Exploring architecture blueprints for prioritizing critical code anomalies: experiences and tool support

E. Guimarães¹, S. Vidal²∗, A. Garcia³, J. A. Diaz Pace² and C. Marcos⁴

¹University of Fortaleza (UNIFOR), Technology Sciences Center, Fortaleza, Brazil
²ISISTAN-CONICET Research Institute, UNICEN, Tandil, Buenos Aires, Argentina
³Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Software Engineering Lab (LES), Rio de Janeiro, Brazil
⁴ISISTAN-CIC Research Institute, UNICEN, Tandil, Buenos Aires, Argentina

SUMMARY

The manifestation of code anomalies in software systems often indicates symptoms of architecture degradation. Several approaches have been proposed to detect such anomalies in the source code. However, most of them fail to assist developers in prioritizing anomalies harmful to the software architecture of a system. This article presents an investigation on how developers, when supported by architecture blueprints, are able to prioritize architecturally-relevant code anomalies. First, we performed a controlled experiment where participants explored both blueprints and source code in order to reveal architecturally-relevant code anomalies. Although the use of blueprints has the potential to improve code anomaly prioritization, the participants often made several mistakes. We found these mistakes might occur because developers miss relationships between implementation and blueprint elements when they prioritize anomalies in an ad-hoc manner. Furthermore, the time spent on the prioritization process was considerably high. Aiming to improve the accuracy and effectiveness of the process, we provided means to automate the prioritization process. In particular, we explored three prioritization criteria, which establish different ways of relating the blueprint elements with code anomalies. These criteria were implemented in the JSpIRIT tool. The approach was evaluated in the context of two applications with satisfactory precision results.

Copyright © 2010 John Wiley & Sons, Ltd.

KEY WORDS: Software Architecture; Code Anomaly; Architecture Blueprints; Empirical Evaluation; Tool Support

1. INTRODUCTION

The unavoidable evolution of software systems results on the increase of their size and complexity, which in turn, might lead to architecture degradation problems. Architecture degradation [1] has often a direct relationship with the progressive insertion of code anomalies [1, 2, 3] in the source code. When code anomalies, also known as code smells [4], are not removed, the architecture of the system is likely to degrade with negative impacts on the project [2, 3]. Some authors have claimed that several types of code anomalies, such as God Class and Divergent Changes, are reifications of architectural

∗Correspondence to: ISISTAN-CONICET Research Institute, UNICEN, Tandil, Buenos Aires, Argentina.
E-mail: svidal@exa.unicen.edu.ar

Copyright © 2010 John Wiley & Sons, Ltd.
problems in the source code [4]. However, it is an arduous task (if not impossible) to identify degradation symptoms directly in the architecture specification. The reason is that architectural decisions are not entirely captured in the specification; instead, they are at best partially represented as blueprints [5], i.e. informal and high-level architecture design models.

Recent studies [2, 3] evidenced that more than 80% of the architectural problems are frequently related to well-known code anomalies. If a code anomaly is related to an architectural problem [6, 2], we say the anomaly is critical* to the software architecture design. Examples of critical anomalies are occurrences of Shotgun Surgery and God Class affecting, for instance, the implementation of a component interface. These anomalies might represent architectural problems, such as bloated component interfaces and scattered architectural concerns [7]. Nevertheless, other studies [3, 8] revealed that a high proportion of code anomaly instances detected using automated code analysis techniques [9, 10, 11], are not critical to the architecture design. According to these findings, existing works [2, 3] showed only a limited proportion of code anomalies, around 40%, being related to architectural problems [7]. Along this line, there is a practical need to provide developers with effective means to prioritize those code anomalies that are harmful to the architectural design, so that they can be removed in a further stage [3].

Previous studies have focused on analyzing metrics of the source code in order to detect critical code anomalies [12, 13]. However, source code analysis alone is often not enough to support detection of critical anomalies [2, 3], since automatically collected measures just reflect properties of the source code structure. Such measures are often agnostic to the architectural design structure and do not help developers to decide which anomalies are relevant to the architecture. We argue that this relevance can be captured by means of architecture blueprints [5]. Blueprints [5] are typically available in software projects from the design outset as they are mainly used to communicate the key architectural decisions [14]. However, there is no investigation in the literature about whether and how blueprints can help in the prioritization of critical code anomalies.

In order to address this gap, in this work we continue a previous publication [15]. Specifically, the current work presents an empirical investigation on the use of architectural information available in blueprints for prioritizing critical anomalies. Our first research question here (RQ1) is whether the use of blueprints by developers enhances their prioritization process. In this case, the prioritization was performed by the developers in an ad-hoc manner, i.e. they judged which information in both the source code and the architecture blueprint are relevant to prioritize certain code anomalies. In this article, we extend that work by conducting a larger experiment in which some subjects explored both architecture blueprints and source code to reveal critical code anomalies, while other subjects did not use the architecture blueprints available (only information from the source code). Even though the experimental results showed that the prioritization process can be improved, the subjects made mistakes in the prioritization, because they missed relationships between code and blueprint elements. In addition, the time spent by the subjects on the prioritization process was considerably high. In order to speed up this process and make it more efficient, the subjects raised the aspect of having automated support for helping them to decide on critical anomalies.

The results from the previous experiment led us to a second research question (RQ2), which investigates to what extent the critical code anomalies can be better prioritized via tool support that takes advantage of architectural information. To address this question, we used an existing tool called JSpIRIT [16], which already supported the prioritization of code anomalies but its criteria were limited with respect to architectural information. We reused a particular criterion provided by JSpIRIT, and furthermore, we extended the tool with three additional criteria. These four criteria were architecture-driven in the sense

---

*We also use the term architecutrally-relevant code anomaly.
that establish different ways of relating architecture information with the source code. Differently from our first evaluation, we assessed these criteria in the context of two software systems in order to guarantee our results were applicable to larger systems. In addition, there was no human intervention in the process of prioritizing the code anomalies of each system (but the developer still provided the input for the criterion used). Using these architecture-driven criteria in JSpIRIT, we observed that the prioritization process improved its precision when identifying critical code anomalies.

The remainder of the article is structured as follows. Section 2 discusses the key concepts that support the experiments performed in this work. Section 3 discusses related work. In Section 4, we describe the steps for the controlled experiment with subjects, and discuss its main findings. Section 5 presents the proposed architecture-based criteria in the context of the JSpIRIT tool. In Section 6, we describe the steps for the empirical evaluation of the criteria. Afterwards, Section 7 covers the threats to validity observed in this work. Finally, Section 8 summarizes our main results and contributions, and outlines future lines of work.

2. BACKGROUND

This section presents the main concepts of our investigation. Section 2.1 discusses the impact of code anomalies on the software architecture of a system. Section 2.2 describes our idea of using blueprints, which are responsible for capturing architectural information (e.g., components, interfaces) when identifying critical anomalies. Section 2.3 provides a formalization of the problem of identifying such anomalies. In Section 2.4, we discuss three different properties an architecture blueprint should have in order to support a good prioritization. Finally, Section 2.5 gives an example of how the presence of critical code anomalies might be associated to architectural problems.

