1 Introduction

Software architecture plays a central role in software development projects as it influences the satisfaction of the key quality attributes, such as modularity and maintainability. Well-designed software architectures usually lead to high-quality software systems (Bass *et al*., 2003). Software architecture is a high-level design involving the description of basic elements – such as components and interfaces – as well as their interactions (Perry and Wolf, 1992). Software architecture is often seen by software industry developers as essential in the software development process (Baltes and Diehl, 2014). In fact, empirical studies (Clements *et al*., 2002)(Baltes and Diehl, 2014) have revealed architectural design decisions usually are, at least, documented as informal models. The architecture models might be incomplete, but the key architecture decisions are part of these models, which are archived and used along software evolution (Baltes and Diehl, 2014). These documented decisions are used to guide the software implementation, the developers' communication, and the software evolution (Clements *et al*., 2002).

The *prescriptive architecture* captures the key design decisions made prior to the system's construction (Taylor *et al*., 2009). In other words, the prescriptive architecture will influence the architecture implementation. However, the actual architecture implementation often does not match the prescriptive architecture. The *descriptive architecture* describes the actual software architecture as observed in the implementation. When software evolves, its prescriptive architecture should ideally be modified first. However, in practice, the descriptive architecture is often directly modified in the implementation without any proper reasoning about the prescriptive architecture. The reasons for this problem range from lack of programmers' awareness about the need of updating the prescriptive architecture to other priorities given the short deadlines in software projects (Taylor *et al*., 2009). These unplanned changes in the implementation reduce the modularity and maintainability of the descriptive architecture realized in the program (Taylor *et al*., 2009).

In this context, the source code changes without considering the prescriptive architecture might hinder the satisfaction of key quality attributes, such as modularity and maintainability. In other words, these changes will introduce structural problems in the implementation of the software system, which are associated with architecture degradation problems. When software developers perform bad architecture decisions in the actual implementation of the system, those decisions can be reflected in the descriptive architecture. Then, architecture degradation symptoms are likely to manifest during the system evolution. Architecture degradation (Hochstein and Lindvall, 2005) is a general term often used to refer to the decay of the architectural design properties of a software system. Therefore, when software architecture degradation symptoms are not properly addressed, the evolution of a software system can be irreversibly compromised. In extreme cases, symptoms of architectural degradation may also cause the reengineering of software systems (Eick *et al*., 2001)(MacCormack *et al*., 2006).

1.1. Motivation

A key concern of software architects is to ensure the modularity and maintainability of the actual architecture, observed in the implementation (descriptive architecture). The reason is that this architecture is the one influencing developers' work when they perform changes in the implementation. The main factor often related with emergence of architecture degradation symptoms is the progressive and unavoidable insertion of code anomalies in a program (Hochstein and Lindvall, 2005).

Code anomalies – also popularly referred to as *code smells* (Fowler *et al*., 1999) – are structural symptoms in the source code, which might also indicate problems in the software architectural design. Examples of code anomalies are God Classes, Long Methods, and Feature Envies (Hochstein and Lindvall, 2005). For example, instances of these code anomalies are used for revealing degradation symptoms in the *descriptive architecture* (Macia *et al*. 2012a)(Macia *et al*. 2012b). In fact, empirical studies (Macia *et al*., 2012a)(Macia *et al*., 2012b) revealed that around 80% of architectural design problems (Garcia *et al*., 2009b) are related with the presence of well-known types of code anomalies, such as God Classes and Long Methods. When

a code anomaly is related with one or more architectural problem, we state this code anomaly is *critical*[1] to the software architecture design.

Figure 1 depicts an example where we can observe architecture decisions present in the *prescriptive architecture* specification (left side) and the *descriptive architecture* (right side). This example will be used to illustrate the elements of our research. For instance, the architectural components *SearchUI*, *SearchController* and *SubscriberController* serve to illustrate cases of architectural degradation symptoms. They manifest symptoms of architectural drift problems related with *Scattered Parasitic Functionality* and *Ambiguous Interface* (Garcia *et al*., 2009b). *Architectural drift* characterizes the introduction of key design decisions into a system's descriptive architecture: (i) that are not included in the prescriptive architecture, but (ii) they do not violate any of the prescriptive architecture's design decisions (Martin, 2003). In other words, architecture drift problems do not represent a violation of a decision made in the prescriptive architecture specification. Typical examples are interfaces of components, which exist in both prescriptive and descriptive architectures, but became too complex due to unplanned changes. An *architecture erosion* problem is the opposite: there is a mismatch between the architecture prescriptive and descriptive design decisions.

In this thesis, we decided to focus on architectural drift problems for three main reasons. First, the manifestation of architectural drift problems in the software project history usually precedes architectural erosion problems (Hochstein and Lindvall, 2005)(Gurgel *et al*., 2014). Second, they are more difficult to be revealed by developers than erosion problems because there is no explicit violation of single architecture design rules. Third, architectural drift often manifests in the descriptive architecture, which can be inferred from the source code. If some design decisions are not properly modified in the source code (descriptive architecture), they will be not properly performed in the evolution of the prescriptive architecture.

The key challenge to software engineers is that certain architectural problems in the implementation cannot be merely detected based only on the source code analysis (Macia *et al*., 2012b). Each architectural problem is often realized by multiple code anomaly occurrences scattered in the implementation (e.g., see Figure 1). Then, the

[1] Also referred to as *architecturally-relevant code anomaly*

detection of the architectural problem requires the developer to spend effort to understand whether and which code anomalies contribute to an architectural problem. Therefore, the effect of occurrences of code anomalies affecting many code elements when detecting problems in the descriptive architecture need to be manually investigated by developers. In addition, it is important to mention that several occurrences of code anomalies are not necessarily related with architectural problems.

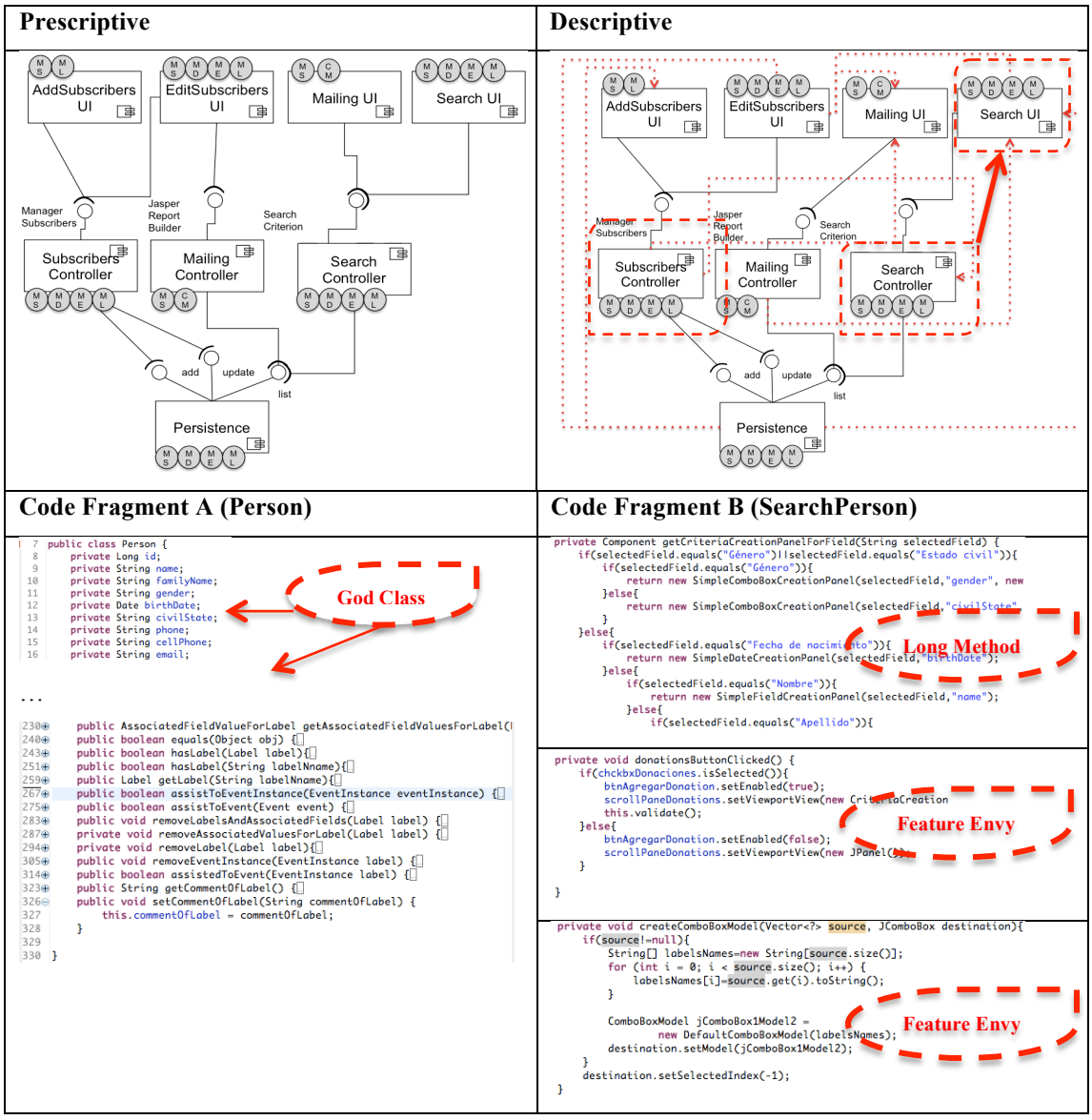

Figure 1 - Code Fragments with Non-Critical and Critical Code Anomalies

Empirical studies (Macia *et al.*, 2012a)(Macia *et al.*, 2012b) revealed that only 40% of code anomalies detected using conventional strategies could be associated with problems in the system architecture. Therefore, software developers are expected to be able to: (i) distinguish the code anomalies associated with problems in the

descriptive architecture, from those that are not; and (ii) rank the critical code anomalies according to their criticality regarding the impact on the architecture degradation. These are time-consuming and error-prone activities as the developers would need to reason about all the code anomalies, understand their relationships. They would need also to reason about the relation of each code anomaly and the architecture problem counterparts in order to identify the relative relevance of each of them.

Motivation Scenario. As just mentioned, occurrences of critical code anomalies must be separated from those not harmful to the architectural design. In addition, software developers must also be able to rank those code anomalies according to the criticality of the architectural problems they might be associated. **Figure 1** depicts a code element *Person* (Code fragment A), which is infected by code anomaly called *God Class*. A *God Class* can be understood as a large class implementing many responsibilities in the software system (Fowler *et al*., 1999). This particular instance of *God Class* might be related to the architectural problem affecting the component *SubscriberController*. This architectural component, among others represented in the specification of the descriptive architecture, suffers from the architectural problem *Scattered Parasitic Functionality* (Garcia *et al*., 2009b). The architectural problem *Scattered Parasitic Functionality* manifests when multiple components are responsible for realizing the same high-level architectural concern. As depicted in **Figure 1**, the *SubscriberController* components are responsible for implementing 4 different concerns - Manage Subscribers (MS), Manage Donations (MD), Manage Events (ME) and Manage Labels for Subscribers (ML). Thus, different concerns are implemented in the code element *Person*, which is responsible for realizing the *SubscriberController* component. Even though, specifically in this case, a single instance of the God Class anomaly is associated with the architectural problem, certain instances of God Classes in the same system are not be associated.

However, other architectural problems are only possible to detect when multiple anomalies in the source code affect several architectural elements. For instance, **Figure 1** (code fragment B) shows the code element *SearchPerson*, which is affected by 3 different code anomalies, namely Long *Method*, *Feature Envy* and *God Class* (Fowler *et al*., 1999). The *SearchPerson* is one of the classes responsible for realizing the *SearchController* component. This architecture component suffers from an

architectural problem called *Ambiguous Interface* (Garcia *et al*., 2009b), which occurs when interfaces offer only a single and general entry-point into a component, reducing the system analyzability and understandability. In this case, the key interface *SearchCriterion* is defined as the only entry-point into the *SearchController* component, which is realized by several classes. Therefore, all the classes realizing this component are forced to implement this interface. In addition, when code anomalies co-exist in classes implementing a given architectural component, undesired dependencies – which are not specified in the prescriptive architecture - might also be observed between the architectural components. All the examples discussed here show that prioritizing and ranking critical code anomalies are important, albeit challenging and time-consuming tasks.

1.2. Problem Statement

Critical code anomalies can lead to the architecture degradation during the system evolution (Macia *et al*., 2012a)(Macia *et al*., 2012b). In order to prevent architecture degradation symptoms (Hochstein and Lindvall, 2005), critical code anomalies must be refactored and removed as early as possible during the software development. Moreover, software developers are usually expected to choose which code anomalies should be refactored first, mainly due to: (i) time constraints, and (ii) attempts to find the correct solution when restructuring a large system. The prioritization of code anomalies is often required for increasing the effectiveness of such refactoring activities. Thus, software developers are expected to distinguish the critical code anomalies, as well as rank those code anomalies according to their impact on problems in the descriptive architecture. When it is not possible distinguishing and ranking critical code anomalies developers will spend more time addressing problems that are not harmful to the system architecture design (Macia *et al*., 2012b).

Despite the existence of several strategies for detecting code anomalies (Marinescu *et al*., 2004)(Moha *et al*., 2006)(Ratzinger *et al*., 2005)(Salehie *et al*., 2006)(Tsantalis, 2008), they fail to assist developers on revealing critical instances of code anomalies associated with problems in the *descriptive architecture*. The problem

is that existing detection strategies are strictly based on source code static analysis. However, analyzing only the source code, which is responsible for realizing the *descriptive architecture*, is not an effective way to reveal architecture degradation symptoms. In addition, those detection strategies detect a high number of occurrences of non-critical code anomalies even in small systems. Therefore, those strategies do not assist developers when distinguishing and ranking the code anomalies considered as candidates to be associated with architectural degradation symptoms. In this sense, despite the existence of several studies on the impact of code anomalies on the quality of software systems (Godfrey and Lee, 2000)(Eick *et al*., 2001)(MacCormack *et al*., 2006)(Knodel *et al*., 2008), developers are still lacking support to characterize which code anomalies are gradually related with architectural problems. Consequently, they have no clue regarding which code anomalies they should refactor earlier in the system development, and hence, those critical code anomalies remain in the source code.

Unfortunately, there is limited to none knowledge about how to prioritize and rank code anomalies critical to the software architecture. Furthermore, there is a lack of empirical investigation on how architecture information, often available on software projects, could be used as means to help developers when revealing concrete problems in the *descriptive architecture*. This lack of knowledge is omnipresent to all existing techniques for architecting and maintaining software systems, from object-oriented and aspect-oriented techniques to model-driven techniques. On the other hand, the identification of architectural problems, by only looking the architectural specification, is not a trivial task. The reason is that the specification of the *descriptive architecture* does not contain all information related with the architecture decomposition. In addition, architectural design decisions usually are not entirely specified in real software projects, but they are only partially represented as informal models[2]. Thus, in order to prevent architecture degradation, software developers should be provided with means for prioritizing and ranking the most critical anomalies as early as possible.

[2] Referred to as *architecture blueprints* in this thesis

1.3. Limitations of Related Work

Several works have investigated the impact of code anomalies on software quality (Arcoverde *et al*., 2011)(Macia *et al*., 2012a)(Macia *et al*., 2012b), as well as the support to specification and detection of code anomalies in general (Fowler *et al*., 1999)(Marinescu *et al*., 2004)(Moha *et al*., 2006)(Ratzinger *et al*., 2005)(Salehie *et al*., 2006)(Tsantalis, 2008). However, existing strategies are mostly focused on defining and applying rules for detection of code anomalies in a program, without considering their relevance according to the software descriptive architecture. As those strategies are more focused on the detection of code anomalies, they were not conceived to perform activities towards the refactoring of the most critical code anomalies. In this sense, they do not provide means for helping developers when improving or maintaining the modularity of the system's descriptive architecture – i.e. helping developers to decide which code anomalies should be refactored first, based on the impact on the actual architectural design.

Moreover, the exclusive use of code anomaly detection strategies has not been succeeded on the prioritizing and ranking critical code anomalies for several reasons (Macia *et al*., 2012a)(Macia *et al*., 2012b). Firstly, recent studies (Macia *et al*., 2012a)(Macia *et al*., 2012b) revealed that even when strategies are calibrated, they fail to support software developers when distinguishing what occurrences of code anomalies are critical to the architectural design. The problem associated with automatically collected measures is the fact they purely represent properties of the source code structure, and therefore, they are often agnostic to the architectural design. Once the architecture decomposition is not explicit in the source code, developers might consider all measures and respective modules have the same relevance in the architecture design. Secondly, existing strategies are context dependent as the choices of metrics and thresholds need to be calibrated depending on the characteristics and complexity of the software project under assessment (Macia *et al*., 2012a)

Although it is highly recommended to detect and prioritize critical code anomalies as early as possible, developers tend to invest more effort on finding new mechanisms that may help on the detection process. Furthermore, software developers

usually spend more time reviewing code anomalies that do not represent any threat to the software architecture design. Recent research has explored the use of other types of available project factors (Wong *et al*., 2010)(Wong *et al*., 2011)(Arcoverde *et al*., 2013), in conjunction or not with program structural metrics, for detecting code anomalies. For instance, extra information ranges from density of (co-) changes in modules (Wong *et al*., 2010)(Raemaekers *et al*., 2012), density of bugs in a module (Arcoverde *et al*., 2013) and similarity measures (Biegel *et al*., 2011). Nevertheless, these strategies do not focus on prioritizing code anomalies according to their architecture relevance. They only perform retrospective analysis of software history data. Moreover, those strategies do not focus on ranking critical code anomalies. The other challenge is that bug reports and change rationale reports in real software projects often do not offer concrete information to enable the understanding about the relevance of anomalous code elements to the architecture design.

On the other hand, architecture blueprints are often available in software projects from the design outset as they are used to communicate key architectural decisions (Clements *et al*., 2002)(Baltes and Diehl, 2014). The use of blueprints has been exploited and assessed in many different software engineering activities, including process evaluation (Alegría *et al*., 2010), model transformation optimization (Jeanneret *et al*., 2011) and test coverage analysis (Araya, 2011). Architecture blueprints can be understood as informal models or sketches about high-level design usually created for communicating developers about the key design decisions in the architecture decomposition of a software system (Baltes and Diehl, 2014).

1.4. Proposed Solution

The solution, proposed in this thesis, is rooted at the assumption that the use of architectural information, available in blueprints, might help developers when deciding what code anomalies are critical to the software architecture. Then, developers can better decide which refactorings should be performed first in order to improve the descriptive architecture in the source code. Therefore, the architecture blueprints are used as additional artifact in the process of prioritizing and ranking critical code anomalies. The blueprints will be used in addition to conventional source

code analysis, typically the solution explored to detect critical code anomalies (Section 1.3). Blueprints often represent key design decisions of the prescriptive architecture (Clements *et al*., 2002)(Baltes and Diehl, 2014). Then, our assumption is that they will serve to infer, at least, which architectural elements are more relevant to the software system.

The prioritization and ranking of critical code anomalies aims at avoiding the degradation of the prescriptive architecture by indicating the need for removing such anomalies at an early stage of software development. As blueprints are often produced before the system is implemented (Clements *et al*., 2002)(Baltes and Diehl, 2014), they might help to prioritize and rank code anomalies already in the first version of a system. Furthermore, exploring architecture blueprints in the prioritization process introduces other challenges. Even in well-documented software systems, there are some difficulties inherent to the use of architecture blueprints on revealing critical code anomalies. The difficulties are discussed as follows.

Architecture blueprints might represent many different characteristics, with different levels of granularity, depending on what type of information the architect are intended to communicate or report to the software developers. For instance, the architecture blueprint in our study of the Mobile Media system represent in more details the descriptive architecture decomposition and the communication between the architectural components. Additionally, as the Mobile Media is a software product line (SPL), all the features implemented in the system are also represented in each component responsible for realizing them. On the other hand, for the Health Watcher system – another system considered in our research, the architecture blueprints represent a more high level view of the system components and interfaces. In summary, architecture blueprints can be produced aiming to attend different interests, which also depend on the nature of the system to be specified and implemented.

Furthermore, architecture *blueprints* can be used for different purposes, such as: (i) informing developers about the full prescriptive architecture to be implemented in the early stage of development, (ii) merely conveying important architectural design decisions; or (ii) reasoning about the construction and evolution of a software system. There is no knowledge about the usefulness of any of these types of blueprints on the prioritization of critical code anomalies in any kind of context. For instance, we can mention the use of blueprints in the model composition context (France and Rumpe,

2007). This context is relevant for architects who use model-driven development for specification, design and implementation of software systems. Many studies have successfully used model composition techniques, either in the industry or academy, for evolving architecture design models. Thus, even considering those different scenarios, it is questionable to what extent the architecture blueprints can be explored as means to guide the prioritization and ranking of critical modules in the system architectures. As main research activities performed in this thesis we aim at:

(i) studying how architecture blueprints (e.g. class diagrams, component diagrams) could add value to the usual process of prioritizing and ranking critical code anomalies;

(ii) performing empirical studies aiming to investigate the impact of using architecture blueprints in the prioritization and ranking process; and

(iii) proposing a blueprint-based approach for prioritizing and ranking critical code anomalies related with architecture degradation symptoms.

Our first goal is associated with the empirical evaluation on the role of architecture blueprints - and the architecture information it represents - in the process of prioritizing and ranking critical code anomalies. For doing so, our empirical evaluation is conducted considering the use of architecture blueprints provided by developers or architects during the system evolution. To guarantee that the architecture design model would, in fact, fit to the concept of architecture blueprints used in this thesis, we have defined 3 properties; level of abstraction, completeness and consistency (see Chapter 2). Those properties also guarantee the architecture blueprints reach a minimum quality so that they can be used in the process of prioritizing and ranking critical code anomalies (see Chapter 5).

Our second goal is to evaluate to what extent the use of architecture blueprint, representing the system descriptive architecture, would improve the process of prioritizing and ranking critical code anomalies. In this sense, we initially performed controlled experiments with participants from different universities and with different technical knowledge. We asked them to perform tasks related to the prioritization and ranking of instances of three well-known code anomalies. After that, we evaluated the results using different metrics (e.g. precision, recall and time) in order to assess how good they performed experimental tasks when blueprints are provided as additional artifact in the process of prioritizing and ranking critical code anomalies. In addition,

our study also investigates what are the main characteristics of instances of critical code anomalies prioritized and ranked as False Positives and False Negatives, when considering software systems with different architectural designs expressed in the blueprints

Our third goal is related with the proposition and evaluation of a blueprint-based approach, where a set of heuristics were created aiming to support developers on the prioritization and ranking of code anomalies that threaten the architectural design. As the set of heuristics relies on relevant information used to evaluate the quality of the software systems' prescriptive architecture, we call it as *architecture sensitive heuristics*. As example of architecture information represented in the blueprints, we can mention: (i) architectural components; (ii) required and provided interfaces; and (iii) dependencies between architectural components. Usually, this information can be either directly observed in the architecture blueprints or inferred from the combination of the source code analysis and the respective blueprint representing the descriptive architecture. These three types of architecture information could be commonly observed in all the architecture blueprints for all the corresponding target applications. Even though the richness of a blueprint varies from a project to another, we consider these three types as the minimum information required to support reasoning about architectural problems in the implementation. Even if there is no detailed information about the architectures, architecture recovery techniques can be used to automatically derive these details (Müller et al, 1993)(Lung, 1998)(Kazman an Carriere, 1998) Maqbool and Babri, 2007).

Besides the source code and the information about the descriptive architecture, our proposed approach will also consider the mapping between elements in both levels of abstraction: architecture elements and the source code elements. The reason is that mapping between source code and architectural elements might help software developers to reveal situations where design principles are violated. In addition, there are several techniques to infer automatically these mappings, with very high accuracy (higher than 90%), only based on the names of architecture elements (Cirilo *et al.*, 2011)(Cafeo *et al.*, 2012)(Nunes *et al.*, 2012). Furthermore, when evaluating the heuristics we considered the combination of different criteria for prioritizing and ranking critical code anomalies. The combination of different criteria might help to improve the accuracy when prioritizing and ranking critical code anomalies.

Nevertheless, the proposed heuristics take into consideration information provided both from the architectural design and source code. It is important to mention that mapping information between the two levels of abstraction aims at improving the accuracy of existing strategies, which are based solely on analyzing source code information. In this sense, the goals of the architecture sensitive heuristics are: (i) to assist developers on the prioritization and ranking of code anomalies harmful to the architecture design during the system evolution; (ii) to allow software developers to prioritize critical code anomalies using different criteria (see Chapter 5) based on the intention of the software architects. For instance, the architecture sensitive heuristics might help developers on identifying problems related either with the communication between architecture components or with the implementation of the system concerns; and (iii) address deficiencies of the existing approaches assisting by improving factors that might lead to many false positives and false negatives.

1.5. Research Questions

Aiming to address our research goals we have defined four research questions (RQs), which are described bellow. Those research questions are important for conducting the empirical studies, as well as for characterizing the evaluation of the architecture sensitive heuristics. Moreover, our empirical studies have been developed aiming to investigate how the existing strategies: (i) provide means to evaluate which code anomalies can be prioritized and ranked by using additional artifacts (i.e. source code, blueprints) provided in the early stages of the system development; and (ii) help developers to reveal how critical instances of code anomalies might be related to drift problems in the descriptive architecture. The expected output of the proposed heuristics is the ranked list of critical code anomalies related with architectural problems according to the different criteria defined by each heuristic (or by the combination of two or more criteria).

RQ_1 – *Does the use of architectural information help developers on revealing architecture problems observable in the source code?*

RQ_2 – *What critical code anomalies are better automatically prioritized when*

exploring architecture blueprints?

RQ_3 – *How the prioritization and ranking of critical code anomalies, when guided by blueprints, might indicate actual symptoms of architecture degradation?*

The first research question (**RQ_1**) aims at investigating what additional architecture information could be used to better explore the architecture blueprints, so that the process of prioritizing and ranking critical code anomalies could be facilitated. Depending on the granularity level on which the information is represented in the architecture blueprint, some characteristics or architectural decisions may not be explicitly represented (see Chapter 3). Thus, software developers and architects should be able to indicate which other information they judge to be useful when prioritizing critical code anomalies. Our empirical studies revealed that inconsistencies in the descriptive architecture could be observed when different paradigms are used.

The second research question (**RQ_2**) aims at assessing how the use of a blueprint, as an additional artifact beyond the source code availability, would help to improve the prioritization of critical code anomalies. In this sense, our first analysis relies on identifying what architectural information represented in the architecture blueprint would be important to improve the prioritization process. In other words, the research question **RQ_2** is mainly concerned with assessing characteristics of the architectural elements that should be indicators of architecture design problems (see Chapter 4). The architectural information is then used for identifying those critical instances of code anomalies that should be prioritized and removed as early as possible. Our initial empirical study demonstrated that architecture blueprints improved the prioritization and ranking process of three critical code anomalies. For instance, we observed that Precision and Recall measures regarding the prioritization process have been expressively improved with respect to a conventional approach, at least for all occurrences of 2 out of 3 types of code anomalies initially investigated. It means developers were able do correctly distinguish critical code anomalies when architecture information was provided.

The third research question (**RQ_3**) aims at evaluating whether the use of architecture blueprint could help developers on revealing how code anomalies are related to architectural problems drift problems in the descriptive architecture. Our study showed that, architecture blueprints should ideally represent information related

to the elements in the system's descriptive architecture (e.g. components, interfaces and relationships between the components), as well as information related with the implementation of the concerns. Therefore, when producing the list of most critical code anomalies, the architecture sensitive heuristics evaluate situations that characterize architectural drift problems regarding: (i) relationships between architectural components; and (ii) problems with the implementation of architecturally-relevant concerns (see Chapter 5). Through the evaluation of the architectural information, software developers will be able to evaluate whether the architecture-sensitive heuristics achieved better accuracy when distinguishing those code anomalies that are really critical to the architecture design from those code anomalies that are not. In addition, the heuristics used this architectural information as means to ranking the most critical code anomalies according to the architectural drift problems they are related. Therefore, we evaluated the accuracy of the proposed architecture-sensitive heuristics when prioritizing and ranking critical code anomalies. In this sense, developers might be able to identify what code anomalies are critical to the architecture design, as well as which code anomalies should be prioritized depending on the criteria adopted by each heuristic. In this sense, the evaluation of the proposed heuristics aims at assisting developers when deciding what code anomalies are critical – and hence should be refactored first. Our results showed that, in average, the prioritization heuristics achieved accuracy higher than 60% when prioritizing and ranking critical code anomalies.

1.6. Outline of the Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of the background needed to understand the thesis and evaluate its contributions. Firstly, we present definitions of code anomalies and introduce the detection strategies existing in the state of art (Section 2.1). Secondly, we describe how the relationship between critical code anomalies and architectural problems might lead to architecture degradation symptoms (Section 2.2). After that, we discuss how the co-occurrence of instances of code anomalies is particularly related with deviations in the architecture design (Section 2.3). Next, we introduce the concept of

architecture blueprint defined in the context of this thesis, as well as the properties we have defined to check whether a design model fits in this concept. Finally, we provided information on how the mapping between architectural and source code elements are performed so that the architecture sensitive heuristics can be properly executed (Section 2.4).

In Chapter 3, we introduced the empirical studies performed to identify how architecture blueprints (and what architectural information) would be useful to assist developers when prioritizing and ranking critical code anomalies (Section 3.3). Our study evaluates the quality of architecture blueprints evolved by using semi-automated model composition techniques (Section 3.4). In this sense, we investigated how we could exploit architecture blueprints, which represents the system's descriptive architecture, as means to predict code anomalies in the actual implementation of different systems. In addition, we evaluated how code anomalies might be related to inconsistencies in AO architecture blueprints during the system evolution (Section 3.5).

In Chapter 4, we describe controlled experiments performed to investigate how the architecture blueprints, which represent the system's descriptive architecture, would help to assist developers when prioritizing and ranking critical code anomalies (Section 4.2). Therefore, we discuss how the use of architecture blueprints has impact in the overall process by evaluating different measures, such as Precision, Recall and Time (Section 4.3). As feedback of the experimental tasks performed in the experiment, we asked the participants to provide information about: (i) what was the rationale when using architecture blueprints in the prioritization and ranking process; (ii) what architecture information they judged as being relevant when performing the experimental tasks. We also asked them to report what information would be relevant to be added in the architecture blueprints in order to facilitate the prioritization and ranking of critical code anomalies (Section 4.4).

In Chapter 5, we describe each of the architecture sensitive heuristics based on two main categories: inter-component and concern-based (Section 5.3). The first set of heuristics exploits information related with code anomaly affecting the communication between architectural components. The investigation of instances of critical code anomaly is related with effects on software maintenance, seeing that those code anomalies can be spread through many architectural elements. The second

set of heuristics investigates the anomalous implementation of architectural concerns, which in turn, can be defined as an architect interest that significantly influences the software architecture. After proposing the heuristics, we evaluated their accuracy when prioritizing and ranking critical code anomalies considering different target applications (Section 5.4). In addition, we describe the study hypothesis, the procedures for data collection and how we obtained the ground truth for the target applications evaluated in this thesis. Regarding the data collection, we describe all the code anomalies investigated in this thesis, as well as the thresholds we have used when collecting all instances of code anomalies for the target applications (Section 5.4.1). Moreover, we describe the main research findings after the architecture sensitive heuristics have been applied for each target application, as well as the test of our study hypotheses (Section 5.4). Finally, we discuss the main contributions when the heuristics are applied solely or combining different criteria aiming to improve the ranked-list of critical code anomalies (Section 5.5).

In Chapter 6, we discuss the main internal, external and construct to validity threats observed in our evaluation (Section 6.1). Finally, we introduce the conclusions and summarize the contributions archived in this thesis (Section 6.2) and points out the activities to be performed as future work (Section 6.3).