2
Background and Related Work

As software systems evolve, their size and complexity tend to grow. In this context, the increasing manifestation of code anomalies is a key symptom of architecture quality degradation. When those anomalies are not detected and systematically removed, the evolution of software systems can be compromised irreversibly, and, eventually, a complete redesign will become inevitable (Eick *et al.*, 2001; van Gurp and Bosch, 2002; Maccormack *et al.*, 2006). Many studies have broadly proposed mechanisms for code anomaly detection (Ratiu, 2004; Lanza and Marinescu, 2006; Mara *et al.*, 2011). However, identifying which anomalies are more likely to adversely impact the system architecture is still a challenging task.

In this context, this chapter outlines the software architecture terminology we will use throughout this document (Section 2.1). Next, it presents the architecture degradation phenomenon in terms of its main causes and the techniques for supporting its prevention (Section 2.2). Afterwards, we 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, the chapter provides a critical review of their open issues that motivate this research work.

2.1.
Software Architecture

Software architecture is the structure of the system, which comprises software elements (e.g. components), the externally visible properties of these elements, and relationships among them (Bass *et al.*, 2003). Additionally, software architecture is the key artifact providing information about the system internal organization (Bass *et al.*, 2003). The development of software architecture can galvanize the diverse stakeholders into action towards a common

goal of realizing the envisaged system, or maintaining the delivered system (Clements *et al.*, 2002). In many cases, it can influence, or even dictate, the organization of the various development teams (Booch, 2007).

From a structural perspective, the architecture captures the structure of a system in terms of architectural components and how they interact (Gorton, 2006). *Architectural components* are architectural elements which address key architectural concerns, including functionalities and behaviors (Taylor *et al.*, 2009). Functionalities refer to the component purpose whereas behaviors represent the component interactions. Components interact by means of *connectors*. In this sense an *architectural concern* is defined as an architect interest that significantly influences the system architecture (IEEE 1471, 2010).

Architectural connectors provide the following types of interaction services between architectural components: communication, coordination, conversion, and facilitation (Mehta *et al.*, 2000). Communication concerns the transfer of data (e.g., messages, computational results, etc.) between components. Coordination concerns the transfer of control (e.g., the passing of thread execution) between components. Conversion is concerned with the translation of differing interaction services between components (e.g., conversion of data formats, types, protocols, etc). Finally, facilitation describes the mediation, optimization, and streamlining of interaction (e.g., load balancing, monitoring, and fault tolerance).

As a conceptual solution, *software architecture* captures the foundational design decisions made early in the development process (Bass *et al.*, 2003; Jansen and Bosch, 2005). These design decisions need to preserve modularity principles and system qualities, which are central to the system success (Clements *et al.*, 2002). *Modularity in architecture* design refers to a logical partitioning of a software architecture that allows it to become manageable for implementation maintenance and evolution (Buschmann *et al.*, 2007).

At any time during the process of engineering a software system, architects will have made a set of architectural design decisions that reflect their intent. These design decisions comprise the system *intended architecture* (or prescriptive architecture). In other words the *intended architecture* comprehends design decisions made by architects that have to be respected during the system implementation (Taylor *et al.*, 2009). Examples of the intended design decisions are: the selection of system components, their interactions, and their constraints.

Implemented architecture (or *actual architecture*), on the other hand, describes how the system has been actually built (Taylor *et al.*, 2009). In an ideal scenario, both architectures (i.e. intended and implemented) would be always modular and identical. That is, the implemented architecture should be a perfect realization of the intended architecture. However, in software systems, the implemented architecture often presents modularity problems and more often does not match the intended architecture (Taylor *et al.*, 2009). Many intended architecture decisions can be undesirably neglected by the actual implementation of a system.

Architectural degradation symptoms represent the mismatches between the intended and the implemented architectures as well as modularity problems in these architectures (Section 2.2). In particular, such symptoms are challenged to be identified when the intended architecture is only partially documented, as it often occurs. In these scenarios, unfortunately the source code is usually the most reliable artifact to be considered in the detection of architecture degradation symptoms. In this research work we are particularly concerned with the identification of architecture degradation symptoms through analysis of the source code.

2.2. Architectural Degradation

Software systems evolve over time as features are added, changed or removed or when corrective maintenance is performed. When these changes are not carefully performed, the system architecture degrades due to *architectural erosion* and *architectural drift* (Hochstein and Lindvall, 2005).