2.1. Impact of Code Anomalies and Architectural Problems

A code anomaly, also known as a “code smell” [4], is an implementation structure that possibly indicates deeper design problems [1], and hence, might hinder software maintenance tasks. Code anomalies can directly contribute to the software architecture degradation [2, 3]. An architectural problem refers to a design decision that negatively impacts the lifecycle properties of a system [17] with the consequent architecture degradation. Unintended design decisions lead to code that either violates the original (prescribed) architecture of the system [18], or goes against well-known modularity principles for software systems. An example of the former can be a code dependency between two components that is forbidden in the architecture blueprint. Classical examples of the latter are: Ambiguous Interface, Component Envy [17], or Cyclic Dependencies [19], among others. These problems are also called “architectural smells”.

Notice that, although an architectural problem often manifests itself in the source code (and might even overlap with an anomaly), it is conceptually different from a code anomaly. Furthermore, not all the architectural problems can be easily identified with automated tools. Some available tools for identifying architectural problems are: Hotspot Detector [20], Arcan [21], Sonargraph*, Structure 101†, and Cast‡, among others.

When software architecture specifications are available and can be mapped to code elements, techniques like the Reflexion model [22] are able to detect architectural violations. Unfortunately, architectural smells are not as straightforward to detect (with tools), and they normally require code inspections by human experts. In this context, those anomalies

---

*https://www.hello2morrow.com/products/sonargraph  
†http://structure101.com  
‡http://www.castsoftware.com/products/architecture-checker
related to architectural smells are considered critical in our vision, because they can “indirectly” point out symptoms of architecture degradation. Our position then is that developers should progressively detect and remove critical code anomalies of a system in order to alleviate architectural problems, thus preventing architecture degradation.

However, the challenge is that most existing techniques/tools fail to assist developers in distinguishing code anomalies critical to the architecture design from those that are not. This is so because current strategies disregard other software information (e.g. architecture blueprints) that might indicate the “architectural relevance” of code anomalies. Therefore, we argue for strategies able to exploit such an information in combination with information from source code artifacts.

2.2. Architecture Blueprints

The concept of blueprint was introduced in [5] to describe the views proposed in the “4+1 View Model”. Architecture blueprints are herein defined as informal models, with a high level of abstraction, and created for the purpose of communicating the system architecture to its stakeholders. In addition, blueprints are often used to inform developers about the key design decisions made by the software architects. Our work focuses on architecture blueprints that capture only a structural view of the software architecture. Figure 1 (top-level part) presents an example of architecture blueprint, which depicts a partial view with the main components of the Mobile Media architecture. In this blueprint, each component is represented by a rectangle, and it can have provided interfaces (lollipop) that are required by other components. An architectural component is a modular abstraction that encompasses one or more coarse-grained functions of the system.

Moreover, Figure 1 depicts the architectural concerns realized by each component. An architectural concern is an architect’s interest or property that significantly influences the design decisions [23]. Concerns can be modularized in one or more architectural components. A component that is responsible for implementing a given concerns is decorated with a small circle that represents that concern. For instance, we can observe...
5 concerns in the blueprint, namely: Counting (C), Sorting (S), Exception Handling (E), Persistence (P) and Favourites (F).

2.3. Formalization

The formalization of architecture blueprints has different purposes: (i) it characterizes the “scope” of a blueprint in the context of our work; and (ii) it explicitly connects the notion of code anomaly (as an implementation element) to the notion of architectural relevance. In addition, the formalization is focused on a component-based, static view of the architecture blueprint, which is common in industry projects [14]. Both implementation elements and code anomalies in a system are modeled at the level of (Java) classes. In the case of an instance of a code anomaly, it is assumed that we can always identify the “main class” hosting the code anomaly.

Due to the need of quantifying blueprints with regards to the corresponding source code, we have initially proposed three metrics (level of abstraction, completeness and consistency) [24]. Furthermore, we deal with the mappings of a blueprint to source code. We provide definitions of the main concepts involved in our approach, which serve as a conceptual framework for Sections 4, 5 and 6. For simplicity, components, interfaces, and concerns are assumed to have unique names.

Definition 1 (Blueprint). We consider an ArchitectureBlueprint as a tuple $<\text{ArchComps}, \text{ArchInterfaces}, \text{ArchConcerns}, \text{realized-by}, \text{provided}, \text{required}>$ that captures a static system view. This tuple includes the following elements:

- $\text{ArchComp} = \{c_1, c_2, \ldots, c_N\}$ is a set of architectural components for the system;
- $\text{ArchInterfaces} = \{i_1, i_2, \ldots, i_M\}$ is a set of component interfaces;
- $\text{ArchConcerns} = \{p_1, p_2, \ldots, p_K\}$ is a set of concerns;
- A binary relation $\text{realizedBy} = \{(p,c) | p \in \text{ArchConcerns} \land c \in \text{ArchComps}\}$ denotes the implementation of particular concerns by architectural components (a given concern can be present in several components);
- A binary relation $\text{provided} = \{(i,c) | i \in \text{ArchInterfaces} \land c \in \text{ArchComps}\}$ models the interfaces offered by the components (a given interface can be provided by more than one component);
- A binary relation $\text{required} = \{(i,c) | i \in \text{ArchInterfaces} \land c \in \text{ArchComps}\}$ refers to the interfaces required by certain components (an provided by other components). In addition, there is a constraint to assure the same interface cannot be simultaneously provided and required by the same component.

Definition 2 (Implementation). For the purpose of our work, we consider an Implementation as a tuple $<\text{Classes}, \text{dependsOn}>$ that represents a code view of the system. The elements of this tuple are:

- $\text{Classes} = \{jc_1, jc_2, \ldots, jc_P\}$ is a set of Java classes (with unique names);
- A binary relation $\text{dependsOn} = \{(jc_1, jc_2) | jc_1 \in \text{Classes} \land jc_2 \in \text{Classes} \land jc_1 \neq jc_2\}$ denotes that a given class requires (or uses) methods defined in other classes. A dependency relation between two classes is often an implementation need.

Definition 3 (Blueprint-Implementation mapping). The mapping between the architecture blueprint and the software implementation is formalized as a partial relation between elements in both levels of abstraction. The elements are described below:

- $\text{mapping} = \{(jc,x) | jc \in \text{Classes} \land x \in \text{ArchComps} \cup \text{ArchConcerns} \cup \text{ArchInterfaces}\}$,
- we say the mapping is a partial relation because for some classes there is not always a correspondence with an architectural element. However, an architectural element can relate to one or more implementation element.

Two additional constraints are also imposed on the mapping pairs:
∀(jc,p): jc ∈ Classes ∧ p ∈ ArchConcerns ⇒ ∃c ∈ ArchComps: realizedBy(p,c) ∧ mapping(jc,c), meaning that each class mapped to a concern must be also mapped to its defining component, and

∀(jc,i): jc ∈ Classes ∧ i ∈ ArchInterfaces ⇒ ∃c ∈ ArchComps: provided(i,c) ∧ mapping(jc,c), meaning that each class mapped to an interface must be also mapped to its defining component.

Note that the mapping relation is based on a static view of both abstraction levels, which are represented by the architecture blueprint and source code implementation. In summary, we defined 3 kinds of links between a given architecture blueprint and the code anomalies present in its implementation, namely: (i) mappings between individual blueprint elements (e.g. components, interfaces, concerns) and corresponding implementation classes; (ii) a set of candidate code anomalies, which captures the “main classes” of those anomalies with mappings to blueprint elements; and (iii) a prioritization function that is applied to the set of candidate anomalies above (see definition 5). This function might also take additional architectural information for computing its results.

