5 Architecture Sensitive Heuristics for Prioritizing and Ranking Critical Code Anomalies

The previous chapter motivated the need for automating the prioritization and ranking of critical code anomalies. Due to the aforementioned reasons, it is not possible to rely on the *ad hoc* use of blueprints. In this chapter, we introduce the *architecture sensitive heuristics* proposed in this thesis. The *architecture sensitive heuristics* aim at assisting software developers when prioritizing and ranking critical code anomalies even in the early stages of the software development lifecycle. We have defined three groups of heuristics based on different architectural information related to the descriptive architecture, which can be exploited in the process of prioritizing and ranking critical code anomalies. By proposing heuristics, we are able to assist developers deciding which code anomalies should be refactored first according to the several drift problems observed in the descriptive architecture. The proposed heuristics consider different criteria to evaluate how critical code anomalies might be related to architecture degradation symptoms. Therefore, the heuristics exploits the combination of architecture blueprints and source code artifacts to better prioritize and rank potential candidates of critical code anomalies that should be refactored first. It is important to mention that the results of this chapter have been submitted to a premier conference on software modularity, with the participation of an international collaborator from *Drexel University*, who has also participated in order research results presented in Chapters 3 and 4 of this thesis.

In this sense, the heuristics provide means for assigning scores to each code anomaly based on different criteria, according to the heuristic under analysis (or even a combination of different heuristics). It is important to reinforce that the main goal on prioritizing and ranking critical code anomalies is to prevent architectural drift problems as early as possible during the system evolution. In this sense, aiming to propose and evaluate the architecture sensitive heuristics, we have initially defined a set of architectural information sources used in the process of prioritizing and ranking critical code anomalies:

Types of architecture blueprint. The architecture blueprints provided in our study represents the high level design of the descriptive architecture. Eventually, the use of architecture blueprints with different level of abstraction (e.g. class and component-connector diagrams) may also be considered in a way that the prioritization and ranking of critical code anomalies is not impaired. It is essential though, that the architecture blueprint reaches a minimal set of properties so that it can be used to guide the prioritization and ranking process. The quality of the architecture blueprints is granted by assessing the three properties (see Chapter 2).

Representing essential information on the architecture blueprint. The information represented on the architecture *blueprints* should allow software developers to: (ii) select architectural problems they want to focus on given the characteristics of the descriptive architecture represented for all software systems under investigation; (ii) minimize the effort regarding the time spent when prioritizing and ranking the most critical code anomalies related to architectural drift problems; and (iii) assist software developers when building strategies for prioritizing and ranking anomalies related with different architectural drift problems.

Representing architectural information. Notations and profiles available in the UML specification (OMG-UML, 2013) can be used to aggregate information in the architecture blueprints. Moreover, there are other ways for mapping system concerns in the source code by using (semi)-automatic tools, such as *ConcernMapper* (Robillard and Warr, 2005) – it allows the representation of architectural concerns in the source code. Therefore, it makes possible, for example, detecting classes responsible for implementing a given architectural component or which class is responsible for realizing one or more system concerns. Nevertheless, the mapping between source code and architectural elements facilitates developers' tasks when prioritizing and ranking critical code anomalies. For instance, when anomalous code elements are prioritized and ranked as critical to the architecture design, software developers can more easily investigate which architecture component each anomalous code element is responsible for realizing. In this sense, the prioritization and ranking process focuses mainly on architectural drift problems occurring in those architectural components and interfaces.

5.1. Criteria Selection and Relation to Architectural Problems

Several factors might indicate to what extent a code anomaly is harmful to the system's descriptive architecture. As previously mentioned in Chapter 2, differently from existing approaches our work focuses specifically on architectural drift problems. The reason is that architectural drift problems impair the adaptability in the software architecture, and therefore, its evolution. In addition, architectural drift symptoms are normally cause by: (i) applying a solution within an inappropriate context; and (ii) applying abstractions in the architecture design with an incorrect level of granularity. For instance, we can mention as architecture drift problem the implementation of a class that provides a lot of methods and implements many architectural concerns. This class might suffer from the code anomaly *God Class*, and violates the design principles: *Single Responsibility Principle* and *Separation of Concerns*.

In addition, the architectural component realized by this class might also be suffering from the architecture drift problems known as *Scattered Parasitic Functionality* and *Concern Component Overload*. As possible solution, this class must be decomposed in many classes, where each of them implements a specific architectural concern. It is important to mention that this type of refactoring operation may require changes in interfaces and architectural components. In this sense, we briefly describe 4 design principles commonly adopted by software architects when modeling the architectural design. These design principles might also be violated when architecture drift problems (see Chapter 2) occurs in the descriptive architecture. In this sense, the architecture sensitive heuristics proposed in this thesis investigate situations where one or more design principles are violated characterizing *architectural drift symptoms*. In the following we briefly describe each of the design principles that might be related to architectural drift problems we not properly addressed in the system architecture.

Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules (Martin, 2003). Both should depend on abstractions. In addition, abstractions should not depend on details, but the details should depend on abstractions.

Interface Segregation Principle (ISP) states that client classes are not forced to depend on interfaces that they do not use (Martin, 2003). When those classes are forced to depend of such interfaces, they are subject to changes performed in the interfaces. As result, we can observe an inadvertent coupling between all the classes in the client. Therefore, the coupling should be avoided and the interfaces should be separated whenever it is possible.

Single Responsibility Principle (SRP) states that there must be no more than one reason to change a class. The importance of separating responsibilities in distinguished classes is due to the fact that each responsibility is an axis of change (Martin, 2003). Therefore, when requirements change, the modifications will manifest through the modification in the responsibility between classes. If a class assumes more than a single responsibility, those responsibilities become coupled. Changes in a responsibility may impair the ability of a class to satisfy other requirements. This type of coupling can lead to a poor design that might break unexpectedly when modified.

Separation of Concerns Principle (SoC) states that a given problem involve different concerns, which should be identified and separated to deal with its complexity (Kiczales *et al.*, 1997), and achieve the factors required to the software quality (e.g. maintainability and reusability). This principle can be applied in many ways, and one might say that the separation of concerns is a ubiquitous principle of software engineering. In addition, the separation of concerns may bring benefits to the software quality properties, such as: (i) facilitate the system reusability; (ii) guarantee the system maintainability; (iii) allow developers to work in independent modules in a software system; and (iv) allow new functionalities to be easily added to existing software.

5.2.
Study Settings

As mentioned in Chapter 2, existing techniques do not provide developers with means for prioritizing and ranking critical code anomalies according to their architectural relevance. All the existing prioritization and ranking techniques are limited to the use of source code analysis (Section 2.1) and, as a consequence, they fail to identify code anomalies hat might be associated to architectural problems

(Garcia *et al.*, 2009a)(Macia *et al.* 2012a)(Macia *et al.*, 2012b). Many cases of software projects, as reported in the literature, resulted in partial or full discontinuation due to degradation problems on their descriptive architecture (Hochstein and Lindvall, 2005). Hence, there is a need for solutions that assist developers in anticipating the emergence of more severe architectural problems by avoiding drift symptoms in their programs.

The use of architecture blueprints, representing information about the system's descriptive architecture, is a promising direction for assisting developers when prioritizing and ranking critical code anomalies. Recent empirical study in several companies revealed that high-level design blueprints or sketches are widely archived and analyzed in industry projects (Baltes and Diehl, 2014). Similarly to our previous studies (see Chapter 3 and 4), we define an auxiliary research question to be addressed in our investigation as well as the study hypotheses. The research auxiliary question (ARQ) is defined as:

- **ARQ^5** - *To what extent critical code anomalies are accurately prioritized and ranked with the heuristics based on descriptive architecture information provided in blueprints?*

The expectation is that the proposed heuristics can assist developers when prioritizing critical code anomalies in the early stages of system development – therefore avoiding architectural degradation. However, the answer to this research question is far from being obvious. Architecture blueprints are very often high-level, incomplete and inconsistent with respect to the descriptive architecture. As a consequence, their use in the heuristics can lead to inaccurate prioritization and ranking results. In order to make clearer the purpose of the study, **Table 15** defines it using the GQM methodology (Basili *et al.,* 1994).

Our study has been organized in 4 different phases: (i) first, we performed the mapping between architectural elements presented in the blueprints and the elements in the system implementation. In this phase, we also evaluate if the architecture design models satisfy the properties defined (see Chapter 2) in order to be characterized as architecture blueprint; (ii) we validate instances of critical code anomalies detected for each target application based on the detection strategies compiled in (Macia *et al.*, 2013); (iii) we have applied the architecture sensitive heuristics for prioritizing and ranking critical code anomalies, and the score was computed for each anomaly given the criteria specified by each heuristic; and (iv) finally, we evaluated the results produced by the heuristics in terms of their accuracy on the process of prioritizing and ranking critical code anomalies. In order the compare the results achieved by the heuristics, we have considered *ground truth* (or reference list of the most critical code anomalies) previously defined by the software developers of each target application.

Table 15 - Study Definition using GQM format

GQM (Goal, Question, Metric)	
Analyze:	The *blueprint*-based prioritization heuristics
For the purpose of:	Evaluating their accuracy for prioritizing code anomalies
With respect to:	Prioritizing anomalous code elements based on inter-connected code anomalies
From the viewpoint of:	Researchers and developers
In the context of:	Two software systems from different domains with different architectural designs

It is important to mention we have proposed and tested two sets of heuristics according to the study hypotheses summarized in **Table 16**. Firstly, we evaluated heuristics for prioritizing and ranking critical code anomalies that affect the communication of two or more architectural components (Section 5.3.1). Secondly, we evaluated the heuristics for prioritizing and ranking critical anomalies related to problems on the implementation of system concerns (Section 5.3.1).

Table 16 - Study Hypothesis for Evaluating the Heuristics

Hypothesis	**Description**
Hypothesis $H_{1.0}$	Inter-component heuristics cannot help developers on prioritizing critical code anomalies.
Hypothesis $H_{1.1}$	Inter-component heuristics can help developers on prioritizing critical code anomalies.
Hypothesis $H_{2.0}$	Architectural concern heuristics cannot help developers on prioritizing critical code anomalies.
Hypothesis $H_{2.1}$	Architectural concern heuristics can accurately identify critical code anomalies.

Our goal is to evaluate the accuracy of the proposed heuristics for prioritizing and ranking code anomalies based on their architectural relevance. To analyze how the heuristics performed in terms of their accuracy for prioritizing and ranking critical code anomalies, we defined three different values: low (0-30%), acceptable (30-80%) and high (80-100%). The thresholds have been similarly adopted in experimentations in software engineering (Wohlin *et al.*, 2000). Therefore, we analyzed the three levels of accuracy to investigate to what extent the proposed heuristics would be effectively assist developers when helpful when prioritizing and ranking code anomalies. For instance, an accuracy level of 50% means that a given heuristic was able to correctly prioritize and rank at least half of all instances of the critical code anomalies.

5.3. Heuristics for Prioritizing and Ranking Critical Code Anomalies

In this section, we proposed 4 different heuristics to assist developers when prioritizing and ranking critical code anomalies based on information regarding the system's descriptive architecture. The proposed architecture sensitive heuristics are organized into two groups based on the main information used according to different criteria adopted on the prioritization and ranking process: (i) inter-component heuristics; and (ii) concern-based heuristics. Each architecture sensitive heuristics is described in more details in the following sections.

5.3.1. Inter-Component Heuristics

Our first set of architecture sensitive heuristics exploit information related with code anomaly affecting the communication between architectural components. The investigation of co-occurrences of code anomaly is related with effects on software maintenance, seeing that those code anomalies can be spread through many architectural elements. Occurrences of such code anomalies can be related with the violation of the *Interface Segregation Principle* and *Single Responsibility Principle* (Martin, 2003). Aiming to prioritize and ranking anomalous code elements according to their architectural relevance, the software artifacts required as input data for these heuristics are: (i) a set of source code artifacts, including software metrics (Marinescu, 2004)(Lanza and Marinescu, 2006); (ii) architecture *blueprints* representing information about the descriptive architecture of the software system (e.g. components, interfaces and concerns); and (iii) mapping between artifacts of both level of abstraction. It is important to mention that before applying the prioritization heuristics, the mapping between architecture and source code elements have already been performed.

5.3.1.1. Heuristic Based External Attractor Component

Our first heuristic investigates occurrences of code elements implementing a given architectural component, whose provided interface is used by many code elements belonging to external architectural components. This situation characterizes an occurrence of the *External Attractor Component*. A code element is considered to be external if it is located in other architectural component than the one under assessment. Moreover, the heuristic helps developers on the identification of architectural components that have, for example, an *Overused Interface* (Garcia *et al.*, 2009) that is responsible for realizing different concerns in the system. Code elements realizing the overused (provided) interface are accessed by the client code.

The problem is that code elements accessed by external code elements, which are responsible for realizing different architectural components, might favor the insertion of code anomalies in the latter. When an accessed code element implements many concerns, its client components are forced to deal with concerns they are not interested. Moreover, it is important to mention that when situation occurs, the *Interface Separation Principle* (Martin, 2003) is neglected, and hence, the internal complexity of the architectural component is increased. Consequently, the maintainability of the component's provided interface used by other external components decreases, and whenever a code element in the interface needs to be changed, the client's component might also be updated. In the following, we provide a formal definition for identifying occurrences of the *External Attractor Component*.

Formal definition. The set of occurrences of the *External Attractor Component (*EAt*)* in a system S is denoted by EAt_S. Considering an architectural component $AC_1 \in AC_S$ *(*set of architectural components in the system S*)*, a code element $CE_1 \in AC_{AC1}$ (set of architectural components realized by anomalous code elements), a set of architectural components $AC_2 \in AC_S$ and a set of code elements $CE_2 \in AC_{CE}2$, the formal definition of EAT_S is:

- $EAT_S = \{CE_1 \cup CE_2 \mid (CE_1, CE_2) \in DCE_1,CE_2 \wedge |CE_1| > th1 \wedge |AC_1| > th2\}$ where:
- DCE_1,CE_2 represents a dependency from the Code element CE1 to the code element CE.

It is important to mention that the generic thresholds *th1* and *th2* can be chosen depending on the characteristics of the software system under analysis and the design decisions defined by the software architect.

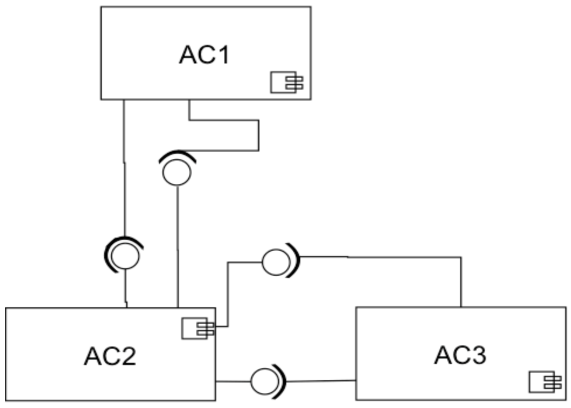

Figure 6 –Scenario for External Attractor Component

Abstract Example. An abstract representation an *External Attractor Component* is depicted in **Figure 6**. In this example, three architectural components are defined in the architecture blueprint. The architectural component AC_2 have an anomalous code element CE_2 that is accessed by other code elements belonging to the external architectural components 3 external components AC_1 and AC_3. That is, the interface provided by the architectural component AC_2 is being overused by other architectural components. In this scenario, a code element C_2 defined in the architectural component AC_2 might be affected by the Overused Interface since it methods are called by many classes. Additionally, it neglects the Single Responsibility Principle. In particular, the methods provided through the interface defined in this component are called by different client classes, which might indicate the inappropriate declaration of those methods. The code elements realizing the components AC_1 and AC_3, may suffer from the Long Method anomaly since they implement different types information that are propagated from code elements defined in AC_2. Furthermore, those code elements might also be affected by several changes due to modifications performed in the CE_2.

5.3.1.2. Heuristic Based on External Addictor Component

Our second heuristic investigates the occurrence of code elements implementing a given architectural component that has lots of dependencies with code elements realizing external components. This situation characterizes an occurrence of an *External Addictor Component*, which is associated with the violation of the *Single Responsibility Principle* and *Interface Segregation Principle*. A code element is considered to be external if it is located in other architectural component than the one under evaluation. In Addition, the heuristic aims at assisting developers to better detect architectural problems where anomalous code elements depend on several code elements realizing external components functionalities. For example, in some cases code elements might be affected as consequence of several changes in other external components, which might lead to many side effects. Those cases situation may also indicate a tight coupling between the anomalous code element realizing a given architectural component, which centralizes the communication between its own component and the adjacent ones. Moreover, side effects and tight coupling between architecture components has been recognized as source of reengineering of software systems and frequently impact on its discontinuation (MacCormak *et al*, 2006).

Formal definition. The set of occurrences of the *External Addictor* (EAd) in a system S is denoted by EAd_S. Considering an architectural component $AC_1 \in ACS$ (set of architectural components in the system S), a code element $CE_1 \in AC_{AC1}$ (set of architectural components affected), a set of architectural components $AC_2 \in AC_S$ and a set of code elements $CE_2 \in AC_{CE}2$, the formal definition of EAd_S is:

- $EAT_S = \{CE_1 \cup CE_2 \mid (CE_1, CE_2) \in DCE_1,CE_2 \wedge |CE_2| > th_1 \wedge |AC_2| > th_2\}$, where:

DCE_1,CE_2 represents a dependency from the Code element CE1 to the code element CE. Similarly to the definition of the *External Addictor Component, th1* and *th2* represent thresholds that can be chosen depending on the characteristics of the software system under analysis and the design decisions defined by the software architect.

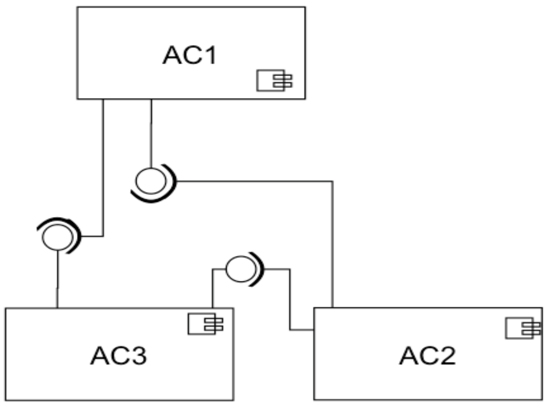

Figure 7 – Scenario for External Addictor Component

Abstract Example. An abstract example characterizing an *External Addictor Component* (Ead) is illustrated in **Figure 7**. Let us assume that the architectural component AC_2 have an anomalous code element CE_2, which access several classes belonging to the other 2 external components AC_1 and AC_3. Thus, the architectural component AC_2 is more interested in accessing the interfaces provided by other components (AC_1 and AC_3) than realizing the functionality it was initially designed to accomplish. We should also assume that architectural components AC_1 and AC_3 are implemented by 2 anomalous code elements CE_5, CE_6, CE_7, and CE_8, respectively and those code elements are infected with *DataClass* anomaly (Fowler *et al.*, 1999). In addition, the code element CE_2 is infected by a *God Class* anomaly (Fowler *et al.*, 1999), and hence, it defines several non-cohesive methods as well as it implements much functionality in this component. In this way, the CE_2 class might propagate several concerns that should only be treated internally. The propagation forces the code element CE_2 to deal with those concerns that were not properly addressed.

5.3.2. Concern-Based Heuristics

Our second set of heuristics investigates the anomalous implementation of system concerns in relation to the descriptive architecture. A system concern (Banker *et al.*, 1989) is defined as an architect interest that significantly influences the system`s descriptive architecture. Therefore, the concern-based heuristics aim at investigating the sources of critical code anomalies that violate the principle of *Separation of Concerns* (SoC) (Kickzales *et* al., 1997). The detection architecture drift problems related to the separation of concerns once those problems can hinder the maintenance of the descriptive architecture as the system evolves. When a component violates the principle of Separation of Concerns, its maintenance may also be impaired since it is responsible for the implementing several architectural concerns. Moreover, cases where the code elements realize many architectural concerns might lead to tight coupling between the components specified in the descriptive architecture represented in the blueprint. The tight coupling between those architectural components is a factor that delimitate the prioritization and ranking of specific anomalous code elements.

5.3.2.1. Heuristic for Concern Overload

Our fourth heuristic aims at identifying architectural components realizing many different concerns. That is, the heuristic detect instances of anomalous code elements realizing the same architectural component, which is responsible for the modularization of several independent concerns. The system concerns are considered independent when each of them could be modularized by different architectural components. In this scenario, we can observe the violation of the principle of *Separation of Concerns* (Kickzales *et al.*, 1997) and *Single Responsibility Principle* (Martin, 2003). Moreover, it is important to highlight that if an architectural component implements several concerns in a software system, it centralizes more than it should actually implement, and therefore, its maintainability can be compromised. Before applying the heuristic, all concerns represented in the descriptive architecture should be identified and validated with the system architect. After that, the mapping between architectural components and code elements responsible for realizing a system concern is performed. Once anomalous code elements are identified for each architectural component, we verify whether the number of concerns modularized by an architectural component, as well as if the number of classes existing in a given architectural component respect the thresholds defined by the system architect. If the architecture component does not respect the thresholds, the anomalous code elements within the architecture component must be identified as an occurrence of concern overload in this architectural component. The verification is performed for all the existing components represented in the architecture blueprint.

Formal Definition. The set of occurrences of *Concern Overload* (CO) in a system S is denoted by CO_S. Before formally defining the CO_S, we need to identify the list of concerns realized by each code element in a software system. Therefore, we define a list of all concerns (CO_S) realized by a code element $CE_{1,c} \in CE_S$ (where CEs represents all the code elements realized in the system S). In this sense, considering an architectural component $AC_1 \in AC_S$ (set of architectural components in the system S) and a code element $CE_1 \in CE_S$, the forma definition for occurrences of CO is represented as:

- $CO_S = \{CE_1 \mid CE_1 \in CE_{ACx} \wedge | CO_S(CE_1) | > th1 \wedge | CE1 | > th2\}$, where*:*
- CE_{ACx} represents all the anomalous code elements realizing a given architectural component AC_x.
- The threshold *th1* represents the maximum number of concerns that a given code element should realize, while the threshold *th2* represents the number of code elements that should be considered in an occurrence of *Concern Overload*.

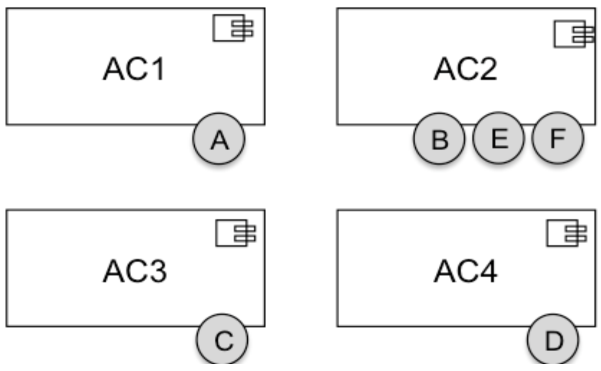

Figure 8 - Scenario for Misplaced Concern

Abstract Example. An abstract example of the *Concern Overload* is depicted in **Figure 8**. As we can observe the architectural component is realized by four different classes. The architectural components AC_1, AC_3, and AC_4 are responsible for three different concerns in the software system. In turn, the AC_2 modularizes 3 different concerns. Thus, code elements realizing the component AC_2 will have to deal with several concerns, and therefore, this architectural component violates both the *Separation of Concerns* and *Single Responsibility Principle*.

5.3.2.2. Heuristic for Misplaced Concern

Finally, our fourth heuristic investigates the occurrence of anomalous code elements that realize the same architectural component and modularize several independent concerns. The concerns are considered independent when each of them could be modularized by different architectural components. When this situation occurs, it is characterized as a *Misplaced Concern*. Instances of *Misplaced Concerns* violate either *Separation of Concerns* or *Single Responsibility* principles. However, the problem can be more severe when an inter-related architectural component also modularizes a misplaced concern. The fact is that dispersed anomalous code elements favor scattering for a concern in the architectural design. Moreover, violations to the separation of concerns often affect the system maintainability since changes in specific concerns can spread over many other components.

Formal Definition. The set of occurrences of *Misplaced Concern* (MC) in a system S is denoted by MC_S. Consider a concern $CO_A \in CO_S$ (represents the list of all concerns realized in a system S*)*, and two different architectural components $AC_1 \in AC_S$ and $AC_2 \in AC_S$ (set of all architecture components in a system S). The formal definition for occurrences of MC in system S is represented as:

- $MCS = \{CE_l \mid CE_l \in ACE_{AC1,COa} |CE_{AC1,COa}| < th1 \wedge |CE_{AC2,COa}| > th2\}$, where:
- $CE_{AC1,COa}$ represents a code element responsible for realizing the architectural component AC_1 and implementing the system concern CO_a.
- $CE_{AC2,COa}$ represents a code element responsible for realizing the architectural component AC_2 and implementing the system concern CO_a.
- The thresholds *th1* and *th2* represents respectively the number of concerns realized in a software system, the maximum number of concerns a given code element should realize. In summary, those values indicate the acceptable measures of which the system concerns are scattered. In this sense, the thresholds th_1 and th_2 must respect the following values $0 \leq th1 \leq 1$ and $0 \leq th2 \leq 1$.

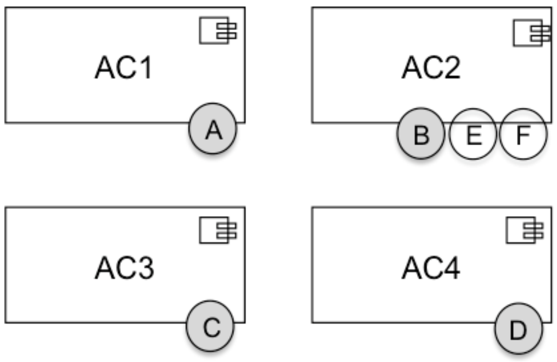

Figure 9 - Scenario for Misplaced Concern

Abstract Example. An abstract example where a component suffers from *Misplaced Concern* is depicted in **Figure 9**. As we can observe, code elements realizing the architectural components AC_1, AC_3 and AC_4 are responsible to deal with only one specific concern that is predominant for each of these architectural components. On the other hand the architectural component AC_2 should implement only on specific concern. However, code elements realizing this architectural component might also have to deal with concerns *E* and *F*, which are not predominant in this component. In other words, the architectural component AC_2 is implementing two concerns whose responsibility should be implemented by other components. Therefore, this situation characterizes an occurrence of an architectural problem related with misplaced concerns in software systems.

5.4. Research Findings on Prioritizing and Ranking Critical Code Anomalies

This section presents the evaluation of the proposed architecture-sensitive heuristics for assisting developers when prioritizing and ranking critical code anomalies. Before evaluating the heuristics, we describe how the detection of individual code anomalies was performed. We also describe the evaluation method used to compare the results and the *ground truth* provided for each target application.

5.4.1. Procedures for Data Collection and Evaluation Method

The first step before evaluating the proposed heuristics was the automatic detection of code anomalies for each target application. We have used well-known detection strategies and thresholds defined in other studies (Macia *et al.*, 2012a)(Macia *et al.*, 2012b). It is also important to mention that we identified types of code anomalies already catalogued in the literature and extensively investigated in other studies (Arcoverde *et al.*, 2012) (Macia *et al.*, 2012a)(Macia *et al.*, 2012b). In this sense, we included the ten most critical code anomalies found in the three target applications, namely: *Divergent Change*, *Shotgun Surgery*, *Middle Man*, *Duplicated Code*, *God Class*, *Small Class*, *Feature Envy*, *Data Class*, *Large Class*, and *Deep Inheritance Tree*. Those code anomalies are documented in different catalogues – i.e. (Fowler *et al.*, 1999)(Piveta *et al.*, 2006)(Marinescu, 2004).

After that, the list of code anomalies was checked and validated by developers and architects of each target application. The validation process is important to guarantee that the detection strategies detected instances of critical code anomalies. Furthermore, we also obtained the *ground truth* to compare results with the top-*N* code anomalies prioritized and ranked by the proposed heuristics. The *ground truth* is a list of the most critical anomalous code elements (according to their architectural relevance) provided by developers and maintainers of each target application. For the Mobile Media, the architects provided the top 10 code elements that they believed to represent the main sources of maintainability problems along the software project history. The architects of the Health Watcher system provided a list with the top 30 code elements that exhibited maintainability problems along its evolution history. Finally, developers from the SubscriberDB system provided a list with the top 15 anomalous code elements considering the impact on the system evolution.

The anomalous code elements listed by the developers as the most critical ones that should be urgently refactored so that severe maintainability problems could be avoided. In summary, the architects and developers of the target applications were asked to reason about the most important and critical classes. For instance, the critical code elements are those responsible for realizing the more important architectural components. In addition, they can also be those code elements s responsible for implementing the key provided and required interfaces.

After the architecture sensitive heuristics have been applied, we compared the list of the most critical code anomalies prioritized and ranked by each heuristic with the *ground truth* provided for each *target application*. The main reasons for performing the comparison analysis are: (i) if we asked developers to produce a ranked list containing all the code elements that could impact on the architecture design, our analysis would be unviable; and (ii) we wanted to evaluate our heuristics in terms of the most critical anomalous code elements. Those code elements should be properly refactored in the early stages of the system development, otherwise deeper maintainability problems are likely to be manifested during the system evolution.

Aiming to analyze set of code elements prioritized and ranked by the architecture sensitive heuristics, we have used the *Size of Overlap* between the different prioritization lists. Calculating this measures is very straightforward, and it allows evaluating whether the heuristics could accurately distinguish the top *k* for each target application. The size of overlap indicates the accuracy of the proposed heuristic when prioritizing and ranking the most relevant code elements according to their architectural relevance.

Table 17 - Additional Architecture Sensitive Metrics

Metric	**Description**
Density of Code Anomalies (DCA)	It calculates the density of code anomalies in a code element realizing a given component.
External Dependencies (ED)	It counts the number of external code elements that a given code elements depends.
Concerns per Code Element (CE)	It counts the number of architectural concerns a measure code element implements
Number of Concerns per Architecture Element	It counts the number of architectural concerns a measure architecture component realizes
Concern Diffusion over Components	It counts the number of code elements affected by the implementation of an architectural concern
Concern Diffusion over Operations	It counts the number of methods and constructors affecter by the implementation of an architectural concern

After the list of the most critical code anomalies has been produced by each heuristic, different criteria might be applied for breaking ties between code anomalies that have the same architectural relevance. For doing so, we introduce different metrics and source code information that may also be used during the prioritization process. Depending on the heuristics used for prioritizing and ranking critical code anomalies, different metrics might be applied according to criteria defined by the system architect. **Table 17** shows the architecture sensitive metrics used as additional measures for breaking ties when prioritizing and ranking critical code anomalies. For instance, the metrics CDC and CDO can be used when detecting cases of the *Misplaced Concern*. These two metrics allow to measure how system concerns are diffused in the actual system implementation. Therefore, we can distinguish whether a given concerns is indeed predominant in a given architectural component.

5.4.2. Target Applications

We selected 3 medium-size applications to evaluate the architecture sensitive heuristics proposed in this thesis. Two of those systems, Mobile Media (Figueiredo *et al*., 2008) and Health Watcher (Soares *et al*., 2002), have already been introduced in our empirical investigations (see Chapter 3 and 4). In this evaluation, we also selected the last version of this application, because it comprises many changes performed during the system evolution. Those changes range from functionality increments and enhancements on error handling policies to the incorporation of design patterns to improve the system modularity. Our third application is the SubscriberDB system, which is a large software of a publishing house. It manages data related with the subscribers of its publications and it supports complex queries on several types of data. There are several other functionalities supported by this system and, for this reason, we selected version 2.4 of this system. The selected version encompasses all the features implemented in the system, as well as it has a more stable version of the system's descriptive architecture. **Table 18** summarizes the main characteristics of the three target applications investigated in our study.

Table 18 - Characteristics of Target Applications

Target Application	**Mobile Media**	**Health Watcher**	**Subscribers DB**
System Type	SPL	Web	Web
Programming Language	Java	Java	Java
Architecture Design	MVC	Layered	MVC
Selected Version	5	8	2.4
KLOC	54	49	100
Number of Architectural Elements	81	48	42
Number of Code Anomalies	260	497	582

These systems were also chosen because they met a number of relevant criteria for our study: (i) these are non-trivial systems and their sizes (varying from 54 to 100 KLOC) are manageable for an in-depth analysis of code anomalies analysis as required in our study; (ii) the applications have been extensively and successfully evaluated in other studies (Arcoverde *et al*, 2012)(Vidal and Marcos, 2012)(Macia et al, 2014)(Oizumi *et al*, 2014); (iii) we needed to rely on the availability of the system's developers to validate our identification of code anomalies instances; and (iv) the architecture blueprints, used to reason about changes requests and produce new versions, were available for all the target applications.

5.4.3. Inter-Component Heuristics

As the first step, components represented in the architecture blueprints should be mapped to the corresponding code elements responsible for realizing them. In addition, it is important to mention that this precondition is related with the completeness of artifacts in both levels of the system representation. Thus, even when the architecture blueprint is incomplete, all the components need be mapped to at least one element in the system implementation. Recalling that our first set of heuristics investigates the occurrence of two different scenarios related with problems in the inter-component communication. The first scenario characterizes when code elements are used by (or *attracts*) several external anomalous code elements. In turn, the second scenario is characterized by code elements that depend a lot (or are *addicted*) of external anomalous code element. Each of these scenarios will be described in the next subsections. Those scenarios might also be related to different types of architectural drift problems introduced in Chapter 2.

This *External Attractor* heuristic is based on the assessment of anomalous code elements that are used by several code elements belonging to other architectural components. In addition, occurrences of *External Attractor Component* might lead to the introduction of critical anomalies in the code elements using the architectural component under assessment. For example, if the anomalous code element under investigation implements different concerns, the external code elements depending on the assessed code element can be forced to deal with concerns that they are not interested. When applying the inter-component heuristics for prioritizing and ranking critical code anomalies, we first detected the anomalous code elements responsible for realizing each architectural component. After that, the heuristic computes the number of anomalous code elements realizing external components that depend on the component under assessment. Finally, the anomalous code elements detected as instance of *External Attractor Component* are ranked according to their architecture relevance.

This heuristic was applied for all the three target applications selected in our study. We observed the results indicated an acceptable accuracy of the heuristics in terms of prioritizing and ranking critical anomalies. **Table 19** shows the results of applying the heuristic for detecting occurrences of *External Attractor Components*. For the Mobile Media, we observed that 5 of 10 measures had low accuracy when compared to the anomalous code elements defined in the *ground truth.* In the case of Health Watcher, the results showed an accuracy level of 40% when prioritizing and ranking critical code anomalies. The recurrent problem is that most part of anomalous code elements identified by the heuristics had the same number of code anomalies. In addition, those anomalous code elements are all implementing the GUI concern. Although this concern is represented in the architecture blueprint by two components, there are 47 elements in the source-code responsible for its implementation. Finally, the heuristics performed better when prioritizing and ranking the anomalous code elements in the Subscriber DB. As we can observe, the accuracy for prioritizing and ranking critical code anomalies was around 67%. However, we observed that in the SubscriberDB the number of instances of code anomalies in many code elements were the same.

Table 19 - Results for Inter-Component Heuristics

Name	**N-ranked CE**	**External Attractor**		**External Addictor**	
		Overlap Size	*Accuracy*	*Overlap Size*	*Accuracy*
Mobile Media	10	5	50%	5	50%
Health Watcher	30	12	40%	11	37%
Subscribers DB	15	10	67%		

This heuristic identifies groups of anomalous code elements that depend (or are addicted) on anomalous code elements belonging to external architectural components. In order to identify the critical code anomalies, we firstly identify the anomalous code elements in the architectural component under assessment. For each anomalous code element, we need to detected external dependencies with other code elements realizing external components At the end, a higher score must be assigned to architectural components where we observe: (i) a higher number of anomalous code elements; and (ii) a higher of number code elements realizing external components depending on the code elements in architectural component under assessment.

Furthermore, Mobile Media represents 18 architectural components, while Health Watcher and Subscriber DB have respectively 6 and 8 architectural components represented in the architecture blueprint. Even with a different level of abstraction for representing the architectural elements, we could only observe instances of *External Addictor Component* in 2 out of 3 target applications. In the Mobile Media, 6 architectural components were involved in occurrences of *External Addictor Component*, while in the Health Watcher we observed 3 architectural components. **Table 19** shows the results of applying our second heuristics for prioritizing and ranking critical code anomalies. For Mobile Media, we observed that 5 of 10 code elements are correctly prioritized and ranked, which indicates an accuracy of 50%. In the Health Watcher, we observed that 11 of 30 code elements are correctly prioritized and ranked as being critical to the software architecture design. That is, this heuristic for detecting instances of *External Addictor Components* achieved an acceptable accuracy when prioritizing and ranking critical code elements that might contribute to architectural drift problems. In addition, as we have not detected instances of *External Addictor* Component in the Subscribers DB system when this heuristic was applied.

5.4.4. Concern-Based Heuristics

Similarly to our first set of heuristics, we discuss how the heuristics evaluation was carried as well as the expected results of applying the concern-based heuristics. This set of heuristic is directly related with problems on the implementation of system's concerns. Therefore, anomalous code elements responsible for implementing a high number of system concerns have to be properly prioritized and ranked, since they violate, for instance, the *Single Responsibility Principle.* Different weights (or level of importance) can be assigned to system's concerns according to the system architect. When this information is not available, a criterion for assigning weights can be also adopted, such as the number of concerns realized by each architectural component. On the other hand, a concern can have high priority when it is implemented by a many code elements - considering the total number of code elements in the system under investigation.

Moreover, architectural problems caused by violation this principle can be observed when anomalous code elements within an architectural component contain several independent concerns. Thus, architecture blueprint should initially be mapped to the source code elements of the existing concerns modularized by the architectural components. Those concerns were validated by the system architects before the heuristic have been applied. When deciding what architectural components have a higher importance, we firstly detected the number of anomalous code elements implementing a given architectural component, which are responsible for realizing the same concerns.

Our third heuristic aims at detecting code elements within the same architectural component that modularizes several independent concerns. A system concern is considered to be independent when it should be modularized by a different architectural component. Firstly, we detected the number of anomalous code elements that implements the same concern within an architectural component. For example, in the Health Watcher system we could detect architectural components that implements more than one concern: GUI (4 concerns), Business Rules (2 concerns), Distribution Manager (4 concerns) and Data Manager (2 concerns). Similarly, the SubscriberDB implements more 8 architectural components (AddSubscribersUI, EditSubscribersUI, MailingUI, SearchUI, SubsriberController, MailingController, SearchController and Persistence). Each architectural component in this application is responsible for realizing 2 concerns.

Code elements implementing one of those architectural components are likely to suffer from *Concern Overload*, since they have to deal with most part of the system's concerns realized for a given architectural concern. Calculating the results for this heuristic can be straightforward. We considered three main measures: (i) number of architectural concerns responsible for realizing more than a system concern; (ii) number of anomalous code elements in a given component, which are responsible for implementing more a concern; (iii) given a list of anomalous code elements in each architectural component, we quantify the number of anomalies affecting code elements within each component to break ties when the prioritization and ranking process. For the Mobile Media, 6 out of 10 (60% accuracy) anomalous code elements are correctly prioritized and ranked when compared to the *ground truth*. The performance for this heuristic is even better for the Health Watcher and SubscriberDB systems. While SubscriberDB achieved 73% of accuracy, in the case of Health Watcher the accuracy of the heuristics when prioritizing and ranking anomalous code elements reached 87% of accuracy. In addition, if we consider only the top 10 anomalous code elements, 9 out of 10 code elements were correctly prioritized and ranked when compared to the *ground truth* of the most critical code anomalies. Similarly, for the SubscriberDB system the heuristic correctly prioritized and ranked 8 out of 10 anomalous code elements.

The fourth heuristic identifies groups of code elements modularizing an architectural concern that is not the predominant one of their enclosing architectural component. A concern is considered predominant in a given architectural component if most of the code elements in this component are dedicated to modularize it. When prioritizing and ranking code elements using this heuristic, we could only identify instances of *Misplaced Concern* on Mobile Media and Health Watcher. Although all the architectural components in the SubscriberDB implement at least two concerns in the system, developers have not provided information of which concern is predominantly addressed by each component. In this sense, we could not apply the heuristic for detecting instances of *Misplaced Concerns* in this system.

Table 20 illustrates the concerns implemented in Health Watcher and Mobile Media. It is important to recall that the concerns of Mobile Media and Health Watcher have already been well documented, respectively, in (Figueiredo *et al.*, 2008) and (Soares *et al.*, 2002). As we can observe, Health Watcher implements 6 architectural concerns, while Mobile Media implements 5 architectural concerns. To define which architectural concern would be more relevant for the prioritization heuristic, we analyzed two additional metrics: (i) Concern Diffusion over Components (CDC) and (ii) Concern Diffusion over Operations (CDO). Those metrics together quantify the degree of scattering of the architecture concerns. A higher measure of scattering means that more code elements are implementing the architectural concerns in a software system.

In this sense, **Table 21** indicates that this heuristic produced good results for both systems. For the Mobile Media system, we observed that 5 (out of 10) measures had acceptable accuracy. When comparing the list of code anomalies provided by the prioritization heuristic and the *ground truth*, we observed that some code elements were equally prioritized. In this scenario, we can use, for instance, the density of code anomalies for breaking ties on the prioritization of anomalous code elements. Moreover, it is important to mention that for this heuristic we only considered the top 10 elements for the Mobile Media (no ties were considered). On the other hand, we observed that 25 of 30 measures in the Health Watcher had high accuracy. That is, the heuristic can produce highly accurate prioritization of critical code anomalies. It indicates that the most part of the elements affected by multiple code anomalies are frequently identified with a high priority. For the Health Watcher system, this heuristic achieved 84% of accuracy when comparing with the ground truth. The Health Watcher also had a higher number of concerns.

Table 20 – Architectural Concerns for Health Watcher and Mobile Media

Target Application	**Concerns**	**CDC**	**CDO**
Health Watcher	Concurrency	8	42
	Distribution	49	76
	Exception	73	294
	Transaction	41	158
	Business	37	222
	View	21	44
Mobile Media	Counting/Sorting	5	42
	Favorites	5	32
	Exception	28	256
	Persistence	25	106
	Media Management	49	68

Table 21 – Results for Concern Based Heuristics

Name	**N-ranked CE**	**Concern Overload**		**Misplaced Concern**	
		Size Overlap	*Accuracy*	*Size Overlap*	*Accuracy*
Mobile Media	10	6	60%	5	50%
Health Watcher	30	26	87%	25	84%
SubscriberDB	15	11	73%		

5.4.5.
Accuracy of the Architecture Sensitive Heuristics

After each architecture sensitive heuristic have been applied, we observed some findings when analyzing the results. On the analysis of occurrences of *External Attractor Components* and *External Addictor Components*, we observed that both scenarios occur for, (at least) 2 out of 3 target application. Those occurrences concentrate more than 60% of dependencies between architectural components in those the target applications. The high percentage indicates a tight coupling between architectural components representing information about the descriptive architecture. This strong coupling is likely to be related with anomalous code elements that realize the communication between the architectural components. When analyzing the results for the *Inter-Component Heuristics*, we observed, in general, the heuristics presented an acceptable accuracy for all the target applications where they have been applied. When analyzing the results for the inter-component heuristics, we also observed that code elements infected by multiple code anomalies are often perceived as high priority. The results observed in the analysis helped rejecting the *null hypothesis* $H_{1.0}$, as the inter-component heuristics were able to prioritize and rank critical code anomalies with an acceptable accuracy in all the target applications under assessment.

On the other hand, the number of occurrences of *Misplaced Concern* and *Concern Overload* indicates a high proportion of anomalous code elements are related with problems on the implementation of system concerns. We observed that in both systems some architectural concerns are crosscutting several code elements. That is, they are scattered through anomalous code elements. In this sense, the mapping of concerns was clearly useful for prioritizing and ranking a significant number of critical code anomalies. Moreover, the results showed that, in general, the heuristics presented an acceptable or high accuracy for all the applications where the heuristics have been applied. In this sense, the analysis of the results indicated that the *null hypothesis* $H_{2.0}$ is rejected, as the *Concern-Based Heuristics* were able to identify code elements containing critical code anomalies in both target applications.

5.5.
Discussions

In addition to the data analysis performed for each architecture sensitive heuristics, we also discuss other intesresting findings observed when analyzing the results. Firstly, we discuss how the architecture sensitive heuristics influenced the number of **False Positives** and **False Negatives** when prioritizing and ranking critical code anomalies (Section 5.5.1). After that, we compare how the architecture sensitive heuristics performed in relation to other existing approach for prioritizing and ranking critical code anomalies (Section 5.5.2).

5.5.1.
Identifying False Positives and False Negatives.

This section discusses the proportion of **False Positives** and **False Negatives** considering the target applications under investigation. This proportion was computed by applying each of the heuristics proposed in thesis: *external attractor* (EAt), *external addictor* (EAd), *misplaced concern* (MC) and *concern overload* (CO). **Table 22** summarizes the number of **False Positives** and **False Negatives** observed for each target application considering the N-ranked code elements (CE) prioritized and ranked according to the proposed heuristics. When analyzing the Mobile Media system, we observed that the heuristics indicated **False Positives** mostly related with the implementation of *Data* (10 instances), *View* (6 instances) and *Controller* (2 instances) concerns. On the other hand, all the code elements prioritized and ranked as **False Negatives** by the heuristics implement the Controller functionality.

Table 22 - False positives and negatives achieved by the prioritization heuristics

System	**Measure**	**N-CE**	**Prioritization Heuristic**			
			EAt	*EAd*	*MC*	*CO*
Mobile Media	FP	10	5	5	4	4
	FN	10	5	5	4	4
Health Watcher	FP	30	18	19	3	4
	FN	30	12	11	3	4
Subscribers DB	FP	15	5			5
	FN	15	5			5

Moreover, we have also analyzed the anomalous code elements prioritized and ranked by the heuristics in the Health Watcher and SubscriberDB systems. For the Health Watcher, we observed that the **False Positives** are mainly related with code elements implementing, respectively, *Data*, *Concurrency* and *Distribution* functionalities. However, one reason for ranking those code elements, as **False Positives** is the fact that they implement a high number of code anomalies when compared to the other ranked code elements. This result is specially observed when the *External Attractor* (Eat) and *External Addictor* (Ead) heuristics are applied.

In the case of the *Misplaced Concern* and *Concern Overload* (CO) heuristics, the code elements prioritized and ranked as **False Positives** are mainly with two important functionalities in the system, which are GUI (19 instances) and Business Rules (3 instances) functionalities. Moreover, the code elements realizing the GUI functionality are responsible for implementing the key interfaces that provide access to all the services available in the system. Although those code elements implement one of the most important components in the descriptive architecture of Health Watcher system, code elements do not have a high number of code anomalies – and therefore the heuristic have not ranked those elements as being critical to the architectural design.

Finally, for the Subscribers DB (S_{DB}), we observed instances of code elements prioritized and ranked as **False Positives** are responsible for implementing different functionalities, namely: Model (1 instance), View (2 instances) and Controller (2 instances). The problem is that the anomalous code elements are similarly distributed through the architectural components represented in the architecture blueprint. In addition, we have not applied the *Misplaced Concern* heuristic, since developers have not provided information about the predominant concerns each component is responsible for realizing according to the descriptive architecture specification.

5.5.2. Comparing Ranking Provided by Different Heuristics

Despite the existence of many different strategies for detecting code anomalies, only a few of them provide support for prioritizing and ranking code anomalies. Even in this restricted scenario, those approaches do not consider architecture information when prioritizing and ranking critical code anomalies. Therefore, developers are not able to distinguish what code anomalies should be correctly prioritized and ranked, for instance, according to architectural problems. In this sense, critical code anomalies are remaining in the source code since they refactoring are not correctly prioritized and architectural problems might emerged as the system evolves. In severe cases, the critical code anomalies can lead to the degradation of the descriptive architecture. In the following, we perform a comparison analysis of different heuristics for prioritizing and ranking critical code anomalies.

Our previous joint work (Arcoverde *et al.*, 2012) proposed heuristics for prioritizing and ranking critical code anomalies based on the evolution history of 4 target applications. Those heuristics are based strictly on source code information, such as number of bugs, number of errors and density of code anomalies. Moreover, the heuristics collect this information considering the evolution history of a software system. Moreover, those heuristics were applied for prioritizing critical code anomalies related to the presence of code anomalies and erosion problems (Hochstein and Lindvall, 2005) in the descriptive architecture. The problem is that the heuristics required the system's history information when prioritizing and ranking critical code anomalies in a given software system. However, this information usually is not available in the early stage of system development. In this sense, **Table 21** shows the results of applying such heuristics. It is important to emphasize that the heuristics were applied for the same version of Mobile Media ad Health Watcher investigated in this paper. As we can see the heuristics performed well in most part of the cases achieving from acceptable to high accuracy. Moreover, the heuristics also prioritized code anomalies associated with architectural problems. For each prioritization heuristic, it was ranked the top 10 code elements according to their architectural relevant, the four heuristics performed well when prioritizing the critical code elements, since all heuristics achieved accuracy higher than 70%.

Table 23 – Relevance based on the system history evolution

Heuristics Based on History Evolution				
Heuristic	**Name**	**N-Ranked**	**ArchRel**	**% ArchRel**
Change-Proneness	Health Watcher	14	10	71%
	Mobile Media	10	7	70%
	PDP	10	10	100%
Error Proneness	Health Watcher	14	10	85%
	Mobile Media	10	8	80%
	PDP	10	8	80%
Architecture Role	Health Watcher	10	4	40%
	Mobile Media	10	9	90%
	PDP	10	10	100%
Anomalies density	Health Watcher	10	5	50%
	Mobile Media	10	9	90%
	PDP	10	8	80%
	MIDAS	10	6	60%
Heuristics Based on Architecture Sensitive Information				

Heuristic	Name	N-Ranked	ArchRel	% ArchRel
External Attractor	Health Watcher	30	21	70%
	Mobile Media	10	9	90%
	Subscribers DB	15	10	67%
External Addictor	Health Watcher	30	22	74%
	Mobile Media	10	5	50%
Misplaced Concern	Health Watcher	30	24	80%
	Mobile Media	10	8	80%
Concern Overload	Health Watcher	30	24	80%
	Mobile Media	10	9	90%
	Subscribers DB	15	11	74%

On the other hand, the architecture sensitive heuristics proposed in this paper exploits architecture information provided in the architecture blueprints[3], which represents the system's descriptive architecture. Those heuristics allow developers prioritize and rank critical code anomalies according to architectural drift symptoms in the early stage of software development. In addition, it is important to mention that architectural drift symptoms are more difficult to detecting than drift problems, since they are not possible to detect by only analyzing the actual system implementation. Moreover, architecture drift symptoms usually are precursor of erosion problems that are likely to be manifested during the evolution of software systems.

Although the proposed heuristics have not presented superior results when compared to the heuristics based on the system history evolution (see **Table 23**), most part of anomalous code elements is related with architectural drift problems. Thus, we performed an analysis to evaluate how accurate the heuristics performed when prioritizing and ranking critical code elements related with architectural problems. The collected data showed most part of the anomalous code elements are related with architectural problems. That is, in average more than 75% of the anomalous code elements identified by the architecture sensitive heuristics proposed in this thesis are related with architectural problems.

5.6. Summary

As previously discussed in Chapter 4, architecture blueprints, which represent the system's descriptive architecture, can be used as means to improve the prioritization and ranking of critical code anomalies. Our controlled experiments

[3] System developers and maintainers produced the architecture blueprints

showed that process of prioritizing and ranking critical code anomalies can be improved in terms of the **Precision** and **Recall** measures. In addition, the results also showed that architecture blueprints did not bring any additional effort in terms of time spent when prioritizing and ranking code anomalies. The problem is that not all participants in the controlled experiment were able to reason about how architecture information could be properly used when performing the experimental tasks.

In this sense, this chapter introduces and evaluates architecture sensitive heuristics. They exploit different information about the system's descriptive architecture in order to prioritizing and ranking critical anomalies. In addition, the heuristics consider different architecture information based on the type of the architecture drift symptoms that architects and developers are interested to investigate. Therefore, we exploit the architecture blueprints to semi-automate the prioritization and ranking of critical code anomalies. In addition, as main contributions of this chapter we can mention: (i) two sets of prioritization heuristics based on different criteria for prioritizing and ranking the most critical anomalous code elements based on their architectural relevance; (ii) the evaluation of the proposed architecture sensitive heuristics regarding the architectural relevance of the anomalous code anomalies; (iii) a discussion on how the architecture sensitive heuristics impact in the process of correctly prioritizing and raking the critical code anomalies in the target applications under analysis.

Furthermore, we discuss how the proposed heuristics performed when they are applied in three medium-size applications by measuring their accuracy according to the size of overlap achieved in comparison to the *ground truth* provided by developers. As main findings observed during the empirical evaluation of the proposed heuristics, we can mention: (i) there are architectural problems involving groups of classes that realize architectural components, which are intended to implement a specific functionality in the system's descriptive architecture; (ii) there are several symptoms of degradation involving architecture components infected by multiple anomalies; and (iii) even for the architectural concerns that are well defined and relevant to a software system, the prioritization heuristics were efficient to pinpoint problems with the implementation of those concerns.

Finally, we evaluated each set of heuristics in a systematic way by comparing to the heuristics defined in our previous work. They are based on analyzing the history about the evolution of software systems; they aim at exploring this history information to prioritize critical code anomalies regarding degradation symptoms. It is important to reinforce that we decided to focus on architectural drift problems since they manifest in the early stages of the system development. Unlike erosion symptoms, their detection is not effective through source code analysis only.