies. Since the proposed approach is based on code anomalies, its success ends up somehow depending on detection strategies for individual code anomalies. However, recent studies suggest that there are no universal metric's threshold that apply to every software project, since software metric values usually follow heavy-tailed distributions (Louridas, Spinellis & Vlachos 2008)(Baxter, Frean, Noble, Rickerby, Smith, Visser, Melton & Tempero 2006). As a result, the effectiveness of the synthesis technique depends on the proper configuration of detection strategies and their respective thresholds. The delegation of the configuration of thresholds for programmers is usually undesired. Programmers can also use methods for selecting specific thresholds tailored to their particular software projects (Vale, Albuquerque, Figueiredo & Garcia 2015). The reason is that even experienced programmers may not have the necessary knowledge for this task. In a future work, we also intend to integrate in our tool a technique for automatically defining thresholds for specific projects.

However, after all, our empirical study (Chapter 4) revealed that the usefulness of our synthesis technique has very limited dependency on the accuracy of the detection strategies. We found that many detected agglomerations are composed of four anomalies or more. Therefore, if some of these anomalies represent false positives, the agglomeration is still useful as long as the other anomalies represent true positives. The programmer can still reflect upon the group of anomalies and infer which of them contribute together to the realization of a design problem. In this process, they can also learn with the false positives and improve the respective strategies for detecting those types of code anomalies.

4
Identifying Design Problems with Agglomerations: A Multi-Case Study

Design problems are certain software structures that indicate the violation of intended design rules or general modularity principles (Suryanarayana, Samarthyam & Sharmar 2014). Every software system suffers from design problems, introduced either during original system development or along software evolution. They are not necessarily faulty or errant, but they negatively impact the design quality of the software. Examples of design problems are Fat Interface (Garcia et al. 2009), Overused Interface (Garcia et al. 2009) and Scattered Concern (Garcia et al. 2009) (Chapter 2).

These problems may have different degrees of severity, but all of them must be detected and removed in the source code somehow. In fact, software systems have been discontinued (MacCormack et al. 2006) or have had to be fundamentally reengineered (Godfrey & Lee 2000) (van Gurp & Bosch 2002)(Schach, Jin, Wright, Heller & Offutt 2002) when design problems were not detected and removed soon after their introduction. Instead, these problems were allowed to persist in a system and to be compounded by other design problems introduced later. The principal difficulty is because frequently a design problem is not localized in a single code element (Moha et al. 2010), instead it is scattered into different code elements of the implementation (Moha et al. 2010).

In order to deal with the scattering of design problems, programmers have tried to detect and to reason about relationships (Abbes et al. 2011)(Macia 2013)(Moha et al. 2010)(Sjobert et al. 2013)(Yamashita & Moonen 2013) of code-level anomalies – popularly known as code smells. In fact, a single code anomaly only represents a possible candidate to identify only one part of a design problem (Yamashita & Moonen 2013). Indeed, most code anomalies are poor indicators of design problems when analyzed individually. (Macia et al. 2012*b*). Even worse, several design problems – such as Fat Interface and scattered Concern – are difficult to find since programmers have to locate and inspect multiple anomalous code elements that are part of the problem. For example, when a programmer is identifying an Overused Interface, he needs to reason about the interface and all the classes that implement the interface.

Furthermore, for a programmer is hard or even impossible to identify which group of code anomalies he should focus on. Even for small software systems, there are hundreds of code anomalies (Macia et al. 2012*c*) and thousands of possible relationships to examine. The key challenge is to understand the relationships between code anomalies that are frequent indicators of critical design problems in an evolving program.

In Chapter 3, we proposed the SCA (Synthesis of Code Anomalies) technique. This technique searches for *code-anomaly agglomerations* and summarizes information about them. In this context, an agglomeration is a coherent group of code anomalies that may represent a design problem. In order to search for agglomerations, SCA considers varied forms of relationship between anomalies. Despite being a promising technique, there is a lack of evidence on the effectiveness of SCA to reveal design problems.

There are few studies investigating questions related to groups of code anomalies (Abbes et al. 2011)(Macia 2013)(Moha et al. 2010)(Sjobert et al. 2013)(Yamashita & Moonen 2013)(Gîrba, Ducasse, Kuhn, Marinescu & Daniel 2007). Most of the resulting techniques assume that relationships in the source code (e.g., type reference, method call, etc) are sufficient to reveal design problems (Abbes et al. 2011)(Macia 2013)(Moha et al. 2010)(Sjobert et al. 2013). Other techniques rely on groups of anomalous code elements derived from code change history (i.e., co-changes) (Gîrba et al. 2007). In addition, each of these techniques is often focused on a few specific design problems. In addition, there are limited (to no) empirical observations about how often these (or other) relationships help to spot recurring design problems faced by programmers. As a result, programmers cannot know which code anomalies relationships are effective at revealing design problems and hence should be the focus of their attention. This task becomes harder in early program versions as there is limited information about the harmfulness of syntactic and co-change relationships of code anomalies. Other recent studies have tried to characterize interactions of code anomalies (e.g., (Yamashita & Moonen 2013)), but again, they provided no empirical evidence of their likelihood of indicating design problems.

Therefore, this chapter presents a multi-case study performing several analyses regarding code-anomaly agglomerations and design problems (Section 4.1). We have used SCA in order to analyze seven systems of different sizes (8 KSLOC to 129 KSLOC) and from different domains. Our analysis involved a total of 5418 code anomalies and 2224 agglomerations. We investigated the circumstances under which agglomerations are related (or not related) to design problems. The analyzed circumstances involve: (i) the statistical significance of the relationship between agglomeration types and design problems, (ii) whether the specific types of agglomerations relate to specific types of design problems, and (iii) to what extent 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 observations (Section 4.2):

1. In most of the target systems, agglomerations were at least twice better than individual code anomalies to indicate the presence of design problems; in some systems, the superiority of agglomerations was even more than five times better.

2. Overall, intra-component, cross-component and hierarchical agglomerations cannot be considered very strong indicators of design problems. They were usually not effective indicators for most of the individual types of design problems. Their statistical significance was not high. Nevertheless, considering all the analyzed systems, approximately 50% instances of these types of agglomerations were related to design problems. This accuracy is much higher than the accuracy of individual code anomalies.

3. Design problems were often much more precisely indicated by concern-based agglomerations. In general, the accuracy was approximately 80% when considering all the design problems and systems analyzed in our study.

4. Analyzing history co-changes in the affected program elements could not be efficient to reveal the relationships of code anomalies and design problems. The reason is that many problems were "congenital", i.e., they were already introduced in the system's initial version. In addition, there were many changes affecting pairs of code anomalies scattered throughout a program. This means that programmers cannot know which of the several co-changes in the corresponding anomalous elements would help them to reveal design problems. Even worse, some co-changes might occur in later versions of a software project, but it is usually too late to identify and remove a design problem. Many clients already depend on the inter-related anomalous code elements that realize the design problem.

5. Based on the aforementioned findings, we can conclude that SCA provide effective means to study the relationship of code anomaly agglomerations and design problems. In addition, SCA-supported agglomerations seem to provide more effective means (than individual code anomalies) to assist locating a wide range of design problems in the source code.

The remainder of this chapter is organized as follows. Section 4.1 describes the settings of our study, including the research questions to the procedure for data collection and analysis. Section 4.2 presents the main results of our study. Section 4.3 outlines the threats to validity of this work. Finally, Section 4.4 summarizes the concluding remarks.

4.1 Study Definition

In this section, we present the evaluation foundations of this study. Section 4.1.1 presents the topologies of code-anomaly agglomerations used in the study. Section 4.1.2 describes the research questions that this study aims to answer. Section 4.1.3 describes the target systems. Finally, Section 4.1.4 addresses the procedure for data collection and analysis.

4.1.1 Agglomeration Topologies

The design problems represent direct symptoms of design degradation (Garcia et al. 2009) and should be removed as early as possible in a software project. However, some studies (Abbes et al. 2011)(Macia et al. 2012*b*)(Macia et al. 2012*c*)(Macia et al. 2012*a*)(Moha et al. 2010)(Olbrich et al. 2010)(Yamashita & Moonen 2013) revealed the individual analysis of code anomalies is not enough to identify design problems. This finding represents a challenge for programmers because they often need to spot design problems based on source code. Moreover, spotting design problems becomes harder because of the lack of up-to-date design documents (Macia et al. 2012*b*).

Programmers may identify a design problem in the source code better when they focus on the relationship between code anomaly than when they focus on code anomalies individually. Code anomaly agglomeration, i.e., a group of interrelated code anomalies, can help programmers to spot a design problem. Therefore, in order to find the best way to interrelate code anomalies, we applied a experiment to verify which forms of agglomeration are better to identify design problems.

To achieve our objective we considered four topologies: (i) intra-component, (ii) cross-component, (iii) hierarchical and (iv) concern-based. In Table 4.1, we present a brief description of each topology. For detailed descriptions and examples about the topologies, refer to Section 3.3 of Chapter 3.

In this study, we opted for not explicitly evaluating the usefulness of intra-class and intra-method topologies. The reason is that these two topologies consider only relationships that would, in theory, be easily spotted by a

Table 4.1: Agglomeration topologies evaluated in this study

Topology	**Description**
Intra-component	A component that contains two or more inter-related anomalous classes.
Cross-component	Two or more anomalous classes located in two or more different components, related through associations, compositions or references in the source code.
Hierarchical	Two or more classes in a common inheritance tree (including interface implementation) that are affected by the same type of anomaly.
Concern-based	Anomalous classes that implement one or more concerns besides implementing the main concern of their enclosing component.

programmer. Moreover, both intra-class and intra-method topologies are not able to directly reveal design problems scattered in different classes and methods. Nevertheless, we will address the evaluation of these two topologies in the controlled experiment presented in Chapter 5.

4.1.2 Research Questions

Previous research (Abbes et al. 2011)(Macia et al. 2012*c*)(Macia et al. 2012*b*)(Macia et al. 2012*a*)(Moha et al. 2010)(Olbrich et al. 2010)(Yamashita & Moonen 2013) revealed that individual code anomalies are not very effective at detecting design problems. Previous research (Abbes et al. 2011)(Macia 2013)(Moha et al. 2010)(Sjobert et al. 2013) showed that groups of inter-related code anomalies may help to reveal maintainability problems. However, there is little knowledge about the relationship between agglomerations and design problems. Moreover, there is little knowledge about how specific anomaly relationships can help programmers to identify specific design problems.

Consequently, programmers still do not know how anomaly agglomerations can help them to locate (typically scattered) design problems. Moreover, design problems need to be revealed in early versions of a program. Otherwise, it is hard to refactor the source code to remove a design problem. Therefore, this study aims at expanding the current knowledge about the code-anomaly agglomerations and design problems. We address three particular aspects associated with this overall goal:

RQ1 Are design problems reflected in code-anomaly agglomerations more often than in individual code anomalies?

RQ2 Which agglomeration topologies are best indicators of design problems?

Table 4.2: 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

RQ3 What proportion of design problems manifest themselves as agglomerations in early versions of a program?