Definition 4 (Code Anomaly). A code anomaly (or code smell) can be seen as a “code pattern” that is usually detected in the source code by means of a particular metric-based strategy. In our work, we assume that a code anomaly always defines a “main class”, which is the source code element where the effects of the anomaly mainly show up. There are different types of anomalies, and each of them might have different instances (i.e., main classes) in the source code. Therefore, we defined that:

• AnomalyTypes is a set of predefined code anomalies. When using the Marinescu’s catalog, AnomalyTypes = {godClass, brainClass, brainMethod, shotgunSurgery, dataClass, dispersedCoupling, featureEnvy, intensiveCoupling, refusedParentBequest, traditionBreaker},
• AnomalyInstances is a subset of Classes (see Definition 2), which correspond actually to the main classes for the anomaly instances,
• A relation typeOf: AnomalyInstances → AnomalyTypes that links a given class in AnomalyInstances (i.e., a main class) to one (or more) of the available types in AnomalyTypes. This means that the same class can participate in several anomalies at the same time.

For the sake of simplicity, the anomaly types and main classes are assumed to have unique names. From Definition 3, note that some of the main classes might participate in architecture-implementation mappings, while other main classes might not.

Definition 5 (Critical Code Anomaly). A code anomaly is regarded as “critical” if it is related to an architectural problem in the form of an architectural smell. In order to avoid the formalization of an architectural smell, we assume that the architect can judge whether a given code anomaly is critical (or not). That is, based on her knowledge of the whole system (including the blueprint), she can say if a given anomaly is, for example, a source of technical debt [25] that hinders system maintenance. Therefore, we have that:

• CandidateCriticalAnomalies is a subset of AnomalyInstances;
• ∀mainClass ∈ CandidateCriticalAnomalies ⇒ ∃comp ∈ ArchComponents: mapping(mainClass,comp), in order to guarantee that the main class of the anomaly is effectively mapped to the blueprint (see Definition 3),
• There is a boolean function criticalAnomaly(): CandidateCriticalAnomalies → {true, false} that reflects the architect’s judgment about the relevance of the code anomalies.

In Section 4, we report an experiment with subjects in which function criticalAnomaly() is a responsibility of human developers. In Sections 5 and 6, we provide automated support (with the JSpIRIT tool), and function criticalAnomaly() is instead realized (in an approximate way) by a prioritization() function that actually assigns a score to the main classes according to their architectural relevance. In the latter case, we would have a numerical function:

• prioritization(): CandidateCriticalAnomalies → R+,
• if \( \text{prioritization}(\text{mainClass}) = k > \text{Threshold} \), then we interpret that \( \text{mainClass} \) is critical to the architecture.

Although it is not modeled here, functions \( \text{criticalAnomaly()} \) and \( \text{prioritization()} \) might include information from sources such as the architecture blueprint, implementation, and mapping (Definitions 1, 2 and 3) to compute the outputs.

### 2.4. Blueprints Properties

Architecture blueprints can be distinguished from other types of models as they hold three properties [26], which are: level of abstraction, incompleteness, and inconsistency. Developers can represent a given architecture design in blueprints with different levels of detail [26]. However, it is rarely the case there is a unitary mapping between architecture and source code elements.

#### 2.4.1. Level of Abstraction

The Level of Abstraction (LoA) denotes “how far” the architecture blueprint is from the architecture implementation. To calculate the level of abstraction of the architectural components in the blueprint, it is required to compute the number of source code elements realizing each of the existing components (see Definition 3 in Section 2.3). The computation of the level of abstraction of an architecture blueprint (\( \text{LOA}_B \)) involves two basic procedures: (i) compute the total number of source code elements representing each architectural component, and (ii) compute the ratio between the total number of code elements participating in the mapping process and the total number of architectural components. After that, we can quantify the level of abstraction not only for each component, but also for the architecture blueprint as a whole. In addition, the LoA must have a value \( 0 < \text{LOA}_B \leq 1 \), as the elements in the architecture blueprint should be mapped to at least one source code element.

#### 2.4.2. Completeness

An architecture blueprint is considered to be complete when it characterizes the components involved in the representation of the actual architecture [5]. For each architectural element (e.g. interfaces, components, concerns) represented in the blueprint (see Definition 1 in Section 2.3), there must be at least one corresponding code element in its counterpart program. When the mapping between architecture and source code elements is performed, each architectural component must be associated with at least one code element (a Java class, according to Definition 2 in Section 2.3) responsible for realizing this component. In order to measure the completeness of the architecture blueprint, we have to compute the following information: (i) number of components (not) mapped; (ii) number of interfaces (not) mapped; and (iii) number of concerns (not) mapped. After collecting those measures, we quantify the completeness of the blueprint using the formula:

\[
\text{CB} = 1 - \left( \frac{\text{AC}_{NM} \text{ to } T_{AE}}{T_{AE}} \right),
\]

where \( \text{CB} \) is the completeness of an architecture blueprint, \( \text{AC}_{NM} \) is the number of elements not mapped and \( T_{AE} \) is the total number of architectural elements.

#### 2.4.3. Inconsistency

An architecture blueprint is said to be fully consistent when it does not present any contradiction in the information represented in the mapping between architecture and implementation elements (see Definition 3 in Section 2.3). In this work, the inconsistencies can be classified into 4 cases. First, dependencies not mapped, i.e. dependencies that exist between classes that implement different components; however, those dependencies do not exist in the architecture blueprint. Second, inverted dependency, i.e. there is a dependency from component A to component B represented in the architecture blueprint; however, when looking to the classes responsible for implementing those components in the source code, the dependency occurs in the other way, that is, from class B to class A. Third, components with no (provided) interface. And fourth, different interfaces or components with the same name. The inconsistency of a blueprint is quantified by computing the total number of inconsistencies for each of the cases above. After that, we
calculate the ratio between the inconsistencies and the number of source code elements affected. If a model is very inconsistent, it does not serve the purpose of our study.

2.5. Example of Critical Code Anomalies

Our investigation assumes that critical code anomalies are usually associated to architectural smells.

Aiming to understand this relationship, let us go back to the example of Figure 1 and consider the components (PhotoListController, PhotoController, AlbumController and PhotoViewController in the MobileMedia architecture. These components are mapped to a (Controller) Java package that contains several classes (bottom-level part of Figure 1). In the detailed design of the package, let us notice that class AbstractController is responsible for handling different commands through the handleCommand(...) method. A closer look at method handleCommand shows that it has an overload of responsibilities, which leads to an architectural problem called Ambiguous Interface (AI) [17]. That is, the AbstractController handles several functionalities that are not exposed in its interface. Furthermore, implementations of handleCommand(...) method in the subclasses of AbstractController are affected by the code anomaly called Dispersed Coupling (DC), which characterizes a method that calls a few methods of several classes. For example, Figure 2 shows a fragment of the handleCommand method in class AlbumController. As shown in Figure 1, a provided interface of component AlbumController is realized by this handleCommand method. The method code makes several calls to classes ScreenSingleton andNewLabelScreen (among others), with the consequent coupling. The code anomaly called Brain Method (BD) is also present in this method, because of its complexity and length. Similar situations can be observed for classes PhotoController and PhotoViewController. Overall, the analysis of these two code anomalies in our example indicates the presence...
of architectural problems about responsibility overload and unwanted coupling, which contribute to architecture degradation of the system.

