4
Impact of Code Anomalies on Architectural Degradation

As discussed in Chapter 2, the violation of modularity principles during the implementation of software systems has been highlighted by several authors as a cause of architectural degradation. For instance, code elements that are tightly coupled often demand a significant maintenance effort, leading systems to be redesigned in extreme cases (Eick et al., 2001; MacCormack et al., 2006). The indicators of maintenance problems in the system implementation are referred to as code anomalies (Fowler *et al.*, 1999). These anomalies can manifest themselves in systems implemented using any modularization technique. Catalogs of code anomalies for object-oriented systems have been widely documented (Webster, 1995; Riel, 1996; Fowler *et al.*, 1999). On the other hand, catalogs of code anomalies for aspect-oriented systems are only beginning to appear (Section 3.2.1). Chapter 3 expanded further the existing catalogs by documenting six new aspect-oriented anomalies.

A number of studies have investigated the occurrences of code anomalies under different perspectives. These studies vary from (i) understanding the manifestations of code anomalies throughout the system evolution to (ii) assessing their correlation with change-proneness, fault-proneness, and maintenance effort (Chapters 2 and 3). However, the impact of code anomalies on architectural degradation has mostly been ignored by existing studies, despite of several authors having called attention to it. This adverse impact could be considered as even more critical due to its silence - in the sense that it does not affect the system's behavior, such as fault-proneness, and it can potentially lead to a complete redesign of the system.

In this context, this chapter presents an empirical study that investigates the interplay of code anomalies and architecture degradation. In other words, this chapter aims at answering our first research question (Section 1.4): *What is the relationship between code anomalies and architectural degradation throughout the evolution of software systems?* To this end, the study to be described

investigates which characteristics of code anomalies are more likely to indicate their harmful impact on the architecture of a system. This study also analyzes to what extent refactoring is applied for removing architecturally-relevant code anomalies.

In order to address the aforementioned goals, the study involves a sample of nearly 2100 anomalous code elements[4] distributed in 40 versions of 6 industry-software systems. The anomalous code elements encompassed heterogeneous forms of code anomaly manifestation. In particular, the study assesses code anomalies that infect aspect-oriented and also object-oriented implementations. This kind of investigation could be performed thanks to: (i) our analysis on when and to what extent code anomalies infect aspect-oriented systems (Chapter 3) and (ii) several reports that document recurrent occurrences of code anomalies in object-oriented systems (Section 2.3.3). The results of this study have been published in two technical papers (Macia *et al.*, 2011b; Macia *et al.*, 2012b).

The remainder of this chapter is structured in four main sections. Section 4.1 describes the study definition and design. Section 4.2 discusses the findings associated with the impact of code anomalies on architectural degradation symptoms. Section 4.3 presents the limitations of the study. Finally, Section 4.4 presents a summary of the key points discussed throughout the chapter.

4.1. Study Definition and Design

The study presented in this chapter aims to answer the first research question of this thesis: *What is the relationship between code anomalies and the degradation of the actual architecture, throughout the evolution of software systems?* This research question was decomposed into three research questions (RQ):

RQ1.1: To what extent are anomalous code elements related to architectural degradation in the actual architecture and vice versa?

RQ1.2: Which characteristics of code anomalies are indicators of their relationship with architectural degradation symptoms?

[4] fragments of programming code such as: attributes, operations and declarations

RQ1.3: How often are architecturally-relevant code anomalies removed by means of refactorings?

A sample of nearly 2100 code anomalies and 1030 architectural degradation symptoms was considered to answer *RQ1.1* (Section 4.2.1). This sample includes aspect-oriented (AO) and object-oriented (OO) code anomalies as well as architectural degradation symptoms observed in these implementations.

For *RQ1.2*, two characteristics of code anomalies were analyzed: their *type* (Section 4.2.2.1), and the *earliness* of their occurrence in a software project history (Section 4.2.2.2). In the context of this study, a code anomaly is considered to be early if it was introduced in the first version of a system. The motivation for the former analysis is that certain types of code anomalies tend to occur more often than others (D'Ambros *et al.*, 2009; Khomh *et al.*, 2009). However, there is no knowledge about how often code anomaly types (Srivisut and Muenchaisri, 2007; Piveta *et al.*, 2006; Chapter 3, Fowler *et al.*, 1999) are sources of architectural degradation symptoms (Garcia *et al.*, 2009; Stal, 2010). The motivation for the later analysis is to understand if code anomalies introduced early in a software project are harmful (or not) to system's architecture; if so, this implies that developers should watch out for harmful early anomalies and anticipate their removal through early refactorings.

Finally, for *RQ1.3*, a sample of 700 refactorings were considered, including those performed in both object-oriented (Fowler *et al.*, 1999) and aspect-oriented systems (Iwamoto and Zhao, 2003; Hannemann *et al.*, 2005) (Section 4.2.3).

In this context, following Wohlin *et al.* suggestion (2000), we defined our study and its goals using the GQM format (Basili *et al.*, 1994) as:

Analyze: code anomalies

For the purpose of: understanding their impact on system's architecture and how often they are removed by means of refactoring

With respect to: architectural violations and architectural anomalies

From the viewpoint of: systems architects and developers and researchers

In the context of: six (06) software systems from different domains, implemented with using different modularization techniques and following different architectural decompositions.

4.1.1. Hypotheses

In order to answer the three aforementioned research questions, we have defined the null and alternative hypotheses as shown in Table 4.1.

Table 4.1: Research questions and hypotheses of the study.

Research Questions	Hypotheses
RQ1.1	For each of the code anomalies studied: **Null Hypothesis, $H1_0$**: The code anomaly is not significantly related to architectural degradation symptoms. **Alternative Hypothesis, $H1_A$**: The code anomaly is significantly related to architectural degradation symptoms.
RQ1.2	**Null Hypothesis, $H2_0$**: Type and earliness are not good indicators of architectural degradation symptoms in the context of the studied code anomalies. **Alternative Hypothesis, $H2_A$**: Type and earliness are good indicators of architectural degradation symptoms in the context of the studied code anomalies.
RQ1.3	**Null Hypothesis, $H3_0$**: Refactorings are not often applied to remove the studied code anomalies related to architectural degradation symptoms. **Alternative Hypothesis, $H3_A$**: Refactorings are often applied to remove the studied code anomalies related to architectural degradation symptoms.

4.1.2. Variable Selection

The following independent and dependent variables have defined the following in order to test our hypotheses.

