6
Detection of Architecturally-Relevant Code Anomalies with Architectural-Sensitive Information

Chapters 4 and 5 discussed how code anomalies (Fowler *et al.*, Chapter 3) can be related to architectural degradation symptoms in systems structured independently of the kind of modularization technique. In particular, Chapter 5 evidenced the inaccuracy of the conventional detection strategies to identify architecturally-relevant code anomalies. This inaccuracy has two main causes. First, the measures used in the conventional strategies quantify only properties of code elements derived from the syntax of programming languages. Consequently, these strategies disregard how the system implementation relates to its architecture.

Second, the conventional strategies are limited to detect only single occurrences of code anomalies. In other words, the conventional strategies do not analyze relationships among anomalous code elements (e.g. inheritance, method call) to indicate their harmful impact on the implemented architecture. However, analyzing single anomalous code elements is not accurate enough to reveal their harmful impact on the implemented architecture (Section 5.2). The reason is that an architectural element is usually implemented by multiple code elements. As a result, several architectural degradation symptoms, such as *Component Concern Overload*, *Overused Interface*, *Redundant Interface*, *Unwanted Coupling* and *Extraneous Connector* could only be observed by analyzing inter-related code elements.

These limitations indicate that the conventional strategies tend to send developers in wrong directions when addressing architecturally-relevant code anomalies, making the system maintenance impracticable in extreme cases (Eick *et al.*, 2001; Maccormack *et al.*, 2006; Knodel *et al.*, 2008). Therefore, there is a crucial need to provide developers with mechanisms that allow them to localize where the architecturally-relevant code anomalies are.

In order to overcome the aforementioned limitations of the conventional strategies and answer the third research question (Section 1.4), *How to accurately identify architecturally-relevant code anomalies?*, this chapter and Chapter 8 propose an approach for detecting and distinguishing architecturally-relevant code anomalies. The proposed approach is based on two novel steps. The first one comprises the detection of anomalous code elements. The novelty in this step is the use of *architecture-sensitive information* in the anomaly detection process to identify architecturally-relevant code anomalies neglected by the conventional strategies. Therefore, the goal is to enhance the recall of the conventional strategies when detecting those relevant code anomalies. This first step is introduced in this chapter.

The second step of the approach aims at distinguishing where the architecturally-relevant code anomalies are by analyzing recurrent relationships among anomalous code elements. The novelty in this step is that the proposed approach is not limited to detect single code elements. This characteristic will enable this approach to indicate the architectural degradation symptoms that emerge from inter-related code anomalies (Section 5.2.4.3). Consequently, the goal is to guide developers in the correct directions when identifying architecturally-relevant code anomalies. This second step will be addressed in Chapter 8.

In order to exploit architecture-sensitive information in the anomaly detection process, this chapter presents a suite of architecture-sensitive metrics and detection strategies. Although there are several measures that focus on quantifying coupling among architectural components, these measures do not quantify to what extent code elements contribute to this coupling (Briand *et al.*, 1993; Lakos 1996; Mancoridis *et al.*, 1998; Martin, 2003; Sarkar *et al.*, 2007, Sant'Anna *et al.*, 2007; Anquetil *et al.*, 2011). Similarly, there are many measures that quantify the scattering and tangling degree of architectural concerns in the system implementation (Ducasse *et al.*, 2006; Sant'Anna *et al.*, 2007). However, these measures do not quantify to what extent properties of code elements (e.g. complexity and cohesion) are affected by the inappropriate modularization of architectural concerns. Therefore, there is a clear need for architecture-sensitive measures; that is, measures that quantify properties of code elements based on how they relate to the system architecture.

In this context, this chapter starts by introducing a generic system meta-model (Section 6.1) which is the base for defining the proposed metrics (Section 6.2). Afterwards, the chapter presents a suite of architecture-sensitive detection strategies in Section 6.3. We evaluated the accuracy of the proposed architecture-sensitive strategies when identifying architecturally-relevant code anomalies in Section 6.4. The key points discussed throughout this chapter are discussed in Section 6.5. The research presented in this chapter has been partially published in (Macia *et al.*, 2013).

6.1. Basic Formalism

This section aims at expressing the concepts that will be used in the proposed detection approach in a consistent and meaningful manner. Some of these concepts were introduced in Chapter 2, but in this section they are formalized by means of set theory similarly to other works (Zhao, 2004; Bartolomei *et al.*, 2006; Figueiredo *et al.*, 2009). The goal of this definition is to provide the basis on which the architecture-sensitive metrics and architecture-sensitive detection strategies will be formalized. This chapter focuses on code anomalies detection, whereas Chapter 7 relies on the formalism introduced in this section for higher level code anomaly analysis. We seek to define a terminology and formalisms that are, as much as possible, extensible as well as independent of the programming language and architectural design (Bryton and Brito, 2007).

6.1.1. System Meta-Model

Figure 6.1 presents a generic meta-model for the analysis of code anomalies in the software system implementation. This meta-model, described using the UML notation (OMG, 2009), is based on previously defined meta-models for measurement (Bartolomei *et al.*, 2006; Briand *et al.*, 1997, Briand *et al.*, 1999). Meta-models for aspect-oriented programming languages were also surveyed (Chavez and Lucena, 2002; Lions *et al.*, 2002; Han *et al.*, 2005). However, for the sake of simplicity, the proposed meta-model only defines elements that are

relevant to the scope of the proposed detection approach. Additionally, the meta-model is kept simple in order to achieve a greater level of generalization.

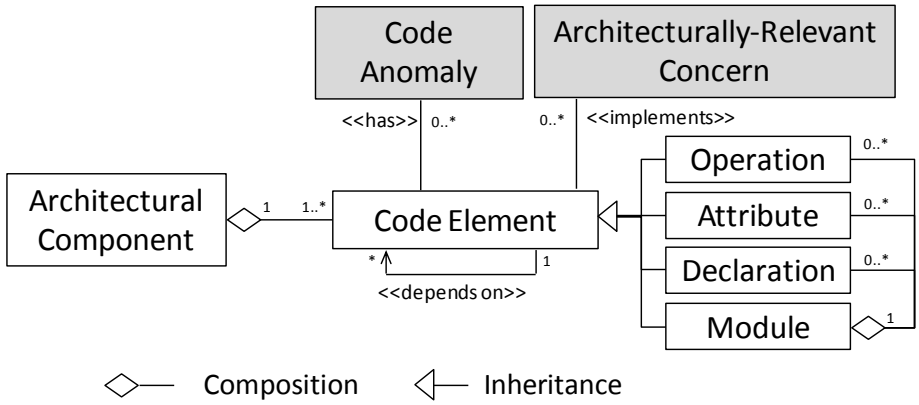

Figure 6.1: System meta-model.

An instance of the meta-model presented in Figure 6.1 is called a *software system*. An *architectural component* is associated with code elements. A *code element* is an *operation*, an *attribute*, a *declaration* or a *module*. The meta-model defines not only the possible relations between the different types of system elements, but also the mappings of such elements to architecturally-relevant concerns. It also describes how code elements of the system can suffer from anomalies.

It is important to note that certain specific information is not included in the proposed meta-model due to its generality. For instance, architectural components were not detailed in the generic meta-model because they are specific to the architectural view used to modeling the system architecture, such as subsystems and connectors. This issue can be addressed though by specifying the "*Architectural Component*" node of the meta-model with all the specific types of components that can be defined in each architectural view.

The key properties of the proposed meta-model are formalized through definitions 1 to 4 using a running example, Figure 6.2. The example was extracted from a logistic system, one of the case studies that will be introduced in Chapter 8. The goal of this system is to manage the oil production that is visualized in scenarios, which in turn are stored in folders. Users can edit the information associated with the oil production. Other system details cannot be provided due to copyright restrictions.

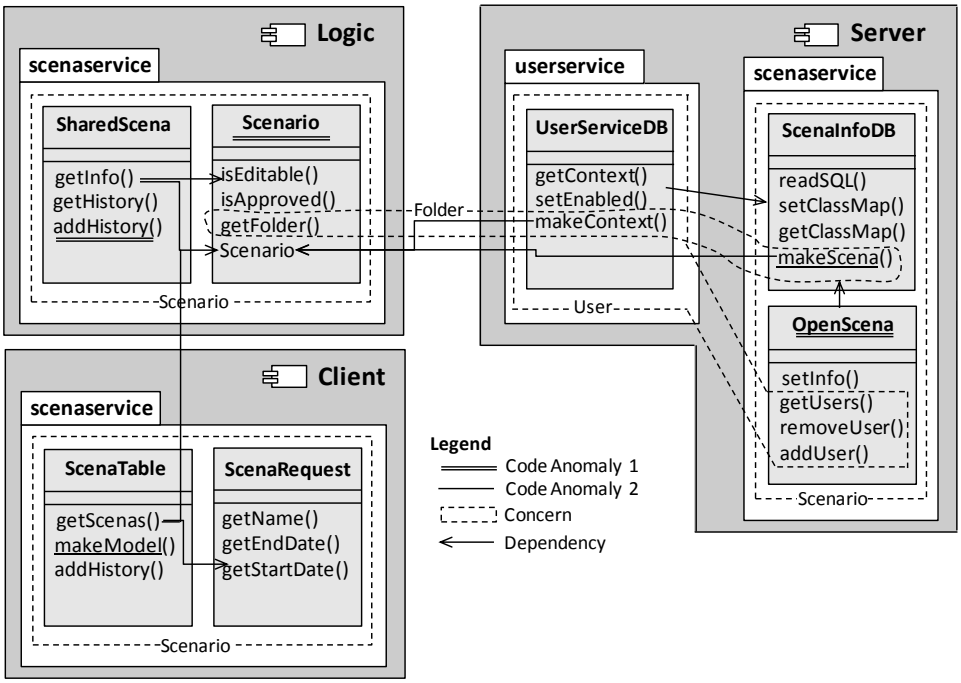

Figure 6.2: A slice of the logistic system design.

Definition 1: System, Architectural Element and Code Element. *A software **system**, S, consists of a set of **architectural components**, denoted by AC_S. An architectural component $co \in AC_S$ is implemented by a set of **code elements**, denoted by $CE_{co} \in CE_S$. A code element can be an attribute, an operation, a declaration or a module. Let Att_{co} be the set of attributes that implement co, Op_{co} be the set of operations that implement co, and Dec_{co} be the set of declarations that implement co, $CE_{co} := Att_{co} \cup Op_{co} \cup Dec_{co}$.*

The notion of architectural component corresponds to those elements into which the system architecture is decomposed (see Section 6.2.2). Also, a declaration is usually defined by aspect-oriented languages. For instance, it can be either a 'parent declaration' or a 'pointcut declaration' in AspectJ (Kiczales *et. al.*, 2001). The system illustrated in Figure 6.2 consists of three different architectural components (Logic, Server, and Client boxes), seven modules (SharedScena, Scenario, UserServiceDB, ScenaInfoDB, OpenScena, ScenaTable and ScenaRequest classes), and twenty-four (24) operations, which are defined within these modules. In that figure, SharedScena and Scenario classes implement the Logic component. UserServiceDB, ScenaInfoDB, and OpenScena classes implement the Service component. Finally, ScenaTable and ScenaRequest classes implement the Client component.

Definition 2: Free and Anomalous Code Element. *Code elements in a software system, S, are often affected by a set of code anomalies denoted by CA_S. Therefore, the code elements in S can be classified in two subsets: ACE_S, the set of* ***anomalous code elements*** *affected by at least one anomaly $a \in CA_S$ and FCE_S, the set of* ***code elements free of anomalies****, where $CE_S := ACE_S \cup FCE_S$. The set of code elements that: (i) implement an architectural component $co \in AC_S$ and (ii) are infected by a particular code anomaly $a \in CAs$, is denoted by* $\boldsymbol{CE_{coa}}$.

In Figure 6.2 there are two anomalies that affect two classes and three methods. While Anomaly 1 infects the ScenaTable.makeModel() and ScenaInfoDB.makeScena() methods, Anomaly 2 affects the OpenScena and Scenario classes, and the SharedScena.addHistory() method.

Definition 3: Dependency between Code Elements. *Let be D_S the set of dependencies between code elements in a system S. A* ***dependency*** *$D_{ij} \in D_S$ between two code elements $c_i \in CE_S$ and $c_j \in CE_S$ is defined as a tuple* $\boldsymbol{(c_i, c_j)}$*, where c_i inherits, calls operations, accesses attributes of c_j or establishes semantic relationship with c_j.*

In Figure 6.2 the SharedScena.getInfo() method depends on the Scenario.isEditable() method because the former calls the latter. The same situation occurs with the ScenaTable.getScenas() and ScenaRequest.getStartDate() methods. Additionally, in Figure 6.2 the UserService and ScenaInfo classes depend on the Scenario class because the first two classes create objects of the last one.

Definition 4: Architectural Concern. *In a software system, S, code elements implement* ***architectural concerns****, which are denoted by C_S. Therefore, an architectural concern C_i is implemented by a set of code elements such that $C_i := CE_i$, where $CE_i \neq \varnothing$. On the other hand, for a given code element $c \in CE_S$, c_{Con} denotes the set of architectural concerns that c implements.*

An architecturally-relevant concern is defined as an *architect's interest*, *architect's purpose* or *system functionality* that significantly influences the system architecture (Chapter 2). As it can be noticed, an architectural concern can be realized by any type of code element – just a single code element or a collection of different ones. Figure 6.2 depicts the implementation of three architecturally-relevant concerns: Scenario, Folder and User. These concerns are considered to be architecturally-relevant because they are important functionalities that ruled the

way in which architects modeled the system architecture. In that figure, the implementation of Scenario and Folder concerns comprises code elements defined in various components. The Scenario concern is implemented by all the code elements in the Logic and Client components and two out of three classes in the Server component. The Folder concern is implemented by methods defined in the Logic and Server components. Finally, the User concern is only implemented by the UserServiceDB class.

6.1.2. Meta-Model Instantiation

The meta-model (Figure 6.1) is abstract enough to be instantiated for different architectural views and programming languages. In the previous section the meta-model was instantiated for a particular software system in order to exemplify the introduced concepts. This section provides a brief illustration of how the meta-model can be instantiated to languages targeting different purposes and levels of abstraction. The depicted instantiation relies on architecture modeling languages for the component-and-connector and module views (Bass *et al.*, 1997), and two programming languages, namely Java and AspectJ (Kiczales *et al.*, 2001). We have chosen these languages for illustration because they are widely used in practice. The goal of Table 6.1 is to illustrate how the meta-model elements can be mapped in software systems structured with these languages. In particular the software system #1 is structured with the component-and-connector view and Java language, whereas software system #2 is structured with the module view and AspectJ language. A blank cell means that the abstraction is not implemented by any element of the language. For example, declarations are usually valid only for aspect-oriented languages, such as AspectJ (Table 6.1).

Table 6.1: Meta-Model instantiation.

	Software System #1	Software System #2
Concern	Data conversion	Data management
Architectural Component	Component and connector	Module
Module	Class, Interface and package	Class, Interface, package and Aspect
Attribute	Class variable and field	Class variable, field and inter-type declaration.
Operation	Method and constructor	Method, constructor, inter-type declaration and advice
Declaration	-	Pointcut and declare statement
Code Anomaly	Long Method and Large Class.	Large Class and Composition Bloat.

6.2. Architecture-Sensitive Metrics

This section defines a suite of metrics that aims at quantifying information that can be extracted from the meta-model presented in Figure 6.1. The proposed metrics can also gathered information from different recovered architectural views such as: component-and-connector view and module view. Before defining the architecture-sensitive metrics in detail, Table 6.2 presents a summary of them. This table provides a catalog with brief definitions for the metrics and their association with the kind of information they quantify. The goal is to provide the reader with a big picture of the proposed metrics and also facilitate the reference to the metric definitions while reading the remainder of the text.

Table 6.2: Summary of the suite of architecture-sensitive metrics.

Information	Metric	Definition
Architectural Component	Number of External Elements (NEE)	Counts the number of external code elements that a measured code element depends on.
	External Fan-out (EFO)	Counts the number of architectural components a measured code element depends on.
	External Fan-in (EFI)	Counts the number of architectural components that depend on the measured code element.
	Architectural Component Locality (ACL)	Counts the relative number of dependencies that the measured code element has in its component in relation to the total number of its dependencies.
Architectural Concern	Number of Architectural Concerns (NAC)	Counts the number of architectural concerns a measured code element implements.
	Architectural Concern Locality (CoL)	Counts the relative number of architectural concerns implemented by a code element within its own component in relation to the total number of architectural concerns it implements.
	Architectural Concern Cohesion (CoC)	Counts the relative number of pairs of methods in the measured class that implement the same architectural concern in relation to the total number of pairs of methods within the class.

As it can be noticed, the proposed metrics are grouped into two categories: architectural components and architectural concerns. The first category refers to how architectural components are implemented by the code elements, while the latter quantifies how architectural concerns are modularized by the code elements. In other words, the proposed metrics are based on previous knowledge about the mapping of architectural information (i.e. components and concerns) on the code elements.

We decided to rely on components and concerns projections due to several reasons. First, empirical evidence (Chapters 4 and 5) indicates that these projections are useful to detect those architecturally-relevant code anomalies neglected by conventional detection strategies. For instance, the projection of architectural components on the system implementation allows identifying code elements related to a high coupling degree among architectural components. Second, the projection of architectural concerns on the system implementation enables us to analyze which code elements modularize several architectural purposes. Finally, both kinds of projections can be recovered from the source code by using third-party tools (Eisenbarth *et al.*, 2003; Maqbool *et al.*, 2007; FEAT, 2009; Garcia *et al.*, 2011; Nguyen *et al.*, 2011). Consequently, the proposed metrics could be gathered even when the systems architects or the architectural documentation are not available.

In the following, each metric is described in terms of: (i) an informal definition, (ii) the measurement purpose, (iii) a formal definition based on set theory, and (iv) an example. The metrics are formally defined in terms of the terminology and formalisms introduced in Section 6.1. The goal of formally definition of the metrics is to express them consistently and unambiguously.

6.2.1. Metrics for Architectural Components

As presented in Table 6.2, this section documents the following metrics: *Number of External Elements*, *External Fan-out*, *External Fan-in* and *Architectural Component Locality*.

Definition 5: Number of External Elements (NEE). *NEE counts the number of "external" code elements used by the measured code element. A code*

element is considered to be external when it belongs to a component different from that where the measured code element belongs. The goal of this metric is to allow engineers to identify cases of code elements that depend on many different external elements.

Formal Definition of NEE. Given a system, S, $\forall\ c_1 \in CE_S$, $NEE(c_1)$ is represented as:

$$NEE(c_1) = |\{ c_2 \mid (c_1, c_2) \in D_S \wedge (\exists\, co_1 \in AC_S \wedge co_2 \in AC_S \mid co_1 \neq co_2 \wedge c_1 \in CE_{co1} \wedge c_2 \in CE_{co2})\}|$$

Example. According to Figure 6.3, the value of *NEE* for the ScenaTable class is one (01) because its methods depend on two classes, ScenaRequest and Scenario, where only the last one is external. That is, the Scenario class is defined in Logic component, whereas the ScenaTable class is defined in the Client component. Note that this metric differs from the conventional coupling metrics.

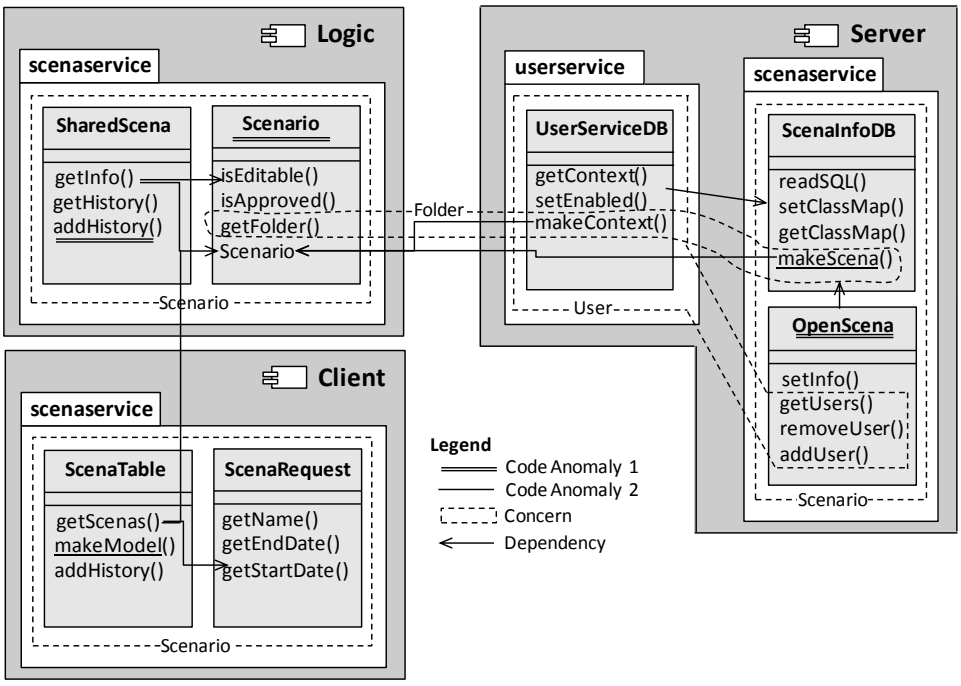

Figure 6.3: A slice of the logistic system design.

We have already presented the slice of the logistic system design. We repeat it here in order to facilitate referring to it during the following discussion.

Definition 6: External Fan-out (EFO). *EFO counts the number of components used by the measured code element. The goal of this metric is to allow engineers to identify code elements that depend on many different architectural components. Note that unlike NEE, which focuses on quantifying the number of code elements, EFO quantifies the number of architectural components.*

Formal Definition of EFO. Given a system, *S*, $\forall c_1 \in CE_S$, $EFO(c_1)$ is represented as:

$$EFO(c_1) = \left|\{ co_2 \mid (c_1, c_2) \in D_S \wedge (\exists co_1 \in AC_S \wedge co_2 \in AC_S \mid co_1 \neq co_2 \wedge c_1 \in CE_{co1} \wedge c_2 \in CE_{co2})\}\right|$$

Example. In Figure 6.3, the value of *EFO* for the UserServiceDB class is one (01) because it depends on a class that is defined in the Logic component.

Definition 7: External Fan-in (EFI). *EFI quantifies the opposite property quantified by the EFO metric. That is, EFI counts the number of architectural components that depend on the measured code element. The idea behind this metric is to enable engineers to identify code elements that might be the source of undesirable ripple effects over the system architecture.*

Formal Definition of EFI. Given a system, *S*, $\forall c_1 \in CE_S$, $EFO(c_1)$ is represented as:

$$EFI(c_1) = \left|\{ co_2 \mid (c_2, c_1) \in D_S \wedge (\exists co_1 \in AC_S \wedge co_2 \in AC_S \mid co_1 \neq co_2 \wedge c_1 \in CE_{co1} \wedge c_2 \in CE_{co2})\}\right|$$

Example. In Figure 6.3, the value of *EFI* for the Scenario.Scenario constructor is two (02) because there are three classes that depend on it, where two of them belong to different architectural components: Client and Server.

Definition 8: Architectural Component Locality (ACL). *ACL counts the relative number of dependencies that the measured code element has in its own component in relation to the total number of its dependencies. The goal of AEL is to help engineers to decide whether a code element depends more on external code elements than on those defined in its own architectural component. This*

situation may indicate that the code element introduce undesirable dependencies among components or it should be moved to another component.

Formal Definition of ACL. Given a system, *S*, $\forall c \in CE_S$, *ACL(c)* is represented as:

$$ACL(c) = NEE(c) / |\{c_2 \mid (c,c_2) \in D_S \wedge c_2 \in CE_S\}|$$

Example. In Figure 6.3, the value of *ACL* for the SharedScenario class is 0.5. The reason is because this class depends on two classes, Scenario and ScenaRequest, where the last one is external.

6.2.2. Metrics for Architectural Concerns

As presented in Table 6.2, this section documents the following metrics: *Number of Architectural Concerns*, *Architectural Concern Locality* and *Architectural Concern Cohesion*.

Definition 9: Number of Architectural Concerns (NAC). *NAC counts the number of architectural concerns that the measured code element modularizes. The goal of NAC is to help engineers to identify code elements that deal with many architectural concerns. A high value of this metric suggests that the measured code element violates the principle of a single functionality, affecting the cohesion of its enclosing component.*

Formal Definition of NAC. Given a system, *S*, $\forall c \in CE$, *NAC(c)* is represented as:

$$NAC(c) = |\{c_{Con}\}|$$

Example. In Figure 6.3, the value of *NAC* for the UserServiceDB.makeContext() method is two (02) because it implements User and Folder architectural concerns. The value of *NAC* in the Scenario class is also two (02). The reason is that, for classes, the metric is computed as the total number of distinct concerns implemented by its methods and attributes.

Definition 10: Architectural Concern Locality (CoL). *CoL counts the relative number of architectural concerns implemented by a code element within*

its own component in relation to the total number of architectural concerns it implements. The purpose of CoL is to support engineers for distinguishing whether a code element implements more external architectural concerns than those implemented within its enclosed architectural component. A concern is considered to be external when it is implemented by external code elements.

Formal Definition of CoL. Given a system, *S*, $\forall c \in CE_{co}$, *CoL(c)* is represented as:

$$CoL(c) = |\{cr \mid \exists co \in AC_S \wedge cr \in C_S \wedge cr \in co_{Con} \wedge cr \in c_{Con}\}| \,/\, NAC(c)$$

Example. In Figure 6.3, the UserServiceDB class implements two architectural concerns: Folder and User, where Folder concern is external because it is also implemented by external code elements (e.g. Scenario). Therefore, the value of *CoL* for UserServiceDB is 0.5.

Definition 11: Architectural Concern Cohesion (CoC). *CoC is defined as the relative number of pairs of methods in the measured class that implement the same set of architectural concerns in relation to the total number of pairs of methods within the class. The purpose of this metric is to enable engineers to identify classes whose methods do not implement the same architectural concerns. When the methods in a class implement distinct concerns, the class can suffer from effects coming from changes associated with any of the concerns implemented within it.*

Formal Definition of CoC. Given a system, *S*, consider a class $c \in CE_S$. In order to define *CoC(c)* two additional functions are required, MethodPairsSameConcerns(c) and MethodsPairs(c). The former function returns the number of pair of methods that modularize the same set of concerns, whereas the latter returns the number of pairs of methods in a given class.

CoC(c) = MethodPairsSameConcerns(c) / MethodsPairs(c)

Example. In Figure 6.3, the value of the metric *CoC* for the Scenario class is 0.33. The reason is that, from the three pairs of methods in this class only one of them (i.e. isEditable, isApproved) implements the same set of architectural concerns: Scenario concern. As shown in Figure 6.3, the other pairs of methods realize different concerns (i.e. Folder and Scenario).

6.3. Architecture-Sensitive Detection Strategies

This section presents a suite of eight (08) detection strategies that support engineers while identifying architecturally-relevant code anomalies. It is our intention to reduce the shortcomings of conventional detection strategies discussed in Chapter 5. Specifically, the presented strategies detect the object-oriented code anomaly types that were related to architectural degradation symptoms in the context of previous studies (Chapters 4 and 5). However, architecture-sensitive strategies can also be defined in order to identify occurrences of a different set of code anomalies.

The proposed strategies combine information gathered from architecture-sensitive metrics (Section 6.2) and conventional code metrics (Chidamber and Kemerer, 1994; Li and Henry, 1993; Lanza and Marinescu, 2006). The goal of these strategies is to enhance the low recall rates of the conventional strategies when identifying architecturally-relevant code anomalies (Chapter 5). Additionally, our intention is to avoid neglecting the detection of code anomalies that do not influence the system architecture. The reason is that code anomalies might induce other maintainability problems (e.g. error-proneness) and are likely to be indicators of architectural degradation symptoms in later versions of the system (Section 4.2.2.2). However, engineers can still distinguish the architecturally-relevant code anomalies using architecture-sensitive strategies. Engineers can focus on using the parts of the detection strategy that explore the architecture-sensitive metrics. Additionally, engineers can use the architecture-sensitive strategies jointly with the mechanism to be presented in Chapter 7 in order to distinguish the architecturally-relevant code anomalies.

Similarly to the aspect-oriented strategies presented in Chapter 3, the architecture-sensitive strategies defined in this section are structured in the form *name<entity> := condition*. The *name* corresponds to the type of the code anomalies detected by the strategy. The *entity* indicates the type of the code element over which the strategy is applied. The *condition* part encompasses the combination of one or more measure outcomes related to the code element under analysis. The condition parts related to the architecture-sensitive metrics are highlighted in gray in the following descriptions of the strategies. Additionally,

the definition of the strategies relies on symbolic constants in the place of thresholds (e.g. LOW and HIGH). The choice of these values will depend on the characteristics of the system and programmers styles.

Furthermore, it is discussed how each proposed strategy would help to identify code elements that may harmfully impact the implemented architecture. In the next sections, the proposed strategies are grouped according to categories presented in Lanza and Marinescu (2006): element anomalies and collaborative anomalies. The *Element Anomalies* category refers to those code anomalies that can be detected by looking at the code element as a single entity. The *Collaborative Anomalies* category groups those code anomalies that emerge from relationships among code elements.

6.3.1. Detection Strategies for Element Anomalies

This section describes the architecture-sensitive strategies proposed to identify element anomalies. Specifically, the proposed strategies detect the following code anomalies: *Feature Envy*, *Misplaced Class*, *Long Method* and *God Class*.

6.3.1.1. Feature Envy (FE)

Feature Envy (Fowler *et al.*, 1999) refers to methods which are more interested in the data of other classes than the one it is actually in. This might be a sign that the infected method was misplaced and that it should be moved to another class. This situation favors the architectural degradation when the accessed data and the infected method are not as architecturally close as they should be (Godfrey and Lee, 2000; Knodel *et al.*, 2008). The reason is that this remote data-method increases the coupling degree, ripple effects and faults among the architectural components.

The goal of the architecture-sensitive strategy is to ensure the detection of *Feature Envy* occurrences that should be moved to another component. This detection cannot be ensured by using the conventional strategy (Lanza and Marinescu, 2006) because it does not consider whether the measured method and

the accessed data belong to different architectural components. In fact, the conventional strategy only focuses on measuring the number of attributes a given method accesses (using the *ATFD* clause) (Chapter 5), without considering in which architectural component these attributes are defined.

The architecture-sensitive strategy is a step beyond because it considers different types of accessed code elements, such as attributes and methods using the *NEE* clause. In particular, it looks for methods that access a great amount of code elements from a few components. To this end, the architecture-sensitive strategy first verifies whether the method accesses more external code elements than internal ones (using the *ACL* clause). Moreover, it checks whether the accessed code elements are located in a few components (using the *EFO* clause). The combination of these types of information suggests that the measured code element should be defined in another component.

FE<method> := (**ATFD** > FEW **or NEE** > FEW) **and**
(**LAA** > ONE THIRD **or ACL** < THIRD) **and**
(**FDP** < FEW **or EFO** < FEW)

where,

ATFD (*Access to Foreign Data*): The number of distinct attributes the measured operation accesses.

LAA (*Locality of Attribute Accesses*): The relative number of attributes that the measured operation accesses on its class.

FDP (*Foreign Data Providers*): The number of classes where the accessed attributes belong to.

6.3.1.2. Misplaced Class (MC)

Misplaced Class (Fowler *et al.*, 1999) occurs when a class depends on classes from other packages more than on those from its own package. This anomaly is very similar to *Feature Envy* (Fowler *et al.*, 1999), but it occurs at the class level. Classes that strongly depend on external ones are likely to have harmful impact on the system architecture since they may introduce unwanted coupling between components as well as architectural anomalies, such as *Scattered Parasitic Functionality*. Thereby, it leads to unwanted ripple effects over different parts of the system architecture (Eick *et al.*, 2001; Knodel *et al.*, 2008).

In this context, the purpose of the architecture-sensitive strategy for *Misplaced Class* is to complement the conventional one in order to detect classes that depend more on external classes than on those defined in their component. To this end, the conventional detection strategy for *Misplaced Class* (Lanza and Marinescu, 2006) is enriched with architecture-sensitive metrics, such as *ACL*, *NEE* and *EFO*. These metrics quantify different kinds of information extracted from the relationships among external code elements.

MC<class> := (**CL** > LOW **or ACL** > LOW) **and**
(**NOED** > HIGH **or NEE** > FEW) **and**
(**EFO** > LOW)

where,

CL (*Class Locality*): The relative number of dependencies that a class has in its own package.

NOED (*Number Of External Dependencies*): The number of classes from other packages on which the measured class depends.

Furthermore, we proposed a second detection strategy for *Misplaced Class*. The new strategy exploits information regarding the architectural concerns a given class modularizes. It verifies whether the measured class modularizes more external concerns than those that are implemented in its enclosing component. If so, the measured class may introduce a scattered concern over components and, hence, it should be moved to another component. Note that this strategy could be defined in the scope of the previous one by using the OR logical operator. However, we opted for documenting it in an independent fashion to facilitate its understanding.

MC<class> := (**CoL** > LOW) **and** (**NAC** > LOW)

6.3.1.3. Long Method (LC)

Long Method (Fowler *et al.*, 1999) occurs when a method has grown too large. This kind of methods is difficult to reuse. However, certain methods need to be large due to their nature (e.g. transactional methods), but they do not affect the system architecture. In particular, long methods tend to adversely impact the software architecture when they modularize concerns that should be implemented by external code elements. (Fowler *et al.*, 1999). In critical cases, methods dealing

with many concerns can get out of control and make their enclosing component hard to maintain.

In this context, the purpose of the architecture-sensitive strategy for *Long Method* is to complement the conventional one in order to detect the architecturally-relevant occurrences. To this end, the strategy is not limited to quantify complex methods in terms of their size (i.e. *Lines Of Code*) and cyclomatic complexity (using the CYCLO clause) as the conventional strategy does (Lanza and Marinescu, 2006). Besides, the architecture-sensitive strategy analyzes the number of architectural concerns that the measured method modularizes (using the *NAC* clause). This strategy is based on the following hypothesis: the higher the number of architectural concerns a method implements, the higher the likelihood of that method to be dealing with a concern that should be modularized in another component is.

LM<method> := (**LOC** > HIGH **or CYCLO** > HIGH) **and** (**NAC** > LOW)

where,

CYCLO (*Cyclomatic Complexity*): The number of linearly independent paths through a measured method source code.

LOC (*Lines Of Code*): The number of lines of code the measured method contains.

6.3.1.4. God Class (GC)

God Class (Martin, 2002) occurs when a class centralizes the system functionalities. However, classes can be large and complex even when implementing a single concern. These classes are not likely to be indicators of deeper design problems (Olbrich *et al.*, 2010, Chapter 4). On the other hand, the existence of *God Classes* modularizing more than one concern may indicate that their enclosing component suffers from the *Component Concern Overload* anomaly. In addition, when these classes are implementing concerns that are also modularized by external elements, it may indicate the introduction of *Scattered Concern Functionality*.

In this context, the architecture-sensitive strategy for *God Class* is not limited to measure the complexity of a class in terms of number and complexity of its methods, as the conventional strategies does. The architecture-sensitive strategy verifies whether a complex class modularizes more than one architectural

concern (using the *NAC* clause). Additionally, it quantifies the cohesion of the class based on the architectural concerns shared by its methods (using *CoC* clause). This quantification is motivated because the *TCC* measure (Marinescu, 2004) used in the conventional strategy (Appendix B) has proved to be inaccurate when quantifying a class cohesion (Chae *et al.*, 2006). The fact that two methods use a common attribute does not necessarily imply that they modularize the same concern.

GC<class> := (**WMC** > MANY **and NAC** > LOW **and CoC** < LOW)

where,

WMC (*Weighted Method Count*): The sum of the complexity of all methods in a class.

6.3.2. Detection Strategies for Collaborative Anomalies

This section describes the architecture-sensitive strategies proposed to identify various collaborative anomalies. Specifically, the proposed strategies detect the following code anomalies: *Shotgun Surgery*, *Intensive Coupling*, and *Dispersed Coupling*. These code anomalies were chosen because they were good indicators of architectural degradation symptoms in previous studies (Chapter 4).

6.3.2.1. Shotgun Surgery (SS)

Shotgun Surgery occurs when a method or a class has many other code elements depending on it (Fowler *et al.*, 1999). The problem is that if a change in the infected code element occurs, other code elements might need to change as well. This scenario makes the software architecture harder to maintain when the required changes are scattered over different components (Eick *et al.*, 2001; Maccormack *et al.*, 2006). The reason is because changes may be required on components that modularize different concerns or have been implemented by different developers. Therefore, it is expected that these changes demand more time and effort to be performed than if they were concentrated in methods or classes belonging to the same component, thus increasing the likelihood of missing an important change.

The purpose of the architecture-sensitive strategy for *Shotgun Surgery* is to complement the conventional one in order to detect those occurrences that would significantly impact several parts of the system architecture. The conventional strategy for *Shotgun Surgery* (Lanza and Marinescu, 2006) does not ensure that kind of detection because it counts only the number of affected classes (using the *CC* clause) and methods (using the *CM* clause), without distinguishing whether such methods and classes belong to the same component or not (Chapter 5). The architecture-sensitive strategy is a step forward because it is not limited to only analyze methods as the conventional strategy does. The proposed strategy can be also applied to other code elements like classes. In addition, it quantifies the number of client components a given code element has (using the *EFI* clause). A high *EFI* value means that there are code elements defined in several components that depend on the measured code element.

Furthermore, the proposed strategy verifies whether the measured code element modularizes several architectural concerns (using the *NAC* clause). The motivation for this verification is that whenever the measured code element implements more architectural concerns, it is likely to be changed more often. Therefore, changes performed in the infected code element could be propagated to components that do not necessarily modularize the modified concern, affecting independent parts of the system architecture.

SS<element> := (**CM** >FEW **and CC** > FEW) **or** (**EFI** >LOW **and NAC** > LOW)

where,

element can be instantiated as a method or a class.

CM (*Changing Methods*): The number of methods that call the measured element.

CC (*Changing Classes*): The number of classes in which the methods that call the measured element are defined.

6.3.2.2. Intensive Coupling (IC)

Intensive Coupling refers to a class or method strongly depends on methods scattered in few classes (Lanza and Marinescu, 2006). The infected element is likely to be the source of architecture degradation symptoms when it depends on external methods (Godfrey and Lee, 2000; Eick *et al.*, 2001; van Gurp and Bosch, 2002; Maccormack *et al.*, 2006; Sarkar *et al.*, 2009b). The reason is that when

Intensive Coupling is introduced among components, it affects the reusability of these components and changes performed in one component might be propagated to the others.

In this context, the purpose of the architecture-sensitive strategy is to complement the conventional one in order to ensure the detection of *Intensive Coupling* at the architecture level. This kind of detection cannot be ensured by using the conventional strategy because it only considers the number of classes (using the *CINT* clause) and methods (using the *CDISP* clause) coupled to the measured method (Chapter 5). That is, the conventional strategy is limited to identify methods as instances of *Intensive Coupling* and does not verify whether the accessed elements: (i) are instances of other elements such as classes and (ii) belong to the same component. Unlike the conventional strategy, the architecture-sensitive one first verifies whether the measured code element, not limited to method, accesses many external code elements (using the *NEE* clause). Secondly, the proposed strategy checks whether the accessed code elements are defined in less than few components (using the *EFO* clause).

IC<element> := (**CINT** > SHORT **and NEE** > MANY) **and**
(**CDISP** < FEW **or EFO** < FEW)

where,

element can be instantiated as a method or a class.

CINT (*Coupling Intensity*): The number of distinct methods called by the measured element.

CDISP (*Coupling Dispersion*): The number of classes in which the called methods are defined is divided by CINT.

6.3.2.3. Dispersed Coupling

A class or a method suffers from *Dispersed Coupling* when it accesses many other classes or methods (Lanza and Marinescu, 2006). Classes or methods infected by *Dispersed Coupling* are likely to be changed due to different reasons (Lanza and Marinescu, 2006). However, not all the *Dispersed Coupling* instances impact the system architecture. Code elements that depend on several components impact the system architecture more than those code elements that only depend on local ones (Eick *et al.*, 2001; van Gurp and Bosch, 2002). The reason is because

the former code elements are likely to be associated with *Overused Interface* anomaly and thus, are targeted as a result of changes performed in different components (Chapter 4). Additionally, code elements that depend on several components are hard to be reused because several external elements must be referenced to as well. In extreme situations, *Dispersed Coupling* among components leads software systems to their complete redesign (Eick *et al.*, 2001; van Gurp and Bosch, 2002; Maccormack *et al.*, 2006; Sarkar *et al.*, 2009b).

In this context, the purpose of the architecture-sensitive strategy is to ensure the detection of *Disperse Coupling* at the architecture level. To this end, the architecture-sensitive strategy looks for classes and methods that access more than few external elements (using the *NEE* clause). Additionally, it verifies whether the accessed code elements belong to more than few components (using the *EFO* clause). Note that this detection is not supported by the conventional strategy (Lanza and Marinescu, 2006) because it only considers methods that access other methods (using the *CINT* clause) from many classes (using the *CDISP* clause). In other words, the conventional strategy: (i) is limited to consider only methods as instances of *Dispersed Coupling* and (ii) does not verify whether the accessed classes belong to different components.

DC<element> := (**CINT** > SHORT **or NEE** > FEW) **and**
(**CDISP** > FEW **or EFO** > FEW)

where,

element can be instantiated as a method or a class.

As it can be noticed, the strategy for *Dispersed Coupling* relies on the same metrics used by the strategy for *Intensive Coupling*. However, unlike the former, *Dispersed Coupling* uses the FEW constant associated with the measure *NEE* in the first clause. The reason is that such strategy does not need to enhance a strong coupling with external code elements. Additionally, *Dispersed Coupling* uses the "*greater than*" operator in the second clause because the strategy needs to ensure that the dependencies of the measured class or method are not concentrated in a few components.

6.4. Assessment of Architecture-Sensitive Strategies

This section describes a study conducted to evaluate the proposed architecture-sensitive detection strategies. Specifically, the study aims at partially answering the fourth and last research question of this work (Section 1.4): *To what extent leveraging architecture-sensitive information and inter-relationships among code anomalies improves the accuracy of conventional strategies when identifying architecturally-relevant code anomalies?* In this study, this research question was decomposed into three research questions (RQ):

RQ4.1: Can the proposed architecture-sensitive detection strategies accurately identify architecturally-relevant code anomalies?

RQ4.2: If so, to what extent each kind of architecture-sensitive information is useful in this process?

RQ4.3: To what extent the different granularity levels of architectural concern mappings influence the accuracy of architecture-sensitive detection strategies?

While there are many tools available to recover mappings between architectural components and code elements, just a few of them is devoted to recover the architectural concerns. In this context, the goal of *RQ4.2* is to understand the contribution of both kinds of architecture-sensitive information in the detection of architecturally-relevant code anomalies. This knowledge allows engineers to be aware of the amount of architecturally-relevant code anomalies that could be missed if a particular kind of architectural information is not leveraged in the detection process.

Concern mappings on the system implementation can be specified at a wide range of levels, from low-level (e.g. code statements) to high-level (e.g. whole packages). It is acknowledged that specifying concerns at method-level is much more time and resource consuming than at class-level. Thus, one could expect that architects and developers prefer to project concerns at class-level instead of at method-level in order to save effort. In this context, the goal of *RQ4.3* is to understand to what extent it is worth to invest efforts on projecting concerns at method-level in order to get an accurate identification of architecturally-relevant code anomalies.

In this context, we defined our study and its goals as:

Analyze: the proposed architecture-sensitive detection strategies

For the purpose of: evaluating their accuracy

With respect to: the identification of architecturally-relevant code anomalies

From the viewpoint of: systems architects, developers and researchers

In the context of: software systems from different domains and following different architectural decompositions.

6.4.1. Hypotheses

In order to answer the three aforementioned research questions, we have defined the null and alternative hypotheses as shown in Table 6.3.

Table 6.3: Research questions and hypotheses of the study.

Research Questions	Hypotheses
RQ4.1	For each of the architecture-sensitive strategies proposed: **Null Hypothesis, $H1_0$**: The architecture-sensitive strategy does not significantly enhance the accuracy of the conventional ones when detecting architecturally-relevant code anomalies. **Alternative Hypothesis, $H1_A$**: The architecture-sensitive strategy significantly enhances the accuracy of the conventional ones when detecting architecturally-relevant code anomalies.
RQ4.2	For each kind of the architecture-sensitive information used in the detection strategies proposed: **Null Hypothesis, $H2_0$**: The kind of architecture-sensitive information does not significantly increase the accuracy of the architecture-sensitive strategies. **Alternative Hypothesis, $H2_A$**: The kind of architecture-sensitive information significantly increases the accuracy of the architecture-sensitive strategies.
RQ4.3	**Null Hypothesis, $H3_0$**: The accuracy of the architecture-sensitive strategies is higher when using concerns mapping at method-level than at class-level. **Alternative Hypothesis, $H3_A$**: The accuracy of the architecture-sensitive strategies is not higher when using concerns mapping at method-level than at class-level.

6.4.2. Variable Selection

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

Independent Variables. In $H1_0$, there are as many independent variables as there are architecturally-relevant code anomalies detected by the proposed strategies. Each variable, $AS_{i,j}$, indicates the number of times that the proposed strategy i detects architecturally-relevant code anomalies in version v_j. As described in Section 6.4.4, all thresholds used when testing the architecture-sensitive detection strategies were confirmed by the architects and developers involved in this process.

In $H2_0$, there are two independent variables, $CP_{i,j}$ and $CN_{i,j}$, indicating the number of times the proposed strategy i detects architecturally-relevant code anomalies using only component and concern projections in version v_j, respectively. In $H3_0$, there are also two independent variables. $CM_{i,j}$ indicates the number of architecturally-relevant code anomalies detected by the strategy i using only concern mappings at the method-level in version v_j. Similarly, $CC_{i,j}$, indicates the number of architecturally-relevant code anomalies detected by the strategy i using only concern mappings at the class-level in version v_j. Section 6.4.4 describes how the concern projection tasks were conducted.

Dependent Variables. This study assesses the accuracy of the proposed strategies when detecting architecturally-relevant code anomalies. Therefore, there is only one Boolean dependent variable, $AR_{i,j}$, for all the null hypotheses, indicating whether the code element i is considered to be architecturally-relevant in version v_j. As described in Section 6.4.4, all the architecturally-relevant code anomalies used in testing these hypotheses were confirmed by the involved architects and developers of the target systems.

6.4.3.
Selection Criteria and Target Systems

Several criteria were established for selecting suitable software systems to this study. The criteria used are presented in Table 6.4.

Table 6.4: Criteria used for the selection of target systems.

	The target system:
C1	was modeled using documented guidelines or well-known architecture styles.
C2	has the intended architecture design available.
C3	has the architects and developers available.
C4	has a manageable size.
C5	is infected by a rich set of code anomalies.
C6	presents multiple symptoms of architectural degradation.
C7	underwent changes.
C8	was implemented by developers with different levels of programming skills.
C9	has architectural concerns implemented at different granularity levels.
C10	has architectural components structured in different groups of code elements.

As it can be noticed, criteria C1-C8 have been used in previous studies (Chapters 4 and 5). Similarly to these studies, we needed to ensure that the target systems were affected by code anomalies and architectural degradation symptoms. Criterion C9 was added to the criteria list because we wanted to observe whether the projection of concerns at different granularity levels affected the accuracy of the proposed strategies. Finally, criterion C10 allows analyzing the accuracy of the proposed strategies when components are implemented by different groups of code elements such as: a single package or a group of classes belonging to different packages.

Based on these criteria, five (05) software systems have been studied. Three of them are desktop applications implemented in Java that aimed at managing oil operations (e.g. production, stock and distribution). Since there are copyright constraints, the fictitious name of S1, S2 and S3 are used in this thesis to refer to them. The main characteristics of these systems are described in Appendix A. The fourth and fifth systems correspond to Health Watcher and MobileMedia, respectively. This study considers only systems implemented using object-oriented programming. Systems implemented using aspect-oriented programming and in accordance with the aforementioned criteria were not found available.

6.4.4. Procedures for Data Collection and Analysis

As the assessment of architecture-sensitive strategies relies on some systems used in previous studies (e.g. Health Watcher and MobileMedia), part of the information gathered in these studies was considered. Additionally, as shown below, we replicated some of the data collection tasks performed in these previous studies to assess the accuracy of architecture-sensitive strategies.

Recovering the Architecture Design. Similarly to previous studies, we counted on the help of architects and developers of S1, S2 and S3 in order to document their architecture design and get information about its correspondence with the system implementation. We spent around ten months in collaborative work with S1, S2 and S3 architects and developers in order to gather that detailed information. Finally, we produced the mappings of the architectural components on the code elements for each target system. In these systems, architectural components were implemented by different sets of code elements, such as groups of classes constituting a single package and classes that belong to different packages.

Recovering the Architectural Concerns. In this stage, for Health Watcher and MobileMedia, we considered the architectural concerns selected n previous studies. In the context of S1, S2 and S3 systems, we asked their architects and developers to select the architectural concerns according to the same criteria used in the previous studies. In other words, architectural concerns that: (i) are clearly relevant according to their knowledge of these systems, (ii) are involved in different important functionalities, (iii) present different scattering degree in the architectural design, and (iv) are projected in the source code at different granularity levels. For each system we counted on at least three architects and developers that had experience on maintaining it for more than four years. Additionally, two researchers were dedicated to support systems architects and developers in that task. All the involved people have deep knowledge on: (i) recovering and documenting system architectures and (ii) guidelines for modeling and implementing software systems.

In the study, we considered six architectural concerns for Health Watcher: Concurrency, Distribution, Persistence, Complaint, Health Unit, and View; nine

architectural concerns for MobileMedia: Security, Concurrency, Screen, Persistence, Photo, Music, Video, Sorting, and Favorite; five architectural concerns for S1: Exportation, Folder, Importation, Scenario, and User; five architectural concerns for S2: Logger, Notification, Route, Point, and Transaction; five architectural concerns for S3: Concurrency, Mixture, Product, Report, and Scenario. Descriptions of the architectural concerns documented for S1, S2 and S3 are provided in Table 6.5, whilst descriptions of concerns documented for Health Watcher and MobileMedia are given in Table 4.3.

Table 6.5: Architectural concerns considered in this study for S1, S2 and S3.

System	Architectural Concern	Description
S1	Export	Manages and defines the rules of products export.
	Folder	Manages and defines rules about how scenarios are stored.
	Importation	Manages and defines the product import rules.
	Scenario	Visualizes and groups the results of all operations (e.g. importation, exportation) in a given time window.
	User	Manages the user's access control, privacy and authentication.
S2	Logger	Saves information about program execution and/or errors.
	Notification	Defines a system notification to users (i.e. email).
	Route	Represents a route of products between two points in the logistics context.
	Point	Manages all the points in the system.
	Transaction	Stores and recovers data from the database and ensuring ACID properties.
S3	Concurrence	Provides a control for avoiding inconsistent information stores in the system database.
	Mixture	Manages all the compositions of products in the system.
	Product	Manages all the products in the system.
	Report	Represents the report exhibition, exportation and printing.
	Scenario	Visualizes and groups the results of all operations (e.g. importation, exportation) in a given time window.

Applying the Conventional Detection Strategies. The conventional detection strategies were only applied in S1, S2, and S3 systems, as we reused the results gathered for Health Watcher and MobileMedia in Chapter 5. In order to apply the conventional strategies in S1, S2 and S3, we followed the same procedures used in previous studies. In other words, we first selected the conventional detection strategies that were used in these studies. Then, we decided to gather the code metrics used in the strategies with well-known code analyzers (Sonar, 2009; Understand, 2009). The outcomes of these metrics were combined according to the strategy definition. Similarly to previous studies, we relied on

these code analyzers because they altogether collect all the required metrics. Regarding the threshold selection, we followed the methodology described in Section 4.1.4. Finally, all the used thresholds were validated with systems architects and developers and, are available in Appendix B.

Applying the Architecture-Sensitive Detection Strategies. The application of the architecture-sensitive detection strategies was supported by our tool, SCOOP (Section 7.7). This tool supports the collection of all proposed architecture-sensitive metrics (Section 6.2). Additionally, SCOOP offers a Domain-Specific Language, allowing engineers to tailor a set of specific thresholds according to the system characteristics. As mentioned before, the thresholds of the proposed metrics were first calibrated using guidelines reported in the literature. Additionally, different thresholds were employed in order to select those that presented the best accuracy rates, similarly to the conventional strategies (Section 4.1.4).

Identifying the Ground Truth of Architecturally-Relevant Code Anomalies. The ground truth of architecturally-relevant code anomalies was built to analyze the accuracy of architecture-sensitive strategies and, the conventional ones when identifying architecturally-relevant anomalies. In order to build the ground truth, architects, developers and researchers worked altogether, similarly to previous studies (Chapter 5). A phase was devoted to identify the architecture degradation symptoms in each target system. Additionally, a code review was performed to reveal the architecturally-relevant code anomalies in these systems. It took around six months to build the ground truth, considering a team of four people. We only considered in the ground truth those anomalous code elements whose impact on the architecture design was confirmed by all the architects and developers involved in this task.

Analyzing the Accuracy of Detection Strategies. This analysis was carried out similarly to its counterpart presented in Section 5.1.4. We measured the accuracy of the architecture-sensitive strategies by calculating precision and recall rates after providing a list of anomalous code elements candidates. The purpose of *precision* is to verify to what extent the proposed detection strategies are able to select only the code elements that harmfully impact the architecture. On the other hand, *recall* measures verify if the strategies are able to detect all these critical code elements. The precision and recall measures were computed

manually based on the list of code anomalies provided by the architecture-sensitive strategies and the ground truth identified by architects and developers. Equations detailing how the precision and recall rates were calculated can be found in Section 5.1.4. A similar process was conducted for analyzing the accuracy of the conventional detection strategies in S1, S2 and S3 systems.

6.4.5. Findings on Architecture-Sensitive Detection Strategies

This section discusses the main findings associated with the three research questions presented in Section 6.4. In particular, Section 6.4.5.1 discusses whether and to what extent architecture-sensitive strategies help engineers to detect architecturally-relevant code anomalies. Section 6.4.5.2 reports the contributions of each kind of architecture-sensitive information (i.e. architectural component and architectural concern projections) in the detection of architecturally-relevant code anomalies. Finally Section 6.4.5.5 analyzes the influence of projecting architectural concerns at different granularity levels on the accuracy of architecture-sensitive strategies.

6.4.5.1. Accurate Detection of Architecturally-Relevant Code Anomalies

Similarly to the assessment of the conventional detection strategies in Chapter 5, we considered that an architecture-sensitive strategy that detects code anomalies of type *T* achieves 100% of precision and 100% of recall if, and only if, it pinpoints all anomalies of type *T* identified in the ground truth. The precision and recall of both architecture-sensitive and conventional strategies on detecting architecturally-relevant code anomalies are presented in Table 6.4. The token '-' is used to represent the cases of code anomalies that did not occur in the target systems.

Table 6.6: Results for the architecture-sensitive detection strategies analyzed.

Code Anomaly	True Positives			False Positives			False Negatives			Precision			Recall		
	HW	MM	S1	HW	MM	S1	HW	MM	S1	HW	MM	S1	HW	MM	S1
Disperse Coupling	23	6	40	14	3	17	3	2	8	0.62	0.67	0.70	0.88	0.86	0.93
Feature Envy	16	7	34	14	4	9	4	2	14	0.52	0.64	0.79	0.80	0.78	0.71
God Class	5	6	26	1	2	12	0	2	8	0.83	0.75	0.68	1.00	0.75	0.76
Intensive Coupling	20	7	40	9	3	21	4	2	12	0.69	0.70	0.66	0.83	0.78	0.77
Long Method	41	12	42	21	7	22	0	5	10	0.57	0.63	0.66	1.00	0.71	0.81
Misplaced Class	3	3	-	2	1	-	0	0	-	0.60	0.75	-	1.00	1.00	-
Shotgun Surgery	11	7	19	7	3	8	4	2	5	0.61	0.70	0.70	0.73	0.78	0.79
	S2	S3		S2	S3		S2	S3		S2	S3		S2	S3	
Disperse Coupling	27	15		10	7		3	1		0.73	0.68		0.89	0.94	
Feature Envy	46	40		10	6		5	2		0.82	0.87		0.91	0.95	
God Class	75	72		12	20		13	15		0.86	0.78		0.84	0.83	
Intensive Coupling	25	42		7	19		2	9		0.76	0.69		0.91	0.82	
Long Method	44	48		15	14		22	20		0.74	0.77		0.67	0.71	
Misplaced Class	-	-		-			-	-		-	-		-	-	
Shotgun Surgery	20	26		15	14		6	4		0.57	0.65		0.77	0.87	

Table 6.7: Results for the conventional detection strategies analyzed.

Code Anomaly	True Positives			False Positives			False Negatives			Precision			Recall		
	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3
Disperse Coupling	31	12	8	82	32	10	17	18	9	0.27	0.27	0.44	0.65	0.40	0.48
Feature Envy	19	24	18	21	32	19	29	27	24	0.48	0.43	0.48	0.40	0.47	0.43
God Class	15	39	31	17	43	52	19	49	56	0.47	0.47	0.37	0.44	0.44	0.36
Intensive Coupling	30	18	26	66	7	44	21	9	25	0.31	0.22	0.37	0.59	0.66	0.51
Long Method	29	33	32	107	92	61	22	33	36	0.21	0.26	0.34	0.57	0.50	0.47
Misplaced Class	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Shotgun Surgery	10	20	14	10	25	13	24	15	16	0.51	0.44	0.52	0.42	0.56	0.46

Precision Analysis. In terms of precision, the results in Table 6.6 show that the architecture-sensitive strategies were highly accurate. They consistently detected architecturally-relevant code anomalies in all analyzed systems. In 87% of the cases, the precision of these strategies was higher than 60% when identifying architecturally-relevant anomalies. On the other hand, in 72% of the cases, the conventional strategies presented precision rates (much) lower than 45% (Table 6.7). Therefore, the architecture-sensitive strategies significantly improved the precision rates of conventional ones when identifying architecturally-relevant code anomalies. On average, the precision rate of the conventional strategies was enhanced in more than 35%. These results suggest that exploiting architecture-sensitive information to detect code anomalies would help engineers to save time when reviewing code that may harmfully impact the system architecture.

Recall Analysis. In terms of recall, Table 6.6 shows that the architecture-sensitive strategies were also accurate to detect architecturally-relevant anomalies in all target systems. Architecture-sensitive strategies achieved recall rates higher than 60% in 100% of the cases. On the other hand, conventional strategies (Table 6.7) presented recall rates lower than 50% in 72% of the cases. Therefore, the recall of the conventional strategies was improved in around 50%. We verified that this superiority of architecture-sensitive strategies was directly related to their ability to better identify those code elements relevant to the architecture decomposition.

We observed that exploiting architecture-sensitive information in the detection strategies had a stronger influence in the recall rates than in the precision ones. We suspect that this occurs because our strategies were built to not only detect architecturally-relevant code anomalies (Section 6.3). That is, they also detect code anomalies that can harm the system in other perspectives (e.g. fault-proneness)are . Therefore, the architecture-sensitive strategies were also able to better detect anomalous code elements that were not necessarily related to architectural degradation symptoms.

6.4.5.2. Impact of Architecture-Sensitive Information is Manifold

Once the precision and recall rates were analyzed, this study investigated the role of each kind of architecture-sensitive information in these rates.

6.4.5.3. Mappings of Architectural Components and Code Elements

First, we have analyzed the influence of architectural component mappings on the accuracy of the architecture-sensitive detection strategies when only considering the architecture-sensitive metrics. To this end, the accuracy of the strategies was analyzed when detecting code anomalies in architectural components that match and do not match the implementation-level packages. The goal was to contrast the accuracy improvement in cases where architectural component mappings matched or not the package structure in the system implementation. In particular, we considered the three collaborative code anomalies (Section 6.3.2) to conduct this investigation; i.e. *Shotgun Surgery*, *Disperse Coupling* and *Intensive Coupling*. These code anomalies were chosen because they are particularly harmful to the architecture design when their code structures introduce relationships between architectural components.

Our analysis revealed that none of the target systems presented a perfect match between its architectural components and implementation-level packages. This result confirmed the intuition that packages often do not correspond to components. Specifically, 64% of the architectural components (45 of 70) did not match at all the implementation-level packages. In these components, the architecturally-sensitive strategies significantly improved the accuracy rates of conventional strategies. Figure 6.3 illustrates this situation in the context of the precision for each analyzed code anomaly.

In particular, the precision improvement of the architecture-sensitive strategies in Figure 6.4 was caused by the fact that a single component grouped several packages. Certain code elements were classified as anomalous by the conventional strategies because they had relationships with elements defined in other packages. However, in several cases, all the involved packages belonged to the same component. Therefore, the relationships introduced by the anomalous element did not affect the system architecture. For example, the method

RouteService.recordArchs() in the S1 system was classified as *Disperse Coupling* by the conventional strategy. The reason is that the latter uses data from different packages such as routedb and archdb. However, RouteService.recordArchs() was correctly neglected by the architecture-sensitive strategy as the packages routedb, archdb and routeservice (where the method is defined) belonged to the same component RouteServer. Similar situations were often observed with the other code anomalies in all target systems.

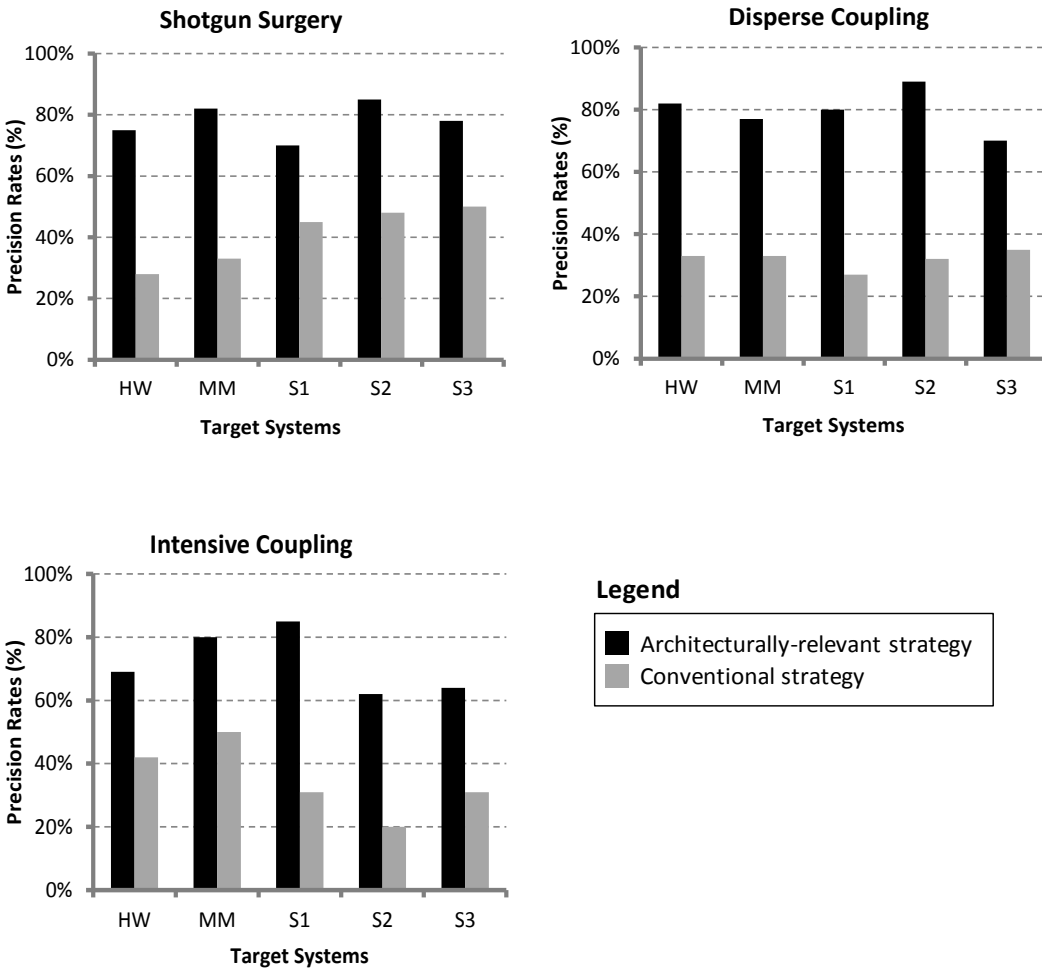

Figure 6.4: Precision rates (%) when components do not match packages.

Figure 6.5 depicts the recall rates of both conventional and architecture-sensitive strategies in the context of components that do not match packages. The recall improvement of the architecture-sensitive strategies in that figure was caused by the fact that several classes defined in the same package were mapped to different components. Several anomalous code elements were neglected by

conventional strategies because they introduced dependencies between classes in the same package. However, in many cases these classes were mapped to different components. Unlike the conventional strategies, the architecture-sensitive ones were able to distinguish these relationships as architecturally-relevant. For example, the method LoginAction.notifyUsers() in the S1 system was not classified as *Disperse Coupling* by the conventional strategy. The reason is that this method accesses data in the classes NotificationData and LoginLocator, which are defined in the same package client.action. However, LoginAction.notifyUsers() was correctly detected by the architecture-sensitive strategy as the classes NotificationData and LoginLocator belonged to different components Notification and Login, respectively. Similar situations were often observed with the other code anomalies in all target systems.

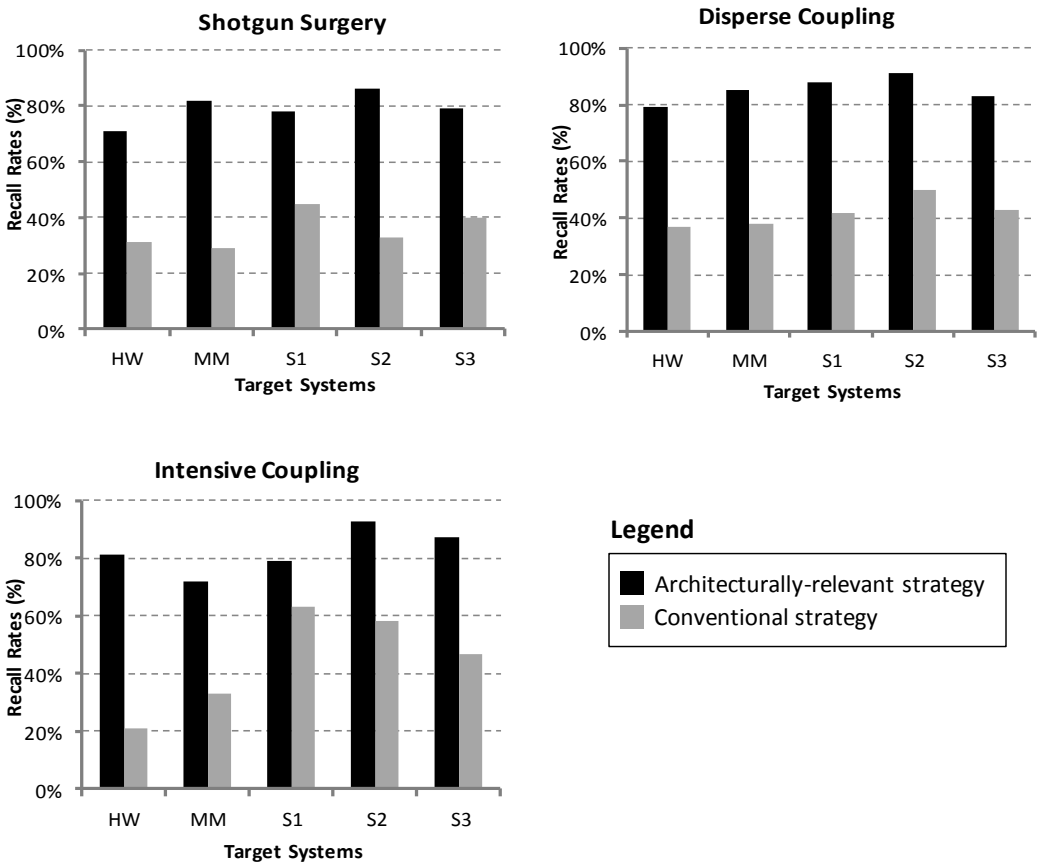

Figure 6.5: Recall rates (%) when components do not match packages.

As expected, architectural component metrics (Section 6.2.1) were more accurate than conventional code metrics to detect architecturally-relevant *Shotgun Surgeries*, *Disperse Couplings*, and *Intensive Couplings*. For example, more than 69% of architecturally-relevant *Shotgun Surgeries* were associated with tight coupling between components. Interesting cases emerge from analyzing other detection strategies. Unexpectedly, the concern metric *ACL* (Section 6.2.2) was not effective in the detection of some architecturally-relevant *Feature Envies*. This occurred because such relevant instances were more related to code elements defined in their own packages than to external code elements. This is a finding that should be tested in the future.

Finally, it was observed that the precision and recall rates between conventional and architecture-sensitive strategies did not significantly vary in components that perfectly match implementation-level packages. This means that architecture-sensitive information did not significantly improve the accuracy of conventional strategies in 36% (15 of 70) of the architectural components. On the other hand, this observation suggests that a great amount of architecturally-relevant anomalies would be missed if architecture-sensitive metrics (e.g. *NEE*, *EFO*, *EFI*) were not used in systems whose architectural decomposition did not correspond to the implementation packages structure.

6.4.5.4. Mappings of Architectural Concerns and Code Elements

We observed that some types of code anomalies benefit the most from concern-sensitive information. Architecture-sensitive strategies for *God Class* and *Misplaced Class* always presented precision and recall rates higher than 65%. A careful analysis of these occurrences revealed that they were the strongest indicators of architectural degradation symptoms in the target systems. On average, 79% of *God Classes* were related to architectural degradation symptoms, as well as 67% of *Misplaced Classes*. Architecture-sensitive strategies were able to detect on average more than 80% of these critical code elements. However, a significant proportion of these critical code elements, more than 50%, were not detected by any conventional strategy.

Complex classes whose methods access attributes in common, but realize different concerns (e.g. Controller in the MobileMedia system), were only detected as *God Class* by the architecture-sensitive strategy. One of the reasons is that the cohesion metric used by the architecture-sensitive detection strategy (*CoC*) is sensitive to the concerns that the class methods implement (Section 6.2.2). Additionally, the metric helped to identify other architecturally-relevant code anomalies like occurrences of *Feature Envy*. For instance, the Controler.showImageList() and Controller.handleCmd(..) methods contributed to the low cohesion of the Controller class. An analysis of these methods revealed that they accessed more data from external classes than from those defined in their own component to realize an external concern, confirming the *Feature Envy* nature.

In particular our results indicated that concern-based metrics (i.e. *CoC*, *NAC* and *CoL*) were the most effective on the detection of architecturally-relevant *God Classes*, *Long Methods* and *Misplaced Classes*. On average, 83% of architecturally-relevant *God Classes* were related to widely-scoped concerns as well as 78% of *Long Methods* and 60% of *Misplaced Class*. This finding shows that developers would not be able to accurately detect a significant amount of critical code elements without considering concern information.

Moreover, we observed that the code anomalies with highest detection accuracy were often caused by an inappropriate modularization of the architectural concerns. A deeper analysis of these anomalies revealed that they were indicators of multiple architectural problems. For instance, *God Classes* were related to *Connector Envy*, *Scattered Functionality*, *Component Concern Overload* and, *Overused Interface*; *Misplaced Classes* were related to *Cyclic Dependencies*, *Ambiguous Interfaces*, and *Scattered Functionality*.

It is important to highlight that strategies for detecting those anomalies classified in the "*Element Anomalies*" category were not the only ones that benefited from the architectural concern information. Such information also contributed to emphasize the architectural relevance of code anomalies classified in the "*Collaborative Anomalies*" category. For instance, around 16% of architecturally-relevant *Intensive Couplings*, 23% of architecturally-relevant *Disperse Couplings*, and 33% of architecturally-relevant *Shotgun Surgeries* were related to the inappropriate modularization of architectural concerns.

The results gathered in this evaluation suggest that mappings of architectural concerns would help engineers to accurately identify the most critical anomalous code elements to the architectural design. Therefore, engineers could rely on using mappings of architectural concerns to better identify which anomalous code elements should be refactored first.

6.4.5.5. Impact of Concern Granularity Level

This subsection analyzes the influence of mapping architectural concerns at different granularity levels (method-level and class-level) on the accuracy of architecture-sensitive strategies. To this end, we assessed the percentage of detected architecturally-relevant code anomalies when considering only architectural concerns projected at method- and class-level.

The results of this analysis are summarized in Figure 6.6. For each analyzed code anomaly, we show the accuracy of architecture-sensitive strategies when exploiting architectural concerns projected at method- and class-level in the target systems. It is important to note that we are only presenting in the figure the results regarding those strategies that exploit architectural concern information. Thus, architecture-sensitive strategies for detecting *Disperse Coupling* and *Intensive Coupling* are not analyzed in this context.

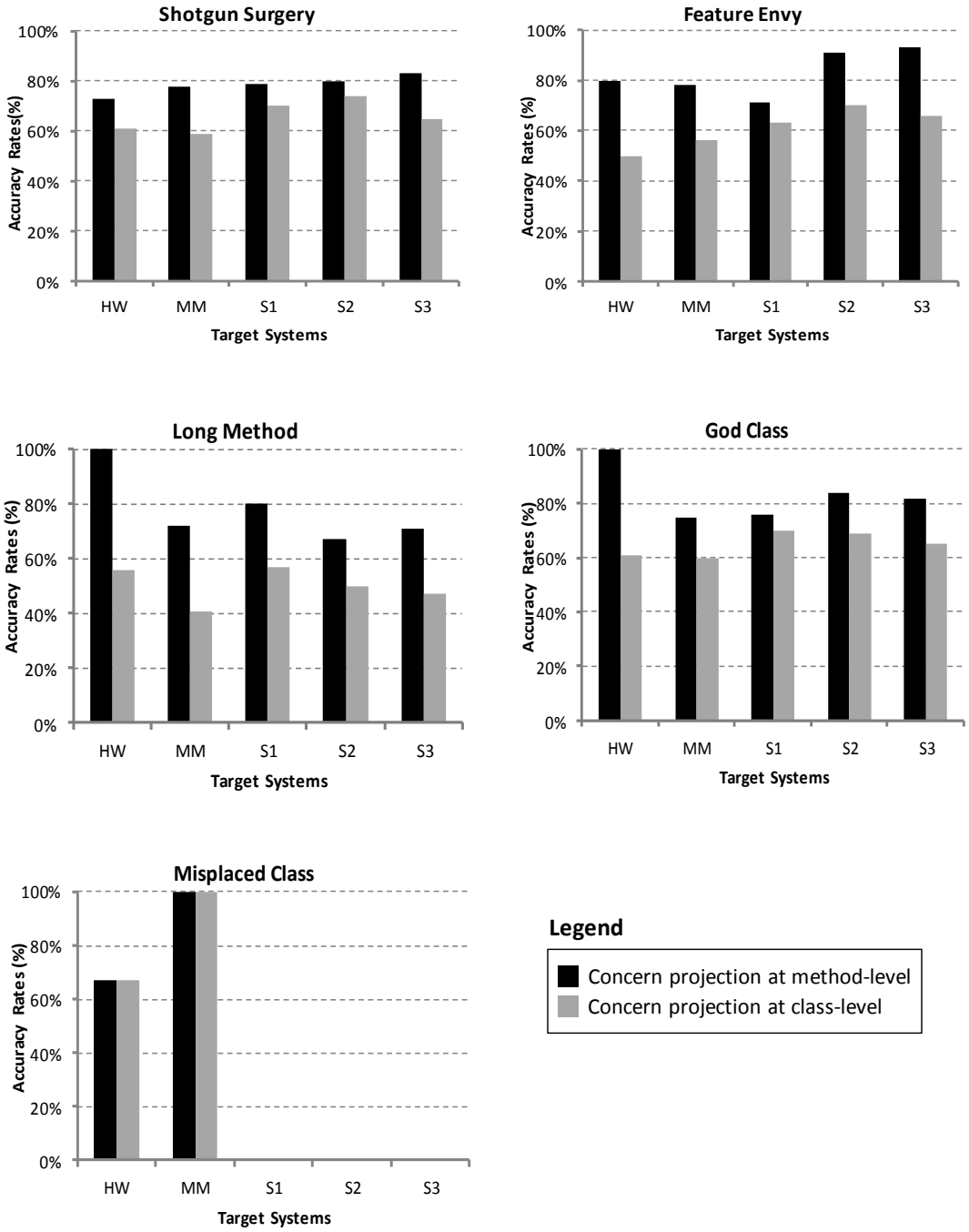

Figure 6.6: Accuracy rates (%) at different granularity levels of concerns.

Our results show that the detection of certain anomalies benefits the most from the projection of architectural concerns at the method-level. For instance, architecture-sensitive strategies that exploit concerns projected at the method-level were able to detect around 40% more *Long Methods*, 32% more *Feature Envies*, and 25% more *God Classes* than strategies that exploit concerns projected at the class-level. This occurred because such detection strategies consider detailed information at the method-level in order to identify anomalous code

structures. For instance, in order to detect *God Classes*, the detection strategy analyzes the class cohesion in terms of the number of common concerns its methods modularize. At first glance, these results might suggest that detection strategies exploiting solely concerns projection at the class-level could miss more than 20% of architecturally-relevant code anomalies.

However, an interesting finding emerges from analyzing the capability of architecture-sensitive strategies to detect occurrences of other code anomalies, besides those that the strategy is defined to detect. For instance, an analysis of the architecturally-relevant *God Class* detected by exploiting concerns at the class-level revealed that near to 35% of the architecturally-relevant *Long Methods* and around 15% of the *Feature Envies* manifested in these anomalous classes. A similar situation was observed for *Misplaced Class* and *Feature Envy* anomalies. In particular, 33% of the architecturally-relevant *Feature Envies* manifested in *Misplaced Classes*. As it can be noticed, this occurred because such code anomalies manifested simultaneously in these cases.

The above results are interesting because they highlight that certain anomalous structures might be indicators of other code anomalies. Even more interesting is the fact that projecting concerns at the class-level could benefit the detection of architecturally-relevant code anomalies that manifest at both class and method levels. In particular, detections strategies that consider solely concerns projected at the class-level would only miss around 17% of those code anomalies identified by exploiting the concern projection at method-level. This is a valuable result for the applicability of the architecture-sensitive strategies since the manual projection of concerns at class-level is much less time and resources consuming than at method-level. Additionally, it suggests that concerns automatically recovered at class-level could be used without significantly affecting the accuracy of the architecture-sensitive detection strategies.

6.4.6. Imperfections in the Detection of Architecturally-Relevant Anomalies

As shown in Table 6.6, the proposed strategies presented imperfections in the identification of the architecturally-relevant anomalies. In particular, around 18% of these code elements still remain undetected even after applying our

strategies. Our analysis indicated that the strategies imperfections are mainly related to three causes: (i) they were not designed to detect only architecturally-relevant code anomalies, (ii) inability to analyze relationships between code anomalies, and (iii) incompleteness and incorrectness of the list of architectural concerns used in the detection process. Other causes, such as the impact of the selected thresholds should be analyzed in the future.

As previously mentioned, the architecture-sensitive strategies detect code anomalies according to their definition, without focusing on a particular anomaly effect. Additionally, code anomalies might favor other maintainability problems (e.g. error-proneness) and are likely to be indicators of architectural degradation symptoms in later versions of the system.

Regarding the second cause, the process of employing architecture-sensitive detection strategies does not capture the relationship between anomalous code elements in order to better distinguish their impact on the architecture design. For instance, the detection of the *Redundant Interface* architectural anomaly requires the analysis of multiple component interfaces to identify common dependencies among them. Since a component interface may be mapped to different classes, the analysis of a single code element may not be accurate enough; i.e., this analysis does not distinguish whether it introduces (or not) redundant dependencies. In particular, it was surprising the proportion of code anomalies related to *Redundant Interface* detected in the target systems (66%). This occurred because such anomalous code elements were associated with code duplications and/or high coupling degree with external code elements. Similar situations occurred with anomalous code elements related to *Extraneous Connector* (Garcia *et al.*, 2009).

Regarding the third cause, several code anomalies were neglected by the architecture-sensitive strategies due to the incompleteness and incorrectness of the architectural concern mappings. Certain architectural concerns were incompletely projected in the systems S2 and S3 (e.g. conversion of data). Therefore, the architecture-sensitive metrics (Section 6.2) did not present high values. For instance, instances of *Long Method* in MobileMedia, S2 and S3 were not detected because the concern-based metrics reported that those elements were just realizing a system concern. Similar situations occurred with *Misplaced Class* in Health Watcher system, where the concern-based metrics did not reported that the class was dealing with concerns realized by other components. Moreover, in some

cases, the value of the concern-based metrics was affected due to mistakes made during the mappings stage. Some methods and attributes were not mapped to certain concerns mainly in concern overlapping cases. As a consequence of these mistakes, certain *God Classes* were not identified in the S2 system. An interesting observation is that architectural component-based metrics were not significantly affected by the projection imperfections. Less than 5% of anomalous code elements were overlooked due to incorrect values gathered by these metrics.

6.4.7. Threats to Validity

Threats to construct validity. A first threat concerns the way the ground truth of code anomalies was identified. We are aware that code anomalies might be accidentally related to architecture problems. However, we limited such threat by considering only the architecturally-relevant code anomalies whose impact on the architecture was confirmed by systems' developers and architects. Another threat concerns the application of detection strategies. We tried to mitigate this threat by involving several architects and developers in the selection of the thresholds. Lastly, construct validity was threatened by how concerns were selected and identified. We intentionally relied on an imperfect concern mapping sample, which presented 8% of mapping mistakes similarly to samples provided by existing feature recovery tools (Eisenbarth *et al.*, 2003; FEAT, 2011; Nguyen *et al.*, 2011). The reason is because concern mapping mistakes seem to be unavoidable even when the samples are provided by the system architects and developers (Nunes *et al.*, 2011).

Conclusion Validity. The number of evaluated systems and assessed anomalies threats the conclusion validity. Five systems from multiple domains, with different architecture decompositions and implemented by different teams were used. Evidently, 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 the architecturally-relevant anomalies identified by architects and developers. Thus, the sample can be seen as appropriate for a first exploratory investigation (Kitchenham *et al.*, 2006). The second issue is the completeness of code anomalies and architectural problems. We analyzed a number of code anomalies and architectural problems similarly to well-known studies (Maccormack *et al.*,

2006; Moha *et al.*, 2010; Olbrich *et al.*, 2010). In addition, certain code anomalies were not discussed (e.g. *Small Class*) since they were not good indicators of architectural degradation symptoms in previous studies (Chapter 4).

External Validity. The main threat to external validity is related to the nature of the evaluated systems. In order to minimize 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 contexts (i.e. academy and industry). However, we are aware that more studies involving a higher number of systems should be performed in the future.

6.5. Summary

By analyzing the state-of-the-art (Chapter 5), we verified that the conventional detection strategies are unable to accurately detect those code anomalies related to architectural degradation symptoms. One of the reasons is that these strategies do not exploit architecture-sensitive information in the anomaly detection process. In order to fill this gap, a suite of seven (07) architecture-sensitive metrics was proposed in Section 6.2. This suite of metrics relies on the proposed formalism (Section 6.1) and quantifies modularity properties (e.g. coupling and cohesion) of code elements based on the architectural components and concerns they modularize Additionally, a suite of eight (08) architecture-sensitive detection strategies was documented in Section 6.3. This suite combines conventional code metrics with the proposed architecture-sensitive ones to identify code anomalies.

The chapter also described a study performed to investigate to what extent the architecturally-sensitive strategies improve the accuracy of the conventional ones on the identification of architecturally-relevant code anomalies. The study involved a sample of nearly 1500 architecturally-relevant code anomalies distributed in five (05) industry software systems, which present different architectural degradation stages (Section 6.4.3). Our results confirmed that architecture-sensitive detection strategies enhance the accuracy rates of the conventional ones when identifying architecturally-relevant code anomalies. The results of such study indicated that using architecture-sensitive information in the

detection of code anomalies would allow engineers to be aware of more architecturally-relevant code anomalies. This means that engineers could promptly identify and address such anomalies upfront, avoiding advanced degradation stages in software projects. This result is even more relevant considering the inability of conventional detection strategies in the identification of these critical code anomalies (Chapter 5). The study also revealed that relying on packages is not an effective approach to detect architecturally-relevant code anomalies. This was often the case as components tend to not matching the package implementation boundaries. This finding raises the concern about the effectiveness of state-of-art mechanisms that are based on such approach (Bouwers *et al.*, 2011).

A natural concern when using the architecture-sensitive detection strategies is the cost to generate and maintain the required architecture-sensitive information. We did not carry out an investigation regarding the cost associated with generating that information. However, we used a sample of architecture-sensitive information with a correctness degree similar to those automatically generated using existing state-of-art tools (Eisenbarth *et al.*, 2003; Garcia *et al.*, 2011; Maqbool and Babri, 2007). Our results suggest that such tools could be used to generate the required information, without significantly impacting the accuracy of the detection strategies. This means that the required information could be generated without prohibitive costs. Of course this finding must be better verified in further studies.

Finally, the chapter discussed the imperfections of the proposed detection strategies. Part of these imperfections is caused by the inability of detection strategies to analyze relationships between code anomalies. In this sense, the next chapter (Chapter 7) will document a set of inter-related code anomalies that often indicate the presence of architectural degradation symptoms. Chapter 7 will also present SCOOP in detail, the tool used in the study to support the collection of the proposed architecturally-sensitive metrics and the automation of the architecturally-sensitive detection strategies.