3. RELATED WORK

Previous work has mostly focused on investigating approaches for detecting code anomalies in different contexts. Previous work has also studied how code anomalies might be associated with architectural problems. In the following, we discuss existing contributions in the literature. Nevertheless, the state-of-art has not identified nor assessed means for prioritizing architecturally-relevant code anomalies.

**Detection of Code Anomalies.** For more than one decade, researchers have investigated approaches for detecting code anomalies [25, 11, 27, 28]. Detection strategies are mostly based on the combination of static code metrics and thresholds. Lanza and Marinescu [10] proposed the use of metrics for detecting code anomalies, reporting a detection accuracy of 60% for the anomalies investigated. Other approaches have also been proposed for detecting anomalies in Java systems, based on the analysis of structural properties of code elements [29]. Some approaches support the detection of specific types of code anomalies by examining change coupling [30, 31]. Palomba et al. [32] present an approach for detecting 5 kinds of code anomalies by mining software history. The authors report a precision over 70% for their detection strategy. Some other approaches have proposed the detection of anomalies using machine learning techniques [33, 34, 35], such as support vector machines, Bayesian networks, or tree-based classifiers. While these works exhibit different precision levels in the detection, they do not distinguish between critical code anomalies and irrelevant ones. Along this line, various prioritization strategies have emerged.

**Prioritization of Code Anomalies.** The prioritization of critical code anomalies can guide developers to solve design problems as early as possible in the lifecycle, and hence, to avoid more severe problems related with architecture degradation. The prioritization of critical anomalies is also important for increasing the effectiveness of software refactoring strategies. Some researchers have used software project repositories and explored their additional information, such as bug reports and change density [36, 14, 37]. Nevertheless, they only perform a retrospective analysis of software history data, rather than revealing anomalies related to architecture degradation symptoms.

Tsantalis and Chatzigeorgiou [38] rank refactoring suggestions to deal with code anomalies based on the analysis of past modifications. In this ranking, those refactorings whose target code was modified in the past are likely to have the highest priority. The approach generates a different ranking for each kind of code anomaly (instead of a unique ranking), which could be confusing for developers. Liu et al. [39] analyze the relationships between a set of 11 code anomalies with the goal of ordering the refactorings to be applied. That is, the prioritization of the anomalies is based on the dependencies among them. The sequence of refactorings depends on how these refactorings affect the source code and it does not consider other factors, such as the severity of code anomalies.

Marinescu [25] measures the impact of code anomalies based on three aspects, namely: (i) the negative influence of each type of anomaly on coupling, cohesion, complexity, and encapsulation; (ii) the kind of entity that the anomaly affects (such as a method or a class); and (iii) the severity of an anomaly regarding the metrics used to identify each kind of anomaly. This impact score can be used to rank anomalies.

Fontana et al. [13] propose an “intensity index” for code anomalies, which can be used to prioritize them. The index value comes from the values of all the metrics used in the detection strategy of the anomaly. In this way, for two anomaly instances of the same kind, it is possible to compute which anomaly is more critical. Based on this intensity index, Fontana and Zanoni [40] automatically classified instances of 4 kind of code anomalies using machine learning techniques. The anomalies are classified according to one of four
possible values, namely: “no-anomaly”, “non-severe anomaly”, “anomaly”, or “severe-anomaly”.

The precedent works have focused on investigating the impact of code anomalies on certain software quality attributes. However, there is no investigation about means to prioritize occurrences of anomalies with respect to their likelihood of indicating software architecture problems.

In prior work [16], we developed an approach to prioritize code anomalies using three criteria, namely: (i) the relation of anomalies with key modifiability scenarios, (ii) the importance of the kind of anomaly; and (ii) the likelihood that the source code related to an anomaly will be modified in future system versions. Although the first criterion conveys some architectural information, we have not focused specifically on design information such as: architectural concerns or components present in an architectural blueprint.

Moreover, some tools for identifying and prioritizing code anomalies (most of them emerging from the previous works) are available, for example: inFusion*, JDeodorant†, JCodeodor‡, Stench Blossom§, and JSpIRIT¶, to name a few.

**Code Anomalies and Architecture Degradation.** Recent studies [6, 2, 41, 42] investigated the negative impact of code anomalies on the software architecture. One of these studies [42] revealed that the software architecture of a larger communication system degraded over the past seven years. The authors observed the coupling between the architectural components, which were increasingly hosting code anomalies, increased over time. Moreover, architectural problems could not be identified based only on conventional source code analysis, since the architectural components were no longer aligned with the modular decomposition of the architecture. Another study [43] revealed that the architecture decomposition of Mozilla’s browser was complex and coupled, hindering system maintainability and evolvability. Developers spent around 5 years for completing a system reengineering process, which included: (i) identification of the code anomalies and the associated architectural problems; and (ii) refactoring actions on more than 2 millions lines of code.

Other studies investigated the impact of inter-related code anomalies and architecture degradation problems[44, 37, 45]. For instance, Yamashita and Moonen [44] performed an empirical investigation for analyzing occurrences of 12 code anomalies, and how their interaction can be related to maintenance problems. They analyzed the occurrences of code anomalies considering the Principal Component Analysis (PCA) as means to identify patterns of co-located code anomalies. The results revealed that inter-related code anomalies occurred across software artifacts, with a comparable negative impact as same-artifact co-location.

4. CONTROLLED EXPERIMENT: USING ARCHITECTURAL BLUEPRINTS FOR PRIORITIZING CODE ANOMALIES

In this section, we present our first study aimed at investigating how the use of architectural blueprints can help developers distinguishing the most critical code anomalies. The added value of architecture blueprints for developers is analyzed by comparing it with circumstances where developers only analyze the source code structure of the system.
Table I. GQM definition for research question 1 (RQ$_1$)

<table>
<thead>
<tr>
<th>Goal / Question / Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analyze:</strong> The role of blueprints</td>
</tr>
<tr>
<td><strong>For the purpose of:</strong> Evaluating the usefulness of architecture information</td>
</tr>
<tr>
<td><strong>With respect to:</strong> Prioritizing critical code anomalies according to their architecture relevance</td>
</tr>
<tr>
<td><strong>From the viewpoint of:</strong> Researchers and Developers</td>
</tr>
<tr>
<td><strong>In the context of:</strong> Evolving software systems</td>
</tr>
</tbody>
</table>

4.1. Research Question and Hypotheses

The prioritization of critical code anomalies can be seen as the process of distinguishing those anomalies that are relevant to the architecture design from the rest. Given the lack of empirical evidence, it is not clear whether blueprints can improve the prioritization process due to the properties discussed in the previous section (i.e. level of abstraction, consistency, and incompleteness). Along this line, our goal is stated in Table I (using the GQM format [46]) and the resulting research question RQ$_1$ is summarized as follows:

- **RQ$_1$:** Can the use of an architectural blueprint improve the prioritization of code anomalies critical to the architecture design of the system?

In order to address RQ$_1$, three different hypotheses are stated. For each hypothesis, we derived the null ($H_{N,0}$) and alternative ($H_{N,1}$) hypotheses.

