5
Analysis of Conventional Detection Strategies

The previous chapter confirmed and provided empirical evidence about the harmful impact of code anomalies on the actual architecture of software systems. Following that, the goal of this chapter is to answer the second main research question of this thesis (Section 1.4): *Whether the conventional detection strategies are able to accurately identify architecturally-relevant code anomalies? If so, to what extent?* Detection strategies (Marinescu, 2004) were selected as a conventional mechanism to identify code anomalies in this thesis due to several reasons. First, they are the most applied and studied mechanism to identify code anomalies in the literature (Section 2.3.1). These strategies have been adopted by several well-known tools that support the static analysis of the source code such as Together, 2010; Marinescu *et. al.*, 2010; Monha *et. al.*, 2011; NDepend, 2011, Mara *et. al.*, 2011. Finally, according to several studies, detection strategies tend to present accuracy rates around 60% when detecting code anomalies (Marinescu, 2004; Lanza and Marinescu, 2006), which is considered an acceptable rate (Khomh *et al.*, 2009; Olbrich *et al.*, 2009, 2010; D'Ambros *et al.*, 2010; Sjobert *et al.*, 2013).

In this context, this chapter describes a study that assesses the accuracy of conventional detection strategies when identifying code anomalies related to architectural degradation symptoms in the actual architecture. In particular, this study investigates the accuracy of conventional strategies when detecting those code anomalies classified by architects and developers as architecturally-relevant in the context of the previous study (Chapter 4). Consequently, the study presented in this chapter relies on information gathered during the data collection process described in Section 4.1.4 such as: target systems and architecturally-relevant code anomalies. The reasons that led us to rely on that information are further discussed in Section 5.1.3. This study has been published in the 11th International Conference on Aspect-Oriented Software Development (Macia *et al.*, 2012c).

PUC-Rio - Certificação Digital Nº 0912915/CA

In order to present the performed study, this chapter follows a similar structure of Chapter 4. That is, Section 5.1 describes the study definition and design. Section 5.2 presents and discusses the results of the study. Section 5.3 describes the limitations of the study and discusses how they were mitigated. Finally, Section 5.4 summarizes the key points discussed throughout this chapter.

5.1. Study Definition and Design

As aforementioned, the study presented in this chapter aims to answer the second main research question of this thesis (Section 1.4): *Whether the conventional detection strategies are able to accurately identify architecturally-relevant code anomalies? If so, to what extent?* In order to carry out this investigation, the analysis was decomposed into two perspectives by observing both violations and architectural anomalies. These perspectives allow us to better analyze the accuracy of conventional detection strategies when identifying code anomalies related to specific kinds of architectural degradation symptoms. Thus, the perspectives lead us to two research questions (RQ) as defined below.

RQ2.1: To what extent are conventional detection strategies able to accurately identify code anomalies related to architectural violations (i.e. erosion symptoms) in the actual architecture?

RQ2.2: To what extent are conventional detection strategies able to accurately identify code anomalies related to architectural anomalies (i.e. drift symptoms) in the actual architecture?

A sample of nearly 600 architecturally-relevant code anomalies was considered to answer both research questions (Section 4.2.1). This sample is representative because it involves 70% of all code anomalies found in the target systems (Section 5.1.3). Additionally, it includes object-oriented and aspect-oriented code anomalies and architectural degradation symptoms observed in the implementation of systems decomposed with such modularization techniques.

PUC-Rio - Certificação Digital Nº 0912915/CA

In this context, following Wohlin *et al.* suggestion (2000), the goal of the study is defined using the GQM format (Basili *et al.*, 1994) as:

Analyze: conventional detection strategies

For the purpose of: assessing their accuracy

With respect to: the identification of code anomalies related to architectural violations and architectural anomalies

From the viewpoint of: architects, developers and researchers

In the context of: five (05) software systems from multiple domains, implemented using various modularization techniques and following different architectural decompositions.

5.1.1. Hypotheses

The following null and alternative hypotheses have been defined in order to answer the aforementioned two research questions as shown in Table 5.1.

Table 5.1: Research questions and hypotheses of the study.

Research Questions	Hypotheses
RQ2.1	For each of the conventional detection strategies studied: **Null Hypothesis, $H1_0$**: The conventional detection strategy accurately identifies code anomalies related to architectural violations. **Alternative Hypothesis, $H1_A$**: The conventional detection strategy does not accurately identify code anomalies related to architectural violations.
RQ2.2	For each of the conventional detection strategies studied: **Null Hypothesis, $H2_0$**: The conventional detection strategy accurately identifies code anomalies related to architectural anomalies. **Alternative Hypothesis, $H2_A$**: The conventional detection strategy does not accurately identify code anomalies related to architectural anomalies.