Independent Variables. Independent variables are those variables manipulated and controlled in the study (Wohlin *et al.*, 2000). There is a Boolean variable for $H1_0$, $C_{i,j}$, that indicates whether (or not) the code element i suffers from at least one anomaly in version v_j. There are as many independent variables for $H2_0$ as there are types of code anomalies analyzed in this study (Section 4.1.4). Each variable, $C_{i,k,j}$, indicates the number of times that a code element i suffers from a code anomaly k in version v_j. Finally, there is a variable or $H3_0$, R_j, representing the number of refactorings applied in version v_j. As will be mentioned in Section 4.1.4, all code anomaly occurrences used when testing these hypotheses were confirmed by developers. Section 4.2.3, presents the process followed to identify the applied refactorings in target systems.

Dependent Variables. Dependent variables refer to those variables that we want to study to observe the effect of changes in the independent variables (Wohlin *et al.*, 2000). There are two Boolean dependent variables, $V_{i,j}$ and $A_{i,j}$, for $H1_0$ that indicate whether (or not) the code element *i* is related to any violation (i.e. architectural erosion) or architectural anomaly in the implemented architecture (i.e. architectural drift) in version v_j, respectively. There are as many dependent variables for $H2_0$ as there are kinds of architectural violations and architectural anomalies. The dependent variables $V_{i,k,j}$ and $A_{i,k,j}$ indicate whether the code element *i* affected by the code anomaly *k* is related to any violation or architectural anomaly in version v_j, respectively. Finally, there is a variable for $H3_0$, ER_j, representing the number of refactorings that were effective to remove at least one architecturally-relevant code anomaly in version v_j. The term *architecturally-relevant code anomaly*, used throughout this text, refers to those code anomalies related to architectural degradation symptoms.

4.1.3. Selection Criteria and Target Systems

Several criteria were established to support the selection of suitable software systems to this study. The criteria used are presented in Table 4.2.

Table 4.2: Criteria used for the selection of target systems.

	The target system:
C1	was modeled using documented guidelines or well known architecture styles.
C2	has the intended architecture design available.
C3	has the original architects and developers available.
C4	has a manageable size.
C5	is infected by a rich set of code anomalies.
C6	presents a rich set of architectural degradation symptoms.
C7	has undergone changes.
C8	was implemented by developers with different levels of programming skills.
C9	was implemented using different programming languages and modularization techniques.

Criterion C1 avoids the analysis of code anomalies and architectural degradation symptoms introduced due to poor software engineering practices. Criteria C2, C3, C4, C5 and C6 allow carrying out a better in depth analysis of code anomaly causes and their impact on the actual architecture design since a variety of code anomalies and architectural degradation symptoms are analyzed.

Additionally, the impact of code anomalies on the actual architecture was validated by system architects and developers, which was also important for increasing the reliability of our analysis. Criterion C7 allows analyzing the harmfulness of code anomalies on architecture design that evolved in different ways and the proportion of refactored anomalies. Criterion C8 supports the observation of whether (or not) architecturally-relevant code anomalies were introduced by specific to developers with certain programming skill level. Finally, criterion C9 ensures the observation of whether (or not) the impact of code anomalies on architectural degradation symptoms is specific to programming languages or modularization techniques.

Based on these criteria, six (06) software systems, totaling 40 versions, were selected. Two of these systems, Aspectual Health Watcher and Aspectual Mobile Media, were introduced in the previous chapter (Section 3.3.1). The third and fourth systems are the Java versions of Health Watcher and Mobile Media. The fifth system, named MIDAS, is a lightweight middleware platform implemented in C++ for distributed, event-based sensor applications (Malek *et al.*, 2006; Garcia *et al.*, 2009). Two versions of MIDAS were assessed in this study, which are the before and after versions of a major restructuring with the widest impact in this system evolution. A high number of architectural and code elements suffered changes in this transition. The last system is a web-based application implemented in C# that allows scenographers to plan and manage scenic sets in television productions. To preserve copyright constraints, the fictitious name of PDP is used in this thesis to refer to it. Similarly to MIDAS, both selected versions of PDP are the before and after versions representing the major changes in this system evolution. The main characteristics of these systems are presented in Appendix A.

4.1.4. Procedures for Data Collection

In order to analyze the impact of code anomalies on architectural degradation symptoms the actual architecture was recovered through reverse engineering. The recovered architecture was then compared with the intended architecture, in order to identify architectural violations; i.e. symptoms of

architectural erosion (Section 2.2). In addition, the recovered architecture was analyzed to detect architectural anomalies - i.e., symptoms of architectural drift (Section 2.2). Then, the recovered architecture was analyzed to detect architectural anomalies and finally, code anomalies were identified. The last activity encompassed the analysis of the impact of code anomalies on architectural degradation. These activities were performed for each version of the target systems, as presented bellow.

Recovering the Architectural Components Design. This activity was based on a semi-automatic process. We used Sonar (2009), Understand (2009) and NDepend (2009) to support the recovery of the actual architecture from the source code (Section 2.2.2) in the target systems. We relied on those tools because they support architecture and code analysis, helping architects and developers to measure modularity in both levels. Additionally, these tools are complementary: Sonar analyzes Java programs, while NDepend and Understand analyze C++ and C# programs, respectively.

Furthermore, Sonar, Understand and NDepend provide mechanisms to extract the relationships (i.e. mappings) between code elements and architectural elements in different granularity levels. That is, by using these tools we could extract the set of code elements (e.g. packages, sets of classes defined or not in the same package) that are in charge of implementing an architectural component (Section 2.1). These mappings allow to better trace and, hence, understand the influence of a code anomaly on the actual architecture design.

Recovering the Architectural Concerns. The recovery of architectural concerns encompassed two main steps: the selection of architectural concerns and their projection on the system implementation. Architects and developers selected only architectural concerns that were clearly relevant according to their knowledge of the target systems. As a result, six architectural concerns were chosen from Aspectual Watcher and Health Watcher: Concurrency, Distribution, Persistence, Complaint, Health Unit, and View; nine architectural concerns were selected from AspectualMedia and MobileMedia: Security, Concurrency, Screen, Persistence, Photo, Music, Video, Sorting, and Favorite; four architectural concerns were selected from MIDAS: Fault Tolerance, Service Discovery, Dynamic Adaptation, and Engine; and six architectural concerns were selected from PDP: Transaction,

Persistence, Service, Photo, Area, and Attachment. Descriptions of these concerns are provided in Table 4.3.

These architectural concerns were chosen because (i) influenced the key architects' design decisions or (ii) represent important functionalities of the system modularized in architectural components, and (iii) are different in terms of functionality, scattering degrees, and granularity levels. Therefore, the selection and assessment of different kinds of architectural concerns were important to enable us to observe the relationship between code anomalies and architectural problems.

Table 4.3: Architectural concerns considered in the study.