Our first hypothesis $H_1$ is concerned with the impact of using architecture blueprints (in addition to the source code structure) when prioritizing critical code anomalies. The null hypothesis $H_{1,0}$ states that the use of architecture blueprints does not improve the prioritization process of critical anomalies. This improvement is measured in terms of precision when comparing the prioritized anomalies with an ideal ranking made by system experts. In turn, the alternative hypothesis $H_{1,1}$ states that the precision tends to increase when developers are provided with an architecture blueprint as an additional artifact for the prioritization process.

Our second hypothesis $H_2$ is concerned with the impact of architecture blueprints on recall. Similarly, the null hypothesis $H_{2,0}$ states that the use of architecture blueprints, as means to improve the prioritization of code anomalies, does not impact on the recall. On the other hand, the alternative hypothesis $H_{2,1}$ states that the recall tends to be higher when developers are provided with architecture blueprints on the prioritization process. Finally, our third hypothesis $H_3$ states that there is no difference in terms of time spent on prioritizing critical code anomalies when developers are provided with blueprints. Moreover, we also discuss whether the time could influence the number of false positives and false negatives observed in the prioritization process.

4.2. Experimental Steps

We performed a set of experimental procedures to conduct the controlled experiment with subjects. First, we performed a training session that presented the main concepts of the experiment to the subjects. During the training session, we introduced, for instance, the concept of code anomalies and give examples to assist subjects when reasoning over the artifacts provided for the prioritization tasks. Second, subjects were organized into two

---

*http://www.intooitus.com/products/infusion
†https://users.encs.concordia.ca/nikolaos/jdeodorant/
§http://essere.disco.unimib.it/wiki/jcodeodor
¶http://multiview.cs.pdx.edu/refactoring/smells/
¶https://sites.google.com/site/santiagoavidal/projects/jspirit
different groups: (i) Non-Blueprint (NBP), representing the subjects that only received source code information (including source code metrics) for prioritizing anomalies; and (ii) Blueprint (BP), for subjects who, in addition to source code information, received architecture information (e.g. component diagrams).

**Preparation of Artifacts.** We used the Mobile Media system as our target application in the first experimental step (more details are presented in Section 4.3). During the preparation of the artifacts, the architecture blueprints were already validated and evaluated by system experts. That is, we performed the mapping between the architectural elements (e.g. components, concerns, interfaces) and the source code (e.g. classes). This mapping helped us to ensure that the architecture blueprints had a minimum quality, in terms of level of abstraction, consistency, and completeness; so that they were useful during the prioritization process. Additionally, all groups received a document containing a partial view of Mobile Media, a brief explanation of its architectural design, as well as a description of the system concerns/responsibilities.

**Experimental Tasks.** Afterwards, we presented the sequence of tasks that should be performed by each group. We assigned the prioritization of 2 out of 3 critical code anomalies for each pair of subjects. Subjects should come up with a set of classes they judged to have critical instances of a code anomaly, as well as explain the rationale they used to achieve the resulting list. Thus, they should indicate what artifacts and metrics they used in the prioritization process. In addition, subjects were asked to order the anomalous code elements according to their architectural relevance. Finally, as a follow-up activity, we also questioned the subjects in order to understand whether and which architectural information was useful to guide the prioritization process (Section 4.6).

**Assessing Technical Knowledge (Questionnaire).** Last, a questionnaire was applied in order to measure the subjects’ technical knowledge and working experience. By gathering this information, we could better decide how the pairwise groups should be defined. More importantly, we should be able to decide which group (BP or NBP) each subject should be assigned to. Given that the technical knowledge and working experience has a wide variation, we had 27 replications for the BP group, while the NBP group had 16 replications.

The subjects were mostly graduate (master and doctorate) and undergraduate students, which certainly explains the variation of working experience and technical knowledge. For master and doctorate students with advanced technical knowledge and more than 2 years of working experience, the controlled experiment was applied individually. Therefore, we performed the assignments of participants to groups as means to ensure the knowledge of each group was leveled.

4.3. Target Application and Code Anomalies Reference List

The target application of our experiment is Mobile Media [47], a software product line that provides several features for manipulation of photos, music and videos on mobile devices. As the main reasons for the selection of Mobile Media, we can mention: (i) availability of the original developers that produced the architecture blueprints; (ii) architecture blueprints were the artifacts used to reason about changes requests and derive new products during the system evolution; (iii) this system is an open-source project of modest complexity, it is a product line with more than one hundred possible configurations for deriving products; and (iv) the system has been successfully used in other studies involving empirical evaluation [47, 48]. In addition, the evolution scenarios of the Mobile Media project range from changes in heterogeneous mobile platforms to additions of many alternatives and optional features. For the controlled experiment, we selected the release R7 of Mobile Media, because the main changes performed during the system evolution are present in this release. Release R7 also presents a more stable architecture when compared to previous releases. Another reason for (the need of) refactoring this release is that it has 54 KLOC and around 45 code anomalies, which are spread through its architectural elements (see section 6.3). More details about the system evolution and its characteristics can be found at [47].
Table II. Reference List of Code Anomalies

<table>
<thead>
<tr>
<th>Code anomalies and their Affected Classes (Mobile Media)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>God Class</strong></td>
</tr>
<tr>
<td>MediaAccessor, MediaController</td>
</tr>
<tr>
<td>** Shotgun Surgery**</td>
</tr>
<tr>
<td>AlbumController, MainUIMidlet, MediaAccessor,</td>
</tr>
<tr>
<td>MediaController, MediaListController, SmsMessaging</td>
</tr>
<tr>
<td>** Divergent Change**</td>
</tr>
<tr>
<td>ImageMediaAccessor, MediaAccessor, MediaController,</td>
</tr>
<tr>
<td>AlbumController, VideoCaptureController, MainUIMidlet,</td>
</tr>
<tr>
<td>MediaListController, MusicPlayController,</td>
</tr>
<tr>
<td>PhotoViewController, PlayVideoController,</td>
</tr>
<tr>
<td>SelectMediaController</td>
</tr>
</tbody>
</table>

Table III. Definition of Precision and Recall

(1) Precision = \( \frac{TP}{TP + FP} \)  
(2) Recall = \( \frac{TP}{TP + FN} \)

To validate the results gathered from the experiments, a reference list with the most critical code anomalies was needed. To this end, we had the collaboration of system experts, who performed a systematic analysis on the target release to identify classes affected by code anomalies they judged as critical. Table II shows the reference list of critical anomalies identified in this exercise. We asked the experts to apply their own strategy when prioritizing the code anomalies in the target application. One of the experts focused on code inspection, while another expert used a set of strategies [6], as a complementary approach to code inspection. For each type of code anomaly, the results showed that both sets of potential instances were not exactly the same. Nonetheless, around 75% of the code anomalies detected by all experts achieved the same result, and therefore, the reference list was produced as result of a joint decision.

4.4. Initial Data Analysis

In this section, we analyze the results based on the data collected through the experiments in order to assess the accuracy of using architecture blueprints in the prioritization process.

4.4.1. Measures and Data Sample  We adopted three different measures: precision, recall and time. We should also mention that precision and recall (see Table III) leverage on other basic metrics: true positives (TP), it computes the number of correctly identified code anomalies; false positives (FP), it computes the number of wrongly identified code anomalies; and false negatives (FN), it computes the number of missing code anomalies.

