1
Introduction

The design of a software architecture plays an important role in the software development process because quality attributes of software systems depend on it, such as maintainability and evolvability. Therefore, a well-designed architecture should lead to high quality software systems. There are different definitions of software architecture in the literature. In this research we focus on the software architecture definition established by Bass *et al.* (2003): "*The software architecture is the structure of the system, which comprises of software components, the externally visible properties of these components (i.e. interfaces), and the relationship among them*".

Software systems can be developed using different architectures, which vary from well-designed to badly-designed. A software architecture, even if initially being well-designed, can degrade as the system evolves over time due to change upon change (Lehman, 1980). A degraded architecture often makes changes in the software system more costly and error prone than should be (Stringfellow *et al.*, 2006). The phenomenon of *architectural degeneration* manifests through architectural erosion and drift processes (Perry and Wolf, 1992; Hochstein and Lindvall, 2005). Architectural erosion corresponds to mismatches between the implemented and the intended one prescribed by architects. A typical example of erosion symptom is an unintended relationship between two components in the implemented architecture. In contrast to erosion, architectural drift occurs when the software architecture violates modularity principles. Typical examples of drift symptoms are components with bloat interfaces or implementing several responsibilities (Garcia *et al.*, 2009; Perry and Wolf, 1992). In extreme cases, symptoms of architectural degradation can cause the software reengineering (Eick *et al.*, 2001; Hochstein and Lindvall, 2005; MacCormack *et al.*, 2006).

More directly, however, the degradation of a software architecture can be observed in its implementation throughout the progressive introduction of code anomalies, commonly referred as "code smells" (Karr, 1996; Fowler *et al.*, 1999;

Eick *et al.*, 2001; Hochstein and Lindvall, 2005). In particular, code anomalies might be introduced in the system implementation as a result of drift symptoms in the intended architecture. On the other hand, degradation symptoms in the software architecture might be introduced by code anomalies, that is writing either unintended or non-modular code for a given architecture (Eick *et al.*, 2001; MacCormack *et al.*, 2006; Knodel *et al.*, 2008; Sarkar *et al.*, 2009b). For instance, anomalous code elements (i.e. those infected by code anomalies) might introduce unintended dependencies among architectural components, making the software architecture hard to evolve. Studying the relationship between code anomalies and architectural degradation is the main focus of this research.

1.1. Motivation

Code anomalies are program structures that may indicate a maintainability problem in the software (Fowler *et al.*, 1999). A high number of code anomalies may emerge in programs structured independently of the kind of modularization technique, including object-oriented programming (Meyer, 2000) and aspect-oriented programming (Kiczales, 1997). Some examples of code anomalies include methods that are too long and contain several functionalities (*Long Method*), and tightly coupled methods that change due to several reasons (*Divergent Change*). Code anomalies are particularly harmful to software maintenance when they are associated with architectural degradation symptoms. An *architecturally-relevant* code anomaly represents an architectural degradation symptom in the implementation.

There are several examples documented in the literature that offer substantial evidence regarding the negative impact of code anomalies on the system architecture. Eick *et al.* (2001) showed how complex code elements related to unintended relationships among architectural components hinder the evolution of the AT&T 5ESS telephone switching system. In particular, the high number of code anomalies related to such undesirable relationships made it impossible to make further changes in the system. MacCormack *et al.* (2006) reported that complex code elements modularizing several responsibilities introduced high coupling between architectural components. These complex code

elements were related to a Mozilla web browser re-engineering, which took about five years to rewrite for seven thousand source files and two million source lines of code (Godfrey and Lee, 2000). Similarly, code anomalies related to architectural problems in the Linux-kernel implementation led to a two-year long restructuring of release 2.6 (van Gurp and Bosch, 2002). Knodel *et al.* (2008) also highlighted the harmful impact of code anomalies on the system architecture. In particular more than 5000 architectural problems were related to anomalous code structures in the Testo AG system. Sarkar *et al.* (2009) showed how the effort to refactor code elements that impact on the architecture design took 2.100 man/day to rewrite and retest 25 MLOC lines of code. These examples stand as evidence of the relevance of supporting software engineers in the detection of architecturally-relevant code anomalies.

1.1.1. Motivating Example

In order to illustrate the adverse impact of code anomalies on the architectural design, Figure 1.1 depicts an example. This figure shows a partial representation of the component-and-connector view (Bass *et al.*, 2003) of a web-based system that manages complaints in public institutions. This system is called Health Watcher (Soares *et al.*, 2002) and is one of the systems used in our assessments. As it can be observed, the architectural view represents the static structure of the system in terms of its components (i.e. View, Distribution, Business and Data) and the relationships among them (Bass *et al.*, 2003). In particular, the components of the Health Watcher system architecture were structured following the *Layers* architectural style (Buschmann *et al.*, 1996).

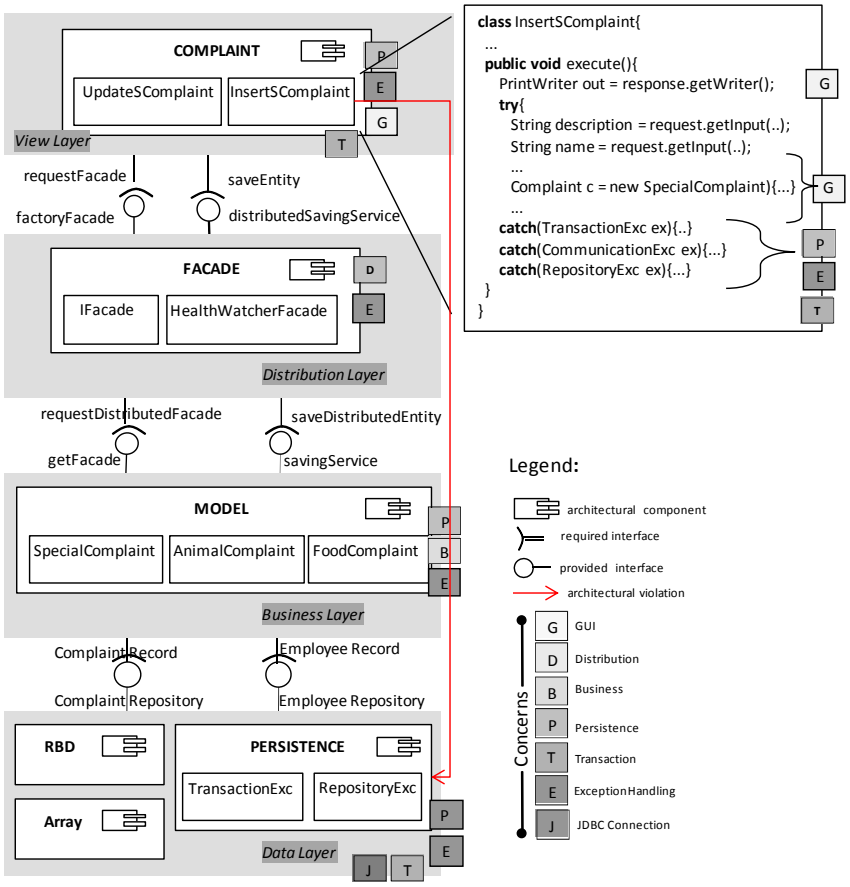

Figure 1.1: A design slice of the Health Watcher system architecture.

Code Anomalies. From the implementation perspective, Figure 1.1 depicts an example of two code anomalies infecting the same code element. First, the InsertSComplaint.execute() method is infected by the *Long Method* anomaly. The problem is that this method deals with several system concerns (i.e. responsibilities or functionalities), such as Persistence and Transaction that should be implemented by other methods. The implementation of these concerns leads the method to present a high complexity in terms of lines of code and cyclomatic complexity (McCabe, 1976). Additionally, the pieces of code dealing with Persistence and Transaction concerns in the InsertSComplaint.execute() method are duplicated in other methods, such as InsertEmployee.execute() and InsertHealthUnit.execute(). Therefore, InsertSComplaint.execute() should be decomposed into several smaller methods in order to improve its readability and modularity. Second, this method also suffers from the *Disperse Coupling* anomaly. In this case the problem is that the method uses information from several classes that belong to different architectural layers (e.g. Distribution and

Data). As a consequence, the InsertSComplaint.execute() method was modified due to changes performed on these architectural layers.

Architectural Degradation. From the architecture perspective, Figure 1.1 presents two examples of architectural degradation symptoms. In particular, it can be noticed how multiple components are responsible for addressing the same system concern and, additionally, some of these components are responsible for independent ones. For instance, View and Data components address the Persistence concern and, at the same time, they implement independent ones such as View and JDBC Connection, respectively. This situation violates the principles of separation of concerns in two ways. Firstly, the Persistence concern is scattered across several architectural layers. Secondly, at least one component modularizes more than one independent system concern. This architectural degradation symptom is known in the literature as *Scattered Parasitic Functionality* (Garcia *et al.*, 2009).

Furthermore, Figure 1.1 depicts an example of a violation of the *Layers* architectural style (Buschmann *et al.*, 1996). The problem is associated with the way exception handling is modularized in this system. Most exceptions are propagated through components interfaces across the system layers, thereby going against the architects' original intent in some cases. For instance, InsertSComplaint.execute() method calls services provided by the requestFacade interface. However, it has to deal with different exceptions (e.g. TransactionExc) propagated by the ComplaintRepository interface, which belongs to the Data layer. The main issue is that these exceptions should be treated internally by the Data component. Specifically, Data should propagate only that exceptional information which is relevant to the final user or other modules, rather than propagate any kind of information. Otherwise, undesirable dependencies are introduced between View and Data layers. This means that accidental code couplings between code elements that belong to the View and Data layers are the sources of architectural violations.

Impact of Code Anomalies on Architectural Design. A deeper analysis of both aforementioned architectural degradation symptoms (i.e. *Scattered Parasitic Functionality* and violation) revealed that they were introduced, in part, by the InsertSComplaint.execute() method, which suffers from a simultaneous occurrence of *Long Method* and *Disperse Coupling* code anomalies. Some of the relationships added by the *Disperse Coupling* occurrence were classified as violations because they introduced undesirable coupling between non-adjacent architectural

components. This was the case, for example, of different exceptions (e.g. TransactionExc) propagated by the Data layer that were dealt by InsertSComplaint.execute() method defined in the View layer, even though these exceptions are not related to the method goal. Also, some concerns implemented by the *Long Method* InsertSComplaint.execute() (e.g. handling the Persistence exceptions) were also addressed by different layer, such as Data. Therefore, the InsertSComplaint.execute() method contributes for the View layer to suffer from *Scattered Parasitic Functionality*. This particular example also helps to illustrate how certain inter-related code anomalies (e.g. *Long Method* and *Disperse Coupling* affecting the same code element) may be an indicator of architectural degradation symptoms.

1.2. Problem Statement

Refactoring (Fowler *et al.*, 1999) is the most common mechanism for removing code anomalies. However, refactoring is often not applied when it is complex, error-prone, time-consuming or, more importantly, when it seems not to be critical to maintain the longevity of the software (Murphy-Hill *et al.*, 2009; Arcoverde *et al.*, 2011). This means that software engineers often need to focus on the identification and removal of the most critical code anomalies. For example, if a code anomaly is associated with architectural degradation symptoms (Figure 1.1) it may require closer, more immediate attention. The problem is that the detection and removal of architecturally-relevant code anomalies are difficult given the high number of code anomalies that often infect the implementation of software systems. For instance, current mechanisms for code anomaly detection (Marinescu, 2004; Lanza and Marinescu, 2006) tend to detect thousands of suspects even in small software systems (Sonarsource, 2010), making even more difficult those processes. Therefore, each code anomaly detected can be considered as a potential candidate to be related to architectural degradation symptoms.

In these cases, developers can only resort to figuring out (or guessing) which code anomalies represent architectural degradation symptoms. The challenge is that there are no attempts to understand the manifestation of code

anomalies that are strong indicators of architectural degradation. There is a conventional wisdom that code anomalies adversely impact the software architecture. However, there is no knowledge about in what proportion this relationship manifests; i.e. do a great amount of code anomalies (not) affect the software architecture? This kind of analysis is important given the need to know whether it is worth focusing on code anomaly analysis to reveal the majority of the architectural problems. Furthermore, it is unknown when the relationship between code anomalies and architectural degradation manifests: (i) during the implementation of the intended architecture, (ii) as a consequence of mismatches between the intended and the actual architectures, and (iii) through the system evolution. Finally, developers do not know which characteristics or relationships among code anomalies are likely to be indicators of their harmful impact on architectural design. As a result, developers do not have any kind of knowledge about how to distinguish such critical code anomalies and, hence, identify refactorings to fix the architectural problems. This means that architecturally-relevant code anomalies may remain in the system implementation over a long period of time, making its maintenance unnecessarily harder.

However, the analysis of architecturally-relevant code anomalies is far from trivial due to several reasons. Firstly, it is questionable if all code anomalies do incur in any damage to the architectural design. Some anomalies might be related to architectural degradation while others are inserted on purpose by programmers as the best solution to a given problem (Fowler *et al.*, 1999). This means that developers might be wasting time on removing anomalies that do not represent any threat to the architecture design. Secondly, code anomalies and their inter-relationships manifest in significantly different ways and usually are scattered across the entire system implementation. Hence, software engineers should be given information not only on where the code anomalies are, but also on its harmful nature. Thirdly, there is no knowledge about whether and to what extent current mechanisms for code anomaly detection are able to identify the architecturally-relevant ones. In other words, conventional mechanisms might be guiding developers in wrong directions when addressing architecturally-relevant anomalies. Finally, and equally importantly, it is common that there is no proper and updated documentation about the system architecture, and only its corresponding source code remains as the reliable artifact.

1.3. The State of Art on Code Anomalies and their Empirical Evaluation

A number of researchers have been interested in documenting code anomalies that affect the modularity of software implementation (Riel, 1996; Fowler *et al.*, 1999; Iwamoto *et al.*, 2003; Hannemann *et al.*, 2005; Monteiro and Fernandez, 2005; Piveta *et al.*, 2005; Srivisut and Muchenraisi, 2007). Other researchers have been interested in classifying code anomalies. Mantyla *et al.*, 2003 grouped code anomalies according to the software modularity property they affect (e.g. complexity, cohesion). Wake (2003) categorized code anomalies considering their scope (e.g. inter-class, intra-class), while Moha *et al.* (2009) classified them according to their nature (e.g. structural). However, the already documented classifications are solely based on the type of the code anomaly rather than considering different sorts of relationships between their occurrences.

Taking into consideration the documented code anomalies, a number of researchers have developed techniques to support their identification (Emden and Moonen, 2002; Ratiu *et al.*, 2004; Marinescu, 2004; Ratzinger, 2005; Lanza and Marinescu, 2006; Murphy-Hill, 2008; Tsantalis, and Chatzigeorgiou, 2009; Marinescu *et al.*, 2010; Moha *et al.*, 2010; Mara *et al.*, 2011). These techniques are based on exploiting information that is extracted from the source code structure varying from the analysis of change couplings (Ratzinger, 2005) to the combination of static code metrics (Mara *et al.*, 201). Such metrics combination, known as *detection strategies* in the literature (Marinescu, 2004), is the most common technique used to identify code anomalies. The reason is that they automatically generate a list of suspects; as a result, a wide range of analysis tools, including visualization ones (Wettel and Lanza, 2008; Carneiro *et al.*, 2010), are based on such strategies. Others researches such as (Moha *et al.*, 2010; Mara *et al.*, 2011) proposed Domain Specific Languages (DSL) to support the construction of detection strategies. Unfortunately, detection strategies do not explore relationships among inter-related code anomalies. We referred to as *conventional detection strategies* those strategies that are only based on static metrics extracted from the source code structure.

Recently, there has been a growing body of relevant work in the literature that analyses the impact of code anomalies. For instance, Kim *et al.* (2005), Mäntylä and Lassenius (2006), Lozano *et al.* (2008), Olbrich *et al.* (2009, 2010), Khomh *et al.* (2009), Rahman *et al.* (2010) and Zazworka *et al.* (2011) studied to what extent automatically-detected code anomalies favor unexpected changes in the system implementation. Others such as Li and Shatnawi (2007), D'Ambros *et al.* (2010) and Zazworka *et al.* (2011) analyzed the correlation of code anomalies with fault occurrences. Other researchers investigated the impact of code anomalies on the maintenance effort, such as Deligiannis *et al.* (2003); Abbes *et al.* (2011); Yamashita *et al.* (2013); Sjobert *et al.* (2013). In particular, the last three works are the only ones that assessed the harmful impact of inter-related code anomalies. These works evidence a recent interest in understanding the effects of inter-relationships among code anomalies. However, these studies were designed to only analyze a specific type of relationship between code anomalies (i.e. when they infect the same code element). Additionally, as it can be noticed, none of the aforementioned studies investigated the impact of such inter-relationships on the architecture design.

Therefore, although there is a vast research in the code anomalies area, all of them do not address relevant challenges in this field, such as those presented in Section 1.2. In particular, two main limitations were identified in the related work. First, they do not carry out assessments regarding: (i) to what extent code anomalies are likely to be indicators of architectural degradation and (ii) the accuracy of conventional detection strategies on the identification of architecturally-relevant code anomalies. Second, they are unable to explore which recurring inter-related code anomalies are likely to degrade the system architecture. As shown in Figure 1.1, inter-related code anomalies might be better indicators of architectural degradation symptoms than single anomalies. Moreover, since an architectural component is often implemented by several code elements, architectural degradation symptoms could be better identified by analyzing inter-related code anomalies. Summarizing, developers are not equipped with knowledge to support the analysis of architecturally-relevant code anomalies.

1.4. Research Questions

Based on the issues discussed in the previous section, the main goal of this thesis is to provide software engineers with a technique that supports them in the identification of architectural degradation symptoms through the analysis of code anomalies. The achievement of this goal was possible only through: (i) understanding the extent of the relationship between code anomalies and architectural degradation, and (ii) assessing to what extent conventional detection strategies are accurate to identify architecturally-relevant code anomalies. The output of the proposed technique is twofold: the lists of single code anomalies and the list of inter-related ones that are likely to be correlated with architectural degradation. In order to develop such a technique the following research questions were addressed in this thesis:

RQ1: What is the relationship between code anomalies and architectural degradation throughout the evolution of software systems?

RQ2: Whether the conventional detection strategies are able to accurately identify architecturally-relevant code anomalies? If so, to what extent?

RQ3: How to accurately identify architecturally-relevant code anomalies?

RQ4: To what extent leveraging architecture-sensitive information and inter-relationships among code anomalies improves the accuracy of conventional strategies when identifying architecturally-relevant code anomalies?

The first research question aims at systematically assessing the proportion of code anomalies that cause architectural degradation symptoms as well as what proportion of architectural degradation symptoms could be removed through refactorings anomalous code elements. The question is also concerned with identifying and assessing characteristics of code anomalies that might be reliable and strong indicators of their harmful impact on the architectural design.

The second research question investigates how accurate the conventional detection strategies are for identifying architecturally-relevant code anomalies. Such investigation is useful to reveal to what extent relying only on the analysis of

the source code structure could benefit or hinder the identification of these critical code anomalies.

The third research question aims at defining a set of detection strategies to help developers identify architecturally-relevant code anomalies. These strategies combine the source code information and how it relates to the system architecture. The proposed detection strategies also aim at reducing the number of neglected architecturally-relevant code anomalies, identified in the previous research question (RQ2). Based on the detected code anomalies, recurrent groups of inter-related ones are identified. The goal of these groups is to encompass the architecturally-relevant code anomalies in order to help engineers in their distinction.

The fourth research question focuses on evaluating the accuracy of our detection technique when identifying architecturally-relevant code anomalies. Such accuracy is also compared with the one obtained by using conventional detection strategies with the same purpose. By performing this comparison we will be able to analyze the advantages and shortcomings of leveraging architecture-sensitive information and inter-relationships among code anomalies in the detection and distinction of the architecturally-relevant ones.

Table 1.1 presents the technical papers that have been published or are under submission and their associations with each research question. Table 1.2 also illustrates published technical papers, but those that only have marginal relation to this thesis.

Table 1.1: Publications directly related to this thesis.

Direct Publications	Research Question(s)
Macia, I.; Detecting Architecturally-Relevant Code Smells in Evolving Software Systems. In Proceedings of the 33rd International Conference on Software Engineering (ICSE) - Doctoral Symposium, Hawaii, USA, 2011.	All
Macia, I.; Garcia, A.; and Staa, A. An Exploratory Study of Code Smells in Evolving aspect-oriented Systems. In Proceedings of the 10th annual International Conference on Aspect-oriented Software Development (AOSD), pp. 203-214, 2011.	RQ1
Macia, I.; Arcoverde, R.; Garcia, A.; Chavez, C.; Staa, A. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), pp. 277-286, 2012.	RQ1
Macia, I. Revealing Architecturally-Relevant Flaws in Aspectual Decompositions. In Proceedings of the 10th International Conference on aspect-oriented Software Development (AOSD), Fourth place at ACM Competition, pp. 85–86, 2011. ACM.	RQ1
Macia, I.; Garcia, A.; Staa, A.; Garcia, J. and Medvidovic, N. On the Impact of aspect-oriented Code Smells on Architecture Modularity: An Exploratory Study. In Proceedings of the 5th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), 2011.	RQ1
Macia, I.; Garcia, A.; Staa, A. Estratégias de Detecção de Anomalias de Modularidade em Sistemas Orientados a Aspectos. In Proceedings of the III Latin American Workshop on aspect-oriented Software Development (LAWASP), 2009.	RQ1
Gurgel, A.; **Macia, I.**; Garcia, A.; Mezini, M.; Eichberg, M.; Staa, A.; Mitschke, R. TamDera: Blending and Reusing Rules for Architectural Degradation Prevention (to be submitted).	RQ1, RQ2
Macia, I.; Garcia, A.; Staa, A. Defining and Applying Detection Strategies for aspect-oriented Code Smells. In Proceedings of the ACM SIGSoft XXIII Brazilian Symposium on Software Engineering (SBES), 2010.	RQ2
Macia, I.; Garcia, J.; Popescu, D.; Garcia, A.; Staa, A, Medvidovic, N. Are Automatically-Detected Code Anomalies Relevant to Architecture Modularity? An Exploratory Analysis of Evolving Systems. In Proceedings of the 11th annual international conference on Aspect-oriented Software Development (AOSD), pp. 167-178, 2012.	RQ2
Macia, I.; Arcoverde, R.; Cirilo, E.; Garcia, A.; Staa, A. Supporting the Identification of Architecturally-Relevant Code Anomalies. In Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), 2012.	RQ3
Arcoverde, R.; **Macia, I.**; Garcia, A.; Staa, A. Automatically Detecting Architecturally-Relevant Code Anomalies. In Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE), held in conjunction with the 34th International Conference on Software Engineering (ICSE), 2012.	RQ3
Macia, I.; Garcia, A.; Chavez, C.; Staa, A. Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies. In Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR), 2013 (to appear).	RQ3, RQ4
Macia, I.; Dantas, F.; Garcia, A.; Staa, A.; Mezini, M. Are Code Anomaly Patterns relevant to the Architectural Design? (to be submitted)	RQ3, RQ4

Table 1.2: Indirect publications.

Indirect Publications
Herrera, J.; **Macia, I.**; Salas, P.; Pinho, R.; Vargas, R.; Garcia, A.; Araújo, J.; Breitman,K. Revealing Crosscutting Concerns in Textual Requirements Documents: An Exploratory Study with Industry Systems. In Proceedings of the ACM SIGSoft XXV Brazilian Symposium on Software Engineering (SBES), pp. 111-120, 2012
Mitschke, R.; Eichberg, M.; Mezini, M. Garcia, A.; **Macia, I**. Modular Specification and Checking of Structural Dependencies. In Proceedings of the 13th annual international conference on Aspect-oriented Software Development (AOSD), pp. 167-178, 2013 (to appear).

1.5. Outline of the Thesis Structure

In the remainder of this thesis, we study the interplay of code anomalies and architectural degradation and demonstrate the inability of conventional detection strategies to identify architecturally-relevant code anomalies. We present the details of our technique to detect code anomalies and its composing elements, such as metrics, detection strategies, code anomaly patterns, and the proposed supporting tool. We also demonstrate how our technique can help a developer to better identify architecturally-relevant code anomalies and, thus, how the architectural information extracted from the source code benefits this process.

Chapter 2 provides an overview of the background material necessary to understand the thesis and evaluate its contributions. Firstly, we present definitions of software architecture (Section 2.1). Then, we describe the architectural degradation phenomenon in terms of its main causes and the techniques for supporting its prevention (Section 2.2). We also discuss code anomalies including definitions, mechanisms for supporting their automatic detection and empirical studies that investigate their harmfulness under different perspectives (Section 2.3). For both research topics architectural degradation and code anomaly, we provide a critical review of their open issues that motivate this research work.

Chapter 3 presents the definition and assessment of code anomalies recurrently observed in three aspect-oriented software systems. It starts presenting the main concepts of the aspect-oriented programming (Kickzales, 1996) in Section 3.1. Then, it describes how the new code anomalies were recurrently observed (Section 3.2). These new code anomalies and the already published ones are classified in three categories according to their common characteristics. We empirically evaluate the anomalies in the context of three software systems

(Section 3.3). The documentation of these code anomalies, their corresponding detection strategies and evaluation, form the *first contribution* of this thesis.

In Chapter 4, we assess the interplay between code anomalies and architectural degradation in the context of five software systems developed using aspect-oriented and object-oriented techniques (Section 4.1). Besides assessing the correlation and cause-effect relationships between code anomalies and architectural degradation, we evaluate the impact of two characteristics of code anomalies, *type* and *earliness*, on such relationships (Section 4.2). We also study how often developers apply refactorings to address architecturally-relevant code anomalies. Finally, the limitations of the study are presented in Section 4.3. The description of the design of the case study, and the discussion of the findings we have encountered, constitutes the *second contribution* of our work.

In Chapter 5, we evaluate the accuracy of conventional detection strategies when identifying architecturally-relevant code anomalies. This evaluation is carried out in the context of the five target systems used in Chapter 4 (Section 5.1). Specifically, fifteen detection strategies including both aspect-oriented and object-oriented ones were assessed in this study. As result the study revealed that conventional strategies are not accurate to identify architecturally-relevant code anomalies (Section 5.2) due to their inability to exploit architecture-sensitive information and relationships among code anomalies. We describe the limitations of the study in Section 5.3. The description of the design of this study, and the empirical evidence and reasons about the inability of conventional detection strategies to identify architecturally-relevant code anomalies, form the *third contribution* of this thesis.

Based on the findings gathered in Chapters 4 and 5, Chapter 6 describes the foundations of the proposed technique to detect architecturally-relevant code anomalies. First, we present the basic terminology and formalism for this detection (Section 6.1). Then, we present a suite of seven architecture-sensitive metrics that is formalized using the proposed formalism and can be gathered from the source code in absence of explicit architecture description (Section 6.2). We also propose a suite of eight detection strategies that rely on these metrics (Section 6.3). Then, we systematically evaluate the accuracy of the architecture-sensitive detection strategies in the context of five software systems (Section 6.4). The proposed architecture-sensitive metrics and architecture-sensitive detection

strategies, form the *fourth contribution* of our work. Additionally, the findings gathered while assessing the strategies accuracy constitute the *fifth contribution* of this thesis.

In Chapter 7, we present the second part of the proposed technique, which involves the distinction of architecturally-relevant code anomalies by analyzing the relationships among anomalous code elements. We present and classify nine recurring inter-relationships among anomalous code elements – code anomaly patterns – observed in a sample of six software systems (Section 7.1). The anomaly patterns are classified in four groups (Sections 7.2 to 7.5) according to their common characteristics. We also discuss some of the possible correlations among the anomaly patterns (Section 7.6). We present our tool, SCOOP, which provides support to the collection of the proposed metrics, application of the proposed strategies and identification of the documented anomaly patterns (Section 7.7). The correlation of the anomaly patterns with architecturally-relevant code anomalies is investigated in Section 7.8. The documented anomaly patterns, our tool and the systematic assessment of their correlation with architecturally-relevant code anomalies, constitute the *sixth contribution* of this thesis.

Chapter 8 presents the final remarks, summarizes the contributions of this work and points out the future directions to be followed.