7
Patterns of Code Anomalies

As introduced in Chapter 5, one of the key obstacles for distinguishing architecturally-relevant code anomalies is that current techniques do not analyze inter-related code anomalies. The importance of analyzing such relationships is due to the fact that an architectural component is usually implemented by multiple code elements. Therefore, some architectural degradation symptoms could be fully detected by analyzing inter-related code anomalies. For instance, *Redundant Dependencies* (Stal, 2011) in the architectural design can only be observed through the analysis of code anomalies that: (i) affect code elements localized in the same component and, (ii) these elements access the same external data. Furthermore, as mentioned in Section 6.4.6, imperfections in the architecture-sensitive detection strategies are also given by their inability in analyzing relationships among code anomalies. Therefore, by documenting inter-related code anomaly occurrences we expect to overcome the accuracy imperfections of both conventional and architecture-sensitive strategies.

However, there is no understanding on which recurring inter-related code anomalies - *code anomaly patterns* - are likely to represent deterioration of the software architecture. In fact, code anomaly patterns can affect the software architecture in significantly different ways. They can range from inter-related code anomalies that infect elements in the same component to inter-related anomalies scattered over the architectural design. Thus, the impact analysis of code anomalies on software architecture could be better supported by considering the systematic classification of code anomaly patterns.

As discussed in Chapter 2, there are several attempts of classifying code anomalies taking into consideration different criteria. Mäntylä and Lassenius (2006) grouped code anomalies according to the software modularity property they affect (e.g. complexity, cohesion). Wake (2003) categorized code anomalies considering their scope (e.g. intra-class, inter-class), while Moha *et al.* (2009) classified them according to their nature (e.g. structural, semantic).

PUC-Rio - Certificação Digital Nº 0912915/CA

These categorizations are preliminary steps towards the definition of a catalogue of the relationships among code anomalies. However, the categorizations suffer from some limitations. First, categorizations of code anomalies are often solely based on the type or a particular characteristic of the code anomaly rather than considering relationships among them. In addition, the existing classifications do not take into consideration how code anomalies relate to the system architecture, which could be an indicator of the their adverse influence on the software architecture.

The previous chapter proposed a terminology and a formalism for code anomaly analysis (Section 6.1). This chapter relies on this terminology and formalism to present a systematic documentation of nine (09) code anomaly patterns. Section 7.1 briefly describes how the code anomaly patterns have been observed. The observed patterns are classified into four categories according to the code anomalies localization. The *Intra-Component Patterns* category comprises patterns that occur in a single component (Section 7.2). The *Inter-Component Patterns* category is formed by patterns that are scattered over various components (Section 7.3). The *Inheritance-based Patterns* category groups patterns that occur in inheritance trees (Section 7.4). This category was created because patterns in inheritance trees can be classified in the two above categories. In other words, an inheritance tree can be completely contained in a component, whereas other tree can group code elements defined in multiple components. Lastly, the *Concern-based Patterns* category comprises patterns related to the inappropriate modularization of architectural concerns (Section 7.5). We decided to create this category even though its patterns could be also classified in the previous categories. The reason is that the new category complements the previous ones since it is not only related to the boundaries of the code anomalies. It represents anomalous implementation of architectural concerns.

Furthermore, Section 7.6 discusses how the proposed anomaly patterns relate to each other. Section 7.7 introduces a tool designed to identify the code anomaly patterns as well as support the collection of the architecture-sensitive metrics (Chapter 6.3), and the application of the architecture-sensitive detection strategies (Chapter 6.4). Section 7.10 presents a study conducted to evaluate the contribution of the anomaly patterns in the identification of architecturally-

PUC-Rio - Certificação Digital Nº 0912915/CA

relevant code anomalies. Finally, Section 7.9 summarizes the key points discussed throughout this chapter.

7.1. Defining Code Anomaly Patterns

The catalogue of code anomaly patterns presented in this chapter is inspired by our previous studies (Chapters 3, 4 and 5). We observed these code anomaly patterns through the evolution history of several software systems. The analyzed systems include MobileMedia (Figueiredo *et al.*, 2008), Health Watcher (Soares *et al.*, 2002; Greenwood *et al.*, 2006), MIDAS (Malek *et al.*, 2006; Garcia *et al.*, 2009), PDP (Chapter 4), Aspectual MobileMedia (Figueiredo *et al.*, 2008) and Aspectual Health Watcher (Soares *et al.*, 2002; Greenwood *et al.*, 2006). As explained in previous chapters, these systems were used because they (i) come from different domains, (ii) present different degrees of size and complexity, (iii) implement architectures of different styles, and (iv) are affected by different types of code anomalies.

Based on these previous observations, we document a catalogue of code anomaly patterns, which are intended to facilitate the code anomaly analysis. Each code anomaly pattern is presented in terms of: (i) a description and motivation highlighting its possible harmful impact on the software architecture, (ii) an example, (iii) a formalization that relies on the formalism introduced in Section 6.1, and (iv) an algorithmic solution to detect its occurrences. As far as examples are concerned, we use a pictorial representation useful for distinguishing the pattern occurrences and a concrete example. The concrete examples are extracted from the software systems used in previous studies (Chapters 4 and 5).

7.2. Intra-Component Patterns

The *Intra-Component Patterns* category represents code anomaly patterns that involve a set of anomalous code elements located in the same architectural component. In this category, software engineers identify sources of modularity principle violations in an architectural component, such as *Single Responsibility Principle* and *Common Reuse Principle* (Martin, 2003). The two patterns defined

PUC-Rio - Certificação Digital Nº 0912915/CA

in this category are: *Multiple-Anomaly Syndrome* and *Similar Anomalous Neighbors*.

7.2.1. Multiple-Anomaly Syndrome

Description and Motivation. The *Multiple-Anomaly Syndrome* pattern consists of code elements that are simultaneously infected by different types of anomalies. Occurrences of *Long Method* and *Feature Envy* (Fowler *et al.*, 1999) infecting the same method is a recurrent instance of this pattern. In particular, this instance indicates that functionalities were incorrectly assigned in the component. The reason is because the anomalous method implements functionalities that should be modularized by other code elements in the component, violating the *Single Responsibility Principle* (Martin, 2003). Therefore, the maintainability of the component is decreased as different functionalities cannot be separately maintained. Note that this negative impact could not be derived from analyzing a single *Long Method* occurrence because some methods are expected to be complex even when implementing a single functionality (e.g. parsers methods).

Moreover, the pattern detection benefits engineers on the choice of an appropriate refactoring strategy to fix the anomalous code element. The application of a proper sequence of refactorings reduces the effort to remove multiple anomalies (Liu, 2011). Considering the previous co-occurrence example (i.e. *Long Method* and *Feature Envy*), during the refactoring of the *Feature Envy*, some parts of the method may be extracted or moved to another code element. Hence, that decomposition would reduce the complexity of the method and simplify the refactoring of (or dispel) the *Long Method* infection. The removal of both anomalies by using a single refactoring strategy could not have been possible if the developers had not been aware of their simultaneous occurrence.

Figure 7.1 presents an abstract representation of instances of the *Multiple-Anomaly Syndrome* pattern. In this figure each color corresponds to a code anomaly. Therefore, when a class is colored, it means that the class is infected by a code anomaly. The infection corresponds to the specific color that appears in the class. In this representation, two of the four classes are instances of the *Multiple-Anomaly Syndrome* pattern because they are simultaneously affected by more than

PUC-Rio - Certificação Digital Nº 0912915/CA

1 code anomaly. Note that this pattern can also occur at the method level. For instance, a method can be considered as an occurrence of the *Multiple-Anomaly Syndrome* pattern when it suffers from multiple code anomalies.

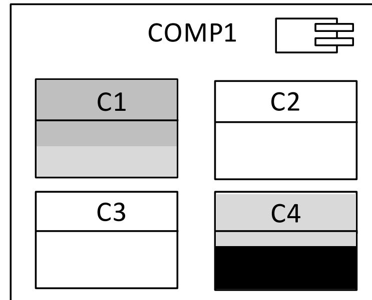

Figure 7.1: Abstract representation of *Multiple-Anomaly Syndrome.*

Formal Definition. Given a software system, *S*, consider a function infect(..) that counts the number of anomalies a given code element is infected by. The set of occurrences of the *Multiple Anomaly Syndrome (MAS)* in *S* is defined in (1) as:

$$MAS\ (S,th) = \{c \mid \exists\, c \in CE_S,\ |infect(c)| > th\}\ (1)$$

where:

$$infect(c) = \begin{cases} 0, c \in FCE \\ > 0, otherwise \end{cases}$$

As defined in Section 6.1, *FCE* represents the set of code elements free of code anomalies. In order to provide a general definition in (1), we opted for using the generic constant, *th*, to represent a given threshold. This constant can be chosen according to the characteristics of the system under analysis and the engineer's design decisions.

According to the abstract representation shown in Figure 7.1, we have that the infect(..) function returns 2 for the C1 and C4 classes, while for C2 and C3 classes the function returns 0. Therefore, C1 and C4 classes are occurrences of the *Multiple-Anomaly Syndrome* pattern if the selected threshold is less than or equal to 1.

Algorithmic Solution. The algorithmic solution for detecting occurrences of *Multiple-Anomaly Syndrome* is presented in Listing 7.1. First, the algorithm identifies code elements infected by anomalies in a given component, using the AnomalousCodeElements(..) function. This function relies on a predefined set of detection strategies - conventional and architecture-sensitive - specifying which

PUC-Rio - Certificação Digital Nº 0912915/CA

types of anomalies will be identified. For each anomalous code element detected (line 4), the algorithm counts the number of code anomalies the measured code element suffers from, using the Anomalies(..) function (line 5). If the number of code anomalies infecting the measured code element is higher than the selected threshold, this code element represents a pattern occurrence (lines 5-7). This algorithm is run for each architectural component of the project under analysis.

Listing 7.1: Detection of *MAS* occurrences.

```
01 Let co be the architectural component under analysis
02 result ={}
03 ACE = AnomalousCodeElements(co)
04 for each c in ACE do
05   if Anomalies (c) > Th then
06     result add c
07   end if
08 end for
09 return result
```

Concrete Example. **Error! Reference source not found.** depicts an example of *Multiple-Anomaly Syndrome* pattern extracted from the MobileMedia system. The MediaController.handleCmd(..) method is the source of three code anomalies. First, this method suffers from the *Long Method* anomaly as it contains many lines of code, presents high cyclomatic complexity, and implements several concerns. For instance, it executes the actions when the method receives a video command as parameter, rather than dispatching to other method in charge of modularize the Video concern. This situation also occurs with other concerns, such as Photo and Favorite. Additionally, the method is affected by the *Feature Envy* anomaly because it implements concerns that should be modularized by other code elements in the Controller layer. Finally, the MediaController.handleCmd(..) method is infected by the *Divergent Change* anomaly because it often changes due to modifications associated with each implemented concern. Therefore, MediaController.handleCmd(..) method should be decomposed into smaller and more cohesive methods.

PUC-Rio - Certificação Digital Nº 0912915/CA

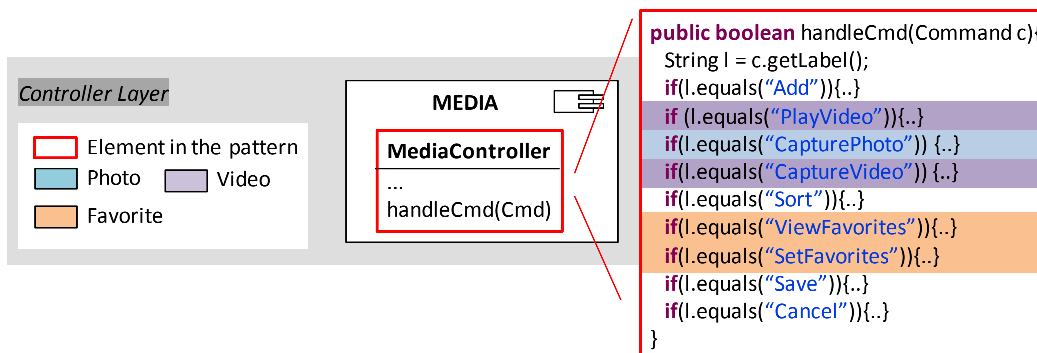

Figure 7.2: Occurrence of the *Multiple-Anomaly Syndrome* in MobileMedia.

As discussed above, occurrences of *Long Method* and *Feature Envy* infecting simultaneously the same code element indicate that functionalities are incorrectly assigned in the component. Additionally, in Figure 7.2 the *Single Responsibility Principle* is neglected in the MediaController.handleCmd(..) method because the method deals with different responsibilities that should be delegated to other code elements. Finally, MediaController.handleCmd(..) method is related to *Ambiguous Interface* architectural anomaly (Section 2.2.3) because it implements the component interface and only has a generic command as parameter. In other words, the method does not specify which commands it executes, making it hard to understand the method's purpose.

7.2.2. Similar Anomalous Neighbors

Description and Motivation. The *Similar Anomalous Neighbors* pattern encloses a set of code elements that: (i) are localized in the same architectural component – they are neighbors, and (ii) all suffer from the same anomaly. The harmful architectural effects of this pattern vary according to anomaly type infecting the code elements. For instance, consider an architectural component $co \in AC_S$ that contains several *Data Classes* (Fowler *et al.*, 1999), where these classes are barely referred by the code elements of *co*. When this occurs the behavior associated with those *Data Classes* are likely to be defined in other components. This situation reduces the component maintainability since the data specification and corresponding behavior are not maintained in the same place.

PUC-Rio - Certificação Digital Nº 0912915/CA

Figure 7.3 presents an abstract representation of a particular occurrence of this code anomaly pattern. As shown in the figure, three out of four classes are infected by the same code anomaly - C1, C2 and C3. These code elements constitute a pattern occurrence, for instance, if the sets of anomalous code elements have to be composed by at least 3 elements.

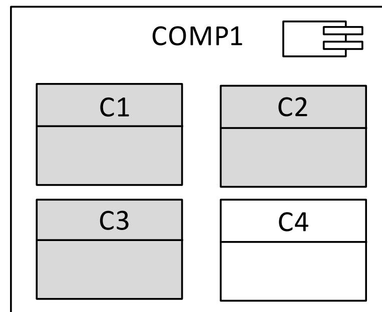

Figure 7.3: Abstract representation of *Similar Anomalous Neighbors*.

PUC-Rio - Certificação Digital Nº 0912915/CA

Formal Definition. Given a system, *S*, the set of occurrences of the *Similar Anomalous Neighbors* (*SAN*) pattern in *S* is formally defined in (2) as:

$$SAN\ (S,th) = \{c_1 \mid \exists\, a \in CAs \wedge co \in AC_S,\ c_1 \in CE_{coa} \wedge |c_1| > th\}\ (2)$$

Similarly to the previous pattern we opted for using a generic constant, *th*, to represent the threshold. This representation allows us to provide a general definition in (2). In order to identify the pattern instance illustrated in Figure 7.2 using this formalism – C1, C2 and C3 classes - we have that: *th* has to be less than or equal to 2.

Algorithmic Solution. The algorithmic solution for detecting *Similar Anomalous Neighbors* is presented in Listing 7.2. First the algorithm detects the set of anomalous code elements that implement a given component (line 3), similarly to the previous algorithm presented in Listing 7.1. Then, the algorithm identifies the list of code anomalies infecting the system under analysis, using the CodeAnomalies(..) function (line 4). For each code anomaly detected, the algorithm identifies the anomalous elements infected by this anomaly, using the ElementsInfectedByAnomaly(..) function (line 7). The algorithm then verifies whether the number of these elements is greater than a given threshold (lines 8-10). If so, the anomalous code elements constitute a pattern occurrence (line 9). This algorithm is run for all components of the system under analysis.

Listing 7.2: Detection of *SAN* occurrences.

```
01 Let co be the architectural component under analysis.
02 Let S be the system under analysis.
03 ACE = AnomaousCodeElements(co)
04 CA = CodeAnomalies(S)
05 result = {}
06 for each a in CA do
07   L = ElementsInfectedByAnomaly(ACE,a)
08   if size(L) > Th then
09     result add L
10   end if
11 end for
12 return result
```

Concrete Example. Figure 7.4 shows an occurrence of the *Similar Anomalous Neighbors* pattern extracted from the PDP system. The Attribute, Photo, Attachment, AreaAttribute, and Area classes are affected by the *Data Class* anomaly because they only contain getters and setters methods, such as getName() and setName(). As explained previously, these classes are architecturally-relevant because their associated behavior is defined in external code elements as illustrated in Figure 7.4. Additionally, the Attribute, Photo, Attachment, AreaAttribute, and Area classes are used together by different components - Proxies and Business. However, these classes did not change together throughtout the system evolution. Therefore, the classes caused that multiple changes associated with different reasons were performed in the Proxies and Business components, violating the *Common Reuse Principle* (Martin, 2003). In the case of the Area class, the methods defined in the Proxies and Business components that manage the Area's behavior (e.g. saveArea(..)) should be moved to the Area class. The Area class should manage its data, rather than other classes, such as Proxy or AreaController. Similar refactorings should be applied to the other classes.

PUC-Rio - Certificação Digital Nº 0912915/CA

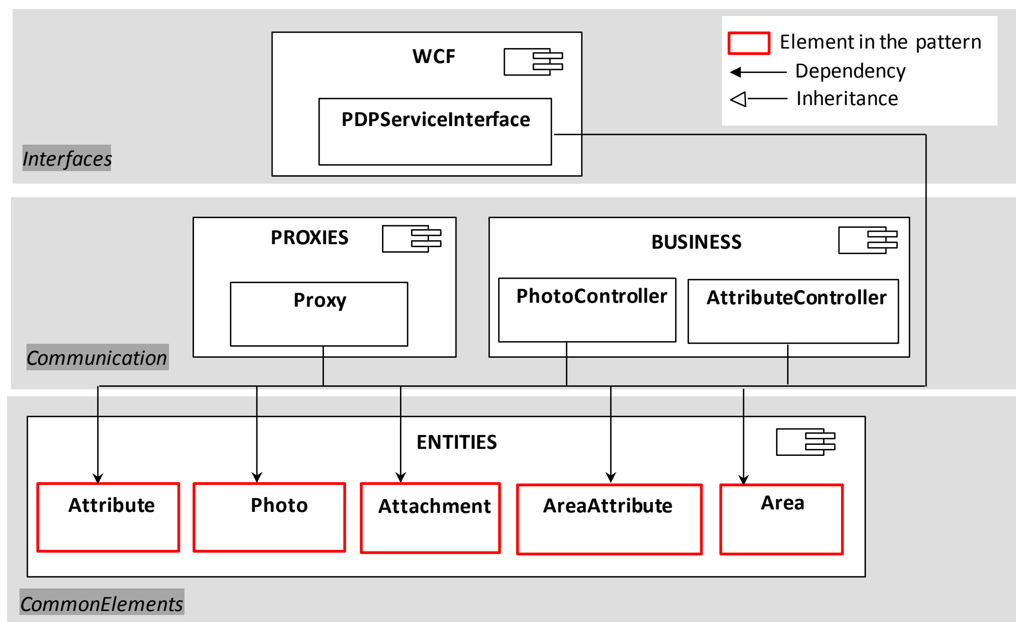

Figure 7.4: Occurrence of the *Similar Anomalous Neighbors* in PDP.

7.3. Inter-Component Patterns

The *Inter-Component Patterns* category includes those patterns related to the communication among architectural components. This means that the software maintenance effects of these patterns are spread over different architectural components. In particular, the category helps engineers to identify code elements that neglect the *Interface Segregation Principle*, the *Common Closure Principle* and the *Single Responsibility Principle* (Martin, 2003) as well as introduce architectural anomalies, such as *Overused Interface* and *Redundant Interface* (Section 2.2.3). The three patterns defined in this category are named: *External Attractor*, *External Addictor* and *Replicated External Network*.

7.3.1. External Attractor

Description and Motivation. The *External Attractor* pattern groups anomalous code elements that are used by (or *attracts*) several external anomalous ones. A code element is considered to be external if it is localized in a component different from where the assessed element is defined. The occurrence of this pattern indicates the existence of different architectural anomalies. First, it

PUC-Rio - Certificação Digital Nº 0912915/CA

suggests that the accessed anomalous code element can be the source of an *Overused Interface* architectural anomaly (Section 2.2.3). A even more harmful situation manifests when: (i) the *Overused Interface* centralizes the realization of different concerns, and (ii) the interface elements realizing each concern are accessed by a different set of client code elements.

Furthermore, this pattern indicates that the accessed code element can favors the further introduction of code anomalies in the client components. When the accessed code element implements different concerns, its client components are forced to deal with these concerns even when they are not interested in them. Therefore, this situation neglects the *Interface Segregation Principle* (Martin, 2003) and increases the internal complexity of the client components. Finally, the *External Attractor* pattern reduces the maintainability of the used component because whenever the server code element needs to be changed, the client components might need to be updated as well. Therefore, the refactoring of this anomaly pattern should be performed as early as possible because its effects impact adversely on multiple parts of the software architecture.

Figure 7.5 depicts an abstract representation of a particular occurrence of this pattern. As it can be noticed, four (04) anomalous classes, localized in three different architectural components, use information (data and behavior) from the same infected class. These classes - C3, C5, C6 and C7 – together with the C2 class constitute a pattern instance because the former group of classes, located in different components, accesses data from C2. Note that, although we are representing a pattern occurrence at the class level, the pattern also manifests at the method level.

PUC-Rio - Certificação Digital Nº 0912915/CA

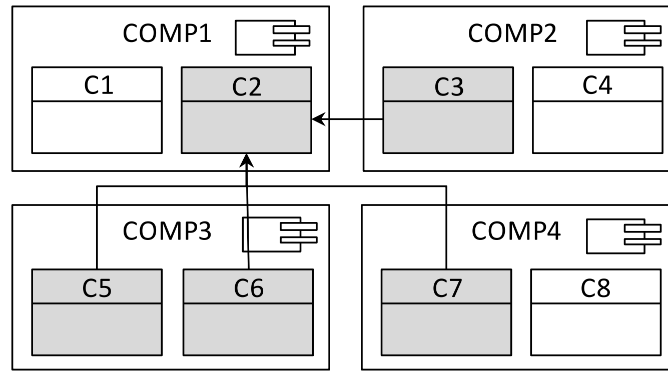

Figure 7.5: Abstract representation of *External Attractor*.

Formal Definition. Given a software system, *S*, the set of occurrences of the *External Attractor (EAT)* pattern in *S* is formally defined in (3) as:

$$EAT(S,th_1,th_2) = \{c_1 \cup c_2 \mid (c_1, c_2) \in D_{c1,c2} \wedge$$
$$\exists\ co_1 \in AC_S \wedge co_2 \in AC_S,$$
$$co_1 \neq co_2 , \wedge c_1 \in ACE_{co1} \wedge c_2 \in ACE_{co2} \wedge |c_1| > th_1 \wedge |co_1| > th_2\} \quad (3)$$

PUC-Rio - Certificação Digital Nº 0912915/CA

In order to provide a general definition in (3), we opted for using generic constants to represent the thresholds, th_1 and th_2. Both constants can be chosen according to the characteristics of the system under analysis and the engineer's design decisions. In order to identify the pattern instance illustrated in Figure 7.5 – C2, C3, C5, C6 and C7 – using this formalism, we have that: th_1 and th_2 have to be less than or equal to 3 and 2, respectively.

Algorithmic Solution. Listing 7.3 presents the solution for detecting instances of the *External Attractor* pattern. First, the algorithm identifies the anomalous code elements in a given component, using the AnomalousCodeElements(..) function (line 3). For each element identified, the algorithm computes the anomalous external elements that depend on the measured element, using the AnomalousExternalClients(..) function (line 6). Then, the algorithm stores each client element and its corresponding component in a temporary structure (lines 8 and 9). If the number of stored client components and external code elements are higher than the corresponding thresholds (lines 11 and 12), the client and server elements constitute a pattern occurrence (line 14). This algorithm is run for each component of the system under analysis.

Listing 7.3: Detection of *EAT* occurrences.

```
01 Let co be the architectural component under analysis
02 result ={}
03 ACE = AnomalousCodeElements(co)
04 for each c in ACE do
05   occurrence = {}
06   EC = AnomalousExternalClients(c)
07   for each client in EC do
08     occurrence{0} add client
09     occurrence{1} add component(client)
10   end for
11   if size(occurrence{0}) > Th1 and
12      size(occurrence{1}) > Th2 then
13        occurrence{0} add c
14       result add occurrence{0}
15   end if
16 end for
17 return result
```

Concrete Example. Figure 7.6 shows an occurrence of *External Attractor* pattern extracted from the Health Watcher system, which hinders the software architecture in multiple manners. In the figure, the IFacade interface is affected by the *Bloat Interface* and *Overused Interface* anomalies because it is large, non-cohesive, and its methods are called by many classes. Additionally, it neglects the *Single Responsibility Principle*. In particular, the methods provided by this interface are called by different client classes and components, indicating the inappropriate declaration of these methods in a single interface. The classes highlighted in the Complaint, HealthUnit and Employee components are infected by the *Long Method* anomaly because they deal with different kinds of information that are propagated by the IFacade interface, such as Persistence and Transaction. Additionally, these classes have been affected by several changes due to modifications performed in the IFacade interface. In summary, this pattern occurrence harms the software architecture in multiple ways, such as introduction of architectural anomalies as well as neglecting modularity principles in the architecture design.

In order to solve these problems, several refactorings should be applied. First, the IFacade interface should encapsulate exceptions coming from components in the Data layer and then, throw business exceptions containing only relevant information for components in the View layer. Therefore, code elements in the View layer will decrease their internal complexity. Second, the IFacade interface should be decomposed into smaller ones that are more cohesive than the original interface structure. Consequently, each component in the View layer will

PUC-Rio - Certificação Digital Nº 0912915/CA

access different interfaces, decreasing the coupling between the IFacade interface and the View layer.

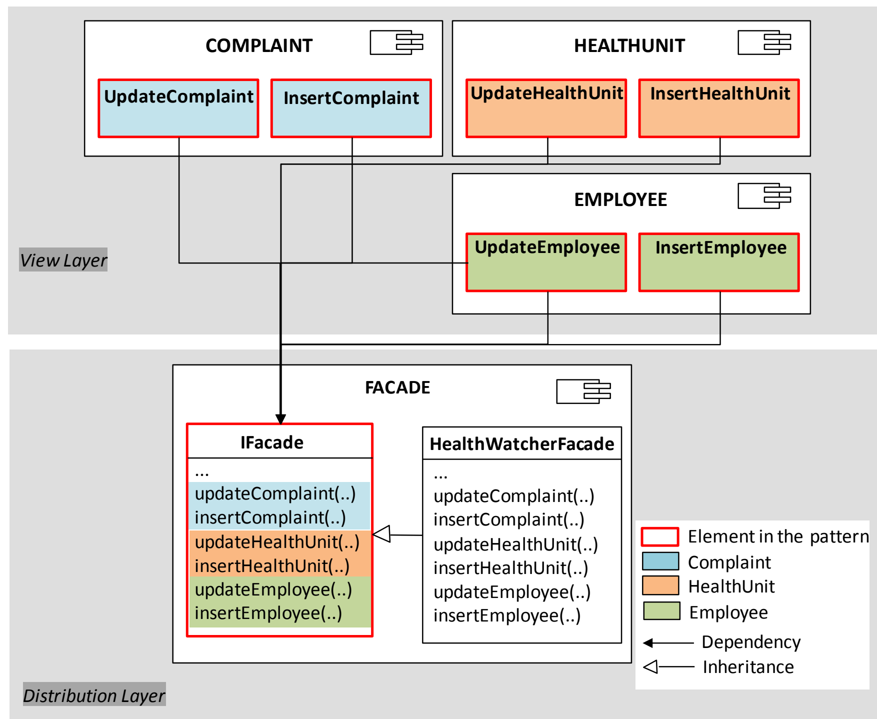

Figure 7.6: Occurrence of the *External Attractor* pattern in HW.

7.3.2. External Addictor

Description and Motivation. In an opposite way of *External Attractor*, *External Addictor* pattern consists of anomalous code elements that depend a lot (or are *addicted*) of external anomalous code elements. The occurrence of this pattern alerts engineers about different architectural degradation symptoms. For instance, the *External Addictor* pattern highlights the presence of an anomalous code element that can be targeted as a consequence of modifications in several components. Thus, when this occurs, change ripple effects are introduced in the system architecture.

Second, this pattern indicates a source of tight coupling degree among architectural components because an anomalous code element centralizes the communication between its enclosing component and the adjacent others. Ripple effects and tight coupling among components are architectural problems that have been recognized as sources of software systems reengineering and discontinuation

PUC-Rio - Certificação Digital Nº 0912915/CA

(Eick *et al.*, 2001; Maccormack *et al.*, 2006; Knodel *et al.*, 2008). These evidences highlight the relevance of engineers being aware of the occurrences of this pattern and, consequently, on performing their early refactoring.

Figure 7.7 illustrates the abstract representation of a particular occurrence of the *External Addictor* pattern. As it can be noticed this representation is very similar to the *External Attractor* pattern; there is a difference in the direction of the communication among components in both patterns. Apart from that, they are similar. An anomalous class named C2 uses information from anomalous classes located in several components – C3, C5, C6 and C7. These five classes constitute an occurrence of the *External Addictor* pattern. It is important to note that, although we are representing a pattern occurrence at the class level, the pattern can also manifest at the method level.

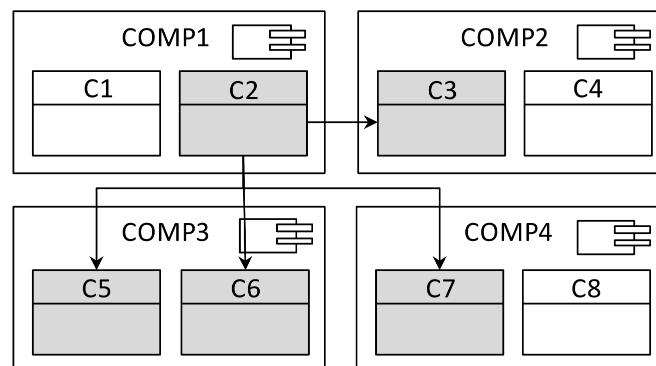

Figure 7.7: Abstract representation of *External Addictor*.

PUC-Rio - Certificação Digital Nº 0912915/CA

Formal Definition. Given a software system, *S*, the set of occurrences of the *External Addictor (EAD)* pattern in *S* is formally defined in (4) as:

$$EAD(S,th_1,th_2) = \{c_1 \cup c_2 \mid (c_1, c_2) \in D_{c1,c2} \wedge \exists\ co_1 \in AC_S \wedge co_2 \in AC_S, co_1 \neq co_2 , \wedge c_1 \in ACE_{co1} \wedge c_2 \in ACE_{co2} \wedge |c_2| > th_1 \wedge |co_2| > th_2\} \quad (4)$$

Similarly to the definition of EAT_S, the used thresholds were represented using generic constants, th_1 and th_2. Both constants can be materialized according to the characteristics of the system under analysis and the engineers' design decisions. In order to identify the pattern instance illustrated in Figure 7.7 – C2,

C3, C5, C6 and C7 – using this formalism, we have that: th_1 and th_2 have to be less than or equal to 3 and 2, respectively.

Algorithmic Solution. Listing 7.4 presents the solution for detecting occurrences of the *External Addictor* pattern. First, the algorithm identifies the anomalous code elements in a given component, using the AnomalousCodeElements(..) function (line 3). For each element identified, the algorithm computes the anomalous external code elements from which the identified element depends, using the AnomalousExternalServers(..) function (line 6). From this point, the algorithm is similar to that presented in Listing 7.3. This algorithm is run for each component of the system under analysis.

Listing 7.4: Detection of *EAD* occurrences.

```
01 Let co be the architectural component under analysis
02 result ={}
03 ACE = AnomalousCodeElements(co)
04 for each c in ACE do
05   occurrence ={}
06   AES = AnomalousExternalServers(c)
07   for each extElem in AES do
08     occurrence{0} add extElem
09     occurrence{1} add component(extElem)
10   end for
11   if size(occurrence{0}) > Th1 and
12      size(occurrence{1}) > Th2 then
13        occurrence{0} add c
14        result add occurrence{0}
15   end if
16 end for
17 return result
```

Concrete Example. Figure 7.8 illustrates an occurrence of the *External Addictor* pattern extracted from Health Watcher system. In this figure HealthUnit, TransactionExc, ObjNotValidExc, ObjNotFoundExc, and RepositoryExc are anomalous classes because they only define attributes or assessors methods. Therefore, these classes are affected by the *Data Class* anomaly. The IFacade class, as aforementioned, is infected by the *God Class* anomaly because, among other reasons, it defines a great amount of non-cohesive methods. Additionally, IFacade propagates several exceptions that should be treated internally. This propagation forces InsertHealthUnit to deal with Persistence and Transaction exceptions when the class should not perform these actions. This phenomenon increases the internal complexity of the InsertHealthUnit's methods. Even worse, the propagation of these exceptions violates the architects' design decisions as dependencies are introduced between View and Data, non-adjacent layers.

PUC-Rio - Certificação Digital Nº 0912915/CA

In order to remove this occurrence, refactorings can be performed in a similar manner to the previous example. In other words, the IFacade interface should encapsulate exceptions coming from components in the Data layer in order to remove dependencies between View and Data layers. Consequently, this refactoring should decrease the internal complexity of the InsertHealthUnit's methods as they do not have to deal with additional information.

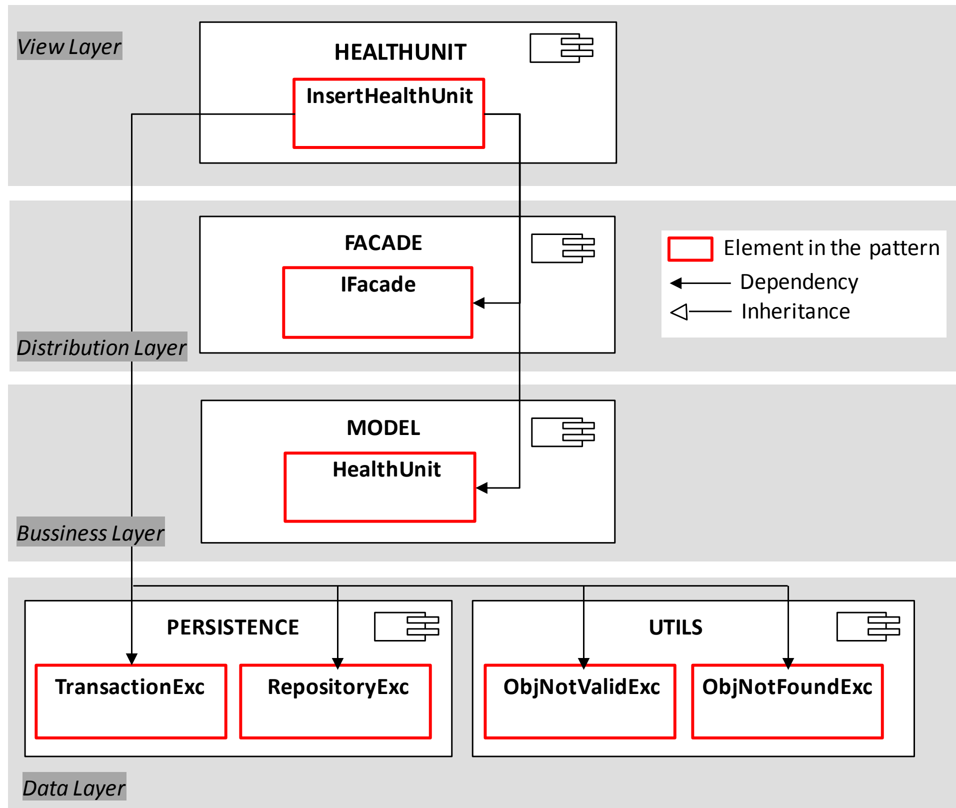

Figure 7.8: Occurrence of the *External Addictor* pattern in HW.

PUC-Rio - Certificação Digital Nº 0912915/CA

7.3.3. Replicated External Network

Description and Motivation. *Replicated External Network* pattern consists of anomalous code elements within an architectural component that depend on the same external code elements. This pattern indicates the presence of three potential architectural degradation symptoms. First, it suggests the lack of a common component interface. In other words, there is no shared interface responsible for establishing the communication between a given component and its adjacent ones. Therefore, any code element within the component can communicate with

external code elements. Second, this pattern suggests the presence of *Redundant Interfaces* anomaly (Section 2.2.3) since different code elements within the component access the same external data. Finally, the occurrence of this pattern introduces the *Extraneous Connector* anomaly (Section 2.2.3) when these interfaces use different mechanisms to communicate the same components (e.g. events and procedure calls).

Figure 7.9 illustrates an abstract representation of this pattern occurrence. This figure highlights how two (02) anomalous classes – C1 and C2 – access data from the same external classes. In this particular example, these two classes constitute a pattern occurrence. Note that, although we are representing a pattern occurrence at the class level, the pattern can also manifest at the method level.

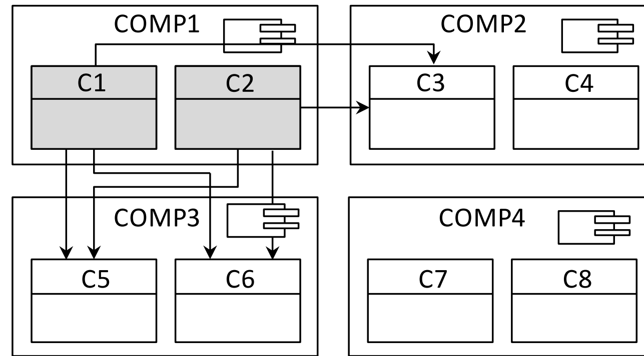

Figure 7.9: Abstract representation of *Replicated External Network*.

Formal Definition. Given a software system, *S*, the set of occurrences of the *Redundant External Network (REN)* pattern in *S* formally defined in (5) as:

$$REN(S,th_1,th_2) = \{c_1 \cup c_2 \mid (c_1, c_3) \in D_{c_1c_3} \wedge (c_2, c_3) \in D_{c_2c_3} \wedge \exists\, a \in CA_S \wedge co_1 \in AC_S \wedge co_2 \in AC_S \wedge c_1 \in CE_{co_1a} \wedge c_2 \in CE_{co_1a} \wedge c_3 \in CE_{co_2}, |c_3| > th_1 \wedge | c_1 \cup c_2 | > th_2 \} \quad (5)$$

where, th_1 and th_2 are generic constants that indicate the undesirable number of anomalous elements and percentage of accessed external code elements, respectively, such that $0 \leq th_2 \leq 1$. In order to identify the pattern instance illustrated in Figure 7.9 – C1 and C2 – using this formalism, we have that: th_1 and th_2 have to be less than or equal to 2 and 1, respectively.

PUC-Rio - Certificação Digital Nº 0912915/CA

Algorithmic Solution. Listing 7.5 presents the solution for detecting occurrences of the *Replicated External Network* pattern. First, the algorithm identifies the anomalous code elements in a given architectural component, using the AnomalousCodeElements(..) function (line 3). Then, for each pair of code elements, the algorithm computes the external code elements they access in common, using the CommonExternalDependencies(..) function (line 6). If the number of external accessed code elements is higher than a given threshold (line 7), both code elements are classified as an occurrence of the pattern (line 8). Finally, the algorithm combines the identified occurrences according to their common external dependencies, using the CombineOccurrences(..) function (line 12). The goal of this combination is not to restrict the algorithm's output to pairs of code elements. It is important to note that the algorithm returns only those combinations that involve: (i) a number of anomalous elements higher than *th1* constant, and (ii) a number of accessed external elements higher than *th2* constant. This algorithm is run for each component of the system under analysis.

PUC-Rio - Certificação Digital Nº 0912915/CA

Listing 7.5: Detection of *REN* occurrences.

```
01 Let co be the component under analysis
02 occurrences ={}
03 ACE = AnomalousCodeElements(co)
04 for each c1 in ACE do
05   for each c2 in ACE do
06     elems = CommonExternalDependencies(c1,c2)
07     if size(elems) > Th1 then
08       occurrences add {c1, c2, elems}
09     end if
10   end for
11 end for
12 return CombineOccurrences(occurrences, th1, th2)
```

Concrete Example. Figure 7.10 illustrates an occurrence of the *External Addictor* pattern extracted from Health Watcher system. The occurrence of this pattern comprises the Complaint, Symptom and Employee classes, which implement part of the *Observer* design pattern (Gamma *et al.*, 1994). These classes are considered to be anomalous because they only define attributes, getters and setters methods and, hence, are affected by the *Data Class* anomaly. In particular, this occurrence is caused by copy and paste practices in the Model component. The source code of the notifyObservers() method in the Complaint, Symptom and Employee classes is presented in Figure 7.8. This figure shows how the implementation of this method is duplicated in these three classes, introducing

high coupling among the Model, Persistence and Utils components. Consequently, whenever the notifyObservers() method suffers a change (e.g. to change the exception treatment or to include a new exception), the modification has to be performed in these three classes. The problem is that these three classes changed together during the system evolution due to modifications in the notifyObservers() method, neglecting the *Common Reuse Principle* (Martin, 2003). In order to remove this pattern occurrence, the notifyObservers() method should be moved to a common parent class in order to decrease the number of dependencies between Business and Data layers. Therefore, the parent class will be the only class responsible for catching and treating the exceptions thrown by the Persistence and Utils components.

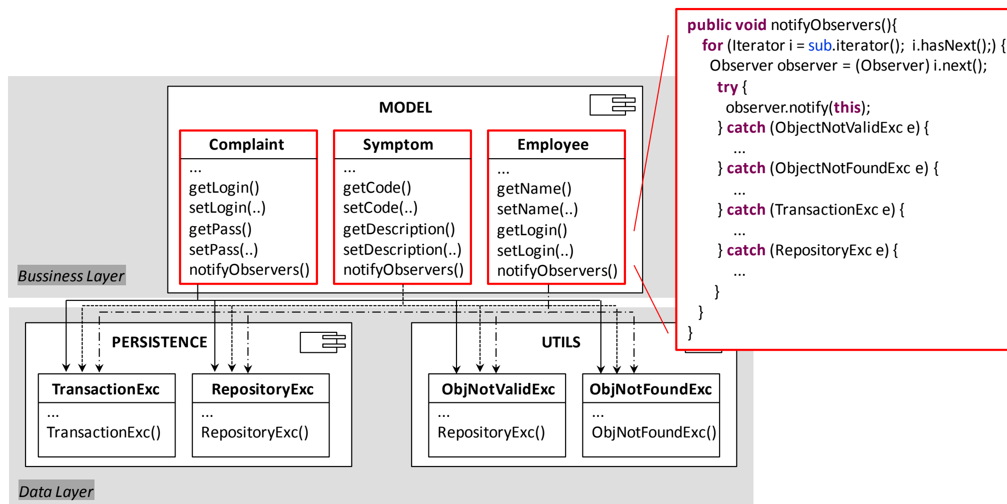

Figure 7.10: Occurrence of the *Replicated External Network* pattern in HW.

7.4. Inheritance-based Patterns

The category *Inheritance-based Patterns* includes two code anomaly patterns named *Hereditary Anomaly* and *Mutant Anomaly*. These patterns consist of anomalous code structures that manifest themselves in inheritance trees. Although these patterns could have been included in the previous categories, we created a new one because code elements making up a hierarchy are not uniformly distributed across the components. In other words, certain hierarchies are fully contained in a single component, whereas others contain code elements located in multiple components. Therefore, instances of the same pattern would be classified in two different categories.

PUC-Rio - Certificação Digital Nº 0912915/CA

Likewise the previous categories, the inheritance-based patterns have been observed in software systems from different domains. In particular, their occurrence is notably severe to the software architecture when they manifest in reusable and long living systems, such as frameworks and program families. When a system extends from an anomalous framework implementation, it can suffer from the code anomalies introduced in the used framework. Also, once the framework is extended, inheritance-based patterns become harder to be refactored because their removal is likely to impact the structure of a client system. Consequently, the best way to avoid this situation is making engineers aware of the presence of such patterns occurrences as early as possible.

7.4.1. Hereditary Anomaly

Description and Motivation. The *Hereditary Anomaly* pattern consists of code anomalies that: (i) infect a parent code element (e.g. class, method) in an inheritance tree and, (ii) are propagated to some descendants in this tree. A code anomaly $a \in CA$ is propagated in an inheritance tree whenever the root code element and an inherited element are infected by a. Anomalies in the *Hereditary Anomaly* pattern do not necessarily affect all the descendants elements. Occurrences of *Hereditary Anomaly* are particularly harmful to the software architecture because they are characterized by the propagation of anomalies over the code elements. In extreme cases such propagation is beyond the boundaries of a component or even a software system.

Furthermore, the refactoring of the anomalous code elements in this pattern implies on a ripple refactoring effect over the inheritance tree. Therefore, this pattern suggests that refactorings should be performed first in the parent element; rather than refactoring each descendant individually. The reason is that the individual refactoring of each anomalous descendant does not completely fix the anomaly because when a new descendant is added into the hierarchy, it is likely to inherit the anomaly.

Figure 7.11 depicts an abstract representation of a particular occurrence of this pattern. As it can be seen some descendants - without any background color - are not infected by the propagated anomaly. Furthermore, we decided not to

PUC-Rio - Certificação Digital Nº 0912915/CA

represent components in this figure as hierarchies can be (or not) fully confined in the same component as previously discussed. In this example, the pattern occurrence is formed by the C1, C2, C4 and C5 classes. It is important to note that, for illustrative purposes we are showing a pattern occurrence at the class level, but the pattern can also manifest at the method level.

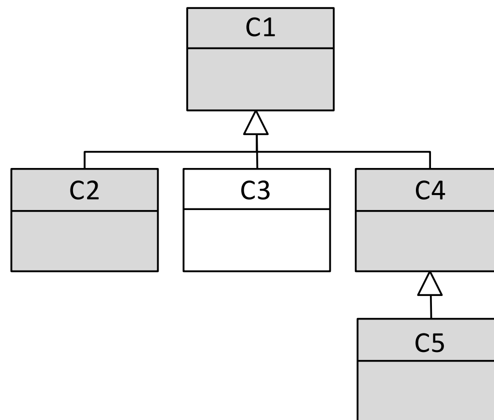

Figure 7.11: Abstract representation of *Hereditary Anomaly*.

PUC-Rio - Certificação Digital Nº 0912915/CA

Formal Definition. Given a software system, *S*, in order to formally define the set of occurrences of the *Hereditary Anomaly (HA)* pattern in *S*, an additional Boolean function, descendant(c_1,c_2) is required. This function indicates whether a code element $c_1 \in CE_S$ inherits from another code element $c_2 \in CE_S$. The *HA* in S is formally defined in (6) as:

$$HA(S,th) = \{c_1 \cup c_2 \mid descendant(c_1, c_2) \wedge$$
$$\exists\, a \in CA_S \wedge c_1 \in CE_a \wedge c_2 \in CE_a,$$
$$c_1 \neq c_2 \wedge |c_2| > th\ \}\ (6)$$

where, *th* is a threshold indicating a number of anomalous descendants. According to the abstract representation shown in Figure 7.11, we have that: the descendant(..) function returns 3 for the C1 class. Therefore, the group encompassed by the C1, C2, C4 and C5 classes is considered to be a pattern occurrence using this formalization, if *th* is less than or equal to 2.

Algorithmic Solution. The algorithmic solution for detecting occurrences of the *Hereditary Anomaly* pattern is presented in Listing 7.6. First, the algorithm computes the set of anomalies infecting the parent code element under analysis, using the CodeAnomalies(..) function (line 3). For each anomaly detected, the

algorithm computes all descendants of the parent element infected by this anomaly, using the AnomalousDescendants(..) function (line 5). If the number of the infected descendants is higher than the threshold (lines 6-8), they constitute a pattern occurrence (line 7). This algorithm is run for each anomalous parent element starting at the root of each tree of the system under analysis.

Listing 7.6: Detection of *HA* occurrences.

```
01 Let c be the anomalous parent element under analysis
02 result = {}
03 CA = CodeAnomalies(c)
04 for each a in CA do
05   AD = AnomalousDescendants(c, a)
06   if size(AD) > Th then
07     result add {AD, a, c}
08   end if
09 end for
10 return result
```

Concrete Example. Figure 7.12 illustrates an occurrence of the *Hereditary Anomaly* pattern extracted from the PDP system. The PDPServices interface is considered a *Bloat Interface* because it defines more than ten (10) methods, which do not implement the same concern. In particular, this interface defines methods that deal with Photo, Attachment, Map, and Area concerns. The PDPServices interface is implemented by three classes: a client interface, PDPServiceInterface, a Proxy interface, Proxy, and a server interface, PDPService. These three classes are affected by the *God Classes* anomaly because they are large, and their methods are complex and non-cohesive. This anomaly was inherited from the PDPServices interface because it forces its descendants to implement may non-cohesive methods. This inherited anomaly is harmful to the system architecture, as changes in the PDPServices interface triggered changes in the three external descendant classes. These classes suffered many modifications associated with each of the implemented concerns through the system evolution, confirming their anomalous nature. In particular, this design was considered a serious architecture degradation symptom in this system due to all the ripple effects across the components.

PUC-Rio - Certificação Digital Nº 0912915/CA

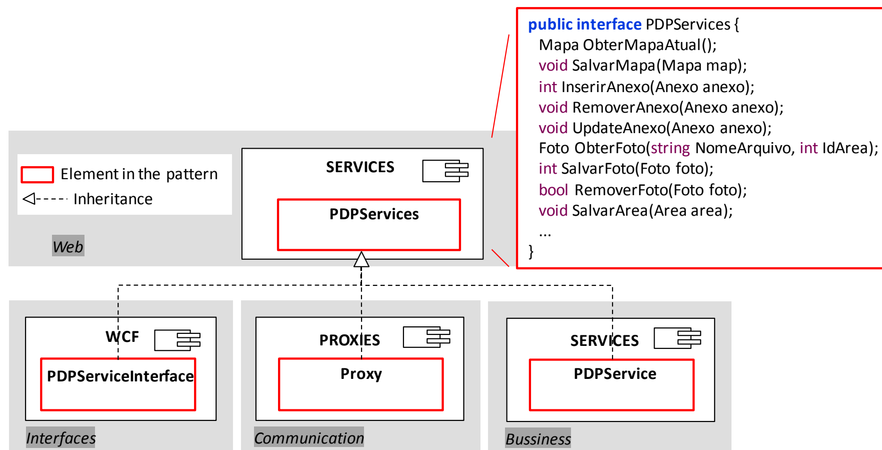

Figure 7.12: Occurrence of the *Hereditary Anomaly* pattern in PDP.

7.4.2. Mutant Anomaly

Description and Motivation. The *Mutant Anomaly* pattern infects descendants in the same inheritance tree similarly to the *Hereditary Anomaly* pattern. However, unlike the previous pattern, in the *Mutant Anomaly* pattern the parent code element has to be free of anomalies. The characteristic of this pattern is that several descendants of a common parent suffering from the same anomaly, suggests that the parent element may induce its descendants to present anomalies. Therefore, developers should analyze the possibility of performing refactorings in the parent element.

Figure 7.13 shows an abstract representation of an occurrence of this pattern. In this illustration, the root code element is free of code anomalies, but all its direct descendants suffer from the same anomaly. In this example, the pattern occurrence is formed by the C2, C3 and C4 classes. It is important to note that, we are only showing a pattern occurrence at the class level, but the pattern can also manifest at the method level. Similarly to the previous pattern, we did not represent architectural components in Figure 7.13.

PUC-Rio - Certificação Digital Nº 0912915/CA

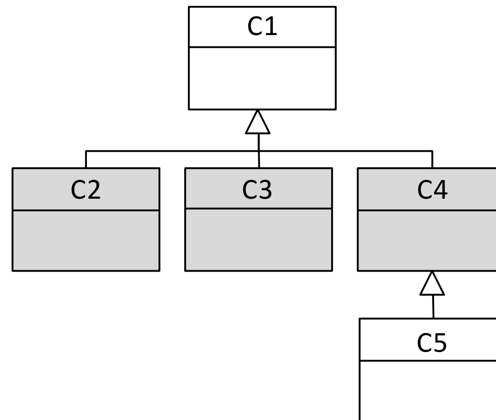

Figure 7.13: Abstract representation of *Mutant Anomaly*.

Formal Definition. Given a software system, *S*, the formal definition of the set of occurrences of the *Mutant Anomaly (MA)* pattern in *S* relies on the function descendant(..) previous defined. The *MA* in *S* is formally defined in (7) as:

$$MA(S,th) = \{c_1 \cup c_2 \mid descendant(c_1, c_2) \wedge \exists\, a \in CA_S \wedge c_1 \in FCE_a \wedge c_2 \in CE_a, |c_2| > th \} \quad (7)$$

where, *th* is a threshold indicating a number of anomalous descendant elements. According to the abstract representation shown in Figure 7.13, we have that: the descendant(..) function returns 3 for the C1 class. Therefore, the group encompassed by the C2, C3 and C4 classes is considered to be a pattern occurrence using this formalization, if *th* is less than or equal to 2.

Algorithmic Solution. The algorithmic solution for detecting occurrences of the *Mutant Anomaly* pattern described in Listing 7.7 is similar to the algorithm presented in Listing 7.6. First, the algorithm computes the list of code anomalies infecting the system under analysis (line 3), using the CodeAnomalies(..) function. For each code anomaly detected, the algorithm identifies the descendants of the parent element under analysis that are infected by this anomaly, using the AnomalousDescendants(..) function (line 6). From this point, the algorithm is similar to that presented in Listing 7.6. This algorithm is run for each anomalous parent element starting at the root of each tree of the system under analysis.

PUC-Rio - Certificação Digital Nº 0912915/CA

Listing 7.7: Detection of *MA* occurrences.

```
01 Let c be the free-anomalous parent under analysis
02 Let S be the system under analysis
03 CA = CodeAnomalies(S)
04 result = {}
05 for each a in CA do
06   AD = AnomalousDescendants(c, a)
07   if size(AD) > Th then
08     result add {AD, a, c}
09   end if
10 end for
11 return result
```

Concrete Example. Figure 7.14 shows an occurrence of the *Mutant Anomaly* pattern extracted from MIDAS system. In this occurrence the SD_EventHandler class represents a root code element in the inheritance tree, which defines the handleRequest(..) method. The descendants of this class are the SD_SD, SD_ClientEventHandler and SD_PubSubEventHandler classes. These classes are anomalous because their corresponding handleRequest(..) method is affected by the *Long Method* anomaly.

In this example the SD_EventHandler.handleRequest(..) method has a single generic parameter and implements the interface of the Engine component. Additionally, the method is called by code elements defined in the Service Discovery, Fault Tolerance, and Monitoring components. Therefore, this occurrence of *Mutant Anomaly* is architecturally-relevant because is related to the propagation of the *Ambiguous Interface* anomaly (Section 2.2.3) in the inheritance tree. In other words, the SD_EventHandler.handleRequest(..) method and, hence, all its descendants accept all invocation requests through a single generic parameter, without dispatching to other methods. Overgeneralized interfaces are even more harmful to the architecture design because they favor additional dependencies and tight coupling among components as the system evolves (Martin *et al.*, 2003; Garcia *et al.*, 2009).

PUC-Rio - Certificação Digital Nº 0912915/CA

Figure 7.14: Occurrence of the *Mutant Anomaly* pattern in MIDAS.

7.5.
Concern-based Patterns

The *Concern-based Patterns* category comprises code anomaly patterns related to not well modularized architectural concerns in the system implementation. It complements the previous categories as the anomalous structures in this category are not only related to the boundaries of the components and their relationships. This category represents anomalous implementation of architectural concerns. Specially, these patterns help engineers to identify architecturally-relevant sources neglecting the concern separation principles (Kickzales, 1996) in the system implementation. The early detection of such sources is relevant because they hamper the maintenance of the affected components. For instance, components tend to not be maintained when they are in charge of implementing many architectural concerns.

PUC-Rio - Certificação Digital Nº 0912915/CA

7.5.1. Concern Overload

Description and Motivation. *Concern Overload* pattern consists of anomalous code elements in the same component that modularize many independent concerns. In this context, two architectural concerns are considered to be independent when they should be modularized by different architectural components. This pattern differs from *Similar Anomalous Neighbors* pattern (Section 7.2.2), in the sense that the former comprises different types of code anomalies. When several anomalous code elements in the same component realize independent concerns, it indicates that the component is not cohesive. Consequently, the affected component might be decomposed into smaller ones that are more cohesive than the original component structure. Furthermore, the occurrence of this pattern suggests that the component centralizes more concerns that it should do, affecting the component maintainability.

Figure 7.15 illustrates an abstract representation of an occurrence of the *Concern Overload* pattern. In this figure, concerns are represented by using filled circles. As it can be seen, a group of two anomalous classes – C1 and C2 – modularize three different concerns. In particular, these classes make the component to deal with several concerns. In this example, C1 and C2 classes constitute a pattern occurrence. It is important to note that, we are only showing a pattern occurrence at the class level, but the pattern can also manifest at the method level.

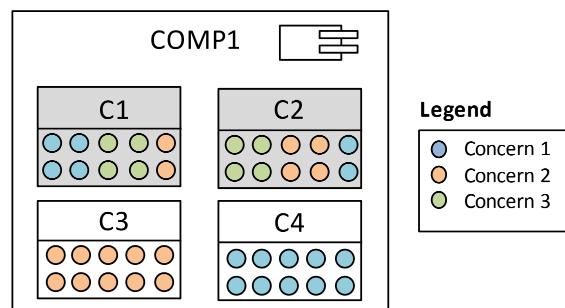

Figure 7.15: Abstract representation of the *Concern Overload* pattern.

PUC-Rio - Certificação Digital Nº 0912915/CA

Formal Definition. Given a software system, S, in order to formally define the set of occurrences of the *Concern Overload (CO)* pattern in S, an additional function, concerns(c_1) is required. This function returns all concerns realized by a code element $c_1 \in CE_S$. The formal definition of CO in S is presented in (8) as:

$$CO(S,th_1,th_2) = \{ c_1 \mid c_1 \in ACE_{co} \wedge \exists co \in AC_S \wedge c_1 \in CA_S, |concerns(c_1)| > th_1 \wedge |c_1| > th_2 \} \quad (8)$$

where, th_1 specifies the maximum acceptable number of concerns that should be realized by a code element and, th_2 specifies the minimum number of anomalous code elements to be considered as a pattern occurrence. According to the abstract representation shown in Figure 7.15, we have that: the concerns(..) function returns 3 for the C1 and C2 classes. Therefore, the group encompassed by the C1 and C2 classes is considered to be a pattern occurrence using this formalization, if th_1 and th_2 are less than or equal to 2 and 1, respectively.

Algorithmic Solution. The algorithmic solution for detecting *Concern Overload* occurrences is presented in Listing 7.8. First, the algorithm detects groups of anomalous code elements that implement the same set of concerns in a given architectural component, using the AnomalousSameConcernImplementers(..) function (line 2). Then, for each group identified, the algorithm verifies whether the number of concerns modularized by this group and the number of code elements contained in the group are higher than the corresponding thresholds (lines 5 and 6). If so, the identified group of elements constitutes a pattern occurrence (line 7). This algorithm is run for each architectural component of the system under analysis.

Listing 7.8: Detection of *CO* occurrences.

```
01 Let co be the architectural component under analysis
02 ACI = AnomalousConcernImplementers(co)
03 result = {}
04 for each c in ACI do
05   if Concerns(c) > Th1 and
06      size(c) > Th2 then
07        result add c
08   end if
09 end for
10 return result
```

PUC-Rio - Certificação Digital Nº 0912915/CA

Concrete Example. Figure 7.16 shows an occurrence of the *Component Concern Overload* pattern extracted from MIDAS system. This occurrence comprises the SD_SD and SD_Engine classes in the Service Discovery component. These classes are affected by the *God Class* anomaly because they are large, their methods are complex and implement several concerns, such as Fault Tolerance, Service Discovery and Dynamic Adaptation. Therefore, the SD_SD and SD_Engine classes contribute to the Service Discovery component centralizes the implementation of several concerns that should be implemented in different components. Additionally, Service Discovery component, in particular these anomalous classes, suffered from many unexpected changes associated with the different concerns. The harmful nature of this incorrect modularization of concerns was recognized by developers, who refactored these classes - moving the Fault Tolerance and Dynamic Adaptation implementations to their own components.

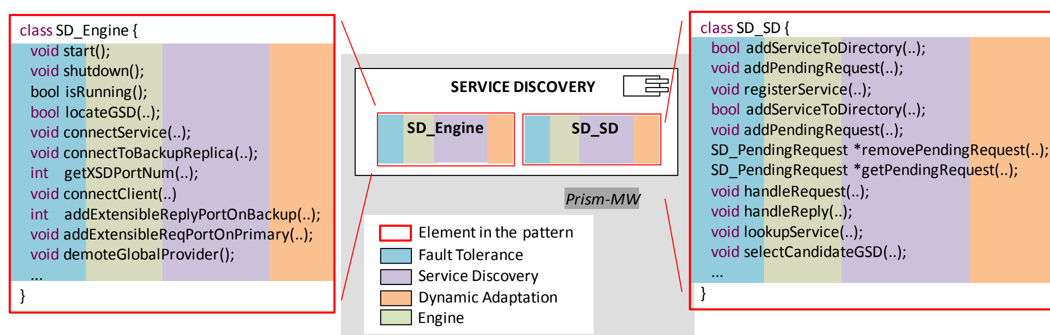

Figure 7.16: Occurrence of *Concern Overload* pattern in the MIDAS system.

7.5.2. Misplaced Concern

Description and Motivation. *Misplaced Concern* pattern refers to another form of violation of the concerns separation principle. This pattern groups anomalous code elements that modularize a concern, which is not the predominant one of their enclosing component. A concern is predominant in a component if most of the code elements in this component are dedicated to modularize the concern. Therefore, the anomalous code elements classified in this pattern should be defined in a different component. This problem is more remarkable when other components also modularize the "misplaced" concern. The reason is that the

PUC-Rio - Certificação Digital Nº 0912915/CA

dispersed anomalous code elements favor the scattering of a concern in the architectural design. The scattering of an architectural concern occurs, for instance, due to copy and paste practices. This neglection of concern separation principle often affects the architecture maintainability because changes in a specific concern will be spread over multiple components.

Figure 7.17 depicts an abstract representation of this pattern. Two anomalous classes stand out among the others in this figure – C3 and C4 – for realizing a concern that is not the predominant one of the component. Additionally, these anomalous classes realize a concern that is also modularized by external code elements. Therefore, the anomalous C1 and C2 classes represent an occurrence of the *Misplaced Concern* pattern in this example. . It is important to highlight that, in this example we are only showing a pattern occurrence at the class level, but the pattern can also manifest at the method level.

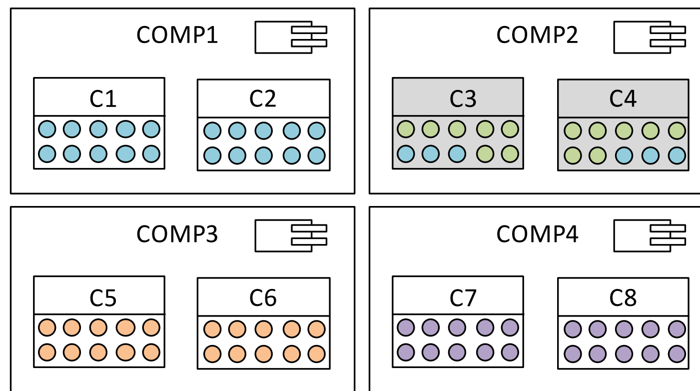

Figure 7.17: Abstract representation of *Misplaced Concern*.

Formal Definition. Given a software system, *S*, the set of occurrences of the *Misplaced Concern (MC)* pattern in *S* is formally defined in (9) as:

$$MC(S,th_1,th_2) = \{c_1 \mid c_1 \in ACE_{co1con} \wedge \exists\, con \in C_S \wedge co_1 \in AC_S \wedge co_2 \in AC_S, co_1 \neq co_2 \wedge |CE_{co1con}| < th_1 \wedge |CE_{co2con}| > th_2\} \quad (9)$$

where, th_1 and th_2 are generic constants that represent acceptable degrees of scattering per concern, such that $0 \leq th_1 \leq 1$ and $0 \leq th_2 \leq 1$. In order to identify the pattern instance illustrated in Figure 7.17 – C3 and C4 – using this formalism,

PUC-Rio - Certificação Digital Nº 0912915/CA

we have that: th_1 has to be higher than or equal to 0.33 and th_2 should be less than or equal to 0.80, respectively.

Algorithmic Solution. The algorithmic solution for detecting *Misplaced Concern* is presented in Listing 7.9. For a given concern (line 1), the algorithm computes the list of components that are weak and strong dedicated to realize this concern, using the WeakDedicatedComponents(..) and StrongDedicatedComponents(..) functions, respectively (lines 2 and 3). Then, for each weak dedicated component, the algorithm detects the anomalous code elements that realize the concern, using the AnomalousCodeElementsPerConcern(..) function (lines 6 - 9). These anomalous code elements constitute a pattern occurrence (line 8). This algorithm is run for each architectural concern of the system under analysis.

Listing 7.9 Detection of *MC* occurrences.

```
01  Let con be the concern under analysis
02  W = WeakDedicatedComponents(con, Th1)
03  S = StrongDedicatedComponents(con, Th2)
04  result = {}
05  if size(W) > 0 and and size(S) >0 then
06    for each co in W do
07      ACEC = AnomalousCodeElementsPerConcern(co, con)
08      occurrence add ACEC
09    end for
10  end if
11  return result
```

Concrete Example. Figure 7.18 depicts an occurrence of the *Misplaced Concern* pattern extracted from MIDAS system. This example is based on that previously shown in Figure 7.16. The occurrence of the *Misplaced Concern* comprises the SD_SD and SD_Engine classes in the Engine component, as well as the Fault Tolerance component. As shown in Figure 7.18, the SD_SD and SD_Engine classes are also being part of a *Misplaced Concern* occurrence. They modularize the Fault Tolerance concern, which should be implemented in its own component. In fact, as aforementioned, these classes were refactored in a later version moving the Fault Tolerance concern to its own component.

PUC-Rio - Certificação Digital Nº 0912915/CA

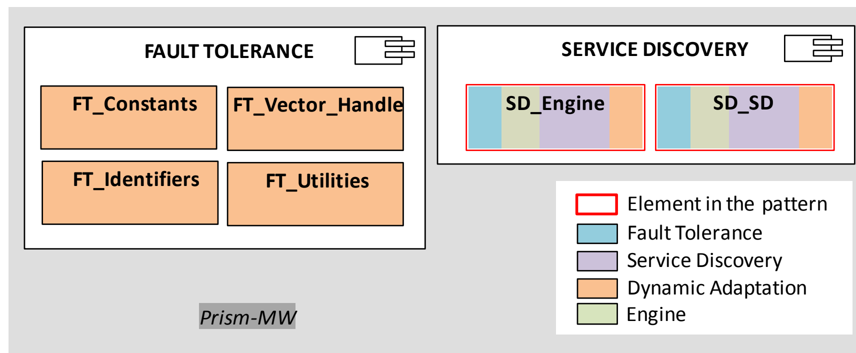

Figure 7.18: Occurrence of *Misplaced Concern* pattern in the MIDAS system.

7.6. Correlating Code Anomaly Patterns

We also observed that anomaly patterns are not fully independent, and the occurrence of one can (in)directly imply another one (and vice-versa). This section discusses the relationships among anomaly patterns that have been observed while carrying out previous studies (Chapters 4 and 5). Note we are not claiming these are the only types of relationships that can exist among anomaly patterns, other relationships could be still identified. Documenting such relationships helps architects and developers to understand and identify alternative reasons for the emergence of a particular code anomaly pattern.

Figure 7.19 illustrates relationships among the patterns where they are represented by "can be related", "can influence" and "cannot be" arrows connecting two patterns. The relationship "can be related" represents the case when a pattern can be seen under the perspective of another pattern. This occurs mainly when two patterns correlate in the sense that the occurrence of a pattern is also affected by another pattern. The relationship "can influence" is a step forward because it means the existence of a pattern can affect the emergence of another pattern. Such cause-effect relationship can be observed either in a single system version or along the system's evolution. Unlike the previous two relationships, the last relationship named "cannot be" corresponds to the case where two patterns cannot simultaneously affect the same code elements.

PUC-Rio - Certificação Digital Nº 0912915/CA

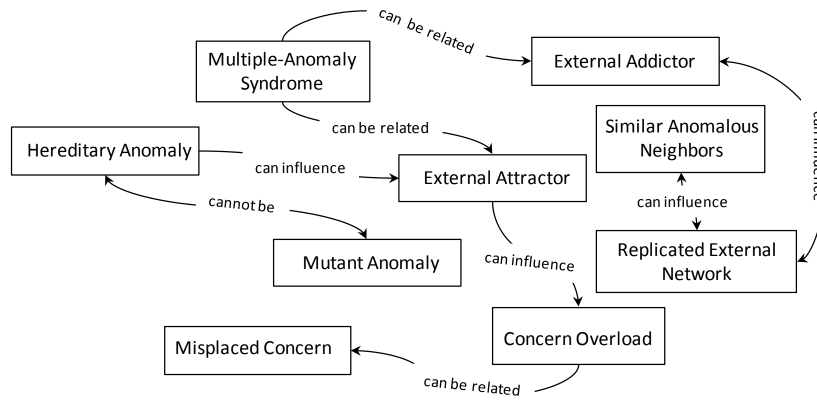

Figure 7.19: Relationships between code anomaly patterns.

An example of "can be related" relationship is defined between *Concern Overload* and *Misplaced Concern*. This relationship can be observed when a given component $co \in AC_S$ realizes many concerns where at least one of them is also realized by a different component. Hence, the component *co* can be also seen under the perspective of *Misplaced Concern* pattern. In this case both patterns *Concern Overload* and *Misplaced Concern* simultaneously affect the component co. Note that *Misplaced Concern* can occur in components that do not realize many concerns. Analogous reasoning applies to the other "can be related" relationships presented in Figure 7.19.

An interesting case of "can influence" relationship can be observed between *External Addictor* and *Replicated External Network*. *External Addictor* can propitiate the emergence of *Replicated External Network* when multiple code elements in the same component begin to increasingly depend on the same external elements. Thus, an occurrence of *Replicated External Network* is introduced. The inverse relationship could be also seen when at least one code element in the *Replicated External Network* pattern begins to depend incrementally on a high number of external anomalous elements. The other cases of "can influence" relationships showed in Figure 7.19 follow the same reasoning. Finally, *Hereditary Anomaly* cannot be classified as a *Mutant Anomaly*. The reason is that *Hereditary Anomaly* encompasses hierarchies where the root element is anomalous, whereas in the *Mutant Anomaly* the root element must be free of anomalies.

PUC-Rio - Certificação Digital Nº 0912915/CA

PUC-Rio - Certificação Digital Nº 0912915/CA

7.7. SCOOP: Detecting Architecturally-Relevant Code Anomalies

As discussed in the previous sections, the main benefits on identifying code anomaly patterns is to improve the accuracy of existing detection strategies. This identification involves the detection of single code anomalies and the analysis of different relationships between them (e.g. access the same external data). As a great amount of code anomalies infect software systems, their patterns detection can be an exhaustive and unviable task even in small systems without a proper tool support. This section describes SCOOP, a tool that leverages: (i) code anomaly patterns and, (ii) architecture-sensitive metrics and strategies to identify architecturally-relevant code anomalies. SCOOP is dedicated to developers, maintainers and code reviewers, who are interested in the continuous improvement of source code quality. SCOOP is implemented as an Eclipse plug-in allowing developers to use it during the implementation of Java systems.

Figure 7.20 depicts an abstract representation of the SCOOP architecture and how its main elements are related to each other. As it can be observed, SCOOP gets as input three different types of information, which are then processed by the SCOOP engine. Section 7.7.1 describes each input. Section 7.7.2 presents the tool engine, detailing how each of its elements work. Finally, Section 7.7.3 shows some screenshots of the user interface.

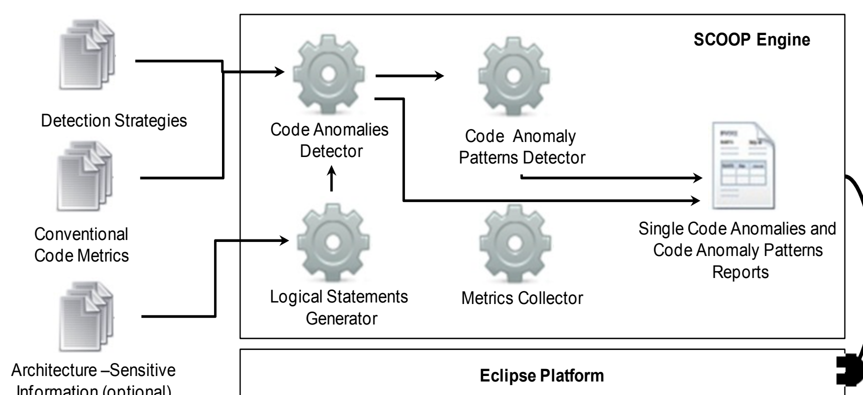

Figure 7.20: Abstract representation of the SCOOP architecture.

7.7.1.
The SCOOP Inputs

Detection Strategies. SCOOP relies on detection strategies to support the identification of code anomalies similarly to state-of-art tools. SCOOP gets an input *Rules.ds* (Appendix D), an internal configuration file that contains the definition of the detection strategies that will be used to identify code anomalies. The strategies are defined using a Domain-Specific Language (DSL) embedded in SCOOP (Appendix C). This DSL was implemented using XText (2010), a framework that provides edition features in order to reduce mistakes in the specification of detection strategies. Moreover, the language is enriched with support for selecting both conventional source code measures (Li and Henry, 1993; Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006) and architecture-sensitive ones (Section 6.2).

By counting on the proposed DSL, developers are able to define or modify their own detection strategies for distinct software systems. SCOOP does not force developers to define their own detection strategies, but they can rely on a pre-defined set of detection strategies provided by SCOOP in the *Rules.ds* file and just perform changes in these strategies according to their needs.

Conventional Code Metrics. SCOOP gets as input a file called *measures* – either an XML or a CSV – containing the conventional code measures collected for each code element in a given Java system under analysis (e.g. LOC, Efferent and Afferent coupling). We decided to get externally-collected code measures as input, as many tools already quantify them (Section 2.3.1). In particular, the current implementation of SCOOP is compatible with well-known code analyzer tools, such as Together and Understand, which were used in the previous exploratory studies (Chapters 4 and 5).

Architecture-Sensitive Information. SCOOP gets input from two kinds of architecture-sensitive information needed to collect the proposed metrics (Section 6.2) and to detect the code anomaly patterns: (i) mappings between code elements and architectural elements and (ii) projections of architectural concerns on the code elements. SCOOP does not support the specification of these kinds of information, since there are many tools documented in the literature with that purpose (Eisenbarth *et al.*, 2003; Maqbool *et al.*, 2007; FEAT, 2009; Garcia *et al.*,

PUC-Rio - Certificação Digital Nº 0912915/CA

2011; Nguyen *et al.*, 2011). The current implementation of SCOOP is compatible with Vespucci (2010) and ConcernMapper (2010). The first is a tool that allows developers to model the software system architecture in terms of its components or modules. The latter is a tool that supports the specification of the system concerns and how they are realized by the code elements. The files generated by these tools are input to SCOOP.

7.7.2. The SCOOP Engine

Logical Statements Generator. The Java system under analysis and the architecture-sensitive information are processed by SCOOP in order to detect architecturally-relevant code anomalies. SCOOP builds a *DecoratedAST*, a structure designed following the *Decorator pattern* (Gamma *et al.*, 1995) and that contains the nodes on the program AST. This structure will be use later to store other information besides that extracted from the programming language syntax. SCOOP uses BAT (*Bytecode Analysis Toolkit*) (Eichberg *et al.*, 2008) to generate a representation of structural properties of code elements as logical facts in *Prolog* (2010). Both kinds of architecture-sensitive information supported by SCOOP are also stored in a Prolog-based representation. This particular representation allows SCOOP to perform queries involving different sources of information. Additionally, the use of Prolog can foster the integration with other programming languages, as we can develop translators from these languages to Prolog. This characteristic is particularly interesting as recent studies have shown that most software projects are currently implemented in four different programming languages (Ubayashi *et al.*, 2010).

Metrics Collector. This module is in charge of collecting and processing the conventional code and architecture-sensitive measures supported by the DSL. To this end, the Metrics Collector module processes the *measures* file of the Java system under analysis and enriches the *DecoratedAST* adding the measures values to each node. The architecture-sensitive measures, which are not received as input, are collected by performing queries over the Prolog representation generated by the module *Logical Statement Generator*. Listing 7.10 illustrates a query used to determine dependencies, in terms of field declarations, between

PUC-Rio - Certificação Digital Nº 0912915/CA

classes that belong to components. As shown, this query relies on the mappings between code elements and architecture elements received as input (lines 2 and 3). All the queries performed by the Metrics Collector module are defined in an SCOOP internal file. Similarly to the conventional code measures, the Metric Collector module decorates the *DecoratedAST* adding the collected architecture-sensitive measures to each node. Therefore, the Metric Collector module returns an instance of the *DecoratedAST* containing the metric values for each node.

Listing 7.10. Prolog query used by SCOOP to detect dependencies.

```
01  field_dcl (pkgsrc, src, pkgdst, fld) :- field(class(pkgsrc, src), fld, dst),
02.                                          mapping(comp_dst, class(pkgdst, dst)),
03.                                          mapping(comp_src, class(pkgsrc, src),
04.                                          comp_src \== comp_dst.
```

Code Anomalies Detector. In order to detect the code elements suspects of being infected by anomalies, this module interprets the specified detection strategies in the *Rules.ds file* and process the *DecoratedAST* structure, which contains the gathered metrics. That is, the module processes the *DecoratedAST* to identify the code elements infected by each code anomaly defined in the *Rules.ds* file. As result, the module decorates the *DecoratedAST* adding the detected list of code anomalies to each node.

Code Anomaly Patterns Detector. Once the anomalous code elements are identified, this module processes the *DecoratedAST* and runs over it the algorithmic solutions presented in Listings 01-09. As result, a list will then be produced containing the occurrences of the code anomaly patterns for the system under analysis. The current version of SCOOP supports the detection of nine code anomaly patterns, but patterns can be included using the tool extension points.

7.7.3. The SCOOP User Interface

SCOOP extends Eclipse Platform with an additional preference page (Figure 7.21). The SCOOP preference page supports the configuration of specific threshold values for the program under analysis. This page also allows the engineers to activate or deactivate the algorithm of a particular code anomaly pattern. For instance, Figure 7.21 shows that all code anomaly patterns are active

PUC-Rio - Certificação Digital Nº 0912915/CA

in this specific project. In particular, according to the used preferences, a group of anomalous code elements constitute an occurrence of the *Concern Overload* pattern if they modularize more than 3 concerns as presented in Figure 7.21.

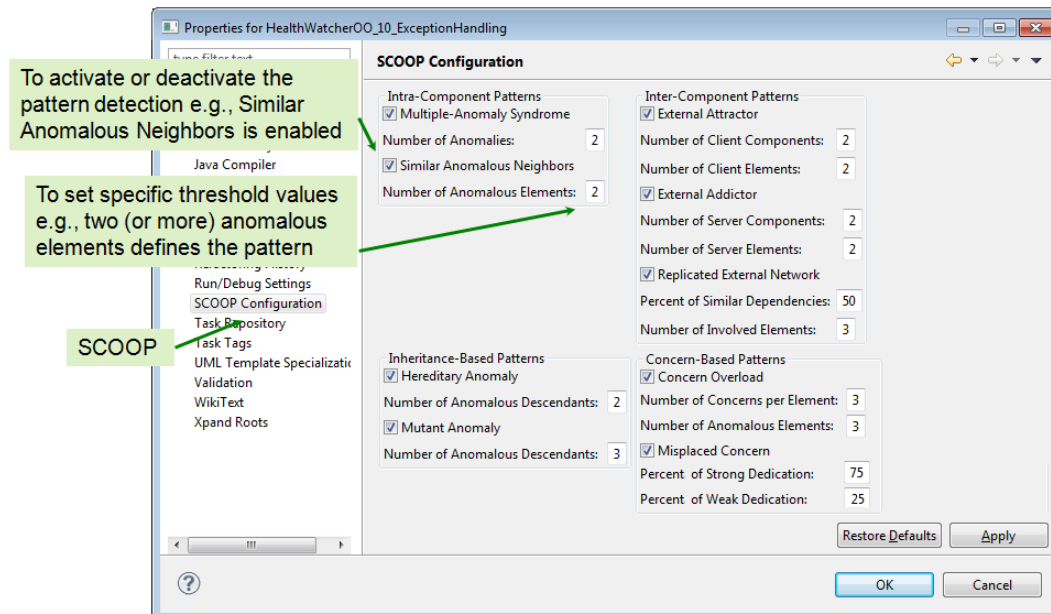

Figure 7.21: Patterns configuration in SCOOP.

The Anomaly Reports element of SCOOP presented in Figure 7.20 adds a new view. Figure 7.22 presents the view of SCOOP that visualizes the anomalous code elements detected. These elements are those identified using the strategies specified in the *Rules.ds* file. We use one of our case studies, called Health Watcher, in the illustrative example of Figure 7.22. As it can be noticed, the anomalous code elements detected are grouped by their type (e.g. class and method) and by the type of the anomaly they suffer from. In this example, UpdateMedicalSpecialityList.execute() and UpdateSymptomSearch.execute() methods are infected by the *Divergent Change* anomaly. By double-clicking on each detected occurrence, the user navigates to the source code of the corresponding code element.

PUC-Rio - Certificação Digital Nº 0912915/CA

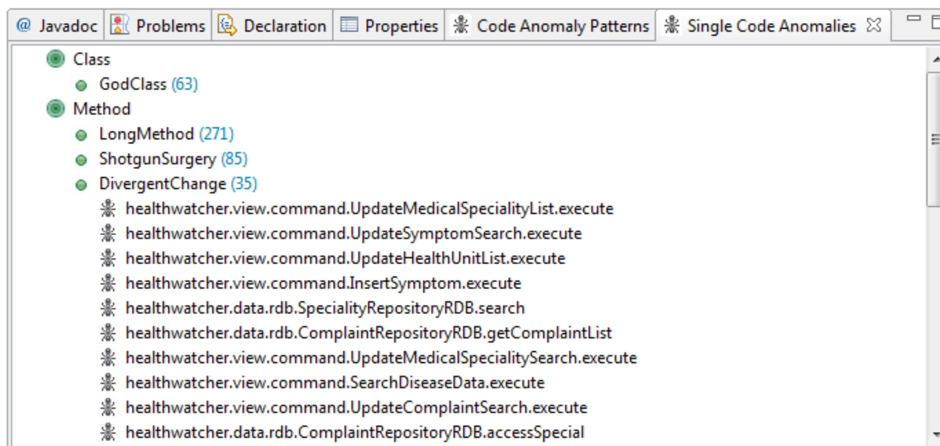

Figure 7.22: Code anomalies view.

SCOOP provides engineers with a second Eclipse view that shows the code anomaly patterns found in the target system. In the example depicted in Figure 7.23, also extracted from Health Watcher system, the FoodComplaint, StateComplaint, FoodComplaintState and SpecialComplaintState classes make up an occurrence of the *Replicated External Network* pattern. Similarly to the code anomalies view, the user can navigate to the source code involved in the pattern occurrence by double-clicking on the visualized element.

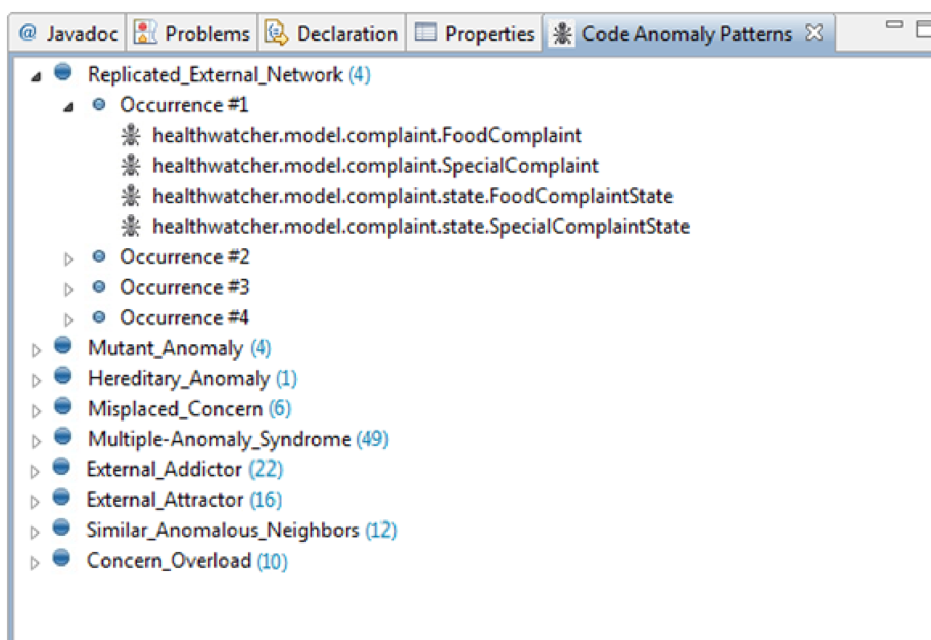

Figure 7.23: Code anomaly patterns view.

PUC-Rio - Certificação Digital Nº 0912915/CA

7.8.
Assessment Code Anomaly Patterns

This section presents a study that evaluates the usefulness of the code anomaly patterns when detecting architecturally-relevant code anomalies. Similarly to Chapter 6, the study presented in this chapter contributes to answer the last research question of this thesis (Section 1.4): *To what extent the proposed technique improves the accuracy of the conventional detection strategies when identifying architecturally-relevant code anomalies?* In the context of this study, this research question was decomposed into the following three research questions (RQ):

RQ4.4: How often do the documented code anomaly patterns manifest themselves in the system's implementation?

RQ4.5: Whether and to what extent code anomaly patterns are better indicators of architectural degradation symptoms than single code anomalies?

RQ4.6: What is the correlation between each type of code anomaly pattern and architectural degradation symptoms?

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

Analyze: the documented code anomaly patterns

For the purpose of: assessing their accuracy

With respect to: grouping architecturally-relevant code anomalies

From the viewpoint of: systems architects, developers and researchers

In the context of: five (05) software systems from different domains and following different architectural decompositions.

PUC-Rio - Certificação Digital Nº 0912915/CA

7.8.1. Hypotheses

In order to answer the three aforementioned research questions, we have defined the null and alternative hypotheses as shown in Table 7.1.

Table 7.1: Research questions and hypotheses of the study.

Research Questions	Hypotheses
RQ4.4	**Null Hypothesis, $H1_0$**: A significant proportion of code anomalies do not belong to the documented patterns. **Alternative Hypothesis, $H1_A$**: A significant proportion of code anomalies belong to the documented patterns.
RQ4.5	**Null Hypothesis, $H2_0$**: Patterns are not better indicators of architectural degradation symptoms than single code anomalies. **Alternative Hypothesis, $H2_A$**: Patterns are better indicators of architectural degradation symptoms than single code anomalies.
RQ4.6	**Null Hypothesis, $H3_0$**: Code anomaly patterns are equally related to architectural degradation symptoms. **Alternative Hypothesis, $H3_A$**: Code anomaly patterns are not equally related to architectural degradation symptoms.

PUC-Rio - Certificação Digital Nº 0912915/CA

7.8.2. Variable Selection

The following independent and dependent variables were defined in order to test the previous hypotheses.

Independent Variables. For $H1_0$, there is an independent variable, C_i, indicating the number of anomalous code elements in a system i. For $H2_0$ and $H3_0$ we only have one independent variable, A_i, indicating the total number of architectural degradation symptoms related to code anomalies in a system i. All occurrences of code anomalies used in testing these hypotheses were confirmed by developers (Section 7.8.4).

Dependent Variables. For $H1_0$, there is a dependent variable, C_i, indicating the total number of anomalous code elements grouped in patterns in a system i. For $H2_0$, there are two dependent variables. P_i indicates the proportion of architectural degradation symptoms encompassed in anomaly patterns in a system i. C_i represents the proportion of architectural degradation symptoms that are not involved in any anomaly pattern in a system i. For $H3_0$, there are as many independent variables as there are types of code anomaly patterns. Each variable

$P_{i,j}$ indicates the number of times a code anomaly pattern j is related to architectural degradation symptoms in a system i.

7.8.3. Selection Criteria and Target Systems

We relied on the target systems used in the previous study (Chapter 6) in order to assess the documented code anomaly patterns due to two main reasons. First, these systems meet a relevant set of criteria to the context of this study, as they (i) are infected by various symptoms of architectural degradation, which have been confirmed by architects and developers, (ii) suffer from a rich set of code anomalies, (iii) were projected following different architectural decompositions, and (iv) are implemented in Java, allowing us to use SCOOP in order to identify code anomaly patterns. Second, in the previous study we used the proposed architecture-sensitive strategies to identify architecturally-relevant code anomalies. Thus, code anomaly patterns could be built over the base of previously identified code anomalies.

7.8.4. Procedures for Data Collection

As discussed in Section 6.4.4 several kinds of information were gathered in the previous study. We were able to reuse that information for performing the study described in this chapter. For example, the set of code anomalies for each system, as well as the ground truth of architecturally-relevant code anomalies were already available. The former is particularly useful to assess to what extent code anomaly patterns encompass these critical code anomalies and, thus, evaluate both hypotheses $H2_0$ and $H3_0$. The only additional information that had to be collected for this study is the code anomaly patterns. We describe the details of such collection in the following.

Detecting Code Anomaly Patterns. The detection of code anomaly patterns was performed automatically, through SCOOP. To the best of our knowledge, SCOOP is the only tool that exploits different kinds of relationships between anomalous code elements to detect those that are likely to cause a harm impact on software architecture.

PUC-Rio - Certificação Digital Nº 0912915/CA

As shown in Figure 7.21, thresholds have to be specified in order to support the patterns identification. Although SCOOP provides a default set of thresholds, we selected a specific set of thresholds for each target system. The goal was to get the best possible results for each system, regarding the detected patterns. All the selected thresholds were confirmed with, at least three, system developers and architects, which were previously instructed about the particularities of each code anomaly pattern. All the involved participants have more than a decade of experience developing software systems. The list of the thresholds used in each target system is provided in Appendix B. After running SCOOP, we were provided with a list containing the occurrences of the anomaly patterns for each target system.

7.8.5.
Study Results

The following subsections present and discuss the main findings associated with the research questions of our study (Section **Error! Reference source not found.**). Section 7.8.5.1 discusses the frequency of the code anomaly patterns by analyzing their occurrences in each target system. Section 7.8.5.2 discusses to what extent code anomaly patterns are indicators of architectural degradation symptoms in the system implementation. Finally, Section 7.8.5.3 analyzes the correlation degree between each kind of code anomaly pattern and architectural degradation symptoms in the implementation of the target systems.

7.8.5.1.
Frequency of Patterns in the Target Systems

There was a significant difference on how often each anomaly pattern manifests in the target systems considering all analyzed code anomalies. The results summarized in Table 7.2 show the frequency of the patterns per system. The last row indicates the number of code anomalies analyzed in a particular system.

PUC-Rio - Certificação Digital Nº 0912915/CA

Table 7.2: Anomaly patterns per target system.

Code Anomaly Pattern	HW	MM	S1	S2	S3
Multiple-Anomaly Syndrome	49	39	65	102	81
Similar Anomalous Neighbors	12	17	32	21	27
Replicated External Network	17	5	12	17	16
External Addictor	22	10	26	18	30
External Attractor	16	6	18	27	14
Hereditary Anomaly	1	2	6	8	8
Mutant Anomaly	4	2	9	16	13
Misplaced Concern	6	5	18	19	9
Concern Overload	10	8	24	32	19
# of pattern occurrences	**137**	**94**	**210**	**260**	**217**

As shown in Table 7.2 code anomaly patterns consistently occurred in all the target systems. This indicates that those patterns may not be specific to a group of developers or particular system characteristics. Additionally, the total number of occurrences of code anomaly patterns varied in the target systems. For instance, S2 presented the highest number of anomaly pattern occurrences. We suspect this happened because it is the most complex system in terms of number of code elements and architectural components; consequently, the S2 system also presented the highest number of code anomalies. .

Individual Analysis of Pattern Categories. An analysis of the pattern occurrence frequency indicates that *Multiple-Anomaly Syndrome* was the pattern that occurred more often. This means that different anomalies infected frequently the same code element in the target systems. A wide diversity of *Multiple-Anomaly Syndrome* instances were observed in the target systems. The most frequent instances of this pattern observed in all target systems were: *God Class/Long Method*, *God Class/Feature Envy*, *Long Method/Intensive Coupling*, and *Disperse Coupling/Feature Envy*. Matching our intuition, co-occurrences of *God Class/Long Method* and *Long Method/Intensive Coupling* were responsible for more than 52% of all the observed simultaneous occurrences of code anomalies. The fact that certain co-occurrences of code anomalies are likely to appear more often than others motivates the investigation of whether the most frequent co-occurrences are the most harmful ones (Section 7.8.5.3).

Furthermore, some findings emerged from the analysis of *Replicated External Network*, *External Addictor* and *Externals Attractor* patterns. These patterns frequently appeared in all the analyzed systems. In particular, they

PUC-Rio - Certificação Digital Nº 0912915/CA

concentrated around 45% of all the relationships between components in the target systems. This high percentage shows how the tight coupling degree between components can be related to anomalous code elements as those patterns are related to the communication among architectural components.

Likewise, the high number of occurrences of *Misplaced Concern* and *Concern Overload* indicates how anomalous code elements can be related to the inadequate modularization of the architectural concerns. This observation seems to suggest that the projection of architectural concerns on the system implementation can be useful to identify a significant number of code anomalies. This suspicion is further investigated in the next section.

Although *Mutant Anomaly* and *Hereditary Anomaly* patterns appeared in all target systems, it was observed that the former occurred more frequently than the later. In many cases it was hard to determine whether the parent element was the source of the anomaly when descendants were classified as anomalous, due to two main reasons. First, parent elements were usually defined in external applications and referenced through the import mechanism. In these situations it was not possible to access the parent code and, thus, detect its code anomalies. Second, parent elements were usually implemented as abstract methods or interfaces. In such cases the code information emerged from the methods signature, which often did not specify any parameter. Thus, there was not enough information to apply strategies and detect code anomalies in the parent element.

Significance of Anomaly Patterns. In order to measure the significance of the pattern occurrences, we analyzed what proportion they represent in the sample of detected code anomalies. Figure 7.24 depicts the proportion of the anomalous code elements participating in anomaly patterns with those anomalous elements that do not make up patterns.

PUC-Rio - Certificação Digital Nº 0912915/CA

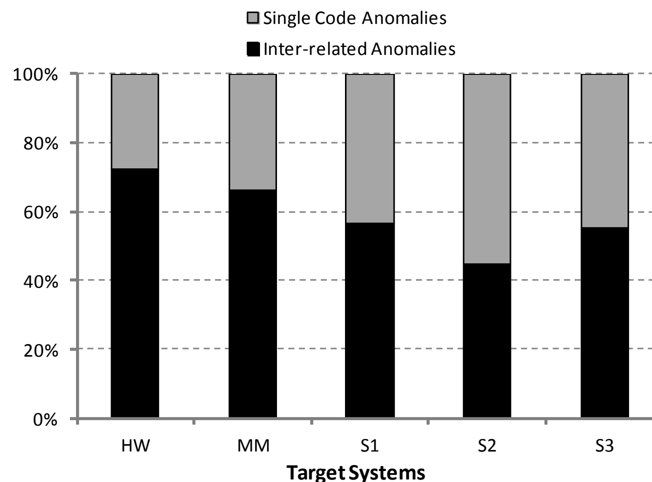

Figure 7.24: Anomalous code elements taking part in the patterns.

In the aforementioned proportions we considered all kinds of anomalous code elements; i.e. anomalous methods and classes. For instance, an anomalous class was considered to be in the patterns proportion if it is involved in any pattern related to classes. As it can be observed, more than 60% of the anomalous code elements in the HW and MM systems are involved in patterns instances. However, anomaly patterns encompass 57%, 45%, and 55% of the total number of anomalous code elements in S1, S2, and S3, respectively. In other words, less than 60% of the anomalous elements contribute to the patterns in these systems. Therefore, there is not enough information to reject $H1_0$. These low proportions might be caused by the fact that only eight code anomalies were considered in this study. This is a suspicion that should be tested in further studies. However, albeit low, these numbers can be relevant depending on whether they are how good indicators of architectural degradation symptoms in the target systems.

7.8.5.2. Code Anomaly Patterns and Architectural Degradation

Once confirmed that code anomaly patterns occurred frequently in the target systems, our study investigated whether and to what extent they were related to architectural degradation symptoms. In order to attempt rejecting $H2_0$, we test whether the proportion of architectural problems significantly varies between code anomalies involved and not involved in the anomaly patterns. We use Fisher's exact test (Shesking, 2007), which checks whether the proportion vary between

PUC-Rio - Certificação Digital Nº 0912915/CA

those two samples. We also compute the *Odds Ratio (OR)* (Shesking, 2007) that indicates the likelihood for an event to occur. The odds ratio is defined as the ratio of the *odds p* of an event occurring in one sample, i.e., the odds that architectural problems manifest in code elements with anomaly patterns (experimental group), to the *odds q* of the same event occurring in the other sample, i.e., the odds that architectural problems occur in code elements without patterns (control group): $OR = (p/(1-p)) / (q/(1-q))$. An OR equals to 1 indicates that the event is equally likely to occur in both samples. An OR greater than 1 indicates that the event is more likely in the first sample, while an OR less than 1 that it is more likely in the second sample.

Table 7.3 reports, for each target system, the number of anomalous classes: (i) contributing both to anomaly patterns and architectural degradation symptoms (*P-AD*), (ii) contributing to patterns but not to architectural degradation symptoms (*P-NoAD*), (iii) containing single anomalies and presenting architectural degradation symptoms (*NoP-AD*), (iv) containing single anomalies and not presenting architectural degradation symptoms (*NoP-NoAD*). This table also reports the result of Fisher's exact test and ORs when testing $H2_0$. Lower p-values indicate that classes involved in anomaly patterns adversely impact the software architecture more than other anomalous classes. In this analysis, a class contributes to an anomaly pattern if it is involved as a whole class in any pattern, or if any of its methods are involved in patterns.

Table 7.3: Contingency table and Fisher's test results.

System	P-AD	P-NoAD	NoP-AD	NoP-NoAD	p-values	OR
HW	62	8	19	18	3.62E-05	7.3
MM	23	6	4	12	5.44E-04	11.5
S1	157	32	123	98	1.36E-09	3.9
S2	287	45	203	215	5.48E-29	6.8
S3	192	28	92	75	1.13E-12	5.6

P = Pattern of code anomalies; AD = Architectural Degradation symptom.

Architectural Significance of Anomaly Patterns. Our analysis revealed a statistically-significant relationship between code anomaly patterns and architectural degradation symptoms in all the analyzed software systems, according to the level of confidence (i.e. 0.001). Also, the odds ratio revealed that the chances for code elements affected by anomaly patterns to be related to

PUC-Rio - Certificação Digital Nº 0912915/CA

architecture problems were sixteen times higher or more than for other code elements. This finding confirms that developers should be concerned with refactoring those elements involved in code anomaly patterns. It is important to note that the significance of this relationship was confirmed in different stages of the software systems evolution (Section 7.8.4). That is, the relationship was confirmed in intermediary (MM and S3) and advanced stages (HW, S1 and S2) of the system evolution. This observation suggests that code anomaly patterns are not only likely to adverse impact on the system architecture in latter versions of the systems. This finding is relevant as it indicates that developers could concentrate on refactoring specific code elements participating in anomaly patterns, and this effort should start in the earliest system version. These pattern-wise refactorings would likely save considerable time and effort when performing latter maintenance tasks. Thus, based on these results, we can reject $H2_0$.

Upstream and Downstream Analyses. In order to complement the Fisher's analysis, we analyzed the upstream and downstream relationships between anomaly patterns and architectural problems. The upstream relationship refers to what extent anomaly patterns in the code (low level of abstraction) were related to architecture problems (high level of abstraction). The downstream is exactly the opposite: it shows to what extent problems in the architectural design were related to code anomaly patterns. These analyses are useful because they show respectively: (i) what proportion of anomaly patterns is critical for systems' architecture, and (ii) what proportion of architectural problems could be fixed by refactoring anomaly patterns. The upstream analysis showed that up to 89% of all anomaly patterns were correlated to architectural degradation symptoms in Health Watcher, 87% in MobileMedia, 82% in S2, 80% in S1, and 75% in S2. The fact that some pattern occurrences were not related to architectural degradation symptoms might have happened because certain architectural problems were neither considered in this study nor detected by architects in the target systems. A downstream analysis revealed that up to 90% of all architectural degradation symptoms were related to anomaly patterns in Health Watcher, 85% in MobileMedia, 85% in S2, 78% in S1, and 73% in S3. The observed results indicated that the vast majority of architectural degradation symptoms was related to code anomaly patterns.

PUC-Rio - Certificação Digital Nº 0912915/CA

The results of these complementary analyses seem to confirm that code anomaly patterns are useful to locate potential sources of architectural degradation symptoms in the system implementation. In particular, we observed that around 80% of code anomaly patterns were indicators of architectural degradation symptoms. This suggests in turn that, by refactoring anomaly patterns developers might avoid spending their time on removing anomalies that do not represent threats to the architectural design. This observation is even more relevant for projects where there is no detailed documented architecture. The reason is that existing tools for architectural problem detection in the source code cannot be used in these situations, as they rely on detailed architectural information. We also found that a proportion of all architectural degradation symptoms, about 15%, were not associated with code anomaly patterns. In other words, there is a sample of architectural degradation symptoms that is not covered or explained by the code anomaly patterns. This means that certain architectural degradation symptoms may affect the system implementation even when code anomaly patterns are not detected.

Neglected Architectural Problems. The previous observations motivated us to analyze which architectural degradation symptoms were not related to code anomaly patterns in the target systems. An analysis of those architectural degradation symptoms indicated that some of them were only related to isolated code anomalies. For instance, about 12% occurrences of the *Overused Interfaces* were used by many anomalous-free code elements. These interfaces were classified as anomalous because changes performed on them caused ripple effects over architectural components. Therefore, these *Overused Interfaces* only involved single anomalous elements. Around 10% of *Unwanted Dependencies* were accidentally introduced by unrelated anomalous code elements. Finally, 7% of *Connector Envies* were only related to single *Long Method* occurrences. In these cases, connector interaction services (e.g. conversion of data formats) demanded high complexity, leading the method to be classified as a *Long Method*.

Moreover, other architectural anomalies were not related neither to code anomaly patterns nor any type of code anomaly. For instance, about 25% of *Extraneous Connector* were related to anomalous-free code elements that were using different connectors type (e.g. procedure call and event-based) to connect the same two components. Around 8% of *Cyclic Dependencies* were related to

PUC-Rio - Certificação Digital Nº 0912915/CA

well-modularized and non-complex methods. We believe that neglecting these architectural degradation symptoms does not constitute a big problem because (i) they represent a small proportion of the whole sample and (ii) some of them (e.g. *Cyclic Dependencies*) can be easily detected by existing code analyzers (e.g. Understand, 2011; Sonar, 2009).

Patterns vs. Single Code Anomalies. The fact that certain architectural degradation symptoms were related to anomalous elements, but not to anomaly patterns, lead us to analyze to what extent code anomaly patterns were better indicators of architectural problems than single code anomalies. Therefore, we compared the proportion of anomaly patterns related to architectural problems with the proportion of single code anomalies related to architectural problems. Figure 7.25 depicts the precision rates of code anomaly patterns, architecture-sensitive strategies and conventional ones when identifying architecturally-relevant code anomalies. This figure shows how code anomaly patterns were better related to architectural degradation symptoms than single code anomalies. In particular, patterns seem to help developers to concentrate their efforts on removing an expressive number of the most critical code anomalies. For instance, in system S1, the proportions of anomaly patterns and single anomalies related to architectural problems differs in 30%.

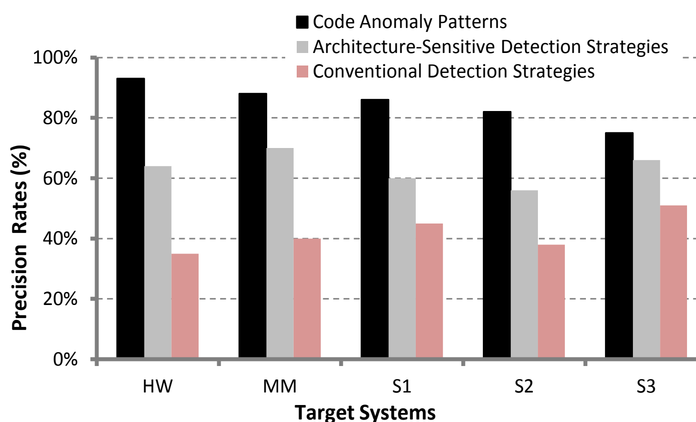

Figure 7.25: Precision of patterns vs. detection strategies in the identification of architecturally-relevant code anomalies.

PUC-Rio - Certificação Digital Nº 0912915/CA

Likewise, Figure 7.26 shows recall rates of code anomaly patterns, architecture-sensitive strategies and conventional ones when identifying architecturally-relevant code anomalies. This figure shows how architecture-sensitive strategies detected the highest number of architecturally-relevant code anomalies. However, the number of architecturally-relevant code anomalies grouped by the documented code anomaly patterns only differs in around 10%. Therefore, code anomaly patterns seem to be the most accurate mechanism to guide engineers in the removal of architecturally-relevant code anomalies.

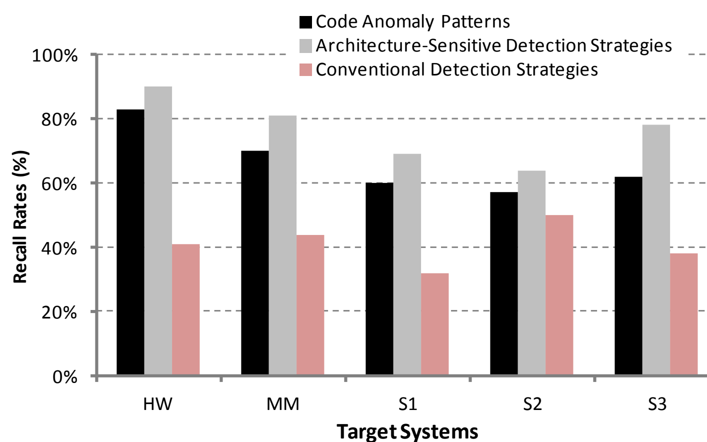

Figure 7.26: Recall of patterns vs. detection strategies in the identification of architecturally-relevant code anomalies.

However, it is still unknown whether such differences are significant. In order to perform this investigation, we use the (non-parametric) Mann-Whitney test (Shesking, 2007). This test compares two sets of variables and assesses whether their difference is statistically significant. Non-parametric tests do not require any assumption on the underlying distributions. We also computed the Cohen's *d* effect size (Shesking, 2007) to indicate the magnitude of the effect of a treatment on the dependent variables. A lower Cohen's d indicates a necessity for larger sample sizes. The effect size is considered small for $0.2 \le d < 0.5$, medium for $0.5 \le d < 0.8$ and large for $d \ge 0.8$.

Table 7.4 reports the results of Mann-Whitney's test and Cohen's *d* effect size. As it can be noticed Mann-Whitney test shows statistically significant differences between architectural degradation symptoms concentrated in code anomaly patterns and single code anomalies. Moreover, Cohen's *d* effect size

PUC-Rio - Certificação Digital Nº 0912915/CA

value is large: 0.84. Therefore, code anomaly patterns have stronger impact on the system architecture decomposition than single code anomalies. We thus can state that code anomaly patterns are significant better indicators of architectural problems than single code anomalies.

Table 7.4: Mann-Whitney's test results and Cohen's *d* effort.

	Mann-Whitney *p*	**Cohen *d***
Patterns vs. Single Anomalies	< 0.01	0.84

7.8.5.3. Specific Code Anomaly Patterns and Architectural Degradation

In order to attempt rejecting $H3_0$, we performed a correlation test between the code elements involved in each anomaly pattern and those elements with architectural problems. As we cannot assume a normal distribution in any of the code elements with anomaly patterns, the specific correlation test used here is the Spearman's rank correlation (Myers and Well, 2003) at the class-level. Furthermore, because the hypotheses are directional, a one-tailed test is performed. When the correlation test shows a moderate to strong correlation, the null hypothesis can be rejected, meaning that a specific code anomaly pattern is correlated with architectural degradation symptoms. Using the thresholds defined by Hopkins (2002) we consider the correlation value lower than 0.1 as trivial, 0.1-0.3 as minor, 0.3-0.5 as moderate, 0.5-0.7 as large, 0.7-0.9 as very large; and 0.9-1.0 as almost perfect.

Table 7.5: Spearman's rank correlation results.

	MAS	**SAN**	**EAD**	**EAT**	**REN**	**HA**	**MA**	**CO**	**MC**
HW	0.67	0.75	1.00	1.00	1.00	1.00	1.00	0.90	0.83
MM	0.66	0.65	0.82	0.83	0.80	1.00	1.00	0.87	0.80
S1	0.46	0.47	1.00	0.88	1.00	0.50	0.44	0.79	0.83
S2	0.49	0.48	1.00	0.96	1.00	0.50	0.43	0.87	0.84
S3	0.49	0.44	0.88	0.85	0.87	0.38	0.46	0.84	0.77

Strong Correlation with Architecture-Relevant Anomalies. Interestingly, our analysis indicates that all three *Inter-Component Patterns* (Section 7.3) present moderate to large correlation with architecturally-relevant code anomalies in the target systems (Table 7.5). This finding may explain why components with tight coupling degree were found to be the main sources of degradation in several studies (Godfrey and Lee, 2000; MacCormack *et al.*, 2006; Knodel *et al.*, 2008).

PUC-Rio - Certificação Digital Nº 0912915/CA

As we can observe, the best indicator of architectural problems was *Replicated External Network*. More than 73% of *Disperse Coupling* and 80% *Divergent Change* occurrences making up that pattern were classified as architecturally-relevant. An interesting finding emerges from analyzing occurrences of the *External Attractor* pattern. Many of its occurrences involved *Data* classes being accessed by external anomalous elements. In particular, these occurrences of the *External Attractor* pattern were harmful to the architecture design when they were often under maintenance and, therefore, suffered from many changes. This finding suggests that other kinds of information, such as change-proneness could be taken into consideration when detecting architecturally-relevant code anomalies.

The group of *Concern-based Patterns* also presents very large correlation with architectural degradation (Table 7.5). As we can observe, these correlation values vary from 0.7 to 0.9 in the target systems. It is important to mention that we could not identify code anomaly types that stand among the others as the most related to these patterns. Occurrences of all code anomaly types were related to the inappropriate modularization of architectural concerns.

Like *Inter-Component Patterns*, the correlation of *Concern-Based Patterns* and architecturally-relevant code anomalies is high since early versions of the systems, such as in S1 system (Section 6.4.3). Therefore, engineers may not have to wait until later system versions to identify these critical code anomalies. However, it is important to note that a higher amount of architecturally-relevant code anomalies were encompassed in patterns in later versions (e.g. Health Watcher and S2 systems). This suggests that patterns might be ‘shaped’ during the system evolution. However, this is suspicion that must be investigated in further studies.

Moderate Correlation with Architecturally-Relevant Anomalies. The correlation between code anomaly patterns and architecturally-relevant code anomalies was not so strong for the other two groups. In fact, we only found high correlation for *Inheritance-based Patterns* (Section 7.4) in MobileMedia and Health Watcher systems; moderate correlation was found in the remainder systems. The analysis of such patterns evidenced that their correlation with architecturally-relevant code anomalies was often observed when the parent element was defined in a different component from those where their descendants were defined. In such cases, descendants were favoring the occurrence of the

PUC-Rio - Certificação Digital Nº 0912915/CA

External Attractor pattern and increasing the coupling between architectural components. This finding raises the assumption that the harmful impact of certain patterns on the architecture could be better observed by analyzing their correlation with other patterns.

On the other hand, *Intra-Component Patterns* were those that presented lower correlation values with architecturally-relevant anomalies (Table 7.5). In few target systems they present moderate correlation values. However, the analyses of these patterns indicated that particular occurrences were good indicators of architectural degradation symptoms. For instance 75% (48 out of 64) of *Shotgun Surgery*/*Divergent Change* co-occurrences, 66% (76 out of 114) of *Long Method/Intensive Coupling* co-occurrences, and 53% (41 out of 78) *Disperse Coupling*/*Feature Envy* co-occurrences were classified as harmful to the architecture design. These occurrences were responsible for introducing a tight coupling among code elements. Likewise, particular occurrences of the *Similar Anomalous Network* pattern were good indicators of architectural degradation symptoms, such as groups of *God Classes* and *Feature Envies*.

7.9.
Threats to Validity

This section discusses the threats to validity of our study following the guidelines of Wohlin *et al.* (2000). These threats are categorized in four categories addressing construct, conclusion, internal, and external validity.

Construct Validity. Threats to construct validity are mainly related to possible errors introduced in the identification of code anomalies and their patterns. We are aware that code anomalies might be accidentally related to architectural problems. A first threat concerns the way the ground truth of code anomalies was identified. However, we limited such threat by considering only code anomalies whose impact on the architecture was confirmed by developers and architects involved in the evolution of the target systems. These developers and architects have experience on detecting code anomalies and architectural degradation symptoms in the systems implementation and architectural decompositions. Another threat concerns the detection of single code anomalies and, as a consequence, code anomaly patterns. We tried to mitigate this threat by

PUC-Rio - Certificação Digital Nº 0912915/CA

involving several architects and developers in the selection of the thresholds for detecting code anomalies and their patterns (Section 7.8.4). We considered the thresholds that were confirmed by all the architects and developers involved in that process. Lastly, construct validity refers to how the architecture design was documented and projected on the system implementation. We intentionally relied on an imperfect concern mapping sample, which presented 8% of mapping mistakes - similarly found in samples provided as output of existing tools for automatic mapping recovery (Nunes *et al.*, 2012).

Conclusion Validity. The number of evaluated systems and target anomalies threats the conclusion validity. Five systems with different architecture decompositions and implemented by different teams were analyzed. Of course, a higher number of systems is always desired. However, the analysis of a bigger sample in this study would be impracticable since we relied on different sorts of information provided by architects and developers, such as the list of architectural concerns, their projection on the system implementation at different granularity levels and, the list of architecturally-relevant anomalies. Thus, our sample can be seen as appropriate for this kind of study. The second issue is the completeness of the list of code anomalies and architectural degradation symptoms. We analyzed a significant number of code anomaly types, similarly to well-known studies in the field. Additionally, we relied on anomalies that affect code elements in significantly different forms. For instance, the selected code anomalies manifest in different code element types (i.e. method and class), range from anomalies that infect a single code element to anomalies that involve collaboration among code elements, and, are related to the inappropriate modularization of different properties (e.g. cohesion and coupling). Likewise, we tried to select heterogeneous forms of architectural degradation symptoms. Thus, we relied on degradation symptoms that: manifest in different architecture views (e.g. component-and-connector, module), range from a single component to relationships among several components, and represent different kinds of problems (syntactic and semantic-based). Additionally, all our conclusions were confirmed by using several statistic tests with a high significance degree. In particular, we paid attention not to violate assumptions of the performed statistical tests. Finally, we mainly used non-parametric tests that do not require making assumptions about the data set distribution.

PUC-Rio - Certificação Digital Nº 0912915/CA

Internal Validity. The main threats to internal validity of our study are the knowledge and experience of all the developers and architects involved in the data validation process. However, we tried to mitigate these threats by: (i) counting on the help of several architects and developers, having more than one decade of experience on code anomaly detection and removal, (ii) all the architects have a similar experience on detection and removal of architectural degradation symptoms in architectural components and system implementation, and (iii) we only considered information that was confirmed by all the developers and architects involved.

External Validity. The main threat to external validity is related to the nature of the evaluated systems. In order to reduce this threat we tried to use systems with different sizes, that suffer from a different set of code anomalies and that were implemented using different architectural styles and for different contexts (i.e. industry and academy labs). In addition, many systems in industry software projects follow the MVC and layered decompositions observed in the target systems of our study. However, we are aware that more studies involving a higher number of systems should be performed in the future.

7.10.
Summary

In this chapter, we have presented a preliminary catalogue of nine patterns of code anomalies identified by performing code anomaly analysis of several heterogeneous systems (Section 6.4.3). The anomaly patterns were classified in four categories (Sections 7.2 to 7.5) according to their common characteristics. We have certainly not claimed that this set of code anomaly patterns is complete; in fact other patterns can be identified in further studies. This chapter also presented orthogonal and overlapping relationships observed in the given definitions of the code anomaly patterns (Section 7.6).

For each code anomaly pattern this chapter described an algorithmic solution for detecting its occurrences (Listings 7.1 - 7.9). These algorithms combine different kinds of information including source code, architecture-sensitive and concern properties. Similarly to the code anomaly patterns, new detecting algorithms can be further defined. In particular, more optimized

PUC-Rio - Certificação Digital Nº 0912915/CA

algorithms can also be proposed. This chapter also presented SCOOP (Section 7.7), a tool that, relying on the architecture-sensitive metrics (Chapter 6) and the algorithms aforementioned, automates the identification of single code anomalies and the documented code anomaly patterns. This tool was implemented as an Eclipse plug-in and offers several extension points. For instance, new detection strategies and algorithmic solutions can be easily incorporated to SCOOP.

Finally, the chapter described the study carried out in order to investigate: (i) the incidence of code anomaly patterns in the architecture of software systems, (ii) the accuracy of code anomaly patterns when grouping architecturally-relevant code anomalies and (ii) the correlation between certain types of code anomaly patterns and architecturally-relevant code anomalies. In order to perform these investigations, a sample of nearly 1100 architecturally-relevant code anomalies, distributed in five (05) software systems was considered.

The key findings of our investigation are summarized below.

- All patterns manifested in the target systems. Some patterns (e.g. *Multiple-Anomaly Syndrome*) manifested more often than others (e.g. *Hereditary Anomaly*) in these systems. This finding suggests that code anomaly patterns are not specific for developer teams or software systems characteristics.
- Around 60% of the anomalous code structures are involved in documented code anomaly patterns. This means that a significant amount of code anomaly occurrences can be removed by applying common refactorings. In other words, a small number of refactorings could be used to remove a significant amount of code anomalies.
- The documented patterns enhanced the precision of architecture-sensitive and conventional strategies in around 30% and 55% respectively. This means that code anomaly patterns are likely to be a better way of reasoning about architecturally-relevant code anomalies. Therefore, our results indicate that developers should invest their effort on detecting and removing anomaly patterns, rather than single code anomalies, when addressing architectural problems.

PUC-Rio - Certificação Digital Nº 0912915/CA

- A vast majority of code anomaly patterns (7 out of 9) presented strong correlation with architecturally-relevant code anomalies. For those patterns which only presented a moderate correlation, we observed that their simultaneous occurrences with other patterns are likely to present (very) strong correlation. Since our results are restricted to the analysis of single code anomaly patterns, further analyses may be required to confirm or refute this finding.

Our empirical study only assessed the impact of code anomaly patterns on the architectural design. However, the patterns harmfulness should be also investigated under different perspectives, such as: code comprehension, maintenance effort, and testing. These studies could provide software engineers with evidence that allow them to determine how to conduct and prioritize their refactorings in order to save maintenance time and effort. Therefore, this research work also contributes to the code anomaly analysis field, encouraging further researches on inter-related code anomalies. The next chapter summarizes this thesis and points out directions for future work.

PUC-Rio - Certificação Digital Nº 0912915/CA