Precision is defined as the ratio of critical code anomalies that were correctly prioritized by the subjects. A high precision implies that a subject prioritized more critical code anomalies than non-critical ones. For example, given a set of anomalies assigned to a subject, the anomalies indicated as a priority by the subject that are also present in the reference list of critical anomalies (II), will be considered as TP. Conversely, the anomalies indicated as a priority by the subject that are not in the reference list will be considered as FP.

On the other hand, recall can be defined as the fraction of critical code anomalies prioritized by the subjects to the total number of anomalies presented in the reference list. For instance, a high recall implies that a subject prioritized most critical code anomalies.

The controlled experiment was replicated in three different institutions. The first two replicas of the controlled experiment were performed in two universities in Brazil: UFBA (8 subjects) and UFMG (42 subjects), with undergraduate and graduate students. The third replica was performed at Drexel University (16 subjects), USA, with only Master and PhD students, totaling 66 subjects. In this context, it was also possible to analyze how the
Table IV. Descriptive Statistics for Precision, Recall and Time

<table>
<thead>
<tr>
<th>CA</th>
<th>Precision (M1)</th>
<th>Recall (M2)</th>
<th>Time Spent (M3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BP</td>
<td>NBP</td>
<td>BP</td>
</tr>
<tr>
<td></td>
<td>Mean (%)</td>
<td>Median (%)</td>
<td>Std. dev.</td>
</tr>
<tr>
<td>DC</td>
<td>82.8</td>
<td>70.8</td>
<td>100</td>
</tr>
<tr>
<td>SS</td>
<td>48.8</td>
<td>54.7</td>
<td>33</td>
</tr>
<tr>
<td>GC</td>
<td>58</td>
<td>61.9</td>
<td>30</td>
</tr>
</tbody>
</table>

subjects’ technical knowledge and experience impact on the conclusions when comparing, for example, the results achieved by the undergraduate and graduate students.

4.4.2. Impact of Blueprints on Precision and Recall

In this section, we present the descriptive statistics and the hypotheses testing for the three measures collected during the controlled experiment. To verify whether the collected data is normally distributed, the Shapiro-Wilk test [49] was applied. The test revealed the collected data did not follow a normal distribution, so we applied a Mann-Whitney non-parametric method to test our hypotheses [49]. We chose the Mann-Whitney test because it is designed to perform a non-paired comparison of two independent samples. We used a confidence level of 95% (p-value = 0.05).

**Precision.** $H_1$ investigates whether higher precision can be achieved in the prioritization of critical code anomalies (CA) when subjects are provided with blueprints. First, we performed a comparative analysis of the precision achieved by the BP and NBP groups (see Table IV, measure M1). We observed that the highest precision was achieved when prioritizing the Divergent Change anomaly (DC). For this code anomaly, we observed an increase of precision of 12% in favor of the BP group when comparing the mean values. On the other hand, for the code anomalies Shotgun Surgery (SS) and God Class, the NBP group achieved better results. In these two cases, and more prominently for the God Class anomaly, we noticed that some subjects were not able to build an interpretation based on the information of the architecture blueprint.

Furthermore, some subjects reported that some code anomalies, according to their knowledge, were more difficult to detect and prioritize. For instance, we could observe different mean values related to precision when prioritizing critical code anomalies. However, most subjects had similar values of precision, which reflects in the median values for the three code anomalies investigated in the controlled experiment.

After applying the test, $H_{1,0}$ could not be rejected with p-value = 0.09. In short, the statistical results did not provide enough evidence that the use of architecture blueprints by the participants helped them to improve precision, considering these three code anomalies under investigation. This finding suggested that a deeper analysis of the mistakes made by developers should be performed, as discussed in Section 4.5.

**Recall.** $H_2$ investigates whether subjects, when provided with architecture blueprints, could achieve better recall. When analyzing the collected data (see Table IV, measure M2), we observed that, in general, the BP group achieved better results. An increase of 7% was observed when prioritizing the anomalies Divergent Changes (DC) and Shotgun Surgery (SS). In turn, better results were observed when prioritizing the God Class anomaly, since the recall increased in 20%. Even observing a better recall for the prioritization of all the three code anomalies, we still need to apply a statistical method to confirm (or refute) $H_{2,0}$.

Overall, we observed that, in average, the recall was higher when subjects were provided with architecture blueprints. This can be explained by the fact that a lower number of false negatives was observed for the BP group. The lower the number of false negatives, the higher the recall. Furthermore, when applying the statistical test, the results showed a p-value of 0.02. Thus, we can reject $H_{2,0}$. In other words, we can say that the prioritization of these three critical code anomalies can be improved in terms of recall when blueprints are employed.

Nevertheless, we observed the participants still made several mistakes with respect to the
DC and SS anomalies, as the recall was lower than 50%. Section 4.5 analyzes and discusses these mistakes in more details.

**Time Spent on the Prioritization.** $H_3$ investigates whether the use of blueprints impacts on the effort (time) spent for prioritizing critical code anomalies. For this task, we analyzed the time (in minutes) the subjects took for prioritizing each code anomaly (see Table IV, measure M3). For the Divergent Changes (DC) anomaly, the BP group spent 4 minutes less, on average, when compared to the NBP group. For the prioritization of the God Class (GC) anomaly, the difference in favor of BP group was higher. On the other hand, for the Shotgun Surgery (SS) anomaly, we observed a very small difference regarding the time spent for each group. In addition, the NBP group achieved a better time for prioritizing this anomaly. When evaluating the median values, we observed that the results are close to each other. Subjects of the BP group spent less time for prioritizing the anomalies Divergent Change and God Class, which was not the case for the Shotgun Surgery, where the NBP group spent less time.

For testing $H_3$, we applied a two-tailed Mann-Whitney U test. For each anomaly, a pair of tasks were given to the subjects so that they should be able to prioritize the code elements containing such a specific anomaly. In addition, subjects should inform the start- and end-time for each pair of tasks, in order to compute total times. When applying the test, we obtained a $p$-value = 0.8, which means that $H_{3,0}$ cannot be rejected. Therefore, we conclude that the use of architecture blueprints does not bring extra effort regarding the time subjects spent for prioritizing the code anomalies under investigation. On the other hand, we noticed that the time reduction is not necessarily high, because developers have to peruse the architectural blueprints manually. This result is a key motivation to provide tool support for exploring architectural information while prioritizing critical code anomalies.

### 4.5. Characterization of False Positives and False Negatives

This section discusses the main characteristics of classes identified as *false positives* and *false negatives*, as well as properties of the architectural blueprints used in the prioritization process. These analyses are intended to help us understanding the mistakes made by the subjects during the experiment. As previously mentioned (Section 4.2), even though the BP participants relied on additional information (i.e. architectural information), we could not observe major enhancements, particularly, with respect to precision.

Aiming to better organize the classes identified as *false positives/negatives*, we used the same package structure recovered from the Mobile Media implementation. This decision was based on the fact that each implementation package contains code elements responsible for implementing the same functionality (or having similar characteristics). In this way, we are able to identify the classes responsible for a high number of *false positives*, which directly impact on precision.