Architectural erosion occurs when the implemented architecture does not reflect the intended architecture. The discrepancies between the intended and the implemented architectures are called *architectural violations* (Perry and Wolf, 1992). Architectural violations can be introduced even in the first version of the software system. They often increase the number of maintainability problems in the system (Perry and Wolf, 1992). As an example of architectural violation, consider a system architecture based on the *Layer* style (Buschmann *et al.*, 1996). This style has a well-known prescribed rule stating that only adjacent layers can

directly communicate with each other. Therefore, the implemented architecture violates the intended architecture when relationships between non-adjacent layers are introduced in the corresponding implementation.

Architectural drift occurs when decisions introduced in the intended or implemented architecture violate modularity principles. Architectural drift symptoms impair the adaptability of system architecture and, therefore, its evolution (Perry and Wolf, 1992). These symptoms are usually caused by applying a design solution in an inappropriate context, applying design abstractions at the wrong level of granularity or misusing of modeling languages (Perry and Wolf, 1992). Unlike erosion symptoms, drift symptoms manifest themselves in the implemented architecture when it perfectly matches the intended one. Similarly to (Garcia *et al.*, 2009), each architectural drift symptom is related to an *architectural anomaly* in the context of this research work. Few catalogs documenting architectural anomalies can be found in the literature (Garcia *et al.*, 2009; Stal, 2011). They will be further discussed in Section 2.2.3.

2.2.1. Causes of Architectural Degradation

Several researchers have discussed possible causes of architecture degradation. Parnas (1994) suggested that software systems that are not properly designed to accommodate changes degrade due to successive modifications. Some of the reasons why this problem occurs include: (i) rushed development, (ii) poor system implementation, (iii) intuitive development which is not modular, (iv) lack of design documentation, (v) poor quality design documentation, (vi) lack of training in design documentation, (vii) time pressure, and (viii) inadequate design and coding tools.

Eick *et al.* (2001) also documented possible causes of degradation, based on their study of a large telecommunication system. Such causes, which are related to requirements and organization factors, include: (i) inappropriate architecture, (ii) violations of the original design principles in the system implementation, (iii) imprecise requirements, (iv) inadequate programming tools, (v) programmer variability, and (vi) inadequate change process.

Finally, van Gurp and Bosch (2002) observed the following causes for design degradation, based on their findings of industrial case studies: (i) lack of traceability of design decisions, (ii) lack of modularity in the system implementation, (iii) increasing maintenance costs, (iv) accumulation of design decisions, and (v) iterative methods. Van Gurp and Bosch's list places more emphasis on the inevitability of degradation, stating that "*even the optimal strategy does not necessarily lead to an optimal design. It just delays inevitable problems like architectural erosion and architectural drift*" (van Gurp and Bosch, 2002).

2.2.2. Prevention of Architectural Violations

It can be observed that architectural degradation could be avoided although this seems to be quite hard (Section 2.1.1). In this sense, a number of techniques address the prevention of architectural violations by defining and enforcing design rules in the source code. The goal of these works is to warn developers when the implemented architecture does not correspond to the intended architecture.

Reflexion Models (1995) - RM - pioneered the idea of encoding the architecture via declarative mappings to the source code. RM is an analytical approach that uses the modeled system architecture to observe deviations between source code and intended architecture, which is reviewed by the architect. The lack of hierarchical organization in Reflexion Models motivated the work on Hierarchical Reflexion Models (Koschke and Simon, 2003). Expected dependencies are inherited, and overriding is only possible if high level dependencies are disallowed and low level exceptions allowed, but not vice versa.

Sangal (2005) and Sangal *et al.* (2005) discuss the scalability issue of architecture descriptions and propose a hierarchical visualization for the method called Design Structure Matrices (DSMs) model (Baldwin and Clark, 2000), which originates from the analysis of manufacturing processes. The key advantage of the matrices is that they facilitate the assessment of a layering pattern (or cycles) in the architecture, via a predominance of dependencies in the lower triangular half of the matrix (or mirroring entries in both triangular halves). It is important to note that, differently from Reflexion Model technique, DSM

does not perform the comparison between both models; it is just interested in providing a mechanism to represent the structure of the architectural design.