System	Architectural Concern	Description
Health Watcher	Complaint	It manages all kind of complaints in the system.
	Concurrency	It provides a control for avoiding inconsistent information stores in the system database.
	Distribution	It externalizes the system services at the server side and supporting their distribution to the clients.
	Health Unit	It manages information regarding health units.
	Persistence	It retrieves and stores the information manipulated by the system.
	View	It processes the web requests submitted by the system users.
MobileMedia	Concurrency	It provides a control for avoiding inconsistent information stores in the system database.
	Favorite	It provides services to set favorite media and visualize them.
	Music	It manages all the music data stored in the system.
	Persistence	It retrieves and stores the information manipulated by the system.
	Photo	It manages all the pictures stored in the system.
	Screen	It processes the requests submitted by users.
	Security	It improves the user's privacy and requires authentication to access to albums (i.e. login and password).
	Sorting	It provides a service for sorting media by the number of accesses.
	Video	It manages all the video data stored in the system.
MIDAS	Dynamic Adaptation	It provides flexible configuration of resources and optimizes the performance of mobile applications.
	Engine	It provides the core services of the middleware.
	Fault Tolerance	It delivers the system services to their recipients under all conditions.
	Service Discovery	It supports the ability of a client to discover or invoke a service without prior knowledge of its physical location.
PDP	Attachment	It manages all the utility equipment in the system.
	Area	It manages all the film sets in the system.
	Photo	It manages all the pictures stored in the system.
	Persistence	It retrieves and stores the information manipulated by the system.
	Service	It manages all the service interfaces in the system.
	Transaction	It manages all the transactions in the system.

A second step was dedicated to manually produce the projection of these architectural concerns on the system implementation. To this end, architects and developers of the target systems worked together for more than 4 months on carefully reviewing the system code to produce such projections. This stage had to be conducted manually since architecture recovery tools do not support the extraction of architectural concerns (Section 2.1) from the system implementation. Some tools were used to mapped the extracted concerns such as ConcernMapper (2010) and Cide (2010). The main motivation for the architectural concern extraction is that there are several architectural anomalies documented in the literature (Garcia *et al.*, 2009, Stal, 2009), which are related to the inappropriate modularization of architectural concerns. Therefore, the mappings between code elements and architectural concerns allow identifying whether and how the inappropriate modularization of such concerns in the code is related to degradation symptoms in the recovered architecture.

Identifying Architectural Degradation Symptoms. In order to identify symptoms of architectural erosion (Section 2.2) we used the Software Reflexion Model (Murphy *et al.*, 2001). The comparison of the actual, extracted architecture (EA), and the intended architecture (IA) was supported by two groups of architects: (i) those that defined the original intended architecture, and (ii) independent reviewers of the system architecture. We measured the architecture conformance in terms of *convergence* (a component or relationship that is in both EA and IA), *divergence* (a component or relationship that is in EA but not in IA), and *absence* relationships (a component or relationship that is in IA but not in EA). Although absence and divergence classifications are natural suspects of possible architectural violations, they must be validated with systems architects and developers. Certain absence and divergence classifications can be motivated by mistakes or wrong decisions introduced in the IA that are subsequently corrected in the EA. Therefore, in those cases, absence and divergence classifications cannot be classified as violations. In the context of our study all the absence classifications were confirmed by architects and developers as violations. That did not occur with divergences, which were only classified as violations in certain cases.

Furthermore, symptoms of architectural drift (Section 2.2.3) were detected by architects in the EA based mainly on: (i) a visual inspection, and (ii) a careful

analysis of the mappings between code elements and architecture elements in that architecture, due to the lack of tools for doing so automatically (Section 2.2.4.2). We also asked the system architects to indicate other architectural drift symptoms observed in the EA beyond those documented in existing catalogs, such as cyclic dependencies. This helped us to better judge whether and which code anomalies are good indicators of architecture problems.

As a result of this stage, architects provided reports describing the architectural problems observed in each system version. These reports described, for instance, the architecture degradation symptom, its location in the design, the architectural elements related to it and, in some cases, an explanation of the problem cause. The final subset of architectural drift symptoms (i.e. architectural anomalies) encompassed those architectural anomalies that were identified by architects in the target systems of our study. These anomalies are summarized in Table 4.4.

Table 4.4: Architectural anomalies analyzed in the study.

Architectural Anomaly	Definition
Ambiguous Inteface	The interface offers only a single general entry-point and, hence, it can handle more requests than it should actually process
Extraneous Connector	Connectors of different types are used to link a pair of components
Cyclic Dependency	A relation between two or more architectural elements that depend on each other either directly or indirectly.
Connector Envy	A component realizes functionality that should be assigned to a connector
Scattered Parasitic Functionality	A high-level concern is spread over multiple modules that implement different concerns.
Redundant Interface	Interface that require the same information of other interfaces
Overused Interface	Interface that requires a lot of data or is required by several interfaces.
Component Concern Overload	A component is responsible for realizing two or more unrelated system's concerns

Detecting Code Anomalies. As a first step, code anomalies were automatically identified using conventional detection strategies (Marinescu, 2004) - similarly to other studies (Khomh *et al.*, 2009; Olbrich *et al.*, 2009; Olbrich *et al.,* 2010; D'Ambros *et al.*, 2010). Therefore, we studied code anomalies for each conventional documented detection strategy. These anomalies are summarized in Table 4.5. However, existing tools for applying detection strategies (Ratiu *et al.*, 2004; Marinescu *et al.*, 2010; Moha *et al.*, 2010; Mara *et al.*, 2011) suffer from limitations that hinder their practical use. First, they do not support all the published strategies. Second, they do not allow developers to adjust metrics and

thresholds according to the particularities of a system. Thus, we decided to collect code metrics used in the strategies with well-known code analyzers (Together, 2009; Understand, 2009) and then, combine their outcome according to the strategy definition. We relied on these code analyzers because they complement each other and together collect all the required metrics.

Table 4.5: Code anomalies analyzed in the study.

Code Anomaly	Definition
Anonymous Pointcut	occurs whenever a pointcut is directly defined in the advice signature.
Composition Bloat	is a complex base computation that is advised by multiple aspects and leads to complex advice implementations in one or more aspects
Disperse Coupling	refers to a code element that accesses many other code elements.
Duplicate Pointcut	is associated with full pointcut expressions equivalent to others that have already been defined.
Feature Envy	refers to a method that seems to be more interested in the data of other elements than those available in its class.
Forced Join Point	is associated with elements (attributes or methods) in the base code that are only exposed to be used by aspects
God Aspect	occurs when an aspect is realizing more than one concern.
God Class	occurs when a class centralizes the system functionalities (realizes more than one purpose).
God Pointcut	occurs when a pointcut has either a complex expression and the respective advice has a complex implementation.
Idle Pointcut	refers to pointcuts that do not match any join point
Intensive Coupling	refers to a code element that has tight coupling with other methods, and these methods are defined in the context of few classes.
Lazy Aspect	is an aspect that has either none or only fragmented responsibility
Long Method	occurs when a method has grown too large.
Lazy Class	refers to classes that are not doing much useful work and should be eliminated.
Long Parameter List	refers to a long list of parameters in a procedure or function make readability and code quality worse.
Misplaced Class	refers to a class that depends on classes from other packages more than those from its own package.
Redundant Pointcut	is associated with partial (not full) pointcut expressions equivalent to others that have already been defined.
Shotgun Surgery	refers to a method that has many other code elements depending on it, hindering the reusability of the infected method.

In this process we selected metrics and thresholds that have shown high accuracy to identify these code anomalies in previous studies (Lanza and Marinescu, 2006; Khomh *et al.*, 2009; Olbrich *et al.*, 2009; Olbrich *et al.,* 2010; D'Ambros *et al.*, 2010). In some cases, the thresholds suffered some minor adjustments in order to maximize the strategy's accuracy. For instance, certain thresholds were calibrated according to the specific programming styles and system characteristics. The goal was to get the best possible accuracy rates with the conventional detection strategies at hand, according to our knowledge on the

software systems. If needed, the changes in the original detection strategies were discussed with the system developers. A complete list of the employed detection strategies and their corresponding thresholds is available in Appendix B.

As a second step, the list of suspects identified by conventional strategies was validated by the developers. That is, developers analyzed whether the detected code structure was really infected by a given anomaly. At least three developers were involved in this process for each system. All the developers had previous experience on the identification and refactoring of code anomalies. As a result, we considered those code structures that were confirmed by all the developers involved in this process. This validation was motivated by the fact that conventional detection strategies present false positives and false negatives when identifying code anomalies (Marinescu, 2004). Consequently, by mixing automatic with manual detection, we aimed at finding a reliable set of code anomalies.

Analyzing the Impact of Code Anomalies on the Architecture. In order to analyze the relationships (i.e. correlation and cause-effect) between code anomalies and architecture degradation symptoms, developers and researchers first analyzed reports provided by the architects. As previously detailed, these reports included fine-grained and accurate details about identified architecture problems, facilitating the correlation analysis. Moreover, the following heuristics were also systematically applied to infer each cause-effect relationship: first, we observed whether the same structural modifications caused simultaneously a code anomaly and an architecture problem. Second, we checked whether the definition of an anomalous code element introduced an architectural problem. Finally, we examined whether changes performed considering the evolution of an anomalous code element caused an architecture problem. In the context of this thesis, the term "anomalous" refers to those code elements infected by at least one code anomaly.

The analysis also relied on a set of criteria to validate the cause-effect relationship between code anomalies and architecture problems. First, the cause-effect relationship was recurrently inferred in almost all systems versions, and for many of the involved code anomaly occurrences and architecture problems. Second, the cause-effect relationship was observed in different components of the same system and, additionally, these components involved the contribution of

different developers. Lastly, all the inferred cause-effect relationships were confirmed by architects and developers involved in this activity. As a result, we produced a ground truth of architecturally-relevant code anomalies for each target system. These lists included fine-grained and accurate details about the code anomalies facilitating further analyses (e.g. Chapter 5). For instance, these lists described the code elements affected by anomalies, the code anomalies type, and the architectural problems such affected code elements are related to.

4.2. Findings on the Impact of Code Anomalies

The following sections present and discuss the main findings associated with each of the aforementioned research questions (Section 4.1). Section 4.2.1 discusses whether anomalous code elements are related to architectural degradation symptoms. Section 4.2.2 reports the observations about the impact of two anomaly characteristics (type and earliness) in the architectural degradation symptoms. Finally, Section 4.2.3 discusses whether and to what extent refactorings were effectively applied to remove architecturally-relevant code anomalies in the target systems.

4.2.1. Are Anomalous Code Elements Architecturally-Relevant?

In order to investigate whether anomalous code elements (classes, aspects, methods and pointcuts) are more related to architectural degradation symptoms than anomaly-free code elements, Fisher's exact test (Shesking, 2007) was first applied. It checks whether the proportion of architectural degradation symptoms varies across classes and aspects with or without code anomalies. This test was selected because the study deals with small samples. Therefore, the use of another test, such as Chi2 (Shesking, 2007) could produce erroneous results because of the approximation.

We have also calculated the *Odds Ratio* (ORs) (Shesking, 2007) to check whether anomalous classes and aspects have the same probability to be related to architecture problems that anomaly-free code elements. Results of Fisher's test for both architectural violations and architectural anomalies are presented in Tables

4.6 and 4.7, respectively. Note that Table 4.6 shows the impact of code anomalies on decisions of the intended architecture, while Table 4.7 shows their impact on the implemented architecture. In these tables, the versions of MIDAS and PDP named as "*BEF*" and "*AFT*" correspond to the version before and after applying major changes, respectively. Also, lower *p-values* indicate that code elements with code anomalies adversely impact architecture design more than anomaly-free code elements. Data for MIDAS and PDP are not presented in Table 4.6 as no violation occurred in these systems. This finding was expected as the development process in these projects strictly enforced architecture conformance.

Table 4.6: Fisher's test results for architectural violations.

Releases	p-values	ORs	Releases	p-values	ORs
Aspectual Watcher			**Health Watcher**		
1.0	< **0.05**	0.8	1.0	< **0.05**	4.2
4.0	< **0.05**	2.4	4.0	< **0.05**	6.0
7.0	< **0.05**	3.3	7.0	< **0.05**	5.1
10.0	< **0.05**	3.9	10.0	< **0.05**	2.8
Aspectual Media			**MobileMedia**		
1.0	0.29	1.7	1.0	0.063	2.0
3.0	0.38	2.5	3.0	0.169	2.3
5.0	< **0.05**	4.2	5.0	< **0.05**	2.9
8.0	< **0.05**	7.3	8.0	< **0.05**	3.1

Table 4.7: Fisher's test results for architectural anomalies.

Releases	p-values	ORs	Releases	p-values	ORs
Aspectual Watcher			**Health Watcher**		
1.0	0.36	1.8	1.0	**< 0.05**	2.6
4.0	< **0.05**	2.4	4.0	< **0.05**	2.0
7.0	< **0.05**	3.1	7.0	< **0.05**	4.2
10.0	< **0.05**	3.9	10.0	< **0.05**	10.2
Aspectual Media			**MobileMedia**		
1.0	< **0.05**	2.3	1.0	< **0.05**	3.3
3.0	< **0.05**	2.1	3.0	< **0.05**	3.2
5.0	< **0.05**	4.5	5.0	< **0.05**	6.4
8.0	< **0.05**	3.6	8.0	< **0.05**	9.1
MIDAS			**PDP**		
BEF	0.07	1.9	BEF	< **0.05**	3.8
AFT	< **0.05**	2.1	AFT	< **0.05**	3.2

Architectural Significance of Code Anomalies. Our analysis revealed a statistically-significant relationship between anomalous code elements and architectural degradation symptoms in 77.5% of the analyzed versions, according to the desired level of confidence (i.e. 0.05). Also, the odds ratio revealed that anomalous code elements were related to architectural degradation symptoms with

two or more times higher probability than anomaly-free code elements. However, the significance of this relationship was not statistically confirmed in the first version of certain systems. In these first versions, anomalous code elements were scattered through all the architecture components, while architecture problems were only found on certain components. On the other hand, degradation symptoms emerged in multiple components over time as a consequence of changes performed on anomalous code elements. Thus, the relationship between anomalous code elements and architectural degradation symptoms was statistically confirmed in later versions. This finding reveals that developers should be more concerned with refactoring anomalous code elements in the first system version. Certain early code anomalies require special attention even when they do not represent a threat to the architecture design (Section 4.2.2.2). Taking into consideration Fisher's test results and the aforementioned gathered evidence, $H1_0$ can be rejected according to the desired level of confidence (i.e. 0.05).

Upstream and Downstream Analyses. In order to complement the Fisher's test analysis, we analyzed the upstream and downstream relationships between code anomalies and architectural degradation symptoms. The *upstream* analysis refers to what extent code anomalies were related to architectural degradation symptoms, while the *downstream* corresponds to what extent architectural degradation symptoms were related to code anomalies. These analyses are useful because they show: (i) the proportion of code anomalies that is critical for the system architecture, and (ii) the proportion of architectural degradation symptoms that could be fixed by refactoring code anomalies. The upstream analysis showed that up to 81% of analyzed code anomalies were correlated to architecture problems in Health Watcher, 72% in MobileMedia, 68% in Aspectual Watcher, 65% in PDP, 63% in Aspectual Media and 51% in MIDAS. As we can notice, a considerable proportion of code anomalies did not impact the architecture design in the analyzed systems. This finding highlights the need for understanding whether certain characteristics of the code anomaly were more likely to be related to the source of architecture problems (Section 4.2.2).

On the other hand, a downstream analysis revealed that up to 86% of all architecture problems were caused by code anomalies in Health Watcher, 83% in Aspectual Media, 80% in Mobile Media, 75% in Aspectual Watcher, 71% in PDP, and 70% in MIDAS. These results indicated that the vast majority of problems in

the extracted architecture were caused by anomalous code elements. We found high cause-effect rates through the evolution of components where conformance of architectural rules was strictly enforced in the source code, which was particularly surprising. The MIDAS project is the best example for such finding: although design rules were not violated, many occurrences of anomalies in the extracted architecture emerged over time. They were found in anomalous code elements with low cohesion and high coupling. Even though most of these code elements implemented a single architectural component, they realized multiple scattered architectural concerns. This phenomenon was often caused by broadly scoped scattered architectural concerns, such as *Service Discovery*, *Fault Tolerance Policies*, and *Dynamic Adaptation*. Each of these architectural concerns should have been modularized in a single component.

The results of these multi-dimensional analyses seem to confirm that the detection of code anomalies is useful to locate potential sources of architectural erosion and drift (Section 2.2). This suggests in turn that the early application of systematic code refactoring could effectively contribute to combat symptoms of architecture degradation. This observation is even more relevant for projects where there is neither effort on proactive maintenance of architecture documentation nor investment on using heavyweight architectural conformance tools (Section 2.2.2). We also found that a considerable proportion of all code anomalies, about 40%, were not indicators of relevant design problems. This means that refactorings cannot be chosen arbitrarily when architecture revisions of the source code are being carried out. Developers should be equipped with guidance and tool support to identify and rank code anomalies according to their relevance to architecture degradation. In this context, the next section discusses whether certain characteristics of code anomalies were better indicators of architecture problems in the analyzed systems.

4.2.2. Are Particular Characteristics of Code Anomalies Indicators of Architectural Degradation Symptoms?

Once confirmed that anomalous code elements were often related to architectural degradation symptoms in the target systems, we investigated the role played by the type and earliness of code anomalies in this context. Additionally,

the results of the previous section also reinforced the importance of studying the earliness of code anomalies.

4.2.2.1. Type of Code Anomalies

A logistic regression model (Hosmer and Lemeshow, 2000) was used to investigate to what extent particular types of code anomalies were related to degradation symptoms in both system architectures. This method was selected because it predicts whether code elements infected by a particular type of code anomaly are likely to be related to architectural degradation symptoms. The closer the value of the regression model is to 1, the higher is the likelihood that the code anomaly relates to an architectural degradation symptom. Differently from the Fisher's test, in this method we considered only those cases where the cause-effect relationship between code anomalies and architectural degradation symptoms were confirmed by architects and developers (Section 4.1.4).

We use regression models as an alternative to the Analysis Of Variance (ANOVA) for dichotomous variables. Then, for each code anomaly and for the 40 analyzed versions, we count the number of times that the *p-values* obtained by the regression model were significant. In this sense, as the regression model demands, we discarded variables that were highly correlated to others. Therefore, the model contains a non-redundant set of code anomalies.

Tables 4.8 and 4.9 show the results of the logistic regression model for the contribution of each type of code anomaly on architectural degradation symptoms. In particular, these tables summarize the total number of analyzed releases for which each type of anomaly was statistically-significant in the regression model. For instance, Table 4.9 shows that *Long Method* was statistically-significant for causing architectural anomalies in 8 of out 10 versions of Health Watcher. Most of the *Long Methods* detected in Health Watcher implemented tangled concerns. We highlight in boldface those anomalies that present a significant *p-value* for at least 70% of the releases where they occurred. This threshold was previously documented in the literature and used in other studies with similar analytical purposes (Khomh *et al.*, 2009).

Table 4.8: Significant p-values for architectural violations.

Type of Code Anomalies	AW	HW	AM	MM
Anonymous Pointcut	5	**-**	3	-
Composition Bloat	3	**-**	4	-
Disperse Coupling	5	**7**	**5**	5
Duplicate Pointcut	4	-	4	-
Feature Envy	1	2	2	2
Forced Join Point	2	-	1	-
God Aspect	4	-	2	-
God Class	3	5	7	5
God Pointcut	**7**	**-**	4	-
Idle Pointcut	0	**-**	0	-
Intensive Coupling	**6**	**9**	5	**8**
Lazy Aspect	0	**-**	0	-
Lazy Class	4	3	3	**6**
Long Method	**7**	**8**	4	**6**
Long Parameter List	**7**	2	5	3
Misplaced Class	1	1	2	3
Redundant Pointcut	4	-	3	-
Shotgun Surgery	3	2	4	5
Lazy Class	4	3	3	**6**

Table 4.9: Significant p-values for architectural anomalies.

Type of Code Anomalies	AW	HW	AM	MM	MIDAS	PDP
Anonymous Pointcut	4	-	2	-	-	-
Composition Bloat	**7**	-	**5**	-	-	-
Disperse Coupling	4	**6**	2	**5**	**2**	**2**
Duplicate Pointcut	3	-	3	-	-	-
Feature Envy	2	2	1	1	0	1
Forced Join Point	2	-	2	-	-	-
God Aspect	5	**-**	**5**	-	-	-
God Class	4	**7**	4	**5**	**2**	**2**
God Pointcut	3	-	**6**	-	-	-
Idle Pointcut	0	-	0	-	-	-
Intensive Coupling	4	**7**	5	5	**2**	**2**
Lazy Aspect	3	-	2	-	-	-
Lazy Class	4	4	3	**6**	**2**	1
Long Method	5	**8**	4	**7**	**2**	**2**
Long Parameter List	1	2	2	1	0	0
Misplaced Class	1	1	2	0	-	0
Redundant Pointcut	2	-	4	-	-	-
Shotgun Surgery	2	**7**	3	4	**2**	**2**

A first analysis of Tables 4.8 and 4.9 suggests that none of the types of code anomalies (Fowler *et al.*, 1999; Chapter 3) stand out as the best indicator of architectural degradation symptoms. None of them were clearly the best indicator across all the systems. Thus, the second null hypothesis, $H2_0$, cannot be rejected for the *type* anomaly characteristic. However, certain types of anomalies were significantly related to architectural degradation symptoms. In certain cases, the high cause-effect relation was consistently observed regardless of the analyzed

system. For instance, 77% of *God Classes*, 71% of *Long Methods*, and 64% of *Intensive Coupling* instances introduced architectural degradation symptoms.

There were also situations where the high cause-effect relationship was observed in the majority of, albeit not all, the systems. This is the case of code anomalies such as *Long Parameter List* and *Lazy Class*, which presented interesting and distinct effects. For instance, about 63% and 22% of *Long Parameter List* introduced architectural violations in later versions of aspect-oriented and object-oriented systems, respectively. Artificial parameters had to be created through aspect-oriented system evolution allowing aspects to access restricted contextual information, thereby breaking the intended encapsulation. This finding suggests that certain types of code anomalies could have been a typical source of architecture degradation symptoms, depending on the type of the adopted programming language and architecture decomposition; however, this suggestion should be tested in further studies.

Our results reveal that no code anomaly type stands out as the best indicator of the code anomaly harmfulness on architectural designs. Other characteristics associated with the code anomaly evolution may be indicators of architectural degradation symptoms. For instance, the incremental increase in the number of changes that a method suffers over time can indicate the presence of *Disperse Coupling*. Therefore, the harmful nature of certain types of code structures might be only confirmed over time. In this context, the next section analyzes whether the earliness of anomalies is a better indicator of their harmful impact on both system architectures.

4.2.2.2. Earliness of Code Anomalies

Interesting results emerged when analyzing the influence of early code anomalies on architecture degradation. We defined early code anomalies as those anomalies that appeared in the first version of each system. In order to analyze the impact of early code anomalies over the systems evolutions we used history-sensitive tools for anomaly detection (Mara *et al.*, 2011; Ratiu *et al.*, 2004). They allowed us to analyze to what extent changes performed on code elements,

infected by early anomalies, resulted in architectural degradation symptoms in later versions.

Harmfulness of Early Anomalies. Our analysis revealed that more than 18% of all architecturally-relevant code anomalies emerged from anomalous code elements in the systems first version. Although 18% is not a high percentage for applying any statistical method and, hence, reject $H2_0$, these early code anomalies were critical to the system architecture: they were related to more than 37% and 31% of all violations and architectural anomalies, respectively. In particular, early code anomalies induced architectural anomalies, which in turn led to violations. Both proportions point out the importance of understanding the impact of certain categories of early code anomalies. Thereby, developers should be able to detect occurrences of architecturally-relevant code anomalies as early as possible and remove them through refactoring.

Figure 4.1 depicts an example of the evolution of the method MediaController.handleCmd(..), an early *Long Method* extracted from the MobileMedia system. This figure shows the major changes performed on this method in terms of: (i) coupling strength, and (ii) its contribution to implement system concerns. The width of each arrow is linearly proportional to the cardinality of low-level dependencies abstracted in the corresponding edge (Lungu and Lanza, 2007). Low-level dependencies, in this context, refer to method invocations, type references and attributes accesses from other classes that a given method executes. Figure 4.1 also relies on the distribution map (Ducasse, 2006), a technique for visualizing system properties throughout evolution, where colors represent the dominant concern in each class.

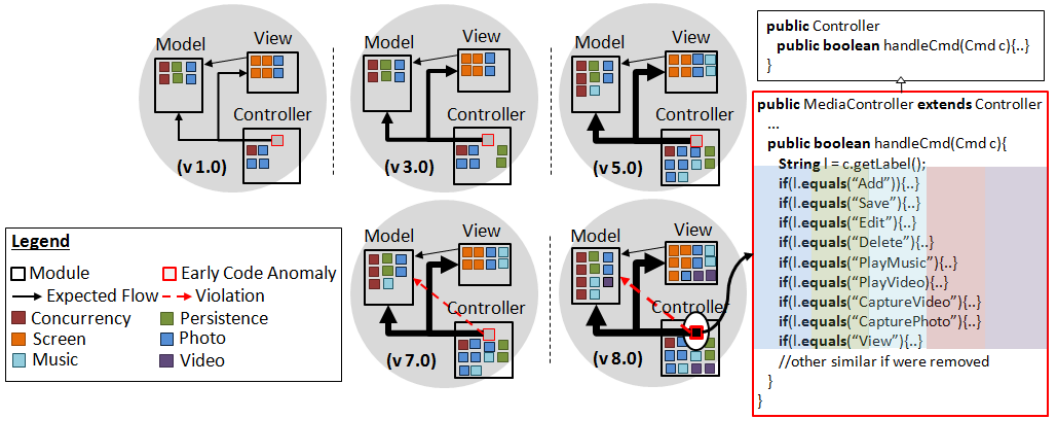

Figure 4.1: Impact of an early code anomaly through the system evolution.

As shown in Figure 4.1, the number of concerns implemented by components increase over time. In some cases, certain code elements were responsible for increasing the number of concerns realized by the same component. For instance, all if statements defined in MediaController.handleCmd() realize multiple concerns in version 8.0 which are represented by vertical bars of different colors. Hence, this method significantly contributed for the introduction of the *Component Concern Overload* and *Scattered Parasitic Functionality* architectural anomalies on the Controller.

Additionally, MediaController.handleCmd(..) handles more types of commands than it will actually process, by accepting the parameter c, typed as the generic typeCmd. This generic parameter makes the method harder to understand because it does not reveal which commands the method is interested in. This situation worsened over time due to the incremental addition of new commands. Therefore, the *Ambiguous Interface* was progressively introduced by the changes made in the method body.

Characteristics of Harmful Early Anomalies. Early code anomalies were recurrently the source of later relevant problems when they infected the API of critical points of the system architecture. Examples of these critical points are component interfaces and super-classes. When code anomalies infect component interfaces, they might be propagated to the: (i) internal code of the component, and (ii) coupled components over system evolution. This situation occurred for 49% of all relevant code anomalies in Health Watcher, 42% in PDP, 38% in MobileMedia, 31% in Aspectual Watcher and MIDAS and 24% in Aspectual

Media. These proportions were directly related to the number of code anomalies infecting critical points in each system.

Moreover, when anomalies infect super-classes they might be propagated from parents to children in inheritance trees. This situation occurred with 53% of all relevant code anomalies in MobileMedia, 50% in Aspectual Media, 45% in Health Watcher, 36% in MIDAS, 33% in Aspectual Watcher and 24% in PDP. The infected super-classes had a considerably negative effect as the architectural violations and architectural anomalies were propagated down through the class hierarchies. For instance, the class MediaController inherits the *Ambiguous Interface* occurrence from its parent - the class Controller (Figure 4.1). As MediaController overrides the handleCmd method, developers were forced to use if statements to handle the commands in which the method is interested.

Finally, it was observed that early code anomalies emerged in code elements that accessed information from classes defined in multiple architecture components. These code elements also implemented several system concerns. The method MediaController.handleCmd(..) is an example of a code element that suffers from this co-occurrence in MobileMedia. About 85% of these code elements introduced architecture problems in the analyzed systems. The issue is that these code elements were the subject of changes associated with each of the system's concerns that they dealt with. Also, the increasing coupling strength was a reason why various changes were applied to this method during its evolution. The diverse nature of the changes confirmed the harmful impact of this co-occurrence on the system architectures.

Based on the gathered results, $H2_0$ cannot be rejected. Although certain types of code anomalies were related to architectural degradation symptoms, none of them emerged as a reliable indicator of degradation symptoms across all the systems. In the context of the earliness characteristic, some anomalies that appear in early versions were related to architectural degradation symptoms introduced in latter versions. However, those code anomalies did not represent a significant proportion of all code anomalies.

4.2.3. Are Architecturally-Relevant Code Anomalies often Refactored?

The aforementioned findings motivated us to investigate whether performed refactorings were targeted at removing architecturally-relevant anomalies in the target systems. To this end we analyzed two groups of refactorings: high-level and low-level refactorings (Murphy-Hill *et al.*, 2009). The former group involves API-level changes – that is, structural modifications that affected interfaces and services. For example *Renaming Public Methods* and *Moving Classes* fall in this category. The latter group implies narrowly-scoped changes, which do not affect the clients of the component being modified. For example, *Renaming Local Variables* or *Extracting Private Methods* (Fowler *et al.*, 1999) are considered low-level refactorings.

We focused on the analysis of both categories of refactoring. Even though there are attempts to automate their detection (Dig *et al.*, 2006; Kim *et al.*, 2011), such tools are not available yet. Therefore, we needed to partially rely on the manual inspection of the source code for identifying their occurrences. We analyzed the commit messages that indicated the occurrence of a refactoring in a given revision. This analysis was possible because the majority of the commit messages followed a template, which allows developers to recover the purpose of each particular commit. Also, we relied on using structural diff tools (ydiff, 2010; Hashimoto, 2008) to analyze refactorings when commit messages were absent. By mixing both strategies we aimed at improving the reliability of our analysis. Finally, we analyzed a total of 217 refactorings in Health Watcher, 160 in MobileMedia, 112 in PDP, 97 in Aspectual Watcher and 72 in Aspectual Media.

Figure 4.2 depicts the proportions of refactorings that contribute and do not contribute to remove at least one architecturally-relevant code anomaly per target system, i.e. effective and non-effective refactorings, respectively. As it can be observed a low number of refactorings removed architecturally-relevant code anomalies. In particular, refactorings were responsible for removing only up to 37% of all architecturally-relevant code anomalies. This means that the majority of architecturally-relevant code anomalies remained in the code as the system evolved.

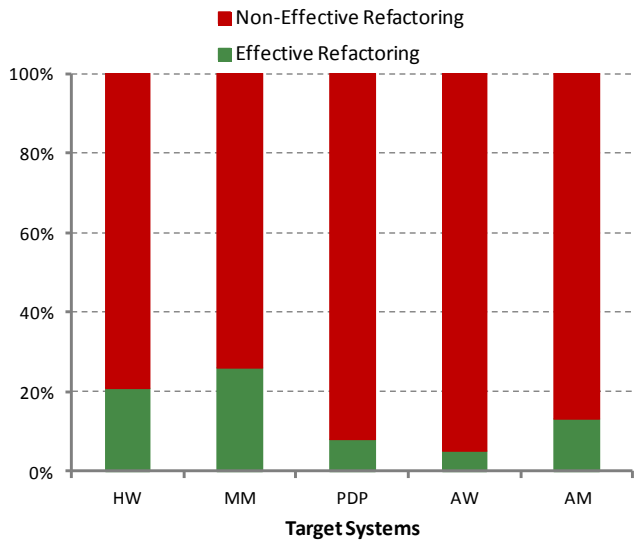

Figure 4.2: Refactorings and architecturally-relevant code anomalies.

In order to investigate the significance of this finding, the Kolmogorov-Smirnov (Shesking, 2007) test was first applied. Due to the samples did not present a normal distribution, the (non-parametric) Mann-Whitney test (Shesking, 2007) was then applied. This test was selected because it compares two sets of variables and assesses whether their difference is statistically significant. Additionally, non-parametric tests do not require any assumption on the underlying distributions. In the context of this study, the two samples to be compared are: the number of applied refactorings in the target systems and the number of refactorings targeting architecturally-relevant code anomalies (Section 4.1.2). Additionally, we compute the Cohen's *d* effect size (Shesking, 2007) to indicate the magnitude of the effect of a treatment on the dependent variables. The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$ and large for $d \geq 0.8$.

Table 4.10 reports the results of Mann-Whitney test and Cohen's effect size. As it can be noticed, Mann-Whitney test shows statistically significant differences between effective and non-effective refactorings. Moreover, Cohen's *d* effect size value is large: 2.59. Therefore, the applied refactorings did not significantly contribute to remove architecturally-relevant code anomalies and, hence, the third null hypothesis, $H3_0$, cannot be rejected.

Table 4.10: Proportions of effective and non-effective refactorings.

	Mann-Whitney *p-value*	Cohen's effect *d*
Effective vs. Non-Effective Refactorings	**< 0.01**	2.59

A deep analysis of this finding revealed that the low rate of removed architecturally-relevant code anomalies may have three main reasons: (i) a low frequency of high-level refactorings, (ii) a high frequency of low-level refactorings, (iii) the cost and effort to perform complex refactorings, and (iv) the inability of current tools for supporting complex refactorings. Even though these were not the unique reasons why architectural degradation symptoms were left in the code, they were the most frequent ones.

High-level vs. Low-Level Refactorings. Only 73 of applied refactorings were of high-level nature in Health Watcher, 41 in MobileMedia, 12 in Aspectual Media, 9 in PDP and 7 in Aspectual Watcher. Thus, less than 34% of refactorings were of high-level nature, which implies that only few refactorings had an impact of wide scope. The most frequent high-level refactorings were: move public member (16%) and extract class or extract super-class (12%). As it can be noticed, they represent a minority of the total number of applied refactorings. These refactorings are stronger candidates for removing architecturally-relevant code anomalies, as they modify the code element signature. However, their application was often confined to later versions where instabilities clearly achieved critical stages. We considered that instabilities achieved critical stages when changes needed to be performed across many code elements, belonging to multiple components, in order to add the new features.

This observation suggests that developers chose to invest their effort on architecturally-relevant refactorings on specific versions. This strategy prevailed in all systems over the option of distributing the effort through consecutive versions. For instance, the highest number of high-level refactorings in MobileMedia was applied in version 7.0 in order to support the inclusion of different requirements. Otherwise, changes associated with such requirements would be scattered and duplicated in many elements belonging to the Controller component. Thus, several classes and super-classes were extracted and many public methods had their signatures changed.

A possible reason for the rare application of high-level refactorings is that developers are not equipped with proper tools for automating this task. High-level refactorings are associated with changes on code element interfaces, which, in some cases, belong to different architectural components. The application of these refactorings, without proper tooling support, may imply in higher risks such as: unexpected breaks on the client code or undesirable semantic changes. This could be one of the reasons why developers need to rely on tools for applying this kind of refactoring, rather than applying it manually. However, recent studies have shown that developers seldom use refactoring tools (Murphy-Hill *et al.*, 2009; Arcoverde *et al.*, 2011), mostly due to usability problems. Specifically, their inability to properly visualize the effects of an applied refactoring was pointed out as a major problem (Murphy-Hill *et al.*, 2009; Arcoverde *et al.*, 2011).

On the other hand, more than 60% of refactorings were of low-level nature. This implies that the impact of most refactorings was of narrow scope. The most frequent low-level refactorings were: rename private members (32%), extract local variable (16%), and move private members (12%). As we can notice, they represent a high proportion of the total number of refactorings applied. However, these refactorings did not contribute significantly to remove architecturally-relevant code anomalies. The reason is that modifications were often confined to the internal code of the class, whereas the removal of architecturally-relevant code anomalies requires modifications in several classes.

4.3. Threats to Validity

This section discusses the threats to validity according to the classification proposed by Wohlin *et al.* (2000).

Construct validity. A first construct validity threat concerns the way we associate code anomalies with architectural problems. We are aware that code anomalies might be accidentally related to architecture problems. However, we limited such threat by considering only the code anomalies and architecture problems whose cause-effect relationship were identified and confirmed by developers and architects. Another threat concerns the set of analyzed code anomalies and architecture problems. We have tried to mitigate this threat by

using systems that suffer from the same set of anomalies than systems used in other studies (Khomh *et al.*, 2009; D'Ambros *et al.*, 2010). Lastly, construct validity could also be threatened considering how refactorings were identified. As we have relied on commit heuristics and diff tools, some refactorings might be missing.

Conclusion and external validity. Threats to conclusion validity are concerned with the relationship between the treatment and the outcome. In our study all the Fisher test and logistic regression model results were statistically-significant at the 95% level. On the other hand, threats to external validity concern the generalization of the findings. We focused on the analysis of medium-size systems. However, the development processes of large-scale systems might differ and lead to different results. We have tried to use systems of different types, implemented using different programming languages, environments (i.e. academy and industry) and with different architecture decompositions. Our target systems also present a similar density of code anomalies to large systems that were used in previous studies (Khomh *et al.*, 2009; Olbrich *et al.*, 2009, 2010; D'Ambros *et al.*, 2010). Another external validity threat concerns the generalization of the results. We plan to apply this kind of study on large open-source and industry systems.

4.4. Summary

This chapter investigated the relationship between code anomalies and architectural degradation symptoms in the implemented architecture. It also investigated whether and to what extent applied refactorings contribute to remove architecturally-relevant code anomalies. To perform these investigations, a sample of nearly 2100 anomalous code elements and 1030 architectural degradation symptoms, distributed in 40 versions of six (06) real-life software systems was considered.

Our results showed that the majority of the architectural degradation symptoms in the actual architecture emerged from anomalous code elements. These results suggest that systematic removal of code anomalies can be used to effectively combat symptoms of architecture degradation in the code. Our study also revealed how certain kinds of early code anomalies cause adverse impact on

the architecture design as the systems evolve. This means that developers should promptly identify and address them upfront; otherwise, those code anomalies are likely to contribute to the anomaly in coupled components, thereby accelerating the architecture degradation processes.

The key findings of this study are summarized as follows:

- Approximately 65% of all code anomalies were related to 78% of all architecture degradation symptoms. This result seems to confirm that detection of code anomalies is useful for locating potential sources of architecture degradation. In turn, this suggests that systematic code refactoring could contribute to address those symptoms (Section 4.2.1).
- Certain types of code anomalies were consistently related to architecture degradation symptoms (e.g. *Long Method*, *God Class* and *Composition Bloat*). However, none of them emerged as the best indicator of degradation symptoms across all the systems. Anomalous code elements often introduced degradation symptoms when implementing non-cohesive functionalities and accessing information from several components; regardless of the type of the code anomaly (Section 4.2.2.1).
- More than 18% of all architecturally-relevant anomalies emerged from anomalous code elements in the systems' first version. These early anomalies were responsible for introducing more than 33% of all architectural problems. This means that certain refactorings should be prioritized to remove code anomalies as early as possible (Section 4.2.2.2).
- About 66% of all refactorings did not contribute to fix architecturally-relevant code anomalies. These refactorings were usually confined to the private members of classes. However, as architecturally-relevant code anomalies often infected public members of interfaces and super-classes, they could only be removed by applying high-level refactoring (Section 4.2.3).

All the aforementioned findings raise questions about the effectiveness of state-of-the-art history-sensitive mechanisms for code anomaly detection (Marinescu, 2004; Ratiu *et al.*, 2004; Mara *et al.*, 2011). These tools rely on change analysis across several system versions to detect anomalies with acceptable accuracy. Therefore, they cannot reveal *early* harmful anomalies, as

their detection might occur too late, when anomaly removal might become impeditive. That concern is exacerbated by the fact that there is no knowledge about the accuracy of current state-of-art mechanisms for identifying architecturally-relevant code anomalies. The investigation of this accuracy is presented in the next chapter.