RQ1 compares the relevance of code-anomaly agglomerations against individual code anomalies. If agglomerations represent more often design problems, it means that SCA would be useful for programmers to identify design problems. Different agglomeration topologies may be more useful than others to identify and locate design problems in the source code. Hence it is important to know which agglomeration topologies are constantly related to design problems in different systems. This was the motivation to define our second research question. We address RQ2 by investigating whether specific agglomeration topologies are (or not) good indicators of design problems. These aspects enable us to reveal to what extent the topology of an agglomeration contributes to the location of design problems in the source code. Someone could speculate that design problems often emerge when changes started to be continuously introduced in the source code. However, it might be that some design problems are “congenital”, i.e., they manifest in the first versions of a program. Then, RQ3 is aimed at investigating the extent of the congenital design problems related to code-anomaly agglomerations. If this relation occurs often in early program versions, it can imply that early detection of design problems can be improved. We address RQ3 by analyzing the available initial versions of the analyzed software systems. With this investigation, we aim to understand if design problems are frequently related to agglomerations when the former are introduced in a system.

4.1.3 Target Systems

In order to address the three research questions, we analyzed systems with a wide range of characteristics. We selected systems of different sizes (8 KSLOC to 129 KSLOC), leveraging different design styles, and spanning different domains. We studied 7 systems in total, of which 2 are from academic research labs and 5 are from industry. Table 4.2 summarizes the general characteristics of each target system.

The first is Health Watcher (HW), a web framework system, aimed at allowing citizens to register complaints regarding health issues in public institutions (Soares, Laureano & Borba 2002). The second is Mobile Media (MM), an academic software product line to derive applications that manipulate photos, videos, and music on mobile devices (Young 2005). 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 of oil, while S4 is intended to support the financial market analysis. The last system is Apache OODT (Object Oriented Data Technology), whose goal is to develop and promote the management and storage of scientific data (Mattmann et al. 2006). For all the target systems, multiple classes implement each component. In OODT, for example, each design component is implemented by an average of 24 classes.

4.1.4 Procedure for Data Collection and Analysis

In this section, we present how this empirical study was defined and executed. Here we present the procedures to (1) detect code anomalies and agglomerations, (2) identify design problems, and (3) analyze the relation of design problems and agglomerations.

Detecting Code Anomalies and Agglomerations

This task was accomplished using detection strategies (Lanza & Marinescu 2006) similar to those used in related studies (Section 2.3 of Chapter 2). Such strategies have proven to be the most effective in other systems, with precision (percentage of true positives) higher than 80% (Macia et al. 2012*b*)(Macia 2013). The process of detection was undertaken with the assistance of a tool called *Organic* (Section 3.4 of Chapter 3). Organic was designed to fully support the use of SCA. It uses conventional detection strategies (Lanza & Marinescu 2006) to detect code anomalies and searches for agglomerations using different forms of relationship. The metrics used by the detection strategies were directly collected by Organic. In this study, we focused in the anomalies detected by Organic, which are described in Chapter 2. As the selected anomalies are widely discussed elsewhere, readers may refer to (Fowler 1999) and (Lanza & Marinescu 2006) for details about each of them.

Organic presents two different outputs. The first is a list of code anomaly instances identified in the system. The second output is composed by the detected agglomerations. For this study, we selected four topologies (Section 4.1.1), according to their characteristics. To the extent of our knowledge, Organic is the only tool that fully supports the synthesis of code anomalies.

Identifying Design Problems

Given that this task had to be performed manually, we tried to avoid mistakes by involving designers of the target systems in the whole process. For all the target systems, the identification of design problems was performed using the source code and the intended design. For systems without design documentation, we relied on a suite of design recovery tools (Garcia et al. 2013). Original designers of the target systems assisted us in all the steps of this task. The procedure for deriving the list of design problems with designers was the following: (i) an initial list of design problems was identified using detection strategies presented in (Macia 2013), (ii) the designers had to confirm, refute or expand the design problems identified, (iii) the designers 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 designers for further feedback. For the description of the design problems, please refer to Chapter 2.

Due to the need of human involvement in the study, it means that we preferred to dedicate much more effort on the reliability of our data set rather than on just increasing our sample. For Mobile Media, Health Watcher, S1, S2 and S3, the lists of design problems had been identified in another study (Macia 2013), which already produced and used this information using the procedure described above. Despite the fact that Mobile Media and Health Watcher were built a long time ago, their designers were still available because one of the authors was somehow involved in those projects. For OODT and S4, we followed the same procedures that were followed in the other study (Macia 2013). OODT development started in 1999, however, this was still being actively developed in 2012-2013, when the list of design problems was validated with one of the leading designers. The leading designer of OODT's development collaborated with us on all the steps of this task. Finally, S4 was developed by a private medium-sized company, which collaborated with us to build the list of design problems, following the same procedure followed with other systems.

Analyzing the Relation of Design Problems and Agglomerations

Aiming to analyze the different agglomeration topologies and how they are related to design problems, we defined the research questions described in Section 4.1.2. To answer our research questions, the criteria used for correlating code-anomaly agglomerations and single code anomalies with design problems were the following. A code-anomaly agglomeration and a design problem are related if they co-occur in, at least, one code element. Even though both agglomeration and design problem usually involve many elements, in our definition is sufficient that they co-occur in one element. Similarly, an individual code anomaly and a design problem are correlated if they occur in one code element together. We considered this decision appropriate because there were no previous evidence to base our study on. Therefore, in this first study we had to analyze the largest set of possible relations between agglomerations and design problems.

In order to answer research question RQ1, we performed both downstream and upstream analyses in all the seven systems. The downstream analysis is from the perspective of the design, i.e., we analyzed how the design problems are related to code-anomaly agglomerations and individual code anomalies. Conversely, in the upstream analysis we analyzed how agglomerations and individual code anomalies are related to design problems.

To answer RQ2 we analyzed the relation of each agglomeration topology with design problems in the target systems. We complemented the answer to RQ2 by analyzing cases of false negatives. This analysis aimed to understand why agglomerations failed in some cases. An agglomeration fails to indicate design problems when its code anomalies are not related to a design problem. To check the statistical significance of our results, we used Fisher's exact test (Fisher 1922). The application of this test was performed with the use of the R tool (Bloomfield 2014).

Finally, in order to answer RQ3, we analyzed the manifestation of agglomerations in initial versions of S1, S2, S4 and OODT. Moreover, we compared the manifestation of agglomerations in two versions - one early and one late - of Mobile Media and Health Watcher. The goal was to understand how different types of agglomerations can help programmers to identify congenital design problems.

4.2 Results and Analysis

This section presents the results and analysis of the study described in Section 4.1. Section 4.2.1 presents a comparison between individual code anomalies and code-anomaly agglomerations. Section 4.2.2 discusses the extent to which code anomaly relationships (i.e., agglomeration topologies) are indicators of design problems. Finally, Section 4.2.3 discusses the manifestation of design problems as code-anomaly agglomerations in early system versions. The findings in this section allow us to understand the value of existing techniques for detecting design problems thorough code change history (Gîrba et al. 2007).

Table 4.3: Relation of design problems with agglomerations and code anomalies. **AG** = Code-anomaly agglomeration; **CA** = Individual code anomaly.

System	**Downstream Design Problem**		**Upstream Design Problem**	
	AG	**CA**	**AG**	**CA**
S1	78%	53%	80%	18%
S2	85%	68%	82%	2%
S3	73%	11%	75%	14%
S4	93%	14%	50%	10%
MM	57%	35%	58%	26%
HW	100%	91%	45%	22%
OODT	62%	38%	71%	58%

4.2.1 Comparing anomaly instances vs. agglomerations

This section addresses question RQ1: "*Are design problems reflected in code-anomaly agglomerations more often than in individual code anomalies?*" A previous study has revealed that design problems are related to individual code anomalies (Macia et al. 2012*b*). At the same time, many code anomalies had no relation to design problems (Macia et al. 2012*b*). Then, the answer to RQ1 would enable us to understand whether the analysis of code-anomaly agglomerations may improve the detection of design problems. As mentioned in Section 4.1.4, we performed the analysis from two perspectives: downstream and upstream.

Downstream Perspective. The first two columns of Table 4.3 present the results of the downstream analysis. The first column (AG) presents, for each system, the proportion of design problems related to agglomerations. The second column (CA) shows the proportion of design problems related to single code anomalies. As it can be seen, the relationship of design problems and agglomerations was always above 57% and in most cases above 70%. The proportions of design problems related to agglomerations were in most cases much higher than the proportions unrelated to agglomerations.

A comparison of the Downstream AG and CA columns of Table 4.3 reveals that the results were very consistent: in all the systems, design problems are much more often related to code-anomaly agglomerations than to individual code anomalies. While there was some variation, the difference between the two groups ranged from 17% to 25% for most systems. There were two cases (S3 and S4) of projects where the difference was much higher than 50% and only one case (HW) where the difference was under 10%. In addition, further analysis confirmed that most of the design problems unrelated to agglomerations were also unrelated to individual code anomalies.

Upstream Perspective. Next, we investigated if the observation of agglomerations in the source code can help to indicate the presence of design problems. To this end, we compared the upstream relationship of code-anomaly agglomerations and individual code anomalies to design problems. The Upstream AG (agglomerations) and CA (individual code anomalies) columns of Table 4.3 show that agglomerations presented even better results from this perspective. With the exception of OODT, the proportion of agglomerations related to design problems was at least twice as large as the proportion of individual code anomalies. Moreover, for five systems (HW, S1, S2, S3 and S4), the proportion of single code anomalies related to design problems was lower than 25%. These results further reinforce the potential of exploring code-anomaly agglomerations as indicators of design problems.

The two analyses above provide evidence that agglomerations are more helpful than individual code anomalies to identify design problems. However, these analyses does not take into consideration the relation of design problems with specific agglomeration topologies. Therefore, in the next Section, we address this question by analyzing the relation of design problems with four specific agglomeration topologies.

4.2.2 Identifying Design Problems with Specific Agglomeration Topologies

This section addresses question RQ2: "*Which agglomeration topologies are best indicators of design problems?*" In order to answer this question, we conducted three different analyses in the context of the seven systems. First, we verified whether the relation of each agglomeration topology with design problems is statistically relevant. The goal was to check if the four categories of inter-related anomalies are good indicators of design problems. Second, for each target system, we analyzed the proportion of agglomerations in each topology related to design problems. Finally, we also complemented our analysis with qualitative observations of interesting cases. As we mentioned in Section 4.1.1, we took into consideration four topologies: intra-component (IC), cross-component (CC), hierarchical (HI), and concern-based (CO).

Statistical Relation between Topologies and Design Problems. Table 4.4 presents a contingency table for anomalous code elements affected by topologies and by design problems (variables). The results in the table represent the overall data, taking all the target systems into consideration. In order to produce this table, we identified anomalous code elements – i.e., classes

Table 4.4: Contingency Table and Fisher's Test Results

Topology	AgDp	AgNotDp	NotAgDp	NotAgNotDp	p-value	ORs
IC	190	479	347	4117	<0.001	4,704096
CC	167	389	370	4207	<0.001	4,878982
HI	82	140	455	4456	<0.001	4,704096
CO	97	133	440	4463	<0.001	7,392637
All	312	890	537	4596	<0.001	2,999686

and methods with at least one code anomaly – that satisfied the following conditions: (i) first column (AgDp): the anomalous code elements were both taking part in an agglomeration topology and realizing a design problem, (ii) second column (AgNotDp): the anomalous code elements were participating in a topology, but do not contribute to a design problem, (iii) third column (NotAgDp): the anomalous code elements are affected by a design problems, but not involved in any topology, and (iv) fourth column (NotAgNotDp): not involved at all in any topology or design problem.

Then, we applied Fisher's exact test and calculated Odds Ratio (Cornfield 1951). The goal was to identify whether the occurrences of each topology (experimental group) are statistically related to the occurrence of design problems. Applying Fisher's exact test, we observed that for every agglomeration topology the p-value was lower than 0.001. Odds Ratio shows the chances of code elements taking part in any agglomeration to be related to design problems were higher than for other code elements. These results suggest the relation between topologies and design problems might be statistically significant. However, despite the statistical relation between topologies and design problems, it is possible to observe that no topology predominantly affected code elements with design problems. For example, only 30% of the elements taking part in cross-component agglomerations were related to design problems. This result was expected as different types of code anomaly relationships are likely to help revealing design problems of different nature.

Proportion of Agglomerations Related to Design Problems. Given evidence that agglomerations are related to design problems, it is important to know which agglomeration topology reveals more design problems. Table 4.5 shows, for each agglomeration topology (column): the percentage of agglomerations related to design problems (Related), the percentage of agglomerations unrelated to design problems (Not), and the number of design problems related to agglomerations (DP). Moreover, for each topology, Table 4.5 shows the median and the standard deviation for the columns Related and Not. Finally, the Total column of Table 4.5 shows, for each system, two interesting measures. First, it shows the total percentage of agglomerations related

Table 4.5: Percentage of agglomerations related and unrelated to design problems

Topology	**Intra-component**			**Cross-component**			**Hierarchical**			**Concern-based**			**Total**		
System	**Related**	**Not**	**DP**	**Related**	**Not**	**DP**	**Related**	**Not**	**DP**	**Related**	**Not**	**DP**	**Related**	**Not**	**DP**
OODT	61%	39%	20	73%	27%	240	78%	22%	21	90%	10%	252	71%	29%	350
MM	40%	60%	6	81%	19%	8	50%	50%	11	100%	0%	4	58%	42%	15
HW	41%	59%	21	45%	55%	163	25%	75%	3	75%	25%	170	45%	55%	187
S1	40%	60%	40	52%	48%	57	52%	48%	28	80%	20%	33	49%	51%	122
S2	35%	65%	41	40%	60%	59	50%	50%	40	66%	34%	46	40%	60%	131
S3	38%	62%	40	39%	61%	57	45%	55%	33	70%	30%	50	44%	56%	146
S4	66%	34%	83	34%	66%	48	0%	0%	0	100%	0%	10	50%	50%	87
Median	**40%**	**60%**		**45%**	**55%**		**50%**	**50%**		**80%**	**20%**		**49%**	**51%**	
SD	**0.12294**		**-**	**0.181108**		**-**	**0.244297**		**-**	**0.138924**		**-**	**0.104881**		**-**

and unrelated to design problems. Second, it shows the total number of design problems related to agglomerations. The reader should notice that the values of the Total column cannot be obtained by using the data in other columns. For example, the total number of design problems is not the result of summing the number of design problems in each topology. The reason is that there are intersections between different topologies, i.e., a design problem might be related to more than one topology.

At a first glance, it is possible to observe the cases of intra-component, cross-component and hierarchical topologies followed a similar trend. The number of false positives was high in several systems. For example, the intra-component topology presented 59% of false positives (or more) in 5 out of 7 systems. The cross-component topology represents strong indicators of several design problems in OODT and MM systems. However, the overall result of this topology is similar to the results of the other syntactically-related code anomalies. Therefore, our data reveal these types of agglomeration might not be much helpful for programmers on effectively spotting design problems. For example, if a programmer decides to analyze the cross-component agglomerations, he would have to analyze more than half of all cases of cross-component agglomerations in a system in order to reach an agglomeration related to a design problem. Given the average number of cross-component agglomerations in our sample is 200, the programmer might spend time analyzing approximately 100 agglomerations before finding a design problem. This situation becomes even worse if we observe that each agglomeration usually encompasses more than two anomalous code elements.

There were much less occurrences of hierarchical topology than occurrences of intra-component and cross-component topologies. However, the former followed, proportionally, the same trends of the latter. For most of the target systems, there was at least 50% of false positives. Moreover, we did not find any hierarchical agglomeration in one of the systems (S4). On the other hand, this finding in S4 helped us to understand that hierarchical topologies were much more useful to revel design problems in systems with extensive use of inheritance, such as software frameworks. Only a few classes of S4 required the advanced use of inheritance and polymorphism. On the other hand, OODT is a framework that provides several low-level services for handling distributed scientific data. As a result, several services are implemented using design patterns and other structures that rely heavily on inheritance. The aforementioned observations may help to explain the discrepancies in the hierarchical topology results.

The concern-based topology was, by far, the most consistent indicators of design problems. This finding is sustained by the fact that the concern-based agglomerations were strong indicators of different design problems across all systems. Concern-based agglomerations presented, proportionally, the highest correlation with design problems, when compared to all other topologies. For example, 90% of the concern-based agglomerations in OODT were related to 252 instances of design problems. The superiority of concern-based agglomerations was also much higher even for systems where the absolute number of related design problems is much lower: (i) the percentage of true positives was very high (from 66% to 100%), and (ii) the percentage of false negatives was very low (from 0% to 25%). Considering the data from all systems, we observed a median of 80% concern-based agglomerations related to design problems. The standard deviation of 13.89% is very low as well. That is, the percentage of concern-based agglomerations related to design problems was relatively high in most of the target systems, with a very small gap from every of them to the median. Regarding the absolute number of design problems (DP), the concern-based topologies were related to a high number of design problems in all the systems.

We also observed that all the cross-component, intra-component and hierarchical agglomerations do not have relation to any specific type of design problem. This finding contradicted our intuition. For example, we expected cross-component agglomerations would serve as consistent indicators of certain design problems, such as Fat Interfaces, Overused Interfaces, Cyclic Dependencies, Unwanted Dependencies and Scattered Concern. All these types of design problems are related to the communication of two or more design components. Then, the manifestation of such design problems in the source code could involve inter-related anomalies in two or more components. However, our expectation was not met. We had similar expectations for the other cases of design problems and agglomerations. However, with the exception of concern-based topology, none of these topologies was frequently related to a specific design problem type through all the systems. There were only a very few minor exceptions.

On the other hand, we observed some design problem types were strongly related to concern-based topologies. For example, the Overused Interface, which was expected to be related to the intra-component topology, had a strong relation to concern-based agglomerations. We observed that the overuse of certain interfaces, in fact, was reflected by the presence of several anomalies in the source code. However, these anomalies were not syntactically related. Their relationship was established by the realization of a single concern by the code anomalies. In general, we observed that the concern-based topology was almost uniformly related to all types of design problems. This result is interesting in the sense that it suggests concern-based agglomerations are useful, no matter which type of design problem manifest in the system. However, existing work (Abbes et al. 2011)(Macia 2013)(Moha et al. 2010)(Sjobert et al. 2013) does not consider concern-based relationships to search for inter-related code anomalies.

Spotting the Locus of Design Problems with Concerns. As the aforementioned results suggests, the use of concerns to spot the location of design problems in the source code seems to be a promising strategy. The location of a design problem in the source code cannot be spotted only by analyzing the source code relationships (type reference, method call, etc) between anomalous code elements. A single design problem may be scattered in code elements unrelated in the source code. Therefore, even with a solid knowledge about the system, designers or programmers may not be able to detect all code elements contributing to a design problem. The use of concern-based agglomerations may help to overcome this problem. In this study, we observed almost all anomalies taking part in concern-based agglomerations are often related to design problems. Moreover, we observed that, even with a low number of occurrences (compared to other topologies), the concern-based topology was correlated to a high number of design problems. In OODT and HW, for example, the concern-based topology was correlated to much more design problems than other topologies. Finally, we observed that concern-based agglomerations are useful to reveal different types of design problems. Therefore, they might be used in any type of system to help in the identification of design problems. In contrast, other topologies presented very different results depending on the system under analysis. The hierarchical topology, for example, might not be useful for most of the commercial systems.

4.2.3 Early Detection of Congenital Design Problems

This section addresses the research question RQ3: "*What proportion of design problems manifest themselves as agglomerations in early versions of a* *program?*" 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. To answer this question we analyzed initial versions of four systems: S1, S2, S4 and OODT. Then, we analyzed cases of false positives and false negatives in early versions for all the systems. False positives are cases of agglomerations in early versions that do not relate to a design problem. False negatives are cases of design problems that are related to an agglomeration.

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 up to 75% for S1, S2 and S4. For OODT the proportion was higher than 60%. Conversely, the proportion of agglomerations related to design problems was up to 40% for all the systems. The analysis of all the initial versions reveals that a considerable number of design problems were introduced in the first version of each target system. A considerable proportion of design problems are related to code-anomaly agglomerations already in the first version. This result shows that change history analysis (Gîrba et al. 2007) would not be an effective solution to reveal many instances of design problems. It would be hard to remove such design problems when eventually the agglomeration's anomalies start to suffer co-changes through the later versions.

Analysis of Early and Late Versions. We also explicitly compared 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. In MM, we compared versions 1 and 8. 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 small. Even with few code elements, the first version already contained, partially or totally, 13% 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. This class is anomalous and is related to another anomalous code element of the same component. In the subsequent versions, this problem expanded to several code elements, causing the emergence of a new agglomeration. More specifically, in version 8, we identified the existence of a hierarchical agglomeration, involving classes that inherit from the *BaseController* class. This suggests that different agglomeration topologies may be useful at different stages of the life cycle of a system. Moreover, the identification and removal of agglomerations in the first version can prevent the introduction of more severe problems in subsequent versions.

In HW, we compared versions 1 and 10. In this case, version 1 (6 KSLOC) was only somewhat smaller than version 10 (8 KSLOC). The first version already contained 31% of agglomerations found in version 10. As in the case of the MM system, we observed that some agglomerations from a late version had already started to form in the first version. For example, in the first version of HW, we identified several combinations of Long Method, Feature Envy, and Divergent Change that evolved into cross-component agglomerations in version 10.

False Positive and False Negative Analysis. In order to finalize our analysis on the relationship between agglomeration topologies and design problems, we further analyzed all the cases of false positives and false negatives in early and late versions. In this work, a false positive is an agglomeration that was identified as not being related to any design problem. A false negative is a design problem not related to any of the agglomerations. The intra-component topology presented a high number of false positives in most systems. This observation is illustrated by the percentages in the columns "Not" in Table 4.5. The table does not represent the number of false negatives, but they were much lower (3 to 5 times) than the number of false positives in each system. We performed an inspection of all these cases and identified factors related to the occurrence of both false positives and false negatives. They are discussed in the next paragraphs.

Other Agglomeration Topologies. Contrary to our expectations, there were some instances of congenital or evolutionary design problems related to intra-method or intra-class agglomerations. The most common cases were co-occurrences of Long Method/Shotgun Surgery and Feature Envy/Shotgun Surgery across all the systems. Even though the pair of Shotgun Surgery and Feature Envy is located in the same method, they were often related to particular cases of Fat Interface problems. This suggests that, even encompassing trivial forms of relationships, both intra-method and intra-class should be considered during the identification of design problems.

Agglomerations Indirectly Related to Design Problems. Even though the cases of false positives and false negatives were high, we observed that agglomerations involved in these cases were located in "outer" classes or "neighbor" methods realizing design problems. In other words, we observed there was a frequent "indirect" relation between the design problem and the agglomeration. This indirect relation with a design problem often occurred in two cases: (1) a class C is indirectly related to an agglomeration A if one or more inner methods of C are related to A; and (2) a method M is indirectly related to an agglomeration A if the enclosing class of M is related to A. In this way, the programmer would spot a design problem if he expands the analysis to consider the outer classes and inner methods of the agglomeration A. The consideration of these indirect relationships would be responsible for reducing an average of 40% of false positives and false negatives across the systems. This result shows that, even though agglomerations might not help to determine the exact location of a design problem, they help to determine the approximate locus of a design problem in the source code. Therefore, in such cases, we confirmed that the contextual information provided by SCA would be fundamental to the identification of design problems.

4.3 Threats to Validity

Construct Validity. The construct validity is threatened mainly by possible errors introduced in the identification of design problems and agglomerations. In order to mitigate this threat, for agglomerations, we relied on the only known tool that is able to identify agglomerations. Finally, regarding the identification of design problems, we mitigated the imprecision of manual inspection involving the original programmers in this process. Furthermore, programmers, who had previous experience on the detection of design problems made the identification of design problems.

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 desired. However, it would be impracticable in our study due to the number of systems (7) and the limited access to the original programmers. A higher number of versions would demand more time for programmers to identify design problems. However, their availability is limited. Therefore, we tried to mitigate this threat by selecting, for each different system, a version in a different life cycle stage. In addition, we compared different versions of two systems. Another threat to the conclusion validity is the use of Fisher's exact test. Although the test is suitable for contingency tables, it might not be enough to assert that the correlation of topologies and design problems is statistically relevant. Therefore, this result does not allow us to make strong assumptions about such relationship. We mitigate this threat by complementing the quantitative analysis with qualitative observations.

Internal and External Validity. The main threat to the internal and external validity is related to the set of analyzed systems. We tried to mitigate this threat using 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 suffer 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. However, we are aware that we should perform more studies involving different systems.

4.4 Concluding Remarks

In this work, we analyzed to what extent recurring relationships of code anomalies (agglomerations) are related to design problems. We studied four agglomeration topologies in a longitudinal multi-case study involving 7 systems of different sizes and from different domains. Analyzing more than two thousand agglomerations, we were able to reach some relevant conclusions. Agglomerations are indeed better than individual code anomalies for the identification of design problems. Intra-component, cross-component and hierarchical topologies were not good indicators of design problems. On the other hand, design problems were often more precisely indicated by concern-based agglomerations. The concern-based relationships were, by far, the best indicators of both congenital and evolutionary design problems, thereby questioning the usefulness of existing techniques to support design problems in the source code.

Based on the aforementioned findings, we can conclude that SCA provides effective means to study the relationship of code anomaly agglomerations and design problems. In addition, SCA-supported agglomerations seem to provide more effective means (than individual code anomalies) to assist locating a wide range of design problems in the source code. In addition, the use of contextual and historical information also proved to be useful to identify design problems that are not detected with only the basic information about the agglomerations.

5
Identification of Design Problems with Anomaly Agglomerations: An Empirical Evaluation

Previous evaluation (Chapter 4) provided evidence that agglomerations are more related to design problems than isolated code anomalies. In fact, a single code anomaly only represents a possible candidate to identify one part of a design problem (Yamashita & Moonen 2013). Indeed, most code anomalies are poor indicators of design problems when analyzed individually (Macia et al. 2012*b*). Even worse, several design problems – such as Fat Interface and Scattered Concern – are difficult to find since programmers have to locate and to inspect multiple code elements that are part of the problem. For example, for a programmer to identify an overused interface; he/she needs to reason about the interface and all the classes that implement the interface. Furthermore, for a programmer, it is hard or even impossible to identify which inter-related code anomalies he/she should focus on. Even for small software systems, there are hundreds of code anomalies (Macia et al. 2012*c*) and thousands of possible relationships to examine. Therefore, programmers should only focus on code elements that are related to design problems, i.e., only elements infected with code anomalies that are part of an agglomeration.

Although our previous evaluation has suggested that programmers would better locate a design problem (and its full extent) when reasoning about anomaly agglomerations rather than single anomalies, we do not know if the programmers will actually find more design problems when they use anomaly agglomerations rather than single anomalies. Therefore, a controlled experiment is required to systematically evaluate if programmers can benefit or not from using anomaly agglomerations to identify design problems in realistic project settings. Moreover, this evaluation allows us to identify the limitations and strengths of an agglomeration-driven technique to identify design problem.

The aforementioned experiment needs to rely on a technique that provides several agglomeration-related resources. Therefore, we have selected the new technique, called *Synthesis of Code Anomalies* (SCA), proposed in this dissertation (Chapter 3). SCA is intended to support the reasoning about anomaly agglomerations and to facilitate the location of code elements realizing a design problem. In fact, to the best of our knowledege, SCA is the only technique that can detect different types of agglomerations and can provide different information about them. As aforementioned, a controlled evaluation is required to determine (1) whether agglomeration (in this case, detected with SCA) is useful or not for the identification of design problems, and (2) which aspects of a state-of-the-art synthesis technique (i.e., SCA) should be improved in order to effectively identify design problems.

In this context, this chapter reports the design and execution of a controlled experiment aimed at addressing the goals above. In this experiment, subjects had to identify design problems using two techniques. The fist technique is SCA, the representative of an agglomeration synthesis technique, while the second one is a conventional technique for code anomaly detection. The later is called here as a "conventional technique" because it represents how programmers identify design problems in source code nowadays.

The conventional technique relies on metrics-based strategies to detect code anomalies (Marinescu 2004) (Section 2.3 of Chapter 2). After detecting individual code anomalies, they are presented as a flat list to programmers. The type of anomaly and the affected code element composes each element in this list. The name of the affected class and/or affected method is an example of code elements. Conventional technique is used by some of the state-of-the-art tools for anomaly detection (Fowler 1999)(Lanza & Marinescu 2006)(Emden & Moonen 2002)(Marinescu 2004)(Moha et al. 2010)(Rapu, Ducasse, Gîrba & Marinescu 2004)(Ratzinger et al. 2005). Each anomaly in the list is used by programmers as an indicator of a design problems. Therefore, the comparison of the conventional technique with SCA is suitable as they can be used by programmers to achieve the same goal.

In the experiment, each subject identified design problems in two different software components using each technique at a time. For example, the subject *S1* starts identifying design problems. He/She applies the *SCA technique* to find design problems in the *software component X*. After finishing this task, the same subject *S1* receives a different software component (*Y*). Then, he/she applies the *conventional technique* to find design problems in *Y*. When subject *S1* finishes the task, the experiment restarts with a new subject (S2). Since the participants apply a different technique in each component, we selected for the experiment two software components (X and Y) that are similar in terms of size (KSLOC), amount of code anomalies and amount of design problems.

After identifying design problems using both techniques, the subject had to answer a questionnaire. The answers were used to collect feedback about the techniques used by him/her. The combination of the experiment results with the subject's feedback enables us to perform a broad qualitative and quantitative analysis about the use of agglomerations for the identification of design problems.

This experiment revealed the participants attempted to guess more when they were using the conventional technique than when they were using the SCA technique. The largest number of guesses usually led them to identify more false positives with the conventional technique (46% of false positives with conventional against 34% with SCA). The less amount of information about each code anomaly may be the reason for the large amount of guesses. Consequently, they were more likely to make wrong conclusions. In fact, most subjects confirmed that SCA provides useful information for the identification of design problems.

Regarding the categories of agglomerations, this study allowed us to evaluate which categories were the most accurate for the identification of design problems. According to the subject's opinion, the accurate useful one was intra-method. This was not an expected result because a design problem is often reified by multiple source code elements. However, the intra-method category only searches for agglomerations grouped in a small portion of the system's design. Following this idea, an intra-method agglomeration would not reveal all information that a programmer needs when trying to identify design problems. However, in a careful analysis of the experiment results, we observed that the success of intra-method is due to the fact that subjects did not have previous knowledge about the component's design. Thus, many of them preferred to use a more simple category of agglomeration.

Despite being the most simple form of agglomeration, intra-method is much more useful than individual code anomalies. In comparison to an individual anomaly instance, an intra-method agglomeration has the advantage of being more likely to help a programmer, as (i) it exposes a group of anomaly instances – instead of an individual anomaly, and (ii) it provides much more information about the anomalies and their surrounding context.

Nevertheless, it is important to mention that other topologies were considered helpful as well. Confirming the empirical evidence of our previous study (Chapter 4), the concern-based category was considered by subjects the second most accurate. This result was reinforced by the following fact: design problems identified with the help of concern-based agglomerations were the most complex and hard to spot. As exposed in the previous study (Chapter 4), agglomerations of this category are more likely to represent design problems than other agglomerations. This happens because, besides revealing inter-related anomalous elements, a concern-based agglomeration also shows the relation of anomalous elements with poorly designed concerns, which are often responsible for introducing design problems.

With respect to improvements of SCA technique, half of the subjects said that SCA should provide means to prioritize agglomerations using alternative criteria. In addition, some subjects told that SCA should provide tips on which design problems each agglomeration category is more likely to represent. Finally, subjects said that detailed graphical representations of code anomalies, agglomerations and the actual program design would be helpful to reason about design problems.

The remainder of this chapter is organized as follows. Section 5.1 presents a detailed description of the study, including empirical procedures, research questions and other relevant information. Section 5.2 presents the data analysis. Section 5.3 discusses the threats to validity. Finally, Section 5.4 concludes this chapter.

5.1 Study Definition

This section contains a detailed description of the experiment. Section 5.1.1 presents the criteria used to select subjects. Section 5.1.2 provides a description about the participants' profile. Section 5.1.3 presents the software components explored in the experimental tasks. Section 5.1.5 presents the empirical procedures followed during the execution of the experiment. Section 5.1.4 presents the research questions addressed by this study.

5.1.1 Selection of Subjects

We needed to select participants to use both SCA and conventional techniques in order to do the evaluation of agglomeration. The usage of both techniques enables us to perform a comparative analysis and highlight benefits and drawbacks of using agglomerations for the identification of design problems. For a subject to be chosen as a participant, he/she would need to meet some requirements. The requirements for each subject were:

- R1: He/She should have 4 years or more of experience with software development and maintenance.
- R2: He/She should be responsible for design decisions in at least one relevant software project. In order to be considered, the software project must have at least (i) 3 programmers involved in the software development, (ii) the size of 10 KSLOC, and (iii) 3 versions released.
- R3: He/She should have basic knowledge about code anomalies and refactoring.

R4: He/She should have at least intermediary knowledge about the Eclipse IDE.

R5: He/She should have at least intermediary knowledge about the Java programming language.

As far as R3, R4 and R5 are concerned, the subjects were required to inform their knowledge level. Table 5.1 shows the levels used to classify the subjects' knowledge in each topic. A description of each level was given to the subjects so that they could have a similar interpretation when answering the questions. Based on the answer, we could check if the subject satisfies or not each requirement.

Table 5.1: Knowledge classification

Classification	**Description**
None	I have never heard about it
Minimum	I have heard about it, but I do not use it
Basic	I have a general understanding, but almost never use it
Intermediary	I have a good understanding, and use basic features sometimes
Advanced	I have a deep understanding, and often use advanced features
Expert	I am a specialist in this topic, and use many features almost every day

After being informed about the knowledge levels, the subjects were asked to fill a form. The questions in the form basically addressed the requirements aforementioned. The answers provided by the subjects were analyzed to determine which subjects were eligible to be a participate in the experiment. We highlight that we have not done any kind of test to verify if the participants had the knowledge that they claimed to have. However, we emphasize to the subjects that they should be as honest as possible. We made clear that the goal of the experiment was not to compare the subjects between each other, but compare techniques. In addition, we have guaranteed them that their identities would not be revealed.

5.1.2 Subjects Profile

We want to analyze if the agglomeration technique (SCA) can help programmers to find the fully location of each design problem better than a conventional technique. In order to perform this analysis, we selected a set of participants to use both techniques. The participants chosen for the experiment were either software development professionals or PhD students. We prioritized the selection of professionals, which work in different software organizations, rather than students. Therefore, we chose **six** professionals and **two** PhD students.

The participants in the study were selected after filling out a characterization form, as described in the previous subsection. The questions allow us to know about the expertise of the participants in the topics related to the study: (a) their experience in software development; (b) their expertise in Apache OODT, and (c) their formal education. The responses obtained through the characterization form allowed us to identify some key characteristics of each participant and consequently, allowing us to create a profile about each participant. This was an important step because it allowed us to select the subjects that are capable of providing valuable feedback. For example, a subject who does not have minimum knowledge about Java would spend too much time trying to understand the programming language. Thus, he/she would not be able to identify design problems within time constraints of the experiment. By analyzing data from the characterization form, we were able to identify and discard this kind of subject. Table 5.2 summarizes the characteristics of each subject.

Table 5.2: Subjects' Profile

ID	**Experience**	**Education**	**DD**	**Knowledge**			
	(in years)			**OODT**	**Java**	**CR**	**Eclipse**
1	5	PhD	Yes	None	Advanced	Advanced	Advanced
2	5	Graduate	Yes	None	Intermediary	Intermediary	Intermediary
3	6	Graduate	Yes	None	Advanced	Basic	Advanced
4	12	Graduate	Yes	None	Expert	Advanced	Expert
5	5	Graduate	Yes	None	Advanced	Advanced	Advanced
6	10	Graduate	Yes	None	Intermediary	Intermediary	Intermediary
7	8	Master	Yes	None	Advanced	Intermediary	Advanced
8	4	PhD	Yes	None	Advanced	Intermediary	Advanced

DD = Has experience with Design Decisions?
CR = Code Anomalies and Refactoring

As exposed in the *Education* column of Table 5.2, 100% of subjects had a formal education in computer science. In addition, the subjects had in average 7 years of experience with software development (*Experience* column). Finally, all the subjects had previous experience with design decisions (*DD* column), according to the R2 requirement described in Section 5.1.1.

Regarding knowledge, 100% of subjects had no previous knowledge about the Apache OODT system (*OODT* column). All the selected participants were considered suitable for this experiment due to the following reasons: (1) they had no less than four years of experience with software development, (2) all of them had formal education in computer science, (3) all of them had at least intermediary knowledge about the Java programming language (*Java* column) and the Eclipse IDE (*Eclipse* column), and (4) they have at least basic knowledge about anomalies and refactoring (*AR* column).

5.1.3 Target Components

We chose two software components to be used in the experimental tasks. In order to avoid bias in the experiment, we selected the components based on their degree of similarity. Therefore, both components used in the experiment are similar in:

– Complexity of the software design: in order to prevent that the identification of design problems was easier in a software component than in another, we had to choose two components with similar complexity, since each subject had to study the design of both components.

– Size (KLOC): each participant had a limited time to find design problems. Therefore we chose both components with similar size to avoid that the participant expent more time in one component than in another because of size difference.

– Amount of code anomalies and design problems: in order to make the identification of design problems fair, both components should have a similar amount of code anomalies and design problems. This allowed the participants to find the same amount of design problems in both components.

Push Pull and *Workflow Manager* were the software components chosen for the experiment. Both of them are components extracted from the Apache OODT. The goal of OODT is to support the management and storage of scientific data (Mattmann et al. 2006). We chose Apache OODT because it is an open source with a well-defined set of design problems. Besides, OODT has components that can be evaluated independently and with similar features as complexity of design and size. A short description about each component is presented below.

Push Pull. This component is responsible for downloading remote content (pull), or accepting the delivery of remote content (push) to a local staging area. Content in the staging area is injected into the File Manager system by a crawler framework. Push Pull is a framework with various extensible points. It also provides a fully tailorable Java-based API for the acquisition of remote content.

Workflow Manager. This component is responsible for description, execution, and monitoring of workflows, using a client-server system. Workflows are typically considered to be sequences of tasks, joined together by control flow and data flow, which must execute in some ordered fashion. Workflows typically generate output data, perform routine management tasks (e.g., sending emails), and/or describe a business's internal routine practices. The Workflow Manager is an extensible software component that provides an XML-RPC – remote procedure call protocol which uses XML to encode its messages – external interface, and a fully tolerable Java-based API for workflow management.

5.1.4 Research Questions

After applying the experiment with all participants, we were able to combine the experiment results with the subject's feedback. The combination enables us to perform a broad qualitative and quantitative analysis of using agglomerations for identification of design problems in contrast with the conventional technique. The analysis was based on the following research questions:

RQ1. Which is the most accurate technique regarding the identification of design problems?

RQ2. In accordance with the participants, which are the most useful categories of agglomeration?

RQ3. How SCA can be improved?

Research question RQ1 allows us to analyze which was the most helpful technique. To conduct this analysis, we compare the two techniques by analyzing the list of possible design problems identified by the subjects. In this analysis, we use a ground truth to confirm or refute each design problem. Then we compare the number of false positives produced with the SCA technique against the number of false positives produced with conventional technique. The higher the number of false positives is the lower is the technique's accuracy.

Research question RQ2 aims at complementing the results obtained in our previous study (Chapter 4). This question evaluates, from the subjects point of view, which categories (i.e., topologies) of agglomeration are the most useful. To answer this question, subjects are asked to report (i) which category of agglomeration was used in the identification of each design problem, and (ii) which categories of agglomeration were the most useful according to his/her opinion.

Finally, research question RQ3 allows us to know how the SCA technique can be improved according to the opinion of experienced programmers. This question is addressed by asking subjects, after the experiment, to report their opinion about the SCA technique. All the questions are designed to gather feedback about key characteristics of SCA, such as contextual information, history information and agglomeration topologies.

5.1.5 Empirical Procedures

The experiment was conducted individually with each subject in order to answer our research questions. As already mentioned, subjects were selected based on a questionnaire. Each selected subject worked in tasks divided into two phases. Both phases involved similar tasks. The only difference between them were the combinations of technique (Section 3) and components (Subsection 5.1.3) used in the phase. For example, a given subject X worked in the first phase using the *SCA technique* and the *Push Pull* component. Then, in the second phase the same subject X will use the *conventional technique* and the *Workflow* component. Therefore, there are four different combinations since there are two techniques and two components. In order to have a fair comparison, all subjects were equally divided into four groups (Table 5.3). Each group corresponds to a different combination of technique, component and phase. The order of execution (phase 1 or 2) was also considered because the learning curve may influence the results. For example, a technique that is always used in phase 2 may artificially outperform the technique and to present better results than the technique that were always used in phase 1. The reasons is that, in phase 2, a subject will have learned in phase 1 and, as a consequence, better perform the tasks.

Table 5.3: Combinations of component, technique and phase

	Phase 1		**Phase 2**	
Subject	**Technique**	**Project**	**Technique**	**Project**
Group 1	SCA	Push Pull	Conventional	Workflow
Group 2	SCA	Workflow	Conventional	Push Pull
Group 3	Conventional	Push Pull	SCA	Workflow
Group 4	Conventional	Workflow	SCA	Push Pull

Since the number of subjects is not enough for a controlled experiment, we applied a *quasi-experiment* (Shadish, Cook & Campbell 2001). A *quasi-experiment* is an experiment in which the units/groups are not assigned to conditions randomly. Moreover, we cannot select subjects randomly because we need to ensure that they meet the requirements described in Section 5.1.1. Our *quasi-dependent variable* (x-variable) is the technique used to identify design problems. While the number of design problems found by the participants, but marked as true positives, is the *dependent variable* (y-variable). We used as *control group* the group of participants that used the conventional technique and the group of participants that used the SCA technique as the *treatment group*.

Tasks

Before starting the experiment, each subject received training about basic concepts and terminologies. This training was given only once for each subject before the first phase of the experiment. It consisted of a presentation that last for around 15 minutes, covering the following topics:

- Software Design
- Software Components and Connectors
- Design Problems
- Code Anomalies

Besides this training, each subject received a document summarizing all the topics covered in the presentation. This document was available to be consulted during the whole experiment. The main objective of this training session was to ensure the subjects were exposed to the mandatory background required to understand and properly execute the experimental tasks. The basic training was followed by the two phases of the experiment. Each phase consisted of four tasks: (1) understand the component, (2) learn how to use the technique, (3) identify design problems, and (4) provide feedback about the technique. Details about the experimental tasks are provided below.

Understand the Component: Before using the technique to find design problems, the participant needed to understand about the component used in the experiment. For example, let us assume the Push Pull component is the first system selected for the participant. Then, we gave to the participant a document with a (i) brief description about the component and its goal, and (ii) a description about the design of the Push Pull, including its constituent sub-components, object model, and extension points. We also gave the subject the source code of the component. As both systems have similar complexity of design and the design is not complex (e.g., few classes, packages and few connections between the components), the participant had 20 minutes to read the document and the source code before going to the next task.

Learn How to Use the Technique: In this task we introduced the technique to find design problems. The participant is expected to apply the technique in the component assigned to him/her in the previous task. Consider that SCA is the first technique that the participant will apply in order to find design problems. Then, he/she received a document explaining the particularities of the SCA technique. Using the document, we explain to the participant how the technique works. For example, for SCA, we explained how SCA uses a group of inter-related code anomalies to indicate possible instances of design problems. We also show to the participants which agglomerations the technique finds and how he/she will run the technique using Eclipse IDE. On the other hand, if the technique selected was the conventional technique, we explained how the conventional technique detects individual code anomalies, giving as output a flat list to programmers use during the identification of design problems. This task lasted for 10 minutes. As the use of each technique is simple, we gave to subjects this limit of time.

Identify Design Problems: In this task, the participant had 40 minutes to apply the technique in the selected component. This is the main task in the experiment. Thus, we emphasized to the participant the importance of achieving the key goal of finding design problems. For each design problem found, the participant was asked to provide the following information: (i) short description of the problem, (ii) possible consequences caused by the problem, (iii) classes, methods and/or packages realizing the design problem in the source code, and (iv) the type(s) of agglomerations (Section 3.3) that helped him/her to identify the design problems. If the participant was using the conventional technique, he/she needed to provide almost the same information, but instead the type of the agglomeration, he/she needed to provide the anomaly or anomalies that he/she used to identify the design problems. We gave the subjects this time constraint based on an pilot experiment that we performed in order to adjust the time required for this task.

Provide Feedback about the Technique: In this task, the participant received a feedback form. This form provides a list of questions, which enables the participant to expose his/her opinion about the used technique. The form requires information about (i) the (dis)advantages of using the technique to identify design problems, (ii) whether he/she understood all information provided by the technique, (iii) which types of information were fundamental to identify design problems, (iv) which he/she believes that should be done to improve the technique, (v) what he/she thought about the use of the agglomerations (or code anomalies), and (vi) how the graphical interface provided by the technique affected his/her performance. After the fourth task was completed, we asked the participant to repeat all tasks, but now with a different component and a different technique.

5.2 Results

This section presents the results and analysis of the empirical study described in Section 5.1. The objective of this study was to evaluate SCA with the participation of several experienced programmers. In this evaluation, SCA was compared to a conventional technique. Besides that comparison, we asked for feedback in order to improve SCA and improve our future work. The remainder of this section is organized as follows. Section 5.2.1 presents a comparison on the use of SCA and the conventional technique. Section 5.2.2 presents a discussion on the most relevant categories of agglomeration. Finally, Section 5.2.3 presents suggestions for the improvement of SCA.

5.2.1 Expectations Confirmed: SCA overcomes the Conventional Technique

This section aims at answering the research question RQ1: "*Which is the most accurate technique regarding the identification of design problems?*". In order to answer this question, we asked subjects to analyze two components with the aim of identifying design problems. As described in Section 5.1.5, the subjects performed this task using different techniques for each component. After completing this task, each subject provided two lists of possible design problem. In other words, in each component, the subjects identified possible instances of design problems that they believed to be real design problems. After this, we analyzed whether the design problem candidates were in fact true positives or false positives. In this context, a true positive is an instance of design problem, which was confirmed by a ground truth analysis or by our own analysis. On the other hand, a false positive is a design problem that was neither confirmed in the ground truth analysis nor in our own analysis.

After performing the aforementioned analysis, we obtained the data presented in Table 5.4. This table summarizes the results for both conventional and SCA techniques. Table 5.4 shows the number of true positives and false positives for each subject (line) and technique (column). Finally, in the last line of the table, true positives and false positives are summarized considering the outcomes of all subjects.

Conventional Technique. According with Table 5.4, subjects guessed more when they used the conventional technique than when they used SCA. First,

Table 5.4: Comparison between conventional and SCA

ID	Conventional		SCA	
	True Positives	False Positives	True Positives	False Positives
1	1	1	2	1
2	1	4	0	3
3	1	4	3	2
4	1	3	2	0
5	3	1	4	0
6	1	0	1	0
7	1	1	1	1
8	3	0	3	0
All	**12**	**14**	**14**	**7**

let us consider only data related to the use of conventional technique. Table 5.4 shows that, using the conventional technique, three subjects (2, 3 and 4) presented a number of false positives much higher than the number of true positives. For subjects 1 and 7 the number of true positives and false positives was equal to 1. Finally, compared to other subjects, subjects 5, 6 and 8 achieved more success, as they were able to identify more true positives than false positives.

Considering all subjects, the differences between true positives and false positives for the conventional technique is not very significant. Considering all guesses – i.e., all the design problem candidates – 46% were false positives while 54% were true positives. This means that more than half of the guesses were wrong. In addition, with the exception of subjects 5 and 8, all the subjects presented unsatisfactory results using the conventional technique. Even though some of them even identified a considerable amount of design problem candidates, the majority of them were false positives. This result reinforces our claim (Chapter 3) that a conventional technique does not provide enough information for the identification of a considerable number of design problems.

Is SCA a better alternative to detect design problems? The results above show that conventional techniques may not provide all the information that a programmer needs to identify and to analyze design problems. Therefore, we want to know if SCA, in fact, overcomes certain limitations of conventional techniques. In order to test this hypothesis, consider Table 5.4 again. The last two columns of this table shows the number of false positives and true positives identified when subjects used SCA. In this case, 4 out of 8 subjects identified only true positives. In other words, half of the subjects found only actual design problems when using SCA. In addition, just the subject number 2 identified only false positives with SCA. In spite of that, it is important to note that the same subject presented unsatisfactory results when using conventional technique as well.

When we consider all subjects, an outstanding result becomes evident. Using SCA, subjects were much more careful in the identification of design problems. This finding is confirmed by the fact that with SCA subjects identified only 21 design problem candidates against 26 ones with conventional technique. From the 21 candidates, 66% were true positives, while only 34% were false positives. The overall analysis of all the individuals show that the use of SCA allowed better or (at least) similar accuracy on the identification of design problems. There was only a single case where the subject achieved slightly better results with the conventional technique.

According to our observations during the experiment, the better results of the SCA technique in comparison to the conventional technique is due to the following facts. In general, subjects were tempted to trust in the results presented by both techniques. As the conventional technique provide limited information about each anomaly instance, they spent very time reasoning about each instance. On the other hand, the better accuracy of SCA (Chapter 4) combined with the contextual and history information allowed subjects to better evaluate each agglomeration. Thus, they were able to identify more true positives.

Given all the evidence presented above, it is possible to conclude that *SCA is more accurate than the conventional technique for the identification of design problems*. The amount and quality of the information provided by SCA helped subjects to conduct more precise analyses. Thus, they ended up identifying more true positives when using SCA than when using the conventional technique. Moreover, in post-experiment interviews, most subjects said the information provided by SCA is relevant and useful. On the other hand, many subjects complained from the incompleteness of the information provided by the conventional technique.

5.2.2 The most useful category of agglomeration

This section aims at answering research question RQ2: "*In accordance with the participants, which are the most useful categories of agglomeration?*". To answer this question, we asked subjects what were the most useful categories of agglomeration. After that, we checked the consistency of their answer with the agglomerations used by them to identify design problem candidates. Table 5.5 summarizes the number of times each category was mentioned as the most useful. The first column of this table shows the category name and the

Table 5.5: The most relevant categories of agglomeration

Category	**# of Mentions**
Intra-method	5
Concern-based	3
Intra-class	2
Intra-component	1
Hierarchical	1

second column presents the number of mentions for each category.

Surprisingly, according to the subject's opinion, the most useful one was the intra-method agglomeration. This was not an expected result because most of the design problems (Macia 2013) are likely to be reified by multiple source code elements. However, the intra-method category only comprises agglomerations grouped in a tiny behavior (method) of the system's design. Therefore, our expectation was that an intra-method agglomeration would not reveal all information that a programmer needs when trying to identify design problems. However, in a careful analysis of the experiment results, we observed that the popularity of intra-method agglomerations amongst most participants were due to the fact that subjects did not have previous in-depth knowledge about the component's design. Thus, many of them preferred to use a more trivial category of agglomeration as they would need to inspect and understand only a single method (rather than various methods, classes or packages).

However, despite being the most simple form of agglomeration, the intra-method category still proved to be more useful than individual code anomalies. In comparison to individual anomaly instances, the advantage of an intra-method agglomeration relies on a higher likelihood of helping a developer to: (i) expose a coherent group of anomalies confined into the method and altogether indicating a design problem, and (ii) provide much more information about the inter-related anomalies and their surrounding context (i.e., about the clients and servers of the problematic method).

Nevertheless, it is important to mention that other topologies were considered helpful as well. Confirming the empirical evidence of our previous study (Chapter 4), the concern-based category was considered by subjects the second most useful. This result was reinforced by the following fact: design problems identified with the help of concern-based agglomerations were the most complex and hard to spot. As exposed in the previous study (Chapter 4), agglomerations of this category are more likely to represent design problems than other agglomerations. This happens because, besides revealing inter-related anomalous elements, a concern-based agglomeration also shows the relation of anomalous elements with poorly designed concerns, which are often responsible for introducing design problems.

5.2.3 Improving SCA

This section aims at answering research question RQ3: "*How SCA could be improved?*". To answer this question, after the experiment, we asked subjects to provide feedback about SCA. We asked them (1) which additional information SCA could provide, and (2) which information provided by SCA is useless.

A considerable proportion of subjects reported that SCA should provide means to prioritize agglomerations using different criteria. This prioritized list of agglomerations would help a developer to progressively analyze the agglomerations that have more chance to represent design problems. This would be especially useful in large legacy systems, in which SCA might detect thousands of agglomerations. Examples of possible prioritization criteria are number of anomalies in the agglomeration, severity of the code anomalies and number of concerns affected by the agglomeration.

Subjects also suggested that SCA should provide tips on which types of design problem each agglomeration is more likely to represent. This would reduce the effort required to decide whether an agglomeration represents a design problem or not. Tips could be based on recurring scenarios extracted from replicated case studies.

Some subjects also said that SCA should use more elaborated graphical resources. This would be useful for a programmer to analyze both code anomalies and agglomerations. This improvement is important because some code anomalies are not easy to spot in the source code. For example, an Intensive Coupling anomaly involves several elements in the source code. A graphical representation of this anomaly would help programmers to identify and analyze elements involved in the anomaly. Finally, a graphical representation of the implemented design would be useful as well. Analyzing the high level design of the system may more easily spot some design problems. Thus, a graphical representation of the implemented design would complement the information provided by SCA.

5.3 Threats to Validity

In this section, we discuss some factors that could invalidate our main findings. Each one of these factors, as well as actions to mitigate their impact on the research results is described below.

The first threat is related to the subjects. We used a sample of 8 subjects. This sample may not be enough to achieve conclusive results, as this sample size does not enable us to achieve statistically-significant results. Therefore, in order to mitigate this threat, we designed the experiment as a quasi-experiment. This format of experiment allowed us to (1) choose subjects without using random assignments, and (2) draw significant conclusions based on a small sample. In addition, the knowledge level of each subject may have direct influence on the results presented in this work. In an effort to minimize this threat, all participants underwent the training sessions. This procedure aimed to resolve any gaps in knowledge or conflicts about the experiment.

Another threat to validity is concerned with the misunderstanding of the participants in relation to some question or task. In order to mitigate this threat, we observed the experiment, providing advice to subjects whenever necessary. This assistance was fundamental to ensure that all participants properly answered the questionnaires. Regarding this last item, it is important to mention that we never interfered in the tasks performed by subjects. In fact, we only helped them to understand all the questions and tasks.

The last threat is related to the possible difficulty of the subjects to understand the source code used in the experimental tasks. This difficulty may occur due to: (i) possible complexity of the software components used in the experiment, and (ii) the time limit to complete each task. To minimize this threat, a pilot experiment was performed aiming to adjust the time required to perform these tasks. Since we used two software components, one software can be "easier" than the other one regarding the identification of design problems. To avoid this, we selected components very similar between each other, in terms of size, complexity, number of anomalies and number of design problems. Moreover, all subjects received a basic training about the two components, and half of them used each of these components with a particular technique. We did not observe any consistent results indicating one of these components was easier to be analyzed.

5.4 Concluding Remarks

In this experiment, we evaluated two techniques in the context of design problems identification. The compared techniques were SCA and the conventional one. SCA is the new technique proposed in this work (Chapter 3). Conventional technique represents the technique commonly supported by most state-of-the-art tools for code anomaly detection. This is a fair comparison because both techniques aim to aid programmers in the identification of design problems. For this comparison, we selected eight subjects – six professionals and two PhD students. Subjects performed the identification of design problems in two distinct software components. For this task, they used one different technique at a time. After the identification of design problems, subjects were asked to provide feedback about the techniques used in the experiment.

The results of this experiment suggest that the SCA technique is better than the conventional technique to aid programmers in the identification of design problems. Overall, subjects identified more false positives with the conventional technique (46%) than with SCA (34%). In addition, with the exception of two subjects, all subjects presented unsatisfactory results when using the conventional technique. On the other hand, half of the subjects identified only true positives when using SCA.

Regarding the categories of agglomerations, this study allowed us to evaluate which categories were the most useful. According to the subject's opinion, the most useful one was the intra-method category. In a careful analysis of the experiment results, we observed that the popularity of intra-method agglomerations is due to the fact that subjects did not have previous knowledge about the component's design. Thus, many of them preferred to use a more simple category of agglomeration. Nevertheless, other topologies were also considered helpful by a subset of the participants. We observed that those participants were amongst the most experienced developers in our sample. Confirming the empirical evidence of our previous study, the concern-based category was considered by subjects the second most useful. This result was reinforced by the following fact: design problems identified with the help of concern-based agglomerations were the most complex and hard to spot.

With respect to improvements of SCA, a considerable proportion of subjects said that SCA should provide means to prioritize agglomerations using a flexible set of criteria. Examples of criteria are number of anomalies and severity of the anomaly types. In addition, several subjects told that SCA should provide tips on which types of design problem may be related to each agglomeration category. Finally, subjects said that detailed graphical representations of code anomalies, agglomerations and the actual program design would be helpful to reason about design problems.

6 Conclusion

Design problems are caused by design decisions that negatively impact the resulting system's quality (Garcia et al. 2009). Therefore, they must be carefully identified and removed. However, several programmers neglect or postpone the identification of design problems as much as they can. This behavior usually leads the system's maintenance to become increasingly costly and time consuming. This is evidenced by the fact that several projects have been discontinued in the history of the software industry due to the presence of design problems (Garcia et al. 2009)(Hochstein & Lindvall 2005)(Macia et al. 2012*b*)(Macia 2013). Even though software design drives software development in real project settings, design is rarely formally documented (Macia et al. 2012*b*). As a result, the identification of design problems cannot be performed with existing documentation-driven techniques (Eichberg, Kloppenburg, Klose & Mezini 2008)(Marwan & Aldrich 2009)(Morgan 2007)(Ubayashi, Nomura & Tamai 2010). Hence, evidence of design problems has to be identified based on the source code analysis (Macia et al. 2012*b*).

Along years of research on the software quality field, different studies (Fowler 1999)(Lanza & Marinescu 2006) have agreed with the idea that code anomalies are relevant indicators of design problems. However, the relationships of code anomalies and their design problems' counterparts are hard to understand and characterize (Macia et al. 2012*a*)(Macia et al. 2012*c*)(Macia 2013). The main difficulty stems from the fact that design problems are usually reified by groups of code anomalies. These groups are not easy to spot. This happens because (1) inter-related code anomalies may be located in "distant" code elements, and (2) the relationships between them may not be trivial. Therefore, programmers need a technique that explores different forms of relationship between anomalies to search for coherent groups of code anomalies, which may be realizing design problems. Nevertheless, to the best of our knowledge, there is no state-of-art technique that fulfills this demand.

In this context, this dissertation addresses the aforementioned gap in the literature, proposing and evaluating the SCA technique. This technique searches for different forms of agglomerations, summarizes relevant information about each agglomeration and allows programmers to specify custom forms of agglomeration.

In order to evaluate SCA, a multi-case study – involving 7 systems with different sizes – was conducted. In this study, four agglomeration topologies were characterized and studied. This study revealed that more than 70% of all design problems are related to agglomerations in most of the target systems. In the opposite (upstream) analysis, it was observed that most agglomerations represent most of the design problems (50-70%). The results confirmed that agglomerations are better than single anomaly instances to indicate the presence of a design problem. Regarding the circumstances in which agglomerations represent design problems, the concern-based topology seems to be the best indicator of design problems. The intra-component and cross-component topologies identified the highest number of design problems. However, they also presented the highest number of instances unrelated to design problems.

Given the relevance of agglomerations, SCA was evaluated in a controlled experiment. This experiment compared SCA to the conventional technique, which is the technique used by state-of-art tools to detect individual code anomalies. In this study, 8 experienced programmers used both SCA and conventional techniques to identify design problems in two software components. The results of this study suggest that the conventional technique leads programmers to identify more false positives, i.e., most of the design problems identified using the conventional technique were not actual design problems. The reason is that the conventional technique provides very little information about code anomalies and their relationships. As a result, subjects had less information to analyze and reason about.

Apart from that, all subjects said the information provided by SCA is relevant to the identification of design problems. Moreover, according to the subjects, the most useful agglomeration topology is the intra-method one. This topology was considered specially useful because it allows a bottom-up analysis, that is, from the method's source code to the system's design. The advantage of intra-method as compared to individual code anomalies is that, (i) it is usually more severe – as it indicates two or more anomalies affecting a single method, and (ii) it provides much more information that helps in the identification of design problems.

6.1 Contributions

In this dissertation, we discussed about the increasingly need for a technique that aids programmers in the identification of design problems. In this context, we designed and proposed SCA: a technique for the synthesis of code anomalies. The proposed technique was evaluated in the context of two empirical studies. Both studies provided evidence that SCA, in fact, overcomes existing state-of-art techniques. In a nutshell, the contributions of this dissertation can be described as follows.

- **A technique for the identification of design problems (Chapter 3).** In order to outperform existing techniques for the identification of design problems, we proposed SCA. The main advantage of SCA is that (1) it explores relationships between code anomalies to search for code-anomaly agglomerations, and (2) it summarizes useful information about each agglomeration. For each agglomeration, SCA provides contextual and history information. The context of an agglomeration is all information related to the relationships of the agglomeration with its surrounding code elements. History consists of all the information about the agglomeration in previous or subsequent versions of the program being analyzed.
- **Tool Support (Chapter 3).** Besides designing SCA, we also provided tool support for the use of SCA. This was provided in the form of a Eclipse (Ecl 2015) plug-in called *Organic*. We designed Organic exclusively for Java (Oracle 2015) programs. This tool was fundamental for the conduction of this dissertation's research, as it was used in the evaluation of SCA. Moreover, the usefulness of Organic is not restricted to this dissertation. We plan to improve Organic in order to further explore it in future studies (Section 6.2).
- **Empirical Findings.** Finally, we conducted two empirical studies to evaluate SCA. A list of the main findings is presented below:

  1. **Agglomerations vs Individual Code Anomalies (Chapter 4).** Our first study revealed that more than 70% of all design problems are related to agglomerations in most of the target systems. In the opposite analysis, it was observed that most agglomerations represent most of the design problems (50-70%). These results confirmed our expectations that agglomerations are better than individual anomaly instances to indicate the presence of a design problem.

2. **Agglomeration Topologies (Chapters 4 and 5).** Our empirical studies provided evidence that concern-based topology is the best indicator of design problems. The intra-component and cross-component topologies were related to the highest number of design problems. However, they also presented the highest number instances unrelated to design problems. The hierarchical topology showed to be more useful in systems with intense use of hierarchical relationships. Finally, the intra-method topology was recognized as the most useful topology when a programmer has limited knowledge about the global design of a system, i.e., about other classes and components of a system.
3. **SCA vs Conventional Techniques (Chapter 5).** Finally, our controlled experiment revealed that programmers attempt to guess more when they use the conventional technique than when they use the SCA technique. The largest number of guesses usually led them to identify more false positives with conventional technique. The less amount of information provided by conventional technique may be the reason for the large amount of guesses. Consequently, programmers are more likely to make wrong conclusions.

6.2 Future Work

The results obtained in this work were just a first step towards the objective of helping programmers to tackle design problems in their source code. Along the controlled experiment, subjects provided feedback for the improvement of SCA. Moreover, new ideas emerged from the results presented in this dissertation. Therefore, the following future work is proposed:

– Our first empirical study revealed that concern-overload agglomerations are better indicators of design problems than other topologies. However, most systems do not have a complete and up-to-date specification of components and concerns. In fact, this happens because the manual identification of components and concerns is a time consuming and error-prone task. Therefore, in order to allow programmers to use concern-overload agglomerations, we plan to integrate SCA with different techniques (Garcia et al. 2013) for automated extraction of components and concerns. With this integration, SCA is able to search for concern-overload agglomerations, without requiring manual effort from the programmer.

– A considerable proportion of subjects said that SCA should provide means to prioritize agglomerations using alternative criteria. This would help a programmer to prioritize the most critical agglomerations. Examples of prioritization criteria are number of anomalies, severity of the code anomalies and number of concerns.

– In the empirical study subjects also suggested that SCA should provide tips on which types of design problem each agglomeration is more likely to represent. This would reduce the effort required to decide whether an agglomeration represents a design problem. Tips could be based on recurring scenarios extracted from case studies. For instance, we observed that Overused Interface problems are often related to concern-based agglomerations.

– Some subjects also said that SCA should use more elaborated graphical resources. This would be useful for a programmer to analyze both code anomalies and agglomerations. This is important because some code anomalies are not easy to spot in the source code. For example, an Intensive Coupling anomaly involves several elements in the source code. A graphical representation of this anomaly would help programmers to identify and analyze elements involved in the anomaly. Finally, a graphical representation of the implemented design would be useful as well. Some design problems may be more easily spotted by analyzing the high level design of the system. Thus, a graphical representation of the implemented design would complement the information provided by SCA.

– The identification of design problems is an important step towards the improvement of a system design. However, other steps are required. A next step would be the removal of design problems. The most common form of removing design problems is by *refactoring* the source code. Refactoring is the process of improving the program structure without changing its behavior. Current state-of-art techniques for refactoring explores very little information about the source code. In general, they only support semi-automated refactoring based on instructions provided by the programmer. Some refactoring techniques use individual code anomalies to suggest refactorings. However, individual code anomalies may not provide enough information for a programmer to decide if the source code should be refactored. Therefore, given the aforementioned limitation, a better refactoring technique is required. A technique that uses SCA would be able to suggest and rank refactorings that are more likely to remove design problems.

Bibliography

Abbes, M., Khomh, F., Gueheneuc, Y. & Antoniol, G. (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', pp. 181–190.

Apa (2015), 'Apache oodt source code'. URL
`https://github.com/apache/oodt`.

Bass, L., Clements, P. & Kazman, R. (2003), *Software Architecture in Practice*, Addison-Wesley Professional.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H. & Tempero, E. (2006), Understanding the shape of java software, *in* 'Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications', pp. 397–412.

Bloomfield, V. A. (2014), *Using R for Numerical Analysis in Science and Engineering*, CRC Press.

Booch, G. (2004), *Object-Oriented Analysis and Design with Applications (3rd Edition)*, Addison Wesley, Redwood City, CA, USA.

Booch, G., Rumbaugh, J. & Jacobson, I. (2005), *The Unified Modeling Language User Guide*, Addison-Wesley, Boston.

Carneiro, G., Silva, M., Mara, L., Figueiredo, E., Sant'Anna, C., Garcia, A. & Mendonça, M. (2010), Identifying code smells with multiple concern views, *in* 'Software Engineering (SBES), 2010 Brazilian Symposium on; Salvador, Brazil', IEEE, pp. 128–137.

Cornfield, J. (1951), 'A method of estimating comparative rates from clinical data. applications to cancer of the lung, breast, and cervix'.

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

Ecl (2015), 'Eclipse integrated development environment'. URL
`http://www.eclipse.org`.

Eichberg, M., Kloppenburg, S., Klose, K. & Mezini, M. (2008), Defining and continuous checking of structural program dependencies, *in* 'Proceedings of the 30th International Conference on Software Engineering; Leipzig, Germany', pp. 391–400.

Emden, E. & Moonen, L. (2002), Java quality assurance by detecting code smells, *in* 'Proceedings of the 9th Working Conference on Reverse Engineering; Richmond, USA', p. 97.

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

Fowler, M. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, Boston.

Freeman, P. & David, H. (2004), 'A science of design for software-intensive systems', *Communications of the ACM* **47**(8), 19–21.

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

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

Gîrba, T., Ducasse, S., Kuhn, A., Marinescu, R. & Daniel, R. (2007), 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', ACM, pp. 83–89.

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

Gorton, I. (2006), *Essential Software Architecture*, Springer-Verlag.

Hochstein, L. & Lindvall, M. (2005), 'Combating architectural degeneration: A survey', *Information and Software Technology* **47**, 643–656.

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

Kim, M., Sazawal, V., Notkin, D. & Murphy, G. (2005), An empirical study of code clone genealogies, *in* 'Proceedings of the 10th European Software engineering Conference; Lisbon, Portugal', pp. 187–196.

Lanza, M. & Marinescu, R. (2006), *Object-Oriented Metrics in Practice*, Springer, Heidelberg.

Louridas, P., Spinellis, D. & Vlachos, V. (2008), 'Power laws in software', *ACM Trans. Softw. Eng. Methodol.* **18**, 1–26.

Lozano, A. & Wermelinger, M. (2008), Assessing the effect of clones of changeability, *in* 'Proceedings of the 24th IEEE International Conference on Software Maintenance; Beijing, China', pp. 227–236.

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

Macia, I. (2013), On the Detection of Architecturally-Relevant Code Anomalies in Software Systems, PhD thesis, Pontifical Catholic University of Rio de Janeiro, Informatics Department.

Macia, I., Arcoverde, R., Cirilo, E., Garcia, A. & Staa, A. (2012*a*), Supporting the identification of architecturally-relevant code anomalies, *in* 'Proceedings of the 28th IEEE International Conference on Software Maintenance; Trento, Italy', pp. 662–665.

Macia, I., Arcoverde, R., Garcia, A., Chavez, C. & Staa, A. (2012*b*), On the relevance of code anomalies for identifying architecture degradation symptoms, *in* 'Proceedings of the 16th European Conference on Software Maintenance and Reengineering; Szeged, Hungary', pp. 277–286.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N. & Staa, A. (2012*c*), Are automatically-detected code anomalies relevant to architectural modularity? An exploratory analysis of evolving systems, *in* 'Proceedings of the 11st International Conference on Aspect-Oriented Software Development; Postdam, Germany', pp. 167–178.

Mara, L., Honorato, G., Dantas, F., Garcia, A. & Lucena, C. (2011), Hist-inspect: A tool for history-sensitive detection of code smells, *in* 'Proceedings of the 10th annual International Conference on Aspect-oriented Software Development; Porto de Galinhas, Brazil', pp. 65–66.

Marinescu (2004), Detection strategies: metrics-based rules for detecting design flaws, *in* 'Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM); Chicago, USA', pp. 350–359.

Martin, R. (2002), *Agile Principles, Patterns, and Practices*, Prentice Hall, New Jersey.

Marwan, A. & Aldrich, J. (2009), Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations, *in* 'Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications; Orlando, USA', pp. 321–340.

Mattmann, C., Crichton, D., Medvidovic, N. & Hughes, S. (2006), A software architecture-based framework for highly distributed and data intensive scientific applications, *in* 'Proceedings of the 28th International Conference on Software Engineering: Software Engineering Achievements Track; Shanghai, China', pp. 721–730.

Mehta, N., Medvidovic, N. & Phadke, S. (2000), Towards a taxonomy of software connectors, *in* 'Proceedings of the 22nd International Conference on Software Engineering (ICSE); Limerick, Ireland', pp. 178–187.

Moha, N., Gueheneuc, Y., Duchien, L. & Meur, A. L. (2010), 'Decor: A method for the specification and detection of code and design smells', *IEEE Transaction on Software Engineering* **36**, 20–36.

Morgan, C. (2007), A static aspect language of checking design rules, *in* 'Proceedings of the 6th international conference on Aspect-oriented software development; Vancouver, Canada', pp. 63–72.

Murphy-Hill, E. & Black, A. P. (2010), An interactive ambient visualization for code smells, *in* 'Proceedings of the 5th international symposium on Software visualization; Salt Lake City, USA', ACM, pp. 5–14.

Oizumi, W. & Garcia, A. (2015), 'Organic: A prototype tool for the synthesis of code anomalies'. URL `http://wnoizumi.github.io/organic/`.

Oizumi, W., Garcia, A., Colanzi, T., Ferreira, M. & Staa, A. (2014*a*), When code-anomaly agglomerations represent architectural problems? An exploratory study, *in* ‘Proceedings of the 2014 Brazilian Symposium on Software Engineering (SBES); Maceio, Brazil’, pp. 91–100.

Oizumi, W., Garcia, A., Colanzi, T., Staa, A. & Ferreira, M. (2015), ‘On the relationship of code-anomaly agglomerations and architectural problems’, *Journal of Software Engineering Research and Development* **3**(1), 1–22.

Oizumi, W., Garcia, A., Sousa, L., Albuquerque, D. & Cedrim, D. (2014*b*), Towards the synthesis of architecturally-relevant code anomalies, *in* ‘Proceedings of the 11th Workshop on Software Modularity; Maceio, Brazil’, pp. 39–52.

Oizumi, W., Garcia, A., Sousa, L., Cafeo, B. & Zhao, Y. (2016), Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems, *in* ‘The 38th International Conference on Software Engineering; Austin, USA (Submitted)’.

Olbrich, S. M., Cruzes, D. S. & Sjoberg, D. I. K. (2010), Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems, *in* ‘Proceedings of the 26th IEEE International Conference on Software Maintenance; Timisoara, Romania’, pp. 1–10.

Oracle (2015), ‘Java 7 programming language’. URL `http://www.oracle.com/java`.

Perry, D. E. & Wolf, A. L. (1992), ‘Foundations for the study of software architecture’, *ACM Software Engineering Notes* **17**, 40–52.

Rapu, D., Ducasse, S., Gîrba, T. & Marinescu, R. (2004), Using history information to improve design flaws detection, *in* ‘Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on; Tampere, Finland’, IEEE, pp. 223–232.

Ratzinger, J., Fischer, M. & Gall, H. (2005), *Improving evolvability through refactoring*, Vol. 30, ACM.

Schach, S., Jin, B., Wright, D., Heller, G. & Offutt, A. (2002), ‘Maintainability of the linux kernel’, *Software, IEE Proceedings -* **149**(1), 18–23.

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

Sjobert, D., Yamashita, A., Anda, B., Mockus, A. & Dyba, T. (2013), 'Quantifying the effect of code smells on maintenance effort', *IEEE Transaction on Software Engineering* **39**, 1144–1156.

Soares, S., Laureano, E. & Borba, P. (2002), Implementing distribution and persistence aspects with aspectj, *in* 'Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications; Seattle, USA', ACM Press, pp. 174–190.

Suryanarayana, G., Samarthyam, G. & Sharmar, T. (2014), *Refactoring for Software Design Smells: Managing Technical Debt*, Morgan Kaufmann.

Taylor, R., Medvidovic, N. & Dashofy, E. (2009), *Software Architecture: Foundations, Theory, and Practice*, Wiley Publishing.

Ubayashi, N., Nomura, J. & Tamai, T. (2010), Archface: A contract place where architectural design and code meet together, *in* 'Proceedings of the 32nd International Conference on Software Engineering; Cape Town, South Africa', pp. 75–84.

Vale, G., Albuquerque, D., Figueiredo, E. & Garcia, A. (2015), Defining metric thresholds for software product lines: A comparative study, *in* 'Proceedings of the 19th International Conference on Software Product Line; Nashville, Tennessee', SPLC '15, ACM, New York, NY, USA, pp. 176–185.

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

Wettel, R. & Lanza, M. (2008), Visually localizing design problems with disharmony maps, *in* 'Proceedings of the 4th ACM symposium on Software visualization', ACM, pp. 155–164.

Wong, S., Cai, Y., Kim, M. & Dalton, M. (2011), Detecting software modularity violations, *in* 'In Proceedings of the 33rd International Conference on Software Engineering; Honolulu, USA', pp. 411–420.

Yamashita, A. & Moonen, L. (2013), Exploring the impact of inter-smell relations on software maintainability: an empirical study, *in* 'Proceedings of the 35th International Conference on Software Engineering; San Francisco, USA', pp. 682–691.

Young, T. J. (2005), Using aspectj to build a software product line for mobile devices. MSc dissertation, *in* 'University of British Columbia, Department of Computer Science', pp. 1–6.