#### 4.5.1. After performing the prioritization tasks, subjects were asked to provide feedback about the usefulness of the architecture blueprint on the prioritization process. That is, they should explain the rationale used to interpret the architectural information provided in the blueprints. In addition, we should mention that the subjects mostly used the component diagrams instead of the class diagrams. According to the subjects, although a component diagram has a higher level of abstraction than a class diagram, the former captured more useful architectural information, such as the main interfaces between components, as well as the concerns each component is responsible for realizing. We decided then to analyze the properties of the blueprint used in the experiment. We specifically analyzed the architecture blueprints in terms of consistency, completeness and level of abstraction.

Given the number of inconsistencies observed in the blueprint and the number implementation elements participating in the mapping, the blueprint of Mobile Media is around 70% consistent. The consistent elements are the core elements of Mobile Media architecture. In addition, around 30% of the inconsistencies might somehow impact on
the effectiveness of the blueprints when guiding subjects on the prioritization of code anomalies.

When analyzing completeness, we observed that the only set of architectural elements that did not reach (at least) 60% of completeness were the required interfaces. Only 22 out of 40 interfaces in the architecture blueprint could be directly mapped to the source code. Other elements (18 architectural components, 23 provided interfaces and 5 concerns) were successfully and directly mapped to several elements in the source code. Overall, we can say the architecture blueprint reached a completeness of around 90% in the mapping.

Lastly, we analyzed the level of abstraction for the component diagram. When analyzing the mapping, we observed that 34 out of 50 code elements were mapped. We should mention that 8 classes are specifically implementing exception handling and 3 are utility classes. The remaining classes are related to Controller, Datamodel or Screen functionalities. Thus, we were able to identify the mapping between 18 components represented in the blueprint and 34 elements (i.e. classes, interfaces) in the system implementation.

The level of abstraction of the blueprint was calculated by considering the level of abstraction in all the architectural components and the total number of architecture elements. The level of abstraction of the component diagram is around 65%, which means that, on average, a component is mapped to 2 or 3 classes in the system implementation. More details about how those properties are calculated can be found at [15].

4.5.2. Code Anomalies and False Positives

We now discuss the proportion of false positives on the prioritization of code anomalies observed in the controlled experiment (see Table V). First, when prioritizing the God Class (GC) anomaly, the BP group identified false positives in classes contained in 5 different packages, while the NBP group identified classes in only 4 packages. The God Class anomaly presented the highest number of instances when compared to the other two code anomalies under investigation. Moreover, most classes in the Controller and Datamodel packages are responsible for more than 80% of instances of false positives.

More specifically, the results for the BP group showed that classes in the Controller package are responsible for 30.61% of false positives, while classes in the Datamodel package are responsible for 42.86%. Based on the feedback provided by the subjects, we noticed a higher number of false positives for the God Class than to any other kind of anomaly, due to difficulty of analyzing the actual relations between the architecture information and code elements.

In fact, we noticed that the NBP group did not make the same mistakes. When analyzing the false positives for the NBP, the classes contained in the Datamodel and Controller packages correspond to 80% of false positives.

4.5.3. Blueprints and False Negatives

In this section, we briefly discuss the characteristics of false negatives, which are classes that should be identified as possible candidate for having a critical code anomaly, but were not. The percentages reported on Table VI are based on the total number of false negatives when considering the three code anomalies under investigation. Surprisingly, in both groups the four most recurrent classes identified
Table VI. Instances of False Negatives by Group

<table>
<thead>
<tr>
<th>Class</th>
<th>% Instances BP</th>
<th>% Instances NBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlbumController</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>ImageMediaAccessor</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>MainUIMidlet</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>MediaAccesssor</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>MediaController</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>MediaListController</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>MusicPlayController</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>PhotoViewController</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>PlayVideoController</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>SelectMediaController</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>SmsMessaging</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>VideoCaptureController</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

as instances of false negatives are AlbumController, MediaListController, MainUIMidlet and SmsMessaging, respectively. In general, those classes are still very difficult to be detected as candidates for having any of the three code anomalies under investigation are even when subjects are provided with architecture blueprints. Each of the other classes presented in the code anomalies reference list is associated to 4 instances of false positives (8%).

4.6. Usefulness of Architecture Blueprints

After prioritizing the critical code anomalies, the subjects were asked to provide feedback about what architecture information they judged as being useful during the experimental tasks. Around 71.4% of subjects judged the provided architecture blueprints as being useful in the prioritization process. In addition, the remaining 28.6% of the subjects claimed the blueprints were not useful, so they used only the set of source code metrics for prioritizing the critical code anomalies. However, when the subjects started with the experimental tasks, they all claimed that it was difficult to determine which architectural information they should focus on for revealing the critical anomalies. In addition, they mentioned it was even harder afterwards to perform the mapping between architecture elements and classes/methods in the source code. These facts can explain many instances of the false positives and false negatives observed for the BP group.

For the subjects indicating that architecture blueprints were useful in the prioritization process, we also asked them to point out what kind of information could be added to the

Figure 3. Information suggested by the Participants
blueprint so that the prioritization process would require less effort (see Figure 3). A 54.55% subjects said that the architecture blueprints were actually complete and sufficient, so no additional information was needed. However, we noticed that these subjects also made several mistakes because they failed to establish certain relationships between specific code elements and architectural information in the blueprints.

Other participants (45.45%) mentioned they would like to rely on other types of architectural information in order to pre-select “architecturally-relevant” classes that might have a specific code anomaly. Then, their suspicion would be confirmed or rejected through the evaluation of source code metrics. A suggestion given by 18.18% of the participants was that information about classes having a high number of architecturally relevant dependencies should be provided. These participants believed this kind of information would have helped them to prioritize classes infected with the code anomalies Divergent Changes and Shotgun Surgery. A smaller fraction of participants (9.09%) suggested it should be easier to determine how fine-grained elements in the source code (e.g. attributes and methods) were related to the architecture. The same amount of participants mentioned that information about classes that changed more often should be available.

4.7. Summary

Our controlled experiment investigated how the process of prioritizing critical anomalies could be improved, when guided by architecture blueprints. Our initial results revealed that blueprints impacted both precision and recall. Even though we observed the use of blueprints improves precision in certain prioritization cases, the statistical tests only showed marginal statistical significance. An analysis of the false positives revealed participants made mistakes where there are complex relationships between the blueprint and the source code. It means a more in-depth investigation is required on this topic. We also observed that recall was more positively affected by the use of architecture blueprints. It means that subjects correctly prioritized instances of critical code anomalies, when compared to the ground truth provided by the systems experts. However, in order to provide developers with a practical solution, the prioritization process should be automated.

Based on the findings above, we argue that the prioritization of code anomalies requires some level of automation, thereby motivating the second part of our investigation (Section 5). Furthermore, subjects tended to invest a significant time when performing the prioritization of critical code anomalies. Automated criteria for supporting the prioritization process are also required because: (i) although the use of architectural information improved the prioritization results, it is not possible to know which specific architectural information should be used when performing the tasks manually; and (ii) the results so far are not much superior to not using blueprints on the prioritization of anomalies. Along this line, we proposed 3 different criteria to automate the prioritization of code anomalies. Furthermore, we also evaluated whether the proposed criteria would achieve better results in terms of precision and recall (Section 6).