Conventional detection strategies are considered to be accurate in the literature when their precision and recall rates are 60% or higher for identifying code anomalies (Marinescu, 2004; Lanza and Marinescu, 2006; Khomh *et al.*, 2009; Olbrich *et al.*, 2009; Olbrich *et al.*, 2010). As code anomalies are related to architectural problems (Chapter 4), we also used this threshold in this study to assess the strategies accuracy when localizing architecturally-relevant code anomalies. That is, in the context of this study a strategy is considered to be accurate whether it detects at least 60% of the architecturally-relevant code anomalies.

PUC-Rio - Certificação Digital Nº 0912915/CA

5.1.2. Variable Selection

In order to test the hypotheses, the following independent and dependent variables are defined.

Independent Variables. There are as many independent variables as there are types of code anomalies to be identified by detection strategies. 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. As described in Section 5.1.4, all code anomaly occurrences and thresholds used in testing these hypotheses were confirmed by more than twelve different developers, who have previous experience on identifying and removing code anomalies.

Dependent Variables. This study investigates the accuracy of conventional strategies to detect code anomalies related to architectural violations and architectural anomalies. Therefore, there are two Boolean dependent variables, $V_{i,k,j}$ and $A_{i,k,j}$, for $H1_0$ and $H2_0$ indicate whether the entity *i* affected by the code anomaly *k* relates to any violation or architectural anomaly in version v_j, respectively. As presented in Section 5.1.4, all the architecturally-relevant code anomalies used in testing these hypotheses were confirmed by architects and developers of the target systems (Section 5.1.3).

PUC-Rio - Certificação Digital Nº 0912915/CA

5.1.3. Selection Criteria and Target Systems

A list of criteria was documented to support the selection of suitable target systems to this study. The criteria used are presented in Table 5.2.

Table 5.2: 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 architects and developers available.
C4	has a manageable size
C5	suffers from a rich set of code anomalies.
C6	Presents multiple symptoms of architectural degradation.
C7	underwent changes
C8	was implemented by developers with different levels of programming skills.
C9	was implemented using different programming languages and modularization techniques.
C10	allows applying conventional detection strategies.

As it can be noticed, all selection criteria used in Chapter 4 (Section 4.1.3) can be used in the context of this study. Unlike the study presented in Chapter 4, criterion C10 is also mandatory for assessing the accuracy of detection strategies. The reason is because we need to ensure that conventional detection strategies can be employed in all the target systems. For instance, we cannot select software systems implemented using programming languages for which there are no detection strategies documented. Conventional detection strategies are available in the literature to identify code anomalies in both object-oriented (Marinescu, 2004; Lanza and Marinescu, 2006) and aspect-oriented implementations (Srivisut and Muenchaisri, 2007; Chapter 3). Consequently, Aspectual Watcher, Health Watcher, Aspectual Mobile, MobileMedia, and MIDAS are also considered as target systems in this study.

5.1.4. Procedures for Data Collection and Analysis

As this study relies on the target systems used in the previous study, several kinds of information have already been gathered for these systems. This implies that such gathered information can be reused in the context of this study, saving effort in the data collection process. For example, code anomalies related to architectural violations were identified and validated by architects and developers of target systems in the previous study. The same occurred for those code anomalies related to architectural anomalies in the target systems. The collection of both kinds of information is particularly relevant because they correspond to the dependent variables in this study (Section 5.1.2).

The independent variables (Section 5.1.2) were also gathered in the context of the previous study (Chapter 4). As described in Section 4.1.4, conventional detection strategies were used as a first step towards the identification of architecturally-relevant code anomalies. Additionally, the employed strategies and their corresponding thresholds were discussed and validated with developers in order to get the best possible results. This means that the code anomalies identified by detection strategies in Section 4.1.4 can be considered in the context of this study.

PUC-Rio - Certificação Digital Nº 0912915/CA

Analyzing the Accuracy of Conventional Detection Strategies. Afterwards, a stage was dedicated to investigate the accuracy of the conventional strategies when detecting the architecturally-relevant code anomalies previously identified by developers (Section 4.1.4). Therefore, this investigation was based on both lists: (i) automatically-detected code anomalies using conventional detection strategies and, (ii) lists of architecturally-relevant code anomalies detected by developers by means of code review (Section 4.1.4). In particular, the lists provided by developers were useful to assess the impact of non-automatically-detected code anomalies on architectural decompositions. In order to reject $H1_0$ and $H2_0$, we calculated the precision and recall measures of detection strategies using the following formulas:

$$\text{precision} = \frac{TP}{TP + FP} \qquad \text{recall} = \frac{TP}{TP + FN}$$

where, *True Positive (TP)* and *False Positive (FP)* encompass all automatically-detected code anomalies that respectively were or were not confirmed as architecturally-relevant by architects and developers of the systems. To determine *False Negatives (FN)* developers performed a code review to detect architecturally-relevant code anomalies that were not automatically-identified by the detection strategies; as described previously (Section 4.1.4).

It is important to reaffirm that we only considered a code element to be infected by a code anomaly when all the architects and developers involved in the process confirmed this. Therefore, a detection strategy that identifies code anomalies of type *C* achieves 100% of precision and 100% of recall if and only if it pinpoints the set of code anomalies of type *C* reported by developers and architects.

In order to illustrate how precision and recall are measured, let's consider the example presented in Figure 5.1. This figure depicts the relationship between the architecture of a software system and its implementation, where such architecture is designed following the *Layers* style. In this example, the anomalous classes C1 and C7 are considered to be architecturally-relevant because they are related to cyclic dependencies between components and violations, respectively. However, only C7 is architecturally-relevant from the seven (07) classes identified by detection strategies (C1-C7). Thus, there are: a true positive (C7), three (03) false positives (C2, C4 and C6), and a false negative (C1), which

PUC-Rio - Certificação Digital Nº 0912915/CA

indicates that the precision and recall of detection strategies are 1/4 and 1/2, respectively.

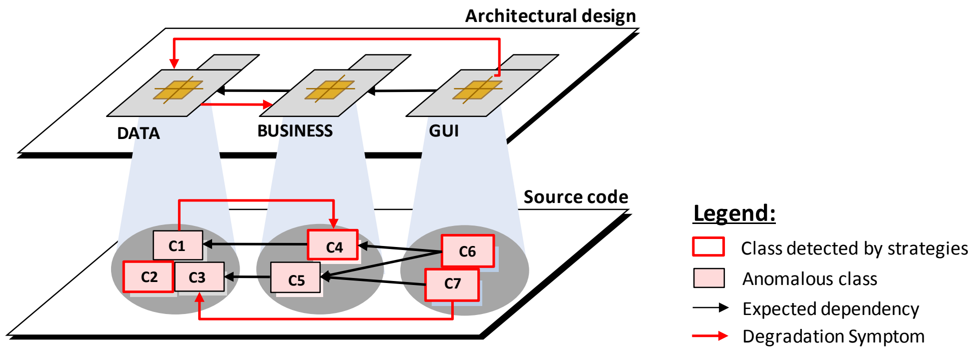

Figure 5.1: Accuracy of detection strategies.

5.2. Study Results

The precision and recall of conventional detection strategies for identifying architecturally-relevant code anomalies is summarized in Table 5.3. The token '-' is used in this table to represent the cases where code anomalies did not occur or they were not related to architectural degradation symptoms. The average of precision and recall are also presented for code anomalies in both object-oriented and aspect-oriented implementations.

In general, our analysis reveals that conventional detection strategies were inaccurate in identifying architecturally-relevant code anomalies in the target systems. Specifically, most of the code anomalies identified by conventional strategies were not associated with architectural degradation symptoms, leading to many false positives. In general, the average of the code anomalies identified by conventional detection strategies represented about 45% (or less) of the total number of code anomalies related to architectural problems. MIDAS was the only exception, which will be discussed later. Consequently, these results might imply a problem to developers who are interested in performing clean-up code revisions to avoid architectural degradation. In these cases, developers are likely to devote most of their time analyzing code anomalies that do not represent a threat to the architecture design.

PUC-Rio - Certificação Digital Nº 0912915/CA

PUC-Rio - Certificação Digital Nº 0912915/CA

Table 5.3: Results for the analyzed detection strategies.

Code Anomaly	True Positives			False Positives			False Negatives			Precision			Recall		
	HW	**MM**	**MIDAS**	**HW**	**MM**	**MIDAS**	**HW**	**MM**	**MIDAS**	**HW**	**MM**	**MIDAS**	**HW**	**MM**	**MIDAS**
Disperse Coupling	7	1	4	14	2	43	19	2	2	0.33	0.33	0.09	0.27	0.33	0.67
Feature Envy	5	2	-	27	6	-	9	3	-	0.16	0.25	-	0.36	0.40	-
God Class	1	3	2	2	4	0	4	5	1	0.67	0.43	1.00	0.33	0.38	0.67
Intensive Coupling	2	5	6	5	8	11	8	6	7	0.29	0.38	0.35	0.20	0.45	0.46
Large Class	1	1	2	2	0	4	4	1	0	0.43	1.00	0.30	0.38	0.50	1.00
Long Method	23	7	6	33	24	37	18	10	4	0.41	0.23	0.34	0.56	0.41	0.50
Long Parameter List	4	-	-	12	-	-	5	-	-	0.25	-	-	0.44	-	-
Misplaced Class	2	1	-	5	2	-	1	2	-	0.33	0.33	-	0.50	0.33	-
Shotgun Surgery	6	2	3	19	6	23	9	7	6	0.24	0.25	0.22	0.40	0.22	0.32
OO Avg. Rates										**0.35**	**0.40**	**0.33**	**0.41**	**0.44**	**0.63**
	AW	**AM**	**-**	**AW**	**AM**		**AW**	**AM**		**AW**	**AM**	**-**	**AW**	**AM**	**-**
Disperse Coupling	7	2	-	10	5	-	10	4	-	0.47	0.29	-	0.41	0.33	-
Feature Envy	16	3	-	14	7	-	14	2	-	0.53	0.30	-	0.53	0.60	-
God Class	1	2	-	1	2	-	7	4	-	0.50	0.50	-	0.13	0.33	-
Intensive Coupling	4	6	-	6	7	-	8	8	-	0.40	0.46	-	0.33	0.43	-
Large Class	1	1	-	2	1	-	5	1	-	0.33	0.50	-	0.17	0.50	-
Long Method	16	4	-	15	16	-	12	5	-	0.52	0.20	-	0.57	0.44	-
Long Parameter List	8	6	-	8	3	-	3	2	-	0.50	0.67	-	0.73	0.75	-
Misplaced Class	3	-	-	4	1	-	8	2	-	0.50	-	-	0.27	-	-
Shotgun Surgery	7	1	-	9	9	-	14	9	-	0.44	0.10	-	0.33	0.10	-
OO Avg. Rates										**0.47**	**0.32**	-	**0.38**	**0.44**	-
Composition Bloat	2	3	-	4	1	-	3	4	-	0.33	0.50	-	0.40	0.43	-
Duplicate Pointcut	8	65	-	11	47	-	3	31	-	0.42	0.58	-	0.72	0.68	-
Forced Join Point	6	1	-	6	2	-	9	6	-	0.50	0.33	-	0.40	0.14	-
God Aspect	11	6	-	11	4	-	17	9	-	0.50	0.60	-	0.39	0.40	-
God Pointcut	10	8	-	20	7	-	14	11	-	0.33	0.53	-	0.42	0.42	-
Redundant Pointcut	52	3	-	17	3	-	32	2	-	0.75	0.50	-	0.62	0.60	-
AO Avg. Rates										**0.47**	**0.50**	-	**0.49**	**0.44**	-

Even worse, many of the code anomalies that are harmful to the extracted architecture were not identified by conventional detection strategies, leading to a high rate of false negatives. Developers will miss a wide range of architectural erosion and drift symptoms. In particular, many of the conventional detection strategies exhibited recall rates close or much lower than 45%. That is, about 55% or more of the non-detected code anomalies were related to architectural problems. These results indicate that conventional detection strategies seem to have a tendency to send developers in wrong directions when addressing code anomalies related to architectural problems.

The next subsections discuss how accurate the conventional detection strategies were when localizing code anomalies related to both types of architectural degradation symptoms: violations (Section 5.2.1) and architectural anomalies (Section 5.2.2).

PUC-Rio - Certificação Digital Nº 0912915/CA

5.2.1.
Accuracy of Detecting Architectural Violations

On average about 41% of the code anomalies related to violations were identified by conventional detection strategies in the target systems. The results also show that code anomalies related to violations emerged in systems developed with both object-oriented and aspect-oriented modularization techniques. In object-oriented systems, these violations were related to undesirable interdependencies between classes responsible for implementing different architectural elements. For instance, 69% of the violations in Health Watcher were related to exception events propagated from the Data layer to the View layer. Consequently, all interfaces between Data and View layers propagated these exceptional events, even though the majority of these exceptions should be treated internally by classes defined in the Data layer according to the designers'. intent. The propagation of exception events introduced several occurrences of *Long Method*, *Misplaced Class*, *Divergent Change*, and *Shotgun Surgery*. However, just about 33% of these architecturally-relevant anomalies were detected by conventional detection strategies.

Other kinds of violations emerged in aspect-oriented systems as they follow a different architectural design. For instance 26% of the total number of

architecturally-relevant anomalies were related to undesirable tight coupling between aspects and the base code. These relations were motivated by the fact that classes were exposing internal information just to be used by aspects. For instance, artificial methods had to be created in later system versions, aiming at allowing the expected composition between aspects. This situation leads to *Interface Bloat* occurrences and to the introduction of relevant *Long Parameter Lists*, *Composition Bloat*, and *Forced Join Points*. However, conventional detection strategies were able only to identify about 40% of these relevant occurrences.

5.2.2. Accuracy of Detecting Architectural Anomalies

Architectural anomalies were mostly related to the inappropriate modularization of architectural concerns in the target systems. Exception Handling for Aspectual Watcher and Connection for Aspectual Media presented the strongest relationship with architectural modularity problems as they are very context-specific with code. Exception Handling, for instance, was scattered among different architectural components and, therefore, it was related to *Scattered Parasitic Functionality* occurrences. On the other hand, the high tangling of Connection with Persistence and Logging led to the architectural components responsible for its modularization were classified as *Component Concern Overload*. The inappropriate modularization of these concerns was associated with several occurrences of *Long Method*, *God Aspect*, *God Class*, *Divergent Change* and *Shotgun Surgery* in the target systems. Exception Handling and Connection were responsible, respectively, for 53 % and 41% of the total of architecturally-relevant code anomalies in Aspectual Watcher and Aspectual Media. However, just about 47% of these relevant anomalies were detected by conventional detection strategies.

5.2.3. Hypotheses and Overall Accuracy Results

Based on the aforementioned results, we can conclude that conventional detection strategies were not accurate in detecting architecturally-relevant code

PUC-Rio - Certificação Digital Nº 0912915/CA

anomalies (Section 5.1). Therefore, we reject both null hypotheses $H1_0$ and $H2_0$ (Section 5.1.1) for all the systems, except MIDAS (Table 5.3). Several conventional detection strategies presented recall rates greater than 60% in MIDAS. That is, more than a half of code anomalies related to architectural degradation symptoms were automatically identified by conventional detection strategies in MIDAS. We also observed that the number of architectural anomalies not related to code anomalies tend to increase compared with the other systems due to incomplete refactorings.

The MIDAS case indicated that conventional detection strategies are more effective in systems where architecture conformance is more strictly enforced in the code. The better the code modularity reflects the architecture decomposition, the fewer the number of code anomalies. This finding was not actually exclusive to MIDAS. Similar results were observed in components of MobileMedia and Health Watcher with highest adherence to the architectural design decisions. In these components (e.g., Model for MobileMedia and Business for Health Watcher) the conventional detection strategies presented precision and recall rates higher than 60%. These components also presented the lowest number of architecturally-relevant code anomalies.

Another relevant characteristic that is likely to favor the success of conventional detection strategies (i.e., accuracy rates higher than 60%) is when the projection of an architectural element occurs in a few code elements. In these cases, single code anomalies will exert a more direct impact on the architectural element that they are implementing. This phenomenon was observed in all target systems.

5.2.4. Analysis of Overlooked Code Anomalies

Once we have discussed the accuracy of the conventional detection strategies, we reflect upon the key factors that contributed to their failure in localizing architecturally-relevant code anomalies (Sections 5.2.4.1 and 5.2.4.2). This discussion can provide insights on how to improve the techniques to detect architecture degradation based on source code analysis.

PUC-Rio - Certificação Digital Nº 0912915/CA

5.2.4.1. Inability to Analyze Properties of Architectural Concerns in the Source Code

Code anomalies were often the source of architectural problems when they were located in modules realizing various architectural concerns. We noticed that 62% of the total number of architecturally-relevant code anomalies exhibited this characteristic. This frequency reinforces that detection strategies should be more sensitive to the degree of concern scattering and tangling in the code. The reason is because the employed conventional strategies were not accurate when detecting anomalies associated with the inappropriate modularization of architectural concerns; they presented precision and recall rates around 43% and 48% respectively.

For instance, the AlbumData class in MobileMedia (Figure 5.2) was classified by developers as an architecturally-relevant occurrence of *God Class* since it defines more than few methods and realizes different architectural concerns (e.g. Photo, SMS, and Persistence). However, differently from MediaController class (Figure 4.1), it was not identified by conventional detection strategies. Although AlbumData was the source of highly tangled and scattered concerns, its methods do not present high complexity in terms of *Line of Code* or *Cyclomatic Complexity* (Section 2.3.1). However, changes associated with each of the architectural concerns were performed in this class, confirming its anomalous nature. This class was particularly related to two architectural anomalies, namely *Component Concern Overload* and *Scattered Parasitic Functionality* (Section 2.2.3).

PUC-Rio - Certificação Digital Nº 0912915/CA

```
public abstract class AlbumData {
  ...
  protected MediaAccessor mediaAccessor;

  public String[] getAlbumNames() {
    mediaAccessor.loadAlbums();
    ...
  }
  ...
  public MediaData[] getMedias(..) throws ...  {
    ...
    result = mediaAccessor.loadMedia(..);
    ...
  }

  public void createNewAlbum(..) throws ... {
    mediaAccessor.createNewAlbum(albumName);
    ...
  }

  public void addVideoData(..) throws ... {
    ((VideoAccessor)mediaAccessor).addVideo(..);
    ...
  }
  ...
  public void addImageData(..) throws .. {
    ...
    ((ImageAccessor)mediaAccessor).addImage(..);
    ...
  }
  ...
}
```

Legend:
Photo Concern | Video Concern | Persistence Concern | SMS Concern

Figure 5.2: Example of neglected *God Class*.

As a conclusion, the results seemed to suggest that conventional detection strategies are not accurate on the identification of architecturally-relevant code anomalies largely due to their lack of sensitivity to properties of architectural concerns in the code. Conventional detection strategies are limited to metrics of structural properties (detected by static analysis tools) of modules in the code. Existing concern metrics (Sant'Anna *et al.*, 2007) and concern tracing tools (FEAT, 2009) should be leveraged to improve the accuracy of detection strategies used to assess architecture degradation.

5.2.4.2. Inability to Identify Architectural Information in the Source Code

Architecturally-relevant code anomalies often occurred in code elements responsible for implementing different architectural elements. Specifically, 49% of the architecturally-relevant code anomalies fell in this category. However,

PUC-Rio - Certificação Digital Nº 0912915/CA

precision and recall rates of the strategies were 36% and 44%, respectively, when identifying these code anomalies.

For instance, the method InsertEmployee.execute() in Health Watcher represents an example of an architecturally-relevant code anomaly that was not automatically detected by conventional strategies. In particular, this method was classified as *Disperse Coupling* by developers since it accesses information and calls methods of classes responsible for implementing different architectural elements. The method InsertEmployee.execute() also introduces undesirable dependencies between non-adjacent layers, condition to be classified as an architecturally-relevant occurrence. However, such method was not identified by conventional detection strategies because the method does not call many methods from other classes. Note that this imperfection cannot be addressed by calibrating the used thresholds. The choice of lower thresholds would lead the strategy to present a higher rate of false positives.

As a result, it was observed that detection strategies were not effective in identifying this kind of code anomaly as they are not sensitive to which architectural elements a code element is responsible for implementing. The key issue is that conventional detection strategies cannot rely on information about how the code elements are associated with architectural elements and their inter-dependencies; this information cannot be extracted using code metrics. This might indicate the need for further investigating how detection strategies could exploit architecture-to-code traceable information.

PUC-Rio - Certificação Digital Nº 0912915/CA

5.2.4.3. Patterns of Code Anomalies

It was observed that certain patterns of code anomalies tend to be better indicators of architectural degradation symptoms than single code anomalies. However, these patterns cannot be directly identified by conventional detection strategies, which focus on identifying individual code anomalies. They do not capture, for instance, a chain of inter-related code anomalies.

Co-occurrences of Code Anomalies. Certain recurring patterns of co-occurring code anomalies tend to be stronger indicators of architectural degradation symptoms. For instance, co-occurrences of *Long Method* and

Divergent Change were associated with architectural problems in all the systems. That is, methods with either many lines of code or realizing several architectural concerns and, high coupling degree with different architectural elements were better indicators than single *Long Method* occurrences. More than 75% of these combined occurrences were associated with architectural problems while just about 43% of single *Long Method* and 36% of *Divergent Change* occurrences were related to architectural problems.

It is important to point out that many of these relevant co-occurrences cannot be detected by simply combining multiple strategies using logical operators (Section 2.3.1). Aiming at identifying these co-occurrences, detection strategies must rely on some kind of architectural information (Section 5.2.4.2). For instance, it would be also useful to consider how many different architectural elements a method is accessing. Otherwise, detection strategies will just detect such relevant co-occurrences that present similar characteristics of non-architecturally-relevant co-occurrences. That is, those co-occurrences that present tight coupling degree with several elements, disregarding their distribution on architectural decompositions.

Code Elements suffering from the Same Anomaly. Interesting findings emerged from analyzing groups of code elements that suffer from the same code anomaly. For instance, when a group of classes that suffer from *God Class* or *Large Class* are implementing the same architectural component *A* and realizing different concerns it may indicate that *A* suffers from *Component Concern Overload*. This assumption departs from the fact that *God Classes* and *Large Classes* are likely to be related to the inappropriate modularization of architectural concerns. Furthermore, when other architectural components and *God Classes* of *A* are sharing the same architectural concern, it may suggest that *A* is affected by *Scattered Parasitic Functionality*. This situation was observed in all the systems.

Propagation of Architectural Problems. The propagation of architectural problems from parents to children in the inheritance trees of all the systems was also often observed. There are two main categories related to such propagation of architectural problems. The first is related to architectural problems that are propagated to all the children in the inheritance tree whereas in the second category the architectural problem is not propagated to all the children, i.e. some children are free of architectural problems. Examples of both categories were

PUC-Rio - Certificação Digital Nº 0912915/CA

found in all systems. For instance, in Health Watcher we observed that several interfaces were introducing undesirable relationships via their parameter types. These interfaces were not identified by detection strategies because they had a well-defined interface (e.g. several members, without a high coupling degree). However, they had a considerable negative effect as these violations were propagated down through the class hierarchies. Usually these undesirable references are left in a system over a long period due to the ripple effects when refactorings are applied to remove them.

The limitations of conventional detection strategies for localizing propagated relevant occurrences of code anomalies are the same for localizing single relevant occurrences. This is due to the propagation of code anomalies in the inheritance trees itself could be detected using static code analysis.

5.2.4.4. Architectural Design and Strategy Accuracy

There was a direct influence of the lack of modularity of certain architectural concerns on the architecturally-relevant anomalies when analyzing different architectural decompositions. We observed that when the modularization of architectural concerns is more explicit in the source code the number of architecturally-relevant anomalies tend to decrease. For instance, object-oriented systems presented a higher number of code anomalies than aspect-oriented systems. We suspect this occurred due to most of the code anomalies being related to the inappropriate modularization of architectural concerns, which are more scattered in object-oriented systems. As aspect-oriented programming mechanisms tend to improve the modularization of crosscutting concerns in single aspects, they may remove relevant anomalies related to this factor. It is not our intention to compare the results in both decompositions, as we discussed in previous sections the inadequate use of aspect-oriented mechanisms may introduce other kinds of architecturally-relevant code anomalies.

Even more interesting is the fact that we have observed how the conventional strategy accuracy for identifying architecturally-relevant anomalies seem to be similar in both kinds of architectural decompositions. This assumption is derived from results regarding to the “average rows” in Table 5.3. The accuracy

PUC-Rio - Certificação Digital Nº 0912915/CA

rates of conventional detection strategies are about 40% for detecting architecturally-relevant code anomalies in all aspect-oriented and object-oriented systems, except in MIDAS.

5.3. Threats to Validity

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

Construct Validity. Threats to construct validity are mainly related to possible errors introduced in the identification of code anomalies and architectural problems. There are different kinds of detection strategies documented in the literature. In particular, we opted for not selecting history-sensitive detection strategies as they tend to be less predictive and require multiple versions of the system (Ratiu *et. al.*, 2004; Mara *et. al.*, 2011). Consequently, they accurately reveal code anomalies just in later releases, when the system may have already achieved critical degradation stages.

We are aware that detection strategies, manual inspections and other mechanisms used to identify code anomalies and architectural problems could introduce imprecision. However, we mitigated this threat by: (i) involving original developers and architects in this process, and (ii) using architectural models where architectural components were mapped to different levels of granularity. That is, the relationships between components and packages were often not 1-to-1. Furthermore, the architectural problems were identified by architects, who had previous experience with the detection of architectural violations and anomalies in other systems. The correlation analysis between code anomalies and architectural problems was also validated with the architects and developers.

Conclusion Validity. We have two issues that threaten the conclusion validity of our study: the number of evaluated systems and assessed anomalies. Two versions of MIDAS, eight versions of MobileMedia, eight versions of Aspectual Media, ten versions of Health Watcher and, ten versions of Aspectual Watcher were used for the purposes of this study, totaling 38 versions. Of course, a higher number of systems would always be desired. However, the analysis of a bigger sample in this study would be impracticable for several reasons.

PUC-Rio - Certificação Digital Nº 0912915/CA

First, the relationship between code anomalies and architectural problems needed to be confirmed by architects. Second, the number of systems with all the required information and stakeholders available to perform this study is rather scarce. Then, our sample can be seen as appropriate for a first exploratory investigation (Kitchenham *et al.*, 2006). All the findings (for example, those discussed in Section 5.2.4) contribute with more specific hypotheses that should be further tested in repetitions or more controlled replications of our study.

Related to the second issue (completeness of code anomalies and architectural problems), our analysis was concerned with a wide variety of code anomalies and problems that occur in system architecture. We analyzed the accuracy of detection strategies for identifying all architecturally-relevant code anomalies that occurred in the target systems. In addition, certain code anomalies were not discussed (e.g. *Small Class*) since their occurrences did not influence the system architectures we have studied.

Internal and External Validity. The main threats to internal and external validity are the following. First, the level of experience of programmers of the systems could be an issue. In order to mitigate this, we used systems that were developed by more than 20 programmers with different levels of software development skills. The main threat to external validity is related to the nature of the evaluated systems as well as their representativeness. In order to minimize these threats we have tried to use applications with different sizes, that suffer from a different set of code anomalies and that were implemented using different architectural styles and context (i.e. academy and industry). Additionally, these applications were developed under different managerial pressures and following distinct software development methodologies (e.g. agile and waterfall). However, we are aware that more studies involving a higher number of systems should be performed in the future.

PUC-Rio - Certificação Digital Nº 0912915/CA

5.4. Summary

This chapter investigated the accuracy of conventional detection strategies when identifying architecturally-relevant code anomalies. To this end, a sample of nearly 800 architecturally-relevant code anomalies distributed in 38 versions of five (05) real-life software systems was considered. Our results confirmed that conventional detection strategies were not accurate to identify code anomalies that attempt against architectural design in the target systems. More specifically:

- More than 50% of the code anomalies identified by conventional detection strategies were not correlated with architectural problems. This means that developers could spend considerable time reviewing code that in fact do not represent threats to the system architectural design.
- Even worse, more than 50% of the false negatives observed using conventional detection strategies were found to be correlated with architectural problems. This means that developers will be lead to not consider code anomalies that are critical to architectural design.
- The inaccuracy of conventional detection strategies cannot be simply addressed by calibrating specific metric thresholds or determining different combinations of particular measures. It seems that their imperfection is largely due to their inability to exploit architectural concern properties or the projection of architecture elements (e.g. components) in the source code.
- Certain recurring patterns of anomaly co-occurrences seem to be better indicators of architecture problems than individual code anomaly occurrences. These patterns usually cannot be directly specified and identified by conventional detection strategies (Lanza and Marinescu, 2006; Srivisut and Muenchaisri , 2007; Moha *et. al.*, 2010; Chapter 3).

The aforementioned findings are interesting because they question the effectiveness of existing strategies and tools in supporting "*architecture revision*" based on the source code. It is important to highlight that current mechanisms for "*architecture revision*" (Section 2.2.2) rely on a detailed specification of the intended architectural design. Furthermore, the findings seem to indicate the need

PUC-Rio - Certificação Digital Nº 0912915/CA

to exploit mappings between architectural design and source code in the code anomaly detection in order to reduce the number of false positives and false negatives. Finally, these findings also revealed the need to analyze recurrent relationships among code anomalies in order to identify their adverse impact on the architectural design of software systems.

PUC-Rio - Certificação Digital Nº 0912915/CA