D'Hondt *et al.* (2001) introduced the Logic-Meta Programming (LMP) in order to enforce the synchronization between design and code. Aldrich *et al.* (2002) proposed *ArchJava*, an extension to the Java programming language. ArchJava unifies architecture with implementation, ensuring that the implementation conforms to architectural constraints. Eichberg *et al.* (2008) proposed *Vespucci*, an approach that uses declarative queries to support architectural rules checking. Marwan and Aldrich (2009) developed *SCHOLIA*, a technique for documenting the system architecture in the source code and checking its conformance with the intended architecture. Terra and Valente (2009) propose a *Dependency Constraint Language* (DCL) that facilitates constructive checking of constraints on dependencies; discrimination of dependencies by kind is also supported. DCL offers a textual *Domain Specific Language* (DSL) for specifying constraints. Oliveira (2011) presented the *PREViA* approach which provides features for defining components and expected interactions in the intended architecture using UML class and component diagrams. Other languages specialized on software constraints, such as LePUS3 (2011), Intensional Views (2011), PDL (2011), and Semmle .QL (2011) can also be used to check detailed design rules, e.g., rules related to design patterns (Gamma *et al.*, 1995).

A number of commercial tools have been proposed for checking dependency among modules and classes using implementation artifacts: Lattix (Sangal *et al.*, 2005), Sonar (2009), Structure101 (2010), Axivion (2010), Klocwork (2010), Coverity (2011), Lindval (2007, 2008), and Bahaus (Raza *et al.* 2006). However, the scope of these tools is limited; they are just able to expose violations of "certain" architectural constraints, such as inter-module communication rules in a layered architecture. In other words, these tools do not provide means for expressing system constraints.

Other works rely on the expressiveness of the mechanisms provided by aspect-oriented programming (Kiczales, 1997) to define design rules in the source code. Sullivan *et al.* (2005) propose Crosscut programming interface (XPI) for specifying design rules between aspects and classes. Dósea *et al.* (2007) propose a language to specify design rules that establish the minimum requirements to

enable the parallel development of class and aspects. Morgan (2007) developed a domain specific language called *Program Description Logic* (PDL). PDL relies on a fully static and expressive pointcut language and allows succinct declarative definitions of programmatic structures which correspond to design rule violations. Ubayashi *et al.* (2010) presented Archface, a programming-level interface to represent the intended architectural design and check its conformance with the source code.

Other researchers have been interested in providing reverse engineering tools for recovering or extracting the implemented architecture from the source code. There are two main categories of reverse engineering tools that can be used for that purpose: filtering and clustering tools and architecture identification tools. Filtering and clustering tools (Lung, 1998; Kazman and Carriere, 1998; Muller *et al.*, 1993) perform an analysis of the source code in order to identify components using function-call dependencies. Through a user interface, users can remove units from the model that are considered unimportant (filtering), and can aggregate units (functions, classes, files) into a cohesive set of modules that form a logical high-level unit (clustering) (Lung, 1998).

Architecture identification tools (Antoniol *et al.*, 1997; Chase *et al.*, 1998) recover information about architectural components and connectors through an analysis of the source code beyond extracting file and function-call dependencies. The main advantage of these tools is the detection of components and connectors that filtering and clustering tools cannot recognize.

2.2.3. Architectural Anomalies

Unlike the existence of a vast research to prevent architectural violations, only few research works are focused on the characterization of architectural anomalies. In fact, catalogs of architectural anomalies are only recently beginning to appear.

Garcia *et al.* (2009) documented a catalog of four (04) architectural anomalies that manifest themselves in component-and-connector architectures, but can be easily adapted to other architectural views, such as modules view. *Ambiguous Interface* refers to interfaces that do not reveal which services the

component is offering. These interfaces usually offer only a single, general entry-point into a component, reducing the system analyzability and understandability. *Scattered Parasitic Functionality* takes place when multiple components are responsible for realizing the same high-level concern and, additionally, some of those components are responsible for independent concerns. The *Scattered Parasitic Functionality* adversely affects maintainability, understandability, testability, and reusability. The reason is because when the shared concern needs to be changed, all the components that realized it can be updated and tested. *Extraneous Adjacent Connector* occurs when two connectors of different types are used to link a pair of components. The problem is that the beneficial effects of each individual connector may cancel each other out. For example, while method call connectors increase understandability, using an additional event-based connector reduces this benefit because it is unclear whether and under what circumstances the additional communication occurs. *Connector Envy* occurs in components that encompass extensive interaction-related functionality that should be delegated to a connector. This anomaly reduces reusability, understandability, and testability. Reusability is reduced by the creation of dependencies between interaction services and application-specific services, which make it difficult to reuse either type of service without including the other. The overall understandability of the component decreases because disparate concerns are blended. Lastly, testability is affected by *Connector Envy* because application functionality and interaction functionality cannot be separately tested.

Stal (2011) documented a catalog of five (05) architectural anomalies. *Dependency Cycle* between architectural components implies that engineers cannot understand, test, or change a given component without addressing other components in the cycle. *Inexpressive Component Names* prevent engineers to understand the architecture without digging deeper into more details. *Component Responsibility Overload* means that a component is implementing too many different responsibilities, preventing clear separation of concerns. *Unnecessary Indirection Layers* do not only negatively affect developmental qualities, like maintainability or extensibility, but also operational quality attributes, such as performance. *Implicit Dependencies* often lead to a shift between desired architecture and implemented architecture. One notable example is violating strict layering, thus introducing unnecessary und unknown dependencies.

2.2.4. Analysis of Existing Techniques to Prevent Architectural Degradation

This section discusses the limitations of existing techniques for preventing architectural degradation symptoms. Such limitations are addressed in the context of this thesis.

2.2.4.1. Lack of Prescribed Design Decisions

As we have observed, existing techniques for preventing architectural degradation rely on the explicit specification of design rules. Thus, those techniques assume the existence or documentation of the intended architecture. However, in industry software systems, architectural design is often not explicitly documented or kept out of date as the system evolves. The problem is that specifying design rules usually demands a large amount of developer and architect effort, as even small systems may contain many constraints. This situation tends to get worse when design rules change, as new requirements are added or modified along the system's evolution. As a consequence, developers often specify the design rules only in the first version of a system. In fact, the lack of coherent design documentation is claimed in different sources to be one of the most recurrent causes of architectural degradation (Section 2.2.1). We suspect that this occurs because these situations hinder the usability of current techniques for both enforcing design rules in the source code and performing an analysis of architectural conformance (Section 2.2.2). Therefore, we can state that the usefulness of current techniques for preventing architectural deviations could possibly be restricted in practice.

2.2.4.2. Lack of Mechanisms to Detect Architectural Anomalies

Although catalogs of architectural anomalies are documented in the literature (Section 2.1.3), there is little empirical foundation about whether and to what extent they impact the quality of the source code. There is also a lack of

empirical studies investigating whether architectural anomalies in the implemented architecture correspond to code anomalies. Furthermore, developers do not know how often architectural anomalies tend to be introduced due to code anomalies while implementing the system. That knowledge is very relevant for a variety of reasons. First, it could warn architects about the importance of carefully analyzing the intended architecture before its implementation starts. Second, developers could understand how architectural anomalies tend to manifest themselves in the source code and, therefore, prioritize the refactoring of potential candidates of architecturally-relevant code anomalies. Third and equally important, without this knowledge the usefulness of these catalogs could be jeopardized.

Furthermore, in contrast to the high number of techniques documented for detecting architectural violations (i.e. erosion symptoms), there is a lack of tools for automatically detecting architectural anomalies (i.e. drift symptoms). As a consequence, architects and developers need to manually analyze the implemented architecture in order to detect them. In some cases, developers need to identify which elements in the source code are responsible for introducing those architectural anomalies. As this process requires a considerable effort, developers and architects tend to avoid this kind of analysis. Therefore, the usefulness of these catalogs may be at risk.

2.3. Code Anomalies

As we have mentioned in Section 2.1.2, some of the documented causes of architectural degradation seem to be unavoidable (e.g. programmer variability). In order to deal with this kind of situation, iterative development methodologies (e.g., *XP* (2011)) advocate for refactoring the code whenever developers suspect that degradation may be occurring. Therefore, in such cases, degradation is not diagnosed by comparing the actual with the intended architecture of the system, but by identifying anomalies that infect code elements and hinder their modularity (Fowler *et al.*, 1999). *Code element* in this context corresponds to fragments of programming language code such as: attributes, operations and declarations.

Many researchers documented code anomalies that affect the modularity of a system implementation. The first evidence on this interest comes from Webster (1995), who wrote a book on anomalies in the context of object-oriented programming. Riel (1996) proposed 61 heuristics in order to detect deviations of good programming practices and provided a basis for improving design and implementation. Fowler *et al*. (1999) introduced the metaphor of "*bad smell*" - referred to in this thesis as a *code anomaly* - in their book as: *a sign of a deeper design problem in the system implementation*. Also, they documented 22 code anomalies that infect methods and classes.

Some researchers (Iwamoto *et al.* 2003; Hannemann *et al.*, 2005; Monteiro and Fernandez, 2005) introduced the notion of code anomaly in aspect-oriented systems by defining refactoring techniques to improve modularity in these systems. Piveta *et al.* (2005) defined a preliminary set of code anomalies in aspect-oriented systems. Srivisut and Muenchaisri (2007) defined five (05) other aspect-oriented anomalies, extending Piveta's catalog and defined metrics that help to identify the anomalies proposed by him.

Moreover, some researchers have been interested in classifying code anomalies. Wake (2003) classified anomalies into two categories according their scope: anomalies *within classes* and code anomalies *between classes*. Mäntylä and Lassenius (2006) proposed a classification for code anomalies according to the modularity property they affect. For instance, *Bloaters* group refers to code anomalies associated with higher complexities like *Long Method* and *Large Class*.

2.3.1. Detection of Code Anomalies

There is a vast amount of techniques and tools that aim at identifying code anomalies. Emden and Moonen (2002) describe *jCosmo*, an approach for detecting code anomalies in Java systems based on the structural properties of code elements. Ratzinger *et al.* (2005) and Wong *et al.* (2011) can detect specific types of code anomalies (e.g. *Duplicated Code*) by examining change couplings. *Detection Strategies* (Marinescu, 2004) is the most common mechanism, and the most referred to in the literature, for identifying code anomalies. This mechanism exploits information that is extracted from the source code structure, relying on

the combination of static code metrics and thresholds into logical expressions. Each detection strategy is a heuristic that identifies code elements that possibly suffer from a particular code anomaly (Marinescu, 2004; 2006). The example below illustrates a well-known detection strategy (Lanza and Marinescu, 2006) for identifying *God Classes.* This strategy and its thresholds have also been used in previous studies (Olbrich *et al.*, 2009; 2010).

GodClass<class> = (WMC > 47) **and** (TCC < 0.3) **and** (ATFD > 5)

In this detection strategy:

- WMC (*Weighted Method Count*) is the sum of the cyclomatic complexity of all methods within the class under analysis;
- TCC (*Tight Class Cohesion*) represents ratio of methods that Access the same instance variable with respect to the total number of methods;
- ATFD (*Access to Foreign Data*) counts the number of attributes in foreign classes accessed by the class under analysis.

A wide range of static analysis tools, including visualization ones (e.g. Wettel and Lanza, 2008; D'Ambros *et al.*, 2010), are based on such strategies. Marinescu *et al.* (2010) also presented *inCode*, a tool used to automate the application of certain detection strategies. Munro (2005) proposed some heuristics for detecting code anomalies. Ratiu *et al.* (2004) approach relied on the use of the historical information to increase the accuracy of the automatic identification of code anomalies through detection strategies. Alikacem and Sahraoui (2006) proposed a language to detect code anomalies. This language allows the specification of detection strategies. Moha *et al.* (2010) presented the *Décor* tool and a domain-specific language to automate the construction of detection strategies. Mara *et al.* (2011) proposed a tool called *Hist-Inspect* to support the definition and automatic application of history-sensitive detection strategies. Table 2.1 summarizes some of the existing tools that support the identification of code anomalies and code analysis based on detection strategies.

Table 2.1. Tools for code anomaly detection and source code analysis

Tool	Platform	Description
CloneDetections / Clever	Java	It supports the detection of *Duplicated Code*
CodeCity	Java	It visualizes source code and its anomalies as interactive, navigable 3D cities.
DECOR	Java	It supports the application of detection strategies. Also, it allows developers to create and adjust detection strategies according to their needs.
Hint-Inspect	Java	It supports the detection of code anomalies based on history-sensitive metrics. It also supports the personalization of detection strategies.
inCode	Java	It supports the detection of a small number of Fowler's anomalies: *Feature Envy*, *Data Class* and *Long Methods*.
JDepend	Java	It supports the collection of metrics for packages such as: number of classes, coupling between classes, and dependencies between packages.
Semmle	Java	It supports the collection of code metrics. It also allows developers to create rules based on *Code Query Language* (CQL).
Sonar	Java, C, PHP, Groovy	It supports the collection of source code metrics and detection of several code anomalies. It also supports the recovery of the implemented architecture.
SourceMiner	Java	It supports the visualization of high coupled code elements.
Together	Java	It supports the collection of source code metrics and detect many of the Fowler's anomalies.
Understand	C, C++, Java, C#	It supports the collection of source code metrics, detection cyclic dependencies and recovery of the implemented architecture
JSLint	JavaScript	It supports the detection of code anomalies and inappropriate programming styles
ReSharper	.NET, JavaScript	It detects code anomalies such as duplicated code. It also allows developers to personalize code inspection process.
Ndepend	.NET	It supports the collection of code metrics, indicating parts of the code that should be reviewed. It also allows the definition of new metrics based on code query language (CQL).
Flay	Ruby	It supports the detection of *Duplicated Code*.
Reek	Ruby	It supports the detection of code anomalies such as: Long Methods, Inappropriate Name and Feature Envy.
Saikuro	Ruby	It supports the detection of complex methods based on the cyclomatic complexity.

In the context of this thesis we rely on detection strategies to identify occurrences of code anomalies due to several reasons. First, they were explicitly conceived with the purpose of supporting the identification of any type of code anomalies. The main focus of other works (e.g. Ratzinger *et al.*, 2005; Wong *et al.,* 2011) is not the identification of code anomalies, even though they can accidentally identify occurrences of specific code anomalies. Second, detection strategies have been widely used and a plethora of tools are based on this technique. Finally, a number of researchers have relied on detection strategies to

investigate the behavior of code anomalies throughout the system evolution as well as their harmful impact on change-proneness, fault-proneness and maintenance effort (Section 2.3.3).

2.3.2. Removal of Code Anomalies by means of Refactorings

The refactoring process aims at removing anomalies by performing changes on the internal structure of the system without changing its external behavior (Fowler *et. al.*, 1999). Several studies have been developed aimed at supporting the automatic identification and application of refactorings. Higo *et al.* (2004) proposed the *Aries* tool to identify possible refactoring candidates based on the number of assigned variables, the number of referred variables, and dispersion in the class hierarchy. Tsantalis and Chatzigeorgiou (2009) proposed a technique for automatically identifying code refactoring needs via static slicing. Vidal *et al.* (2012) proposes an expert software agent that assists developers when refactoring an object-oriented system into an aspect-oriented one. It analyzes the user's interaction history for improving the agent's effectiveness over time, guiding developers through the steps they should take. Xi *et al.* (2012) also propose a refactoring recommendation mechanism based on the observation of manual refactoring steps. Their goal is to monitor common sequences of previous changes on code structures in order to detect the occurrence of refactorings, and recommend their automation on-the-fly, while the developer is programming.

Van Gurp and Bosch (2002) suggest that refactoring techniques cannot effectively improve the global maintainability, especially when there are complex structural problems which are widely dispersed over multiple components. In addition, Murphy-Hill (2009) and Arcoverde *et al.* (2011) suggest that refactorings are not usually applied if they are complex, error prone, time-consuming or there is no evidence that they are effective in maintaining the system's modularity. A major factor that hinder the proper choose of refactorings to be applied is that anomalies usually occur simultaneously (Liu *et al.*, 2011). In these situations, developers need to evaluate the characteristics of the different anomalies in order to define the most appropriate sequence of refactorings. For instance, removing the most critical anomalies leads to the removal of other anomalies that affect the same piece of code (Liu *et al.*, 2011).

2.3.3. Empirical Studies about Code Anomaly Side Effects

Many works documented the impact of code anomalies on system maintainability and evolvability. A number of researchers investigated the impact of code anomalies on change-proneness. Mäntylä and Lassenius (2006) investigated to what extent code anomalies can be used as a basis for subjective evaluation of code evolvability. Olbrich *et al.* (2009, 2010) and Khomh *et al.* (2009) investigated the evolution of code anomalies. The authors analyzed whether the number of code anomalies increases over time, and the anomaly influence on how often a code element changes. Olbrich *et al.* (2009) analyzed the historical data of two open-source projects focusing on the *God Class* and *Shotgun Surgery* code anomalies. An important conclusion of their analysis was that the classes infected by the examined anomalies suffer more changes than non-infected ones and they are affected by larger changes. In particular, this analysis suggested that *God Classes* are more change prone than other classes. A similar conclusion was reached in Khomh *et al.* (2009), where statistical analysis of 29 anomalies in several releases of two open-source projects revealed that classes with anomalies are more likely to be the subject of changes. Also, Khomh *et al.* (2009) suggested that specific anomalies are more correlated than others to change-proneness. Zazworka *et al.* (2011) confirmed these results revealing that *God Classes* are 5-7 times more change prone than others. Kim *et al.* (2005) and Lozano *et al.* (2008) analyzed the impact of duplicated code in system changes. While Kim *et al.* (2005) observed that 36% of the duplicated code needed to be changed consistently, Lozano *et al.* (2008) found that at least 50% of the methods with duplicated code required more change effort than the methods without such duplications. Kapser *et al.* (2008), Jürgens *et al.* (2009) and Rahman *et al.* (2010) also studied the impact of duplicated code in change-proneness, obtaining similar results. In the context of these studies, Wong *et al.* (2010) was the only identified that attends to observe how changes associated with code anomalies are likely to drift from the intended design. In particular, they observed that when code duplications change frequently together they often deviate from the intended designer decisions.

The effects of code anomalies have also been studied from the perspective of system defects. Li and Shatnawi (2007) investigated the relationship between the class error probability and code anomalies, based on three versions of the Eclipse project (2011). Their result showed that classes which are infected with code anomalies (e.g. *Shotgun Surgery*, *God Class* or *God Methods*) are more likely to present errors than non-infected classes. Also, D'Ambros *et al.* (2010) investigated the influence of code anomalies on software defects in six open-source systems. Their investigation suggested that an increase in the number of code anomalies is likely to generate software defects. However, there is no a single code anomaly that was consistently correlated to errors more than others across the totally of the systems. Recently, Zazworka *et al.* (2011) suggested that *God Classes* are 4-17 times more defect prone than other classes (affected or not by any anomaly).

Other researchers studied the impact of code anomalies on maintenance effort. Deligiannis *et al.* (2003) showed that a design (not code) without a *God Class* was judged and measured to be better (in terms of time and quality) than a design for the same system with a *God Class*. Deligiannis *et al*. (2004) observed that a design without a *God Class* had better completeness, correctness and consistency than a design with a *God Class*. From the empirical studies identified, only 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.

2.3.4. Analysis of Existing Research on Code Anomalies

We have identified several particularities of existing research on code anomalies that are likely to plaster the early detection of architectural degradation symptoms in the system implementation. These particularities are discussed as follows.

2.3.4.1. Lack of Extensive Catalogs of Aspect-Oriented Code Anomalies

As introduced in Chapter 1, the main focus of this research work is to analyze the interplay between code anomalies and architectural degradation in software systems implemented with different modularization techniques. However, unlike object-oriented systems, there are few research works that focused on the characterization of code anomalies in aspect-oriented systems. Piveta *et al.* (2005) defined five anomalies occurring in AO systems. Srivisut and Muenchaisri (2007) extended Piveta's work in order to define metrics that help identifying the code anomalies proposed by him. Moreover, such studies are limited and mostly mimic classical well-known problems in object-oriented programs, such as *Lazy Aspect* and *Large Aspect* that are minor adaptation from object-oriented anomalies. Nevertheless, the expressive power of aspect-oriented mechanisms might facilitate the introduction of particular code anomalies. That is, code anomalies might emerge due to misuses of specific AOP facilities, such as pointcut descriptions. These anomalies differ from those found in object-oriented systems, as pointcuts are aspect-oriented specific constructs. Finally, existing research on aspect-oriented anomalies only focus on the definition of code anomalies, without empirically studying whether and how often these anomalies manifest in software systems. Therefore, there are basically two needs: (i) the documentation of code anomalies associated with the inappropriate use of the aspect-oriented mechanisms and (ii) the investigation of the aspect anomalies impact in the system maintenance.

2.3.4.2. Lack of Knowledge about the Code Anomaly Influence on Architectural Design

Even though several researchers have highlighted the impact of anomalies on architectural decompositions (Fowler *et al.*, 1999; Eick *et al.*, 2001; Hoschtein and Lindvall, 2005; Maccormack *et al.* 2006; Wong *et al.*, 2011), none of the existing reports (Section 2.3.3) investigate the impact of code anomaly on system architecture. Compared to the widely-studied anomaly effects (i.e., changes, faults and maintenance effort), their impact on system architecture is even more harmful

since it can impair the continuity of the project, leading to architectural degradation (Eick *et al.*, 2001; van Gurp and Bosch, 2002; Maccormack *et al.*, 2006). The identification of these code anomalies is specially challenging when the architectural designs are not explicitly documented or are incomplete - recurrent situations in industry software systems. Clearly, then, there is an actual need for empirical understanding of which particular characteristics of code anomalies adversely impact architectural designs.

2.3.4.3. Lack of Documentation on Code Anomaly Patterns

It has been observed that the vast majority of documented researches investigate the impact and behavior of isolated anomaly occurrences and specific types of code anomalies (Fowler *et al.*, 1999). Only Abbes *et al.* (2011) brings up the notion of interaction effects across code anomalies. However, their study is rather limited since it only focuses on investigating the impact of simultaneous occurrences of *God Classes* and *God Methods* on system defects. Therefore, none of the existing empirical studies have explored whether and to what extent inter-related code anomalies might be indicators of architectural degradation symptoms. Even worse, relationships among code anomalies cannot be detected with existing mechanisms, because they solely focus on identifying isolated occurrences of code anomalies. This means that developers may not be able to perform the appropriated sequence of refactoring strategies in order to completely remove the code anomaly.

Code anomaly patterns can alert developers that the occurrence of an anomaly might be a sign that other anomalies are also affecting the same code element. This is particularly handy when one of the co-occurring anomalies is not easy to identify. As mentioned in the previous chapter, code anomaly patterns might also allow developers to identify the existence of architectural problems in the system implementation. In particular, there are architectural problems that cannot be detected without analyzing relationships between code elements (e.g. *Component Responsibility Overload*). Finally, the detection of code anomaly patterns benefits the proper choice and application of refactoring techniques and consequently, reduces costs and resources during the system implementation

stage. Therefore, there is a need for documenting recurrent relationships between code anomalies as well as assessing their impact on the system architecture.

2.4.Summary

This chapter presented the main concepts addressed in this thesis. It also presented an overview of existing studies and a critical discussion of their limitations. Section 2.1 presented the definitions of the main terms discussed throughout this research work, such as software architecture, intended architecture and implemented architecture. Section 2.2 presented the definition of architectural degradation and discussed the symptoms by means it manifests: architectural erosion and architectural drift. There is a variety of techniques and tools that help engineers to identify violations of the intended architecture in the system implementation (Section 2.2.2). However, the use of these techniques is restricted because they depart from the assumption of the existence or documentation of prescribed design decisions and such rules are often not explicitly documented or is not kept up to date as the system evolves. Section 2.2.3 presented catalogs documenting symptoms of architectural drift. Nevertheless, there are neither techniques nor tools that automatically identify such symptoms in the system implementation. There is also little empirical foundation about whether and to what extent these symptoms impact the quality of the source code. As a consequence, architects and developers need to manually analyze the system implementation in order to detect the architectural drift symptoms.

Section 2.3 reported the existing research on code anomalies. In particular, this section presented catalogs of code anomalies in aspect-oriented and object-oriented programming. Section 2.3.3 presented current research that explores the impact of code anomalies on change-proneness, system defects and maintenance effort. However, there is no work that analyzes code anomalies with the intent of providing evidences about their harmful impact on the architectural design, even though such impact has been recognized. Unlike the studied anomaly effects, their impact on system architecture is even more harmful since it can impair the continuity of the project, due to severe architectural degradation. Consequently, although there are a lot of efforts to provide developers with mechanisms that

support the anomaly detection (Section 2.3.1), it is unknown to what extent such mechanisms accurately identify the architecturally-relevant code anomalies.