5. AUTOMATING THE PRIORITIZATION OF CODE ANOMALIES: JSPIRIT

This section explores how automated support for prioritizing critical code anomalies could assist developers in this regard. In the remaining subsections, we explain how code anomalies are automatically detected (Section 5.1) and, we propose 4 criteria to distinguish which code anomalies are critical (Section 5.2).

5.1. Detection of Code Anomalies with JSpIRIT

Aiming to provide automated architecture-based criteria, we extended a tool called JSpIRIT [16] that supports the detection and prioritization of code anomalies. The tool works as an Eclipse plugin and allows developers to browse instances of different types of anomalies in
EXPLORING BLUEPRINTS FOR PRIORITIZING CRITICAL CODE ANOMALIES

Table VII. Code Anomalies currently supported by JSpIRIT

<table>
<thead>
<tr>
<th>Code anomaly</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brain Class</td>
<td>Complex class that accumulates intelligence by brain methods</td>
</tr>
<tr>
<td>Brain Method</td>
<td>Long and complex method that centralizes the intelligence of a class</td>
</tr>
<tr>
<td>Data Class</td>
<td>Class that contains data but not behavior related to the data</td>
</tr>
<tr>
<td>Dispersed Coupling</td>
<td>Method that calls one or few methods of several classes</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>Method that calls more methods of single external class that their own</td>
</tr>
<tr>
<td>God Class</td>
<td>Long and complex class that centralizes the intelligence of the system</td>
</tr>
<tr>
<td>Intensive Coupling</td>
<td>Method that calls several methods that are implemented in one or few classes</td>
</tr>
<tr>
<td>Refused Parent Bequest</td>
<td>Subclass that does not use the protected methods of its superclass</td>
</tr>
<tr>
<td>Shotgun Surgery</td>
<td>Method called by many methods that are implemented in different classes</td>
</tr>
<tr>
<td>Tradition Breaker</td>
<td>Subclass that does not specialize the superclass</td>
</tr>
</tbody>
</table>

the source code of a Java project. The tool is publicly available to be downloaded*. Internally, JSpIRIT performs a static code analysis and builds a graph of classes and their dependencies to other classes. Furthermore, for each class, JSpIRIT computes a number of traditional metrics, such as: number of lines of code, cyclomatic complexity, and number of methods, among others. Regarding prioritization, the tool comes with a few basic criteria to rank the detected anomalies.

Furthermore, another reason for choosing JSpIRIT is that developers can programmat-ically extend its object-oriented framework to provide additional strategies for detecting and prioritizing anomalies. Indeed, we relied on this feature to incorporate blueprint information and architecture-related criteria into the tool. Currently, JSpIRIT supports the identification of 10 anomalies (see Table VII), following the detection strategies presented in the catalog of Lanza and Marinescu [10]. In such detection strategies, each code anomaly is expressed as a rule combining different metrics, which have to satisfy predetermined thresholds. The threshold values were also determined by [10] based on statistical information from practical experiences.

For example, the identification of a God Class combines three metrics, namely: Weighted Method Count (WMC) to measure the complexity of the class, Access To Foreign Data (ATFD) to measure the coupling with external attributes, and Tight Class Cohesion (TCC) to measure the internal cohesion of the class. In this way, a God Class is determined by the rule:

\[
\text{GodClass} = (\text{WMC} > \text{VERYHIGH}) \land (\text{ATFD} > \text{FEW}) \land (\text{TCC} < \text{LOW})
\]

where VERY HIGH, FEW, and LOW are predefined thresholds with values 44, 2, and 0.3, as defined in [10].

At this point, we should emphasize that the definition of the detection strategies, along with the metrics/thresholds, is not a contribution of our article. We serve from a well-known set of detection strategies for predefined set of code anomalies, which have been validated in the literature [10]. In addition, if other kinds of code anomalies would need to be incorporated to JSpIRIT in the future, the current metrics/thresholds might need to be

---

*https://sites.google.com/site/santiagoavidal/projects/jspirit
revised. Therefore, different detection strategies might need to be defined, and an in-depth empirical study should be conducted (for the detection strategies themselves), which is out of the scope of our work.

5.2. Proposed Architecture-Based Criteria

Since our work focuses on architectural information (from blueprints), we defined three criteria that take advantage of specific architectural elements, namely: components, modifiability scenarios, and concerns. Figure 4 shows the rankings of code anomalies generated by JSpIRIT, and presented to the developer, using the component criterion (note that in some cases the ranking values have ties, e.g. 4-5; in these cases, the ranking position is random). A criterion can be seen as a function that returns a score value in the range [0..1] for each code anomaly. This score indicates how critical the anomaly is for the system (0=no critical, 1=very critical). Thus, the ranking is calculated based on the score given to each anomaly. For example, in Figure 4, the God Class MediaController has a score of 1 while the Brain Method PhotoViewController.handleCommand has a score of 0.93. That is, the God Class instance is more critical than the Brain Method instance. In our approach, developers can obtain a ranking of all the anomalies found using a single criterion (components, modifiability scenarios, or concerns) or they can use a voting rule as a means to combine the different criteria. Each of our criteria, as well as their rationale, is described next.

5.2.1. Component criterion

This criterion analyzes the relationship between a code anomaly and the corresponding components of the blueprint. The intuition behind this criterion is that code anomalies related with a large number of architectural components are more critical than others. To make the criterion work, the developer must load all the components specification into the tool. For instance, the developer needs to provide, for each component, its name and the classes to which it should be mapped. For doing so, JSpIRIT provides a simple interface to load the component information (see Figure 5). Thus, the automated support provided by JSpIRIT is achieved by automatically identifying and ranking the anomalies. However, the developer still has to provide the input for the criterion used (in this case, the names of the components and their code mappings).

To calculate the ranking score of each code anomaly, we analyzed all the classes affected by the anomaly. The set of classes affected by an anomaly are determined, for each type of code anomaly, by taking into account those classes that should be refactored to fix it (the worst case is assumed). For example, classes affected with an instance(s) of a Feature Envy anomaly are: (i) the main class in which the method of the anomaly is implemented and (ii) the classes that invoke and are invoked by the method. Once the affected classes are
determined, we count the number of components of the blueprint related with the affected classes. This number of components will be the score of the anomaly.

In order to obtain a value between 0 and 1, we normalize the score values. For example, let us consider the Feature Envy in method `getImageFromRecordStore` (from Mobile Media), as shown in Figure 6. The classes affected by the anomaly are: `ImageAlbumData` (the main class of the anomaly), `MediaData`, `ImageMediaAccessor`, and `MediaAccessor`. These four classes are (partial) realizations of two components: `AlbumPhotoData` (mapped to `MediaData` and `ImageAlbumData`) and `ImageAccessor` (mapped to `MediaAccessor` and `ImageMediaAccessor`). For this reason, the score value for the Feature Envy instance, according to this criterion is 2. To normalize this value, it is divided by the code anomaly with the highest score. In
5.2.2. **Concern criterion** This criterion analyzes the relationship between a given code anomaly and an architectural concern represented in the blueprint. Recall from Section 2.2 that a concern represents some important part of the problem (or domain) that the architects want to treat in a modular way at the architecture specification. The rationale behind this criterion is that a code anomaly affecting several concerns can lead to (future) architectural problems, and thus, should be brought to the developer's attention. Note that, although concerns are allocated to architectural components, the organization of concerns is not necessarily aligned with that of the components within a blueprint.

JSpIRIT offers a simple interