Oizumi *et al. Journal of Software Engineering Research and Development* (2015) 3:11
DOI 10.1186/s40411-015-0025-y

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

On the relationship of code-anomaly agglomerations and architectural problems

Willian N. Oizumi[1*], Alessandro F. Garcia[1], Thelma E. Colanzi[2], Manuele Ferreira[1] and Arndt V. Staa[1]

*Correspondence: woizumi@inf.puc-rio.br
[1]OPUS Group - Informatics Department, PUC-Rio, Marques de Sao Vicente Street, 225, 22451-900 Rio de Janeiro, Brazil
Full list of author information is available at the end of the article

Abstract

Several projects have been discontinued in the history of the software industry due to the presence of software architecture problems. The identification of such problems in source code is often required in real project settings, but it is a time-consuming and challenging task. Some authors assume that architectural problems are reflected in source code through individual code anomalies. However, each architectural problem may be realized by multiple code anomalies, which are intertwined in several program elements. The relationships of these various code anomalies and their architecture problems' counterparts are hard to reveal and characterize. To overcome this limitation, we are studying the architecture impact of a wide range of code-anomaly agglomerations. An agglomeration is a group of code anomalies that are explicitly related to each other in the implementation – e.g. two or more anomalies affecting the same class or method in the program. In our empirical study, we analyzed a total of 5418 code anomalies and 2229 agglomerations within 7 systems. In particular, our analysis focused in understanding (i) how agglomerations and architectural problems relate to each other, and (ii) how agglomerations can support the diagnosis of well-known architectural problems. We observed that most of the anomalous code elements related to architectural problems are members of one or more agglomerations. In addition, this study revealed that, for each agglomeration related to an architectural problem, an average of 2 to 4 anomalous code elements contribute to the architectural problem. Finally, the result of our study suggests that certain types of agglomerations are better indicators of architectural problems than others.

Keywords: Code anomaly; Architectural problem; Source code analysis

1 Introduction

Several projects have been discontinued in the history of the software industry due to the presence of software architecture problems (Garcia et al. 2009; Hochstein and Lindvall 2005; Macia et al. 2012; Macia 2013) Such problems are caused by architectural design decisions that negatively impact the resulting system's quality (Garcia et al. 2009). Even though software architecture drives software development in real project settings, architectural design is rarely formally documented (Macia et al. 2012). As a result, analysis of architectural problems cannot be performed with existing documentation-driven techniques (Eichberg et al. 2008; Marwan and Aldrich 2009 Morgan 2007; Ubayashi et al. 2010). Hence, evidence of architectural problems has to be identified based on the source code analysis Macia et al. (2012). Along years of research on the

software quality field, different studies (Fowler 1999; Lanza and Marinescu 2006) have agreed with the idea that source code anomalies, such as Long Methods and God Classes (Fowler 1999), are relevant indicators of different kinds of maintainability problem, including architectural problems.

Our recent studies (Macia 2013; Macia et al. 2012a,b) started to investigate the relation of code anomalies and architectural problems. However, these studies revealed the relationships of code anomalies and their architectural problems' counterparts are hard to understand and characterize (Macia 2013; Macia et al. 2012a,b). As an architectural component is often implemented by several code elements, the aim of identifying architectural problems might not be fulfilled by analyzing individual code anomalies in isolation (Macia et al. 2012; Macia 2013; Macia et al. 2012b). As an architectural problem is likely to affect several elements of the implementation, it might be the presence of two or more code anomalies better indicate an architectural problem (Macia et al. 2012,b). It might be even the case that these inter-related code anomalies more often indicate architectural problems than individual code anomalies. Unfortunately, there is little knowledge about the manifestation of inter-related code anomalies in software projects. A few studies investigated when inter-related code anomalies adversely affect the software architecture (Macia 2013; Oizumi et al. 2014). Nevertheless, to the extent of our knowledge, there is no work investigating the extension of the relation between architectural problems and inter-related code anomalies. In addition, existent studies do not investigate whether and how inter-related code anomalies can support the diagnosis of well-known architectural problems.

Therefore, in this paper, we present a study about the relationship between code-anomaly agglomerations and architectural problems. An agglomeration consists of a group of code anomalies that are related to each other in the source code. There are different types of relationships amongst the code anomalies involved in an agglomeration. For example, the relation between two or more code anomalies can be established through a single method shared by these anomalies. That is, a group of two or more code anomalies may simultaneously affect the same method. Therefore, in this case, we consider this group of code anomalies (affecting the same method) is an agglomeration of code anomalies. In the study, we take into consideration a categorization of four types of code-anomaly agglomerations. Each of these types determine the elements and relationships that compose its agglomerations. We analyze seven systems of different sizes (8 KSLOC to 129 KSLOC) and domains. Our sample of systems goes from older versions (e.g. version 0.2 of OODT) to newer versions (e.g. version 10 of Health Watcher). We perform several analyses to understand whether and when code-anomaly agglomerations represent architectural problems. The analyses of this study can be summarized as follows:

1. We analyze if architectural problems are reflected in code-anomaly agglomerations more often than in individual code anomalies.
2. We compute the proportion of elements in each agglomeration related to architectural problems.
3. We investigate which types of agglomerations are more (or less) related to architectural problems.
4. We discuss concrete examples of how each of these agglomeration types can be used to reveal architectural problems.

In our previous work (Oizumi et al. 2014), we have not addressed the second and fourth analyzes described above. The execution of these analyses altogether allowed us to further reinforce that agglomerations are significantly better than individual code anomalies to indicate the presence of architectural problems. In most of the target systems, agglomerations were at least twice better than individual code anomalies to indicate the presence of architectural problems; in some projects, the superiority of agglomerations was even more than five times better. As far as the relation of agglomerations and architectural problems is concerned, we observed that most of the anomalous code elements related to architectural problems are members of one or more agglomerations. Moreover, we observed that each agglomeration has, on average, 2 to 4 elements related to architectural problems. Finally, analyzing the circumstances in which agglomerations represent architectural problems, we observed that some types of agglomeration are better indicators than others. We have collected representative examples of how agglomerations can be used to support the diagnosis of significantly-different architectural problems.

The remainder of this paper is organized as follows. Section 2 contains the background and literature review. Section 3 presents the study settings. Section 3.4 presents the empirical procedures. Section 4 presents the results and findings. Finally, Section 5 summarizes our main conclusions.

2 Background and literature review

When software changes are made, the system's architecture can degrade due to different architectural problems (Hochstein and Lindvall 2005). These problems may be related to code anomalies (Fowler 1999). We refer to an individual manifestation of an architectural problem as an architectural problem instance.

2.1 Architectural problem

An architectural problem occurs due to the addition of unintended design decisions that either violate (1) the original, intended architecture of a system or (2) general software modularity principles (Perry and Wolf 1992). To illustrate the violation of an intended architecture, consider the Health Watcher architecture in Fig. 1. For each figure in this paper, we rely on a UML-like notation (Booch et al. 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 Fig. 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 Fig. 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 architectural 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 architecture of Health Watcher. However, they were implemented in the actual architecture, thus violating the intended architecture.

Architectural problems also occur when elements of an architecture violate modularity principles. A catalog of such architectural problems can be found in (Garcia et al. 2009). As an example, consider the *webgrid* component from the Apache OODT (Object Oriented Data Technology) system in Fig. 2. The *webgrid* component (1)

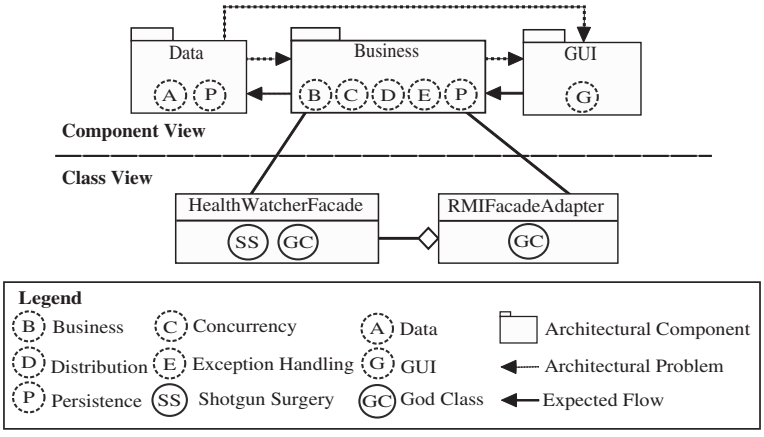

Fig. 1 Concern-based agglomeration in the Health Watcher system

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 Connector Envy problem, 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 Connector Envy problem.

2.2 Architectural problems and code anomalies

Architectural problems have caused the discontinuation or reengineering of several software projects (Hochstein and Lindvall 2005). As previously mentioned, developers frequently need to detect such problems in the source code due to the lack of formal

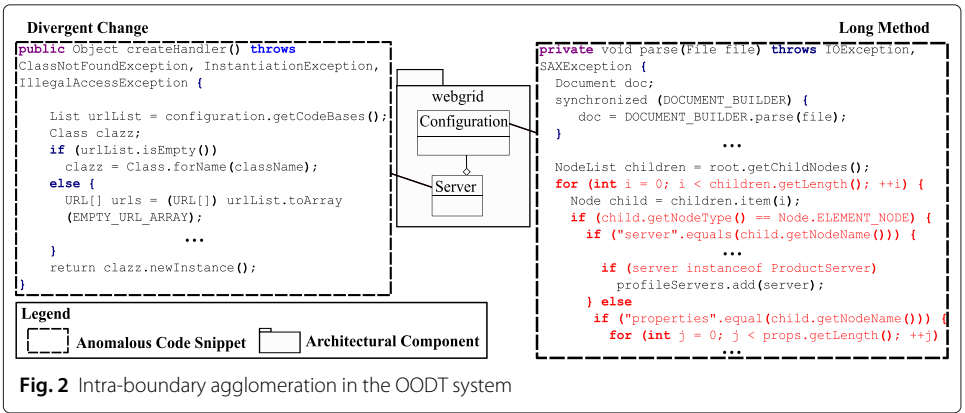

Fig. 2 Intra-boundary agglomeration in the OODT system

architecture documentation. In other words, it is expected that a wide range of architectural problems are reflected in a system's implementation through code anomalies (Fowler 1999; Hochstein and Lindvall 2005). A code anomaly, popularly known as a "bad smell", is a symptom observed in a program's structure and usually indicates deeper maintenance problems, such as architectural problems. A code anomaly negatively impacts the maintainability of classes and methods. The manifestation of a code anomaly in a code element is called a code-anomaly instance. 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 Fig. 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 Connector Envy problem from *webgrid*. As we mentioned, a group of inter-related classes of this component contribute to the same problem. Therefore, the fully removal of Connector Envy depends on the refactoring of multiple classes.

2.3 Studies on the impact of code anomalies

The impact of code anomalies has been largely studied. Khomh et al. (2009), Kim et al. (2005), Lozano et al. (2008) and Olbrich et al. (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. (2010) observed that code anomalies are often related to software defects. Sjoberg et al. (2013) showed that single code anomalies were not related to maintenance effort. Macia et al. (2012) analyzed the relevance of code anomalies to identify architectural problems. This research revealed that none of the studied code anomalies was consistently strong indicators of architecture problems. The results also revealed that a higher proportion of individual code anomalies did not impact the system architecture.

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. (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 (2013) observed that inter-related code anomalies negatively affect systems maintenance. None of the aforementioned authors investigated the relation of architectural problems and code-anomaly agglomerations. In this context, Macia (2013) observed that a specific set of nine inter-related code anomalies are better indicators of architectural problems than individual code anomalies. The results of her study revealed a statistically-significant relationship between some of these inter-related anomalies and architectural problems. However, to the extent of our knowledge, our previous work (Oizumi et al. 2014) was the first to investigate when and whether different forms of

agglomerations represent architectural problems. However, our previous study has not analyzed in details the relation between architectural problems and agglomerations. In addition, we did not present concrete scenarios where agglomerations would be useful to effectively reveal architectural problems. Therefore, this paper extends our previous work by (i) providing a deeper analysis on the relationship of architectural problems and agglomerations, (ii) presenting a more detailed description about the concept and classification of code-anomaly agglomerations, and (iii) providing a discussion on various concrete examples of agglomerations extracted from our target systems. These examples serve to indicate how agglomerations could be used to find or anticipate (the side effects of) architectural problems.

3 Methods

In this section, we present the key settings of our study. Section 3.1 presents our definition and categorization of code-anomaly agglomerations. Section 3.2 describes the research questions answered in this paper. Finally, Section 3.3 describes the target systems of our study. Section 3.4 will present more detailed procedures of our empirical study.

3.1 Code-anomaly agglomeration

Prior work (Macia et al. 2012b) has shown that current techniques are unable to detect architecturally-relevant code anomalies. In this paper, we investigate to what extent code anomaly agglomerations represent architectural problems. In order to do a thorough analysis, the first step of our study was the categorization of code-anomaly agglomerations into topologies, considering their particular characteristics. We considered four topologies of code-anomaly agglomeration: (i) intra-boundary, (ii) cross-boundary, (iii) hierarchical, and (iv) concern-based. We focused in these four topologies because they are simple to describe and understand. In addition, they can be automatically (and reliably) detected with existing tool assistance (Macia et al. 2012a). Section 3.4 carefully describes the procedures followed for data collection and analysis.

Before providing a detailed description of each topology, in Fig. 3 we present a meta-model that characterizes the concept of code-anomaly agglomeration. It also establishes the relation of this concept with other concepts relevant to our study. In the meta-model (Fig. 3), each *topology* is associated with a detection *strategy*. Conversely, each *strategy* is used by only one *topology*. Each *strategy* considers different types of *element* and *relationship*. *Element types* can be source code element (*methods* and *classes*) or architectural component. *Relationship types* can be *hierarchy*, *dependency* or *common concern*.

Each *agglomeration* has a specific *topology*. Each *agglomeration* comprises one or more *relationships* that determine the grouping of code anomalies. There may be none, one or more agglomerations of each *topology* type in a program. Moreover, each agglomeration is composed by one or more code anomalies. In addition, each *agglomeration* uses one or more *relationships* to group *code anomalies*. Each *code anomaly* affects a specific *code element*, which may be affected by several *code anomalies*. A *code element* may realize multiple *architectural problems*. Conversely, an *architectural problem* is realized by one or multiple *code elements*. Finally, an *agglomeration* may help to diagnose an *architectural problem* by grouping anomalous *code elements* that collaborate to the realization of an *architectural problem*. We present below a detailed description of each topology accompanied by illustrative examples.

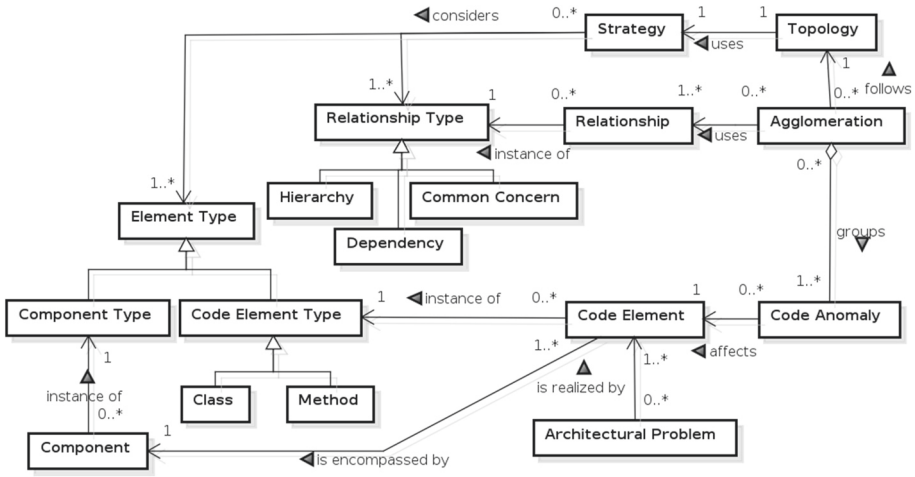

Fig. 3 Code-anomaly agglomeration meta-model

3.1.1 Intra-boundary agglomeration

It is an agglomeration composed of anomalous code elements that implement the same architectural component. Falls into this topology the agglomerations located within a single component and composed by: (1) inner anomalous code elements that are syntactically related or (2) inner anomalous code elements infected by the same type of code anomaly. In both cases, none of the code elements takes part in the implementation of a different architectural component. In the second case, there is no syntactical relationship between the anomalous code elements. To understand what we consider a syntactic relationship, assume we have two classes, T and X, and two methods M and N. The classes T and X are syntactically related if X is referenced inside T or vice versa. The methods M and N are syntactically related if M calls N or vice versa. We do not consider the inheritance relationship, because it characterizes the occurrence of a hierarchical topology. Figure 2 shows an example of intra-boundary agglomeration. In this case, the method *createHandler* is infected by Divergent Change and the method *parse* is infected by Long Method. As both of them are enclosed by the same architectural component (*webgrid*) and there is an association between them, the *createHandler* and *parse* elements form an intra-boundary agglomeration.

3.1.2 Cross-boundary Agglomeration

It is an agglomeration composed of anomalous code elements syntactically related, but located in the implementation of different architectural components. The syntactical relationships considered are the same as for the intra-boundary agglomerations. A cross-boundary agglomeration always involve two or more components. There might be more than one anomalous code element in each of the involved components. However, there must be at least one anomalous element within each agglomeration's component (two or more). Agglomerations composed by code elements of a single component falls into the intra-boundary category. As an example of cross-boundary agglomeration, consider the architecture diagram from OODT in Fig. 4. This diagram presents a partial view from four components of OODT: *metadata*, *crawler*, *filemgr* and *pushpull*. The code

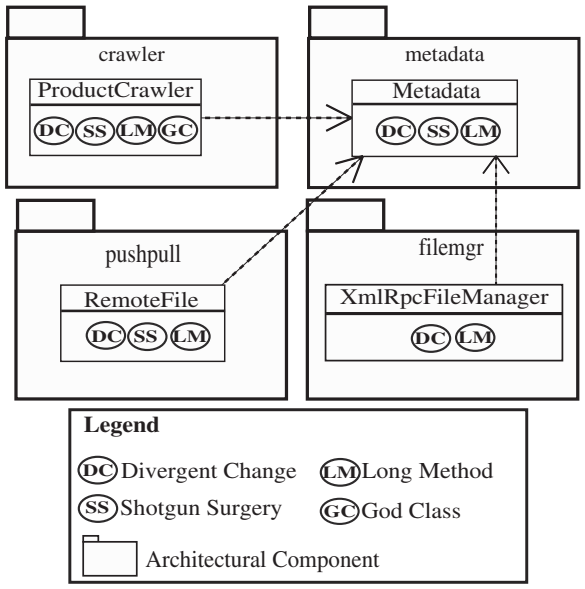

Fig. 4 Cross-boundary agglomeration in the OODT system

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

3.1.3 Hierarchical agglomeration

It is an agglomeration composed of anomalous code elements that take part in the same inheritance tree. The hierarchies might occur either in the implementation of a single component or across multiple components. In this topology, we consider only hierarchies where the code elements are infected by the same type of code anomaly. The recurrent introduction of the same code anomaly in different code elements may represent a bigger problem in the hierarchy. An example of this agglomeration topology is in the *Versioner* hierarchy (Fig. 5). All the classes of *Versioner* are affected by the same anomalies – Divergent Change, Feature Envy, Long Method and Shotgun Surgery – thus, they can be grouped into a hierarchical agglomeration.

3.1.4 Concern-based agglomeration

It is an agglomeration composed of anomalous code elements correlated through architectural concerns. On the one hand, an architectural component only represents the overall purpose of its classes, i.e. their main purpose. On the other hand, classes of each component might be realizing other multiple and specific concerns, which are not the main purpose of the component. As a consequence, each architectural concern might not be well modularized. When this phenomenon happens, an architectural concern is implemented by several classes in a single or multiple components. Classes comprising

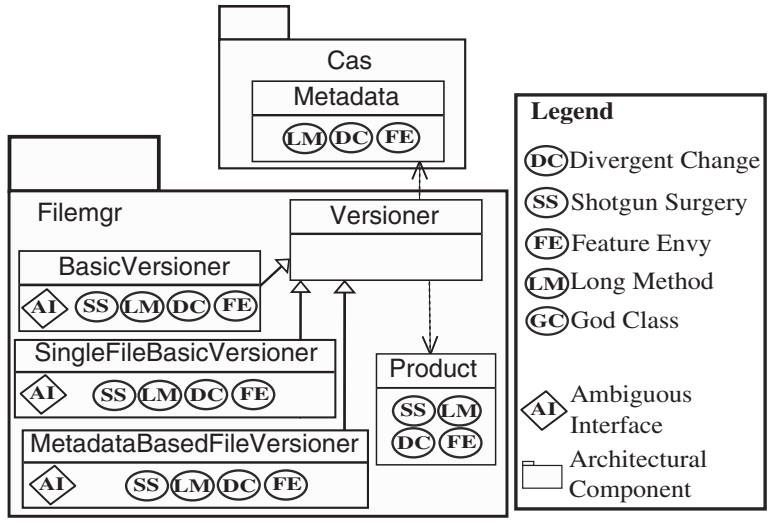

Fig. 5 Hierarchical agglomeration in the OODT system

a concern-based agglomeration may be grouped through: (i) an overload of concerns in their enclosing component, or (ii) through a single concern that is spread across several code elements in which it is not the primary concern.

The concern-based topology differs from others by also considering the purposes (concerns) associated with the realization of architecture components. Figure 1 shows a partial view from the Health Watcher architecture. Characters represent the concerns addressed by the code elements of *Business* and 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 *Business* is responsible for realizing several concerns, such as *Persistence* and *Exception Handling*, which are not related to its main concern. Hence, the anomalous code elements of *Business* are inter-related through an overload of concerns.

3.2 Research questions

Previous research (Macia 2013) showed that code-anomaly agglomerations may help to reveal architecturally-relevant code anomalies. However, there is little knowledge about the relationship between agglomerations and architectural problems. Therefore, we aim at expanding existing literature about code-anomaly agglomerations. To this end, we focus the degree of the relationship involving code-anomaly agglomerations and architectural problems. We address three particular research questions associated with this general objective:

RQ1. Are architectural problems reflected in code-anomaly agglomerations more often than in individual code anomalies?
RQ2. Are agglomerations fully related to architectural problems?
RQ3. In which circumstances agglomerations are related (or not) to architectural problems?

RQ1 compares the relevance of code-anomaly agglomerations against individual code anomalies. If agglomerations represent more often architectural problems, it means

developers need complementary techniques to reveal them in the source code. RQ2 measures the proportion of elements in each agglomeration related to architectural problems. This question is addressed by investigating the number of elements in each agglomeration realizing architectural problems. With RQ2, we intend to investigate the extension of the relation between each agglomeration with respect to architectural problems. In this context, the extension is measured by the proportion of elements in each agglomeration contributing to architectural problems. Finally, RQ3 investigates whether specific agglomeration topologies are better indicators of architectural problems than others. This investigation enables us to reveal when the topology of an agglomeration contributes to the detection of architectural problems in the source code.

3.3 Target systems

The focus of this study is to investigate the relationship between architectural problems and code-anomaly agglomerations. In order to study this relationship, we analyzed systems with a wide range of characteristics. Aiming at reducing the influence of particular software characteristics in the results, we selected systems of different sizes (8 KSLOC to 129 KSLOC), leveraging different architectural styles, and spanning different domains. We studied 7 systems, of which 2 are academic and 5 are from industry. Table 1 summarizes the general characteristics of each target system. The first 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 stock of oil, while S4 is intended to support the financial market analysis. The next is Mobile Media (MM), an academic software product line to derive applications that manipulate photos, videos and music on mobile devices (Mobile Media Source Code 2015; Young 2005). The next is Health Watcher (HW), a web framework system, whose objective is to allow citizens to register complaints regarding health issues in public institutions (Health Watcher Source Code 2015; Soares et al. 2002). The last system is Apache OODT (Apache OODT Source Code 2015), whose goal is to develop and promote the management and storage of scientific data (Mattmann et al. 2006). For all the target systems, several classes implement each component. In OODT, for example, each component is implemented by an average of 24 classes.

3.4 Empirical procedures

In this section, we present the empirical procedures of our study. Section 3.5 presents the procedures to detect code anomalies and agglomerations. Section 3.6 presents the procedures to identify architectural problems. Finally, Section 3.7 presents the procedures to analyze the relation of architectural problems and agglomerations.

Table 1 Target systems. Characteristics of each target system

System	Application type	Architectural design	KSLOC
S1	Desktop Application	Client-Server	122
S2	Desktop Application	Client-Server	118
S3	Desktop Application	Client-Server	93
S4	Web Application	MVC	116
MM	Software Product Line	MVC	10
HW	Web Framework	Layers	8
OODT	Middleware	Components and Connectors	129

3.5 Detecting code anomalies and agglomerations

This task was accomplished using detection strategies (Lanza and Marinescu 2006) similar to those used in related studies. 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, Macia 2013). The process of detection was undertaken with the assistance of a tool called SCOOP (Macia et al. 2012a). SCOOP was designed to use architectural information to identify code anomalies. It uses architecturally-sensitive metrics and strategies to try to identify architecturally-relevant code anomalies. We collected the metrics using a third-party tool called Understand (Understand: User Guide and Reference Manual 2015).

SCOOP does not detect a fixed set of code anomalies. Instead, it provides a DSL for the specification of detection strategies. That is, using SCOOP it is possible to select the metrics and thresholds for each code anomaly. In this study, we specified metrics for 6 code anomalies (Table 2). As the selected anomalies are widely discussed elsewhere, readers may refer to (Fowler 1999) and (Lanza and Marinescu 2006) for details about each of them.

SCOOP presents two different outputs. The first is a list of code anomaly instances identified in the system. As we mentioned, SCOOP detects only code anomalies specified by the user. The second output is composed by groups of inter-related code anomalies. In this study, we categorized the inter-related code anomalies into four topologies (Section 3.1), according to their characteristics. To the extent of our knowledge, SCOOP is the only tool that exploits the several forms of relationships between code anomalies.

3.6 Identifying architectural problems

Given that this task had to be performed manually, we tried to avoid mistakes by involving architects of the target systems in the whole process. For all the target systems, the identification of architectural problems was performed using the source code and the intended architecture. For systems without architectural documentation, we relied on a suite of architecture recovery tools (Garcia et al. 2013). Original architects of the target systems assisted us in all the steps of this task. The procedure for deriving the list of architectural problems with architects was the following: (i) an initial list of architectural problems was identified using detection strategies presented in (Macia 2013), (ii) the architects had to confirm, refute or expand the architectural problems identified, (iii) the architects provided a brief explanation to the researcher on the (ir)relevance of the architectural problem, and (iv) when we suspected there was still inaccuracies in the confirmed list of architectural problems, we asked the architects for further feedback. In Table 3, we briefly describe each type of architectural problem identified in this procedure.

Table 2 Code anomalies

Code anomaly	References
Data Class	(Lanza and Marinescu 2006)
Divergent Change	(Fowler 1999; Lanza and Marinescu 2006)
Feature Envy	(Fowler 1999; Lanza and Marinescu 2006)
God Class	(Lanza and Marinescu 2006)
Long Method	(Fowler 1999; Lanza and Marinescu 2006)
Shotgun Surgery	(Fowler 1999; Lanza and Marinescu 2006)

Due to 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 architectural problems have 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 architects 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 architectural problems was validated with one of the leading architects. The leading architect 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 architectural problems, following the same procedure followed with other systems.

3.7 Analyzing the relation of Architectural Problems and Agglomerations

Aiming to analyze the different agglomeration topologies and how they are related to architectural problems, we defined the research questions described in Section 3.2. To answer our research questions, the criteria used for correlating code-anomaly agglomerations and single code anomalies with architectural problems were the following. A code-anomaly agglomeration and an architectural problem are related if they co-occur in, at least, one code element. Even though both agglomeration and architectural problem usually involve many elements, in our definition is sufficient that they co-occur in one element. Similarly, an individual code anomaly and an architectural 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 architectural 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 architecture, i.e., we analyzed how the architectural problems are related to code-anomaly agglomerations and individual code anomalies. Conversely, in

Table 3 Architectural problem types

Name	Description
Ambiguous Interface	Interface that offers only a single, general entry-point, but provides two or more services.
Architectural Violation	Dependency that violates an intended architectural rule.
Connector Envy	Component that encompasses extensive interaction-related functionality that should be delegated to a connector.
Concern Overload	Component that is responsible for realizing two or more unrelated system's concerns.
Cyclic Dependency	Two or more components that directly or indirectly depend on each other to function properly.
Extraneous Connector	Two connectors of different types that are used to link the same pair of components.
Scattered Functionality	Multiple components that are responsible for realizing the same high level concern, with some of them responsible for orthogonal concerns.
Unused Interface	Interface that is never used by external components.

the upstream analysis we analyzed how agglomerations and individual code anomalies are related to architectural problems. Furthermore, to answer research question RQ2, we analyzed the relation between architectural problems and agglomerations. To measure the extension of this relation, we conducted two complementary analyses. First, we calculated the average of anomalous elements, which take part in individual agglomerations and directly contribute to an architectural problem. Second, we analyzed the proportion of anomalous elements related to architectural problems and taking part in one or more agglomerations. To answer RQ3 we compared the number of agglomerations related to architectural problems in each topology. Finally, to illustrate our findings, we selected insightful examples from our target systems.

4 Results and discussion

This section presents the findings of our study. Section 4.1 presents a comparison between individual code anomalies and code-anomaly agglomerations. Section 4.2 presents an analysis on the extension of the relation between architectural problems and agglomerations. Section 4.3 discusses in which specific circumstances agglomerations are related to architectural problems. Finally, Section 4.4 presents threats to validity of this study.

4.1 Comparing anomaly instances vs. agglomerations

This section addresses the question RQ1: "Are architectural problems reflected in code-anomaly agglomerations more often than in individual code anomalies?" A previous study have revealed that architectural problems are related to individual code anomalies (Macia et al. 2012). At the same time, many code anomalies had no relation to architectural problems (Macia et al. 2012). Then, the answer to RQ1 would enable us to understand whether the analysis of code-anomaly agglomerations may improve the detection of architectural problems. As mentioned in Section 3.4, we performed the analysis from two perspectives: downstream and upstream.

4.1.1 Downstream perspective

The first two columns of Table 4 present the results of the downstream analysis. The first column (AG) presents, for each system, the proportion of architectural problems related to agglomerations. The second column (CA) shows the proportion of architectural problems related to single code anomalies. As it can be seen, the relationship of architectural problems and agglomerations was always above 57 % and in most cases above 70 %. The proportions of architectural 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 reveals that the results were very consistent: in all the systems, architectural 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 architectural problems unrelated to agglomerations were also unrelated to individual code anomalies.

Table 4 Relation of architectural problems with agglomerations and code anomalies

System	Downstream Architectural problem		Upstream Architectural 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 %

AG = Code-anomaly agglomeration; CA = Individual code anomaly

4.1.2 Upstream perspective

Next, we investigated if the observation of agglomerations in the source code can help to indicate the presence of architectural problems. To this end, we compared the upstream relationship of code-anomaly agglomerations and individual code anomalies to architectural problems. The Upstream AG (agglomerations) and CA (individual code anomalies) columns of Table 4 show that agglomerations presented even better results from this perspective. With the exception of OODT, the proportion of agglomerations related to architectural 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 architectural problems was lower than 25 %. These results further reinforce the potential of exploring code-anomaly agglomerations as indicators of architectural problems.

The two analyses above provide evidence that agglomerations are more helpful than individual code anomalies to diagnose architectural problems. However, these analysis are not enough to determine if architectural problems and agglomerations have a strong relation. Next, in Section 4.2, we address this issue by further analyzing the relation between agglomerations and architectural problems.

4.2 Agglomerations and architectural problems: are they fully related?

The previous research question does not consider the fact that code-anomaly agglomerations have higher probability of being related to architectural problems. The probability increase stems from the fact that each agglomeration involves more code elements than a single code anomaly. Then, it would be interesting to analyze to what extent various code elements of the agglomeration are, in fact, contributing to the realization of an architectural problem. In order to address this concern, we performed a further analysis about the relation between agglomerations and architectural problems. As a first step, for each system, we separated the anomalous code elements related to architectural problems in two sets: (1) anomalous code elements participating in one or more agglomerations; and (2) anomalous code elements not participating in any agglomeration. The union of the two groups forms the set of all anomalous code elements involved in architectural problems.

Table 5 shows, for each system, the number of anomalous elements related to architectural problems. The AG column represents how many of these elements take part in agglomerations. The NoAG column shows the number of anomalous elements not

involved in agglomerations. As we can observe, for each system, the number of anomalous code elements encompassed by agglomerations and related to architectural problems was substantially much higher.

Some code anomalies related to architectural problems are member of more than one agglomeration. However, we do not consider that those individual anomalies are better than agglomerations to diagnose architectural problems. Even though the effort to detect agglomerations is higher, we observed that the vast majority of the individual agglomerations had from 2 to 4 anomalous code elements simultaneously related to the same single architectural problem in all systems. Almost 98 % of the anomalies, taking part in agglomerations, were related to architectural problems. It means that, when fixing a particular architectural problem, the developer can reveal which set of anomalous code elements need to be involved in the refactoring strategy. In addition, the scope of analysis is reduced to the list of code anomalies taking part in the agglomeration (rather than a huge list of individual anomalies). All these results reinforce the observation that agglomerations are almost fully related to architectural problems and are very likely to contribute to the diagnoses of architectural problems.

To illustrate this relation between architectural problems and agglomerations, consider the *webgrid* component depicted in Fig. 2. As we already discussed, *webgrid* suffers from the Connector Envy problem (Section 2) and contains an intra-boundary agglomeration (Section 3.1). Analyzing the anomalous elements of the intra-boundary agglomeration, we observed that all of them contribute to the Connector Envy problem. That is, all elements in the agglomeration mix their main functionalities with interaction services. This example reinforce the findings of this section: agglomerations are useful to diagnose architectural problems since each of them reveals multiple anomalies that realize a unique architectural problem. Even in the agglomerations where some anomalies do not contribute to the architectural problem, the agglomeration is helpful to developers since it (i) reduces the scope of analysis and (ii) provides information about code elements that may be affected when removing the architectural problem.

4.3 Relevance of agglomeration topologies

We investigated the architectural relevance of specific agglomeration topologies in terms of their likelihood of indicating architectural problems. We selected a sub-set of systems aiming to illustrate the key findings with respect to the RQ3: "In which circumstances agglomerations are related (or not) to architectural problems?" To answer this question, we analyzed circumstances defined by the four agglomeration topologies considered in

Table 5 Anomalous code elements involved in architectural problems

System	AG	NoAG
S1	157	123
S2	287	203
S3	192	92
S4	87	5
MM	23	4
HW	62	19
OODT	212	92

AG = Encompassed by agglomeraions; NoAG = Not encompassed by agglomerations

this paper (Section 4.3.1). We have also collected concrete examples on how the use of agglomerations can help to indicate the presence of architectural problems (Section 4.3.2).

4.3.1 Architecturally-relevant topologies

Table 6 shows, for each agglomeration topology (line): (A-AP) the number of agglomerations related to some architectural problems, and (A-NoAP) the number of agglomerations not related to architectural problems. Across all systems, the concern-based agglomerations presented, proportionally, the highest correlation with architectural problems. In OODT, 48 out of 53 instances (90 %) of the concern-based topology were related to architectural problems. In HW, 3 out of 4 concern-based instances (75 %) were related to architectural problems. In S4 the 4 instances were related to architectural problems. Finally, considering all the systems 153 out of 183 concern-based instances (83 %) were related to architectural problems. In terms of absolute values, the cross-boundary and intra-boundary topologies presented the highest number of architecturally-relevant instances. Despite the high number of instances, there were several cross-boundaries and intra-boundaries instances not related to architectural problems. In MM for example, more than 50 % (7 out of 11) of the cross-boundaries agglomerations were not related to architectural problems. This means that, analyzing MM, a developer or an architect could have to inspect more than 50 % of the cross-boundaries agglomerations to find one architectural problem.

Summarizing, we observed that the concern-based topology is the best indicator of architectural problems. Even though the intra-boundary and cross-boundary topologies identify higher numbers of architectural problems, several instances of them are unrelated to architectural problems. The hierarchical topology presented a low number of instances in most of the target systems. However, analyzing individual instances, we found interesting examples of hierarchical agglomerations related to architectural problems. The examples are presented and further explored in Section 4.3.2.

4.3.2 Diagnosing architectural problems with agglomerations: examples

In this study we collected evidence that (i) code-anomaly agglomerations are related to architectural problems more often than single code anomalies; (ii) most of the agglomerations are almost fully related to architectural problems, and (iii) the concern-based topology is more related to architectural problems than other topologies. Next, to illustrate these findings, we present concrete examples (extracted from the analyzed systems) of how different agglomeration types can support the diagnosis of architectural problems.

Table 6 Number of agglomerations related and unrelated to architectural problems

Agglomeration topology	A-AP	A-NoAP
intra-boundary	506	234
cross-boundary	748	467
hierarchical	69	22
concern-based	153	30
TOTAL	1476	753

A-AP = Agglomeration instances related to at least one architectural problem; A-NoAP = Agglomeration instances unrelated to architectural problems

Wide architectural problem in mobile media During the evolution of the Mobile Media system, the *Controller* component suffered a major refactoring in the system's sixth version. This refactoring affected the vast majority of the other system components. The *Controller* component had to be decomposed into three components. This "wide" refactoring had to be made: it was no longer possible to maintain all the emerging controller responsibilities in a single component. Until then, both the interface and the classes realizing the component in the program were being artificially changed 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 components, which were realizing other requirements. In other words, there were increasingly-critical ripple effects being observed as the system evolved. This overload of adjacent responsibilities in the *Controller* component reflected in the architecture as the Concern Overload and Scattered Functionality problems (Garcia et al. 2009).

In this context, we observed an instance of concern-based agglomeration (see concern-based topology in Section 3.1) involving the *Controller* component occurring in the first version of Mobile Media. In other words, the concern-based agglomeration of code anomalies could be used in the first version to early reveal the manifestation of a congenital architectural problem. This agglomeration provided useful information about how different anomalies were contributing to such a major architectural problem. Then, the high cost involving this particular case of wide architectural refactoring could be avoided if developers had reasoned about the manifestation of the concern-based agglomeration while developing the first version. We observed that similar cases of early manifestation of code-anomaly agglomerations also occurred often in the other systems.

Architectural problem in a single component Another interesting case of usage of code-anomaly agglomerations occurred in the OODT system. Differently from the example above, we observed an architectural problem instance confined to the *RPCWorkflowManager* component. This component encloses two main classes: *XmlRpcWorkflowManager* and *XmlRpcWorkflowManagerClient*. Both classes contribute to the realization of an architectural problem called Connector Envy (Garcia et al. 2009). Connector Envy indicates that the component's implementation mix its main functionalities with connector functionalities. This problem can dramatically reduce the component's maintainability. The mixed functionalities force developers to reason about two or more different concepts while changing a single class. This can hinder developers, leading them (i) to make the code structure more complex or (ii) to introduce bugs. In a system such as OODT it would not be easy to diagnose this problem. OODT contains several complex functionalities implemented by hundreds of classes. As a result, diagnose architectural problems in this system requires the analysis of several classes. Due to the large scope of analysis, this task would be exhaustive and time consuming. This scenario often leads developers to neglect the diagnose of architectural problems.

For architectural problems like Connector Envy, it is possible to help developers by revealing intra-boundary agglomerations. In our example, *XmlRpcWorkflowManager* and *XmlRpcWorkflowManagerClient* compose an intra-boundary agglomeration because (i) they are enclosed by the same component (*RPCWorkflowManager*) and (ii) both of them are God Classes. This combination of God Classes indicates that the main classes of *RPCWorkflowManager* are highly complex and implement multiple functionalities. This

combination of complex classes is an evidence of a bigger problem. Knowing that both classes are complex and implement multiple functionalities, a developer could diagnose the Connector Envy problem in *RPCWorkflowManager*. It is important to note that the analysis of code anomalies in isolation would not be as helpful as the analysis of agglomerations. Developers would have to manually inspect several anomalies, mentally trying to identify possible relations between them.

Architectural problem in a hierarchy In the OODT data-grid sub-system, the *Versioner* hierarchy is responsible for managing and storing versions of different *Product* types using different 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* only deals with "flat" products). Consequently, the *Product* type handled by each *Versioner* implementation cannot be discovered from the *createDataStoreReferences* interface. Hence, according to the OODT developers, the *Versioner* implementations were realizing Ambiguous Interfaces (Garcia et al. 2009), which are interfaces that expose multiple functionalities through a general interface.

While an Ambiguous Interface decouples components and simplifies 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. Therefore, the architectural problem cannot be simply detected by observing a single code anomaly. In fact, the analysis of code anomalies in isolation would not be useful to detect architectural problems, such as Ambiguous Interfaces. However, as we already discussed, the *Versioner* hierarchy contains multiple occurrences of a hierarchical code-anomaly agglomeration. More specifically, in several classes the *createDataStoreReferences* method is affected by the Shotgun Surgery, Long Method, and Feature Envy code anomalies. These anomalies in isolation may not represent severe problems. Nevertheless, they may indicate a deeper problem in the system when they are inter-related. In our example, the hierarchical agglomeration could help developers to early diagnose and remove the Ambiguous Interface problem, by revealing the recurring introduction of anomalies in different *Versioner* implementations. The early removal of the Ambiguous Interface problem would prevent the problem to grow during the addition of new *Versioner* implementations. We observed that the most relevant instances of hierarchical agglomeration occurred in system such as OODT, which uses polymorphism to implement abstractions. This suggests that hierarchical agglomerations are more suitable to diagnose architectural problems in systems with intensive use of polymorphism.

Usefulness of agglomerations In this section we highlighted several situations in which code anomaly agglomerations are useful to diagnose architectural problems. One of the given examples showed that some architectural problems are difficult to diagnose due to being spread through diverse unrelated modules. In these cases, our findings suggests that the Concern-based agglomerations are the most helpful in diagnosing architectural

problems. On the other hand, other agglomerations may be useful in other situations. For example, hierarchical agglomerations showed to be useful to reveal architectural problems that involves several classes in a unique hierarchy.

4.4 Threats to validity

Construct validity The construct validity is threatened mainly by possible errors introduced in the identification of architectural problems, code anomalies and code-anomaly agglomerations. Recent studies suggests that there are no metric's threshold that applies to every project, since software metric values usually follow heavy-tailed distributions (Baxter et al. 2006; Louridas et al. 2008). Therefore, in order to mitigate this threat for code anomalies, we selected detection strategies and thresholds that presented satisfactory results in a previous study (Macia et al. 2012; Macia 2013). For code-anomaly agglomerations, we relied on the only known tool that is able to identify agglomerations using architectural information. The use of a tool to detect code anomalies and agglomerations is another threat, since there is no code anomaly tool with 100 % of accuracy. The choice of SCOOP mitigates this threat since SCOOP presented near 80 % of accuracy for code anomaly detection in a previous study (Macia 2013). This level of accuracy was never reported by other techniques (and their supporting tools) studied in the literature. Furthermore, we are not aware of other tools that can be used to support the detection of agglomerations, in particular concern-based agglomerations. Regarding the identification of architectural problems, we mitigated the imprecision of manual inspection involving the original developers and architects in this process. Architects, who had previous experience on the detection of architectural problems and code anomalies, made the identification of architectural problems. Finally, the procedure used to correlate code anomalies and agglomerations with architectural problems (Section 3.4) is another threat to the construct validity. As we already discussed, agglomerations have higher chance to be related to architectural problems than individual anomalies as they naturally affect more code elements than individual code anomalies. We mitigated this threat by answering RQ2 (Section 4.2).

Conclusion validity The main threat to the conclusion validity is the number of evaluated versions of each system. A study involving several versions of each system is always desired. Nevertheless, it was impracticable in our study due to the number of systems (7) and the limited availability that original developers and architects had to help us. Therefore, we tried to mitigate this threat by selecting, for each different system, a version in a different life-cycle stage. However, in order to evaluate other aspects of the relation between code-anomaly agglomerations and architectural problems, studies involving several versions of a system should be performed in the future. Another threat to conclusion validity is related to the lack of statistical tests. In order to have a significant sample, we would have to consider the full history of architectural problems along several versions of each system, which are not available. In order to reduce the impact of this threat, we made our best effort in: (i) selecting an heterogeneous set of systems in terms of domain, complexity and degree of architecture decay, and (ii) selecting systems with varied types of architectural problems and code anomalies. Our results encourage more rigorous analyses to be developed in the future. We intend to continue expanding our data set over the next years so that statistical tests can be successfully applied.

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 architectural styles and that suffer from a different set of architectural 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.

5 Conclusions

In this work, we analyzed in which circumstances code-anomaly agglomerations represent architectural problems. We conducted an exploratory study involving 7 systems with different sizes. We characterized and studied four agglomeration topologies and compared them to individual code anomalies. We observed that more than 70 % of all architectural problems are related to agglomerations in most of the target systems. In the opposite (upstream) analysis, we noticed that most agglomerations represent most of the architectural problems (50–70 %). Our results confirmed that agglomerations are better than single anomaly instances to indicate the presence of an architectural problem. Moreover, in all target systems, most of the architecturally-relevant anomalous code elements (i.e. realizing an architectural problem) are encompassed by agglomerations. Regarding the circumstances in which agglomerations represent architectural problems, we observed that the concern-based topology is the best indicator of architectural problems. The intra-boundary and cross-boundary topologies identified the highest number of architectural problems. However, they also presented the highest number instances unrelated to architectural problems. The hierarchical topology may be more useful in systems with intense use of hierarchical relationships.

5.1 Extending the analysis of code anomaly agglomerations

Based on the aforementioned results, we plan to perform further studies to address issues. First, further studies need to identify and characterize possible causalities between agglomerations and architectural problems. Several steps need to be made to address this issue. A first and simpler step is enriching the data sample and conducting a more rigorous statistical analysis. For example, the architectural problems analyzed in this study were identified from the perspective of the software architects and developers. However, automated analysis of the implemented architecture (i.e. directly extracted from the source code) could present different and, possibly, relevant results. Therefore, by combining these two points of view, we may possibly reach complementary findings.

5.2 Design and implementation of a synthesis technique

Given our study results, the implementation of a generic technique for detecting agglomerations is required. Unlike the methodology applied in this study, we have to design and implement a synthesis technique; that is, a technique able to automatically detect agglomerations using information from different artifacts of a system beyond the source code. The idea is to help the users to identify architectural problems by (i) revealing different topologies of agglomeration in a software system, and (ii) allowing users to

define their own topologies according to their needs. Before implementing a synthesis technique, it is necessary to define the main requirements that such a technique must satisfy. Hence, the data analyzed in this paper may be explored to define such requirements.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
WO conducted the data collection and analysis, participated in the design of the study, and drafted the manuscript. AG conceived the study, participated in its design, coordinated the research activities and helped to draft the manuscript. TC participated in the design of the study, and in the data analysis and helped to polish the text for external readers. MF participated in the data collection, and in the data analysis and helped to draft the manuscript. AS helped to conceive the study and participated in its design. All authors read and approved the final manuscript.

Acknowledgements
This work was funded by CNPq (productivity grant 305526/2009-0 and Universal Project grant number 485348/2011-0) and FAPERJ (MSc Scholarship, distinguished scientist grant E-26/102.211/2009, project grant number E-26/111.152/2011 and grant E-26/103.366/2012). We also thanks Nenad Medvidovic, Isela Macia, Chris Mattmann and Joshua Garcia for the contribution to the data collection and revision of the manuscript.

Author details
[1]OPUS Group - Informatics Department, PUC-Rio, Marques de Sao Vicente Street, 225, 22451-900 Rio de Janeiro, Brazil.
[2]Informatics Department, State University of Maringa, Colombo Avenue, 5790, 87020-900 Maringa, Brazil.

Received: 8 December 2014 Accepted: 30 June 2015
Published online: 10 July 2015

References
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. IEEE Computer Society, Oldenburg, Germany. pp 181–190
Apache OODT Source Code (2015). https://github.com/apache/oodt
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. ACM. pp 397–412
Booch G, Rumbaugh J, Jacobson I (2005) The Unified Modeling Language User Guide. Addison-Wesley, Boston
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. IEEE Computer Society, Zhangjiajie, China. pp 23–31
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. ACM, Leipzig, Germany. pp 391–400
Fowler M (1999) Refactoring: Improving the Design of Existing Code. Pearson Education, India
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
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. IEEE Computer Society, Palo Alto, USA. pp 486–496
Health Watcher Source Code (2015). http://ptolemy.cs.iastate.edu/design-study/#healthwatcher
Hochstein L, Lindvall M (2005) Combating architectural degeneration: a survey. Inf Softw Technol 47:643–656
Khomh K, Penta MD, 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. IEEE Computer Society, 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. ACM, 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. IEEE Computer Society, Beijing, China. pp 227–236
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. ACM, Orlando, USA. pp 321–340
Macia I, Arcoverde R, Garcia A, Chavez C, Staa A (2012) On the relevance of code anomalies for identifying architecture degradation symptoms. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering. Computer Society, Szeged, Hungary. pp 277–286
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 (2012a) Supporting the identification of architecturally-relevant code anomalies. In: Proceedings of the 28th IEEE International Conference on Software Maintenance. Computer Society, Trento, Italy. pp 662–665
Macia I, Garcia J, Popescu D, Garcia A, Medvidovic N, Staa A (2012b) 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. ACM, Postdam, Germany. pp 167–178
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. ACM, Shanghai, China. pp 721–730
Morgan C (2007) A static aspect language of checking design rules. In: Proceedings of the 6th International Conference on Aspect-oriented Software Development. ACM, Vancouver, Canada. pp 63–72
Moha N, Gueheneuc Y, Duchien L, Meur AL (2010) Decor: A method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36:20–36
Mobile Media Source Code (2015). http://ptolemy.cs.iastate.edu/design-study/#mobilemedia
Oizumi W, Garcia A, Colanzi T, Ferreira M, Staa A (2014) When code-anomaly agglomerations represent architectural problems? An exploratory study. In: Proceedings of the 2014 Brazilian Symposium on Software Engineering (SBES). IEEE Computer Society, Maceio, Brazil. pp 91–100
Olbrich SM, Cruzes DS, Sjoberg DIK (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. IEEE Computer Society, Timisoara, Romania. pp 1–10
Perry DE, Wolf AL (1992) Foundations for the study of software architecture. ACM Softw Eng Notes 17:40–52
Sjobert D, Yamashita A, Anda B, Mockus A, Dyba T (2013) Quantifying the effect of code smells on maintenance effort. IEEE Trans Softw Eng 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. ACM Press, Seattle, USA. pp 174–190
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. ACM, Cape Town, South Africa. pp 75–84
Understand: User Guide and Reference Manual (2015). https://scitools.com/documents/manuals/pdf/understand.pdf
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. IEEE Press, San Francisco, USA. pp 682–691
Young TJ (2005) Using aspectj to build a software product line for mobile devices, University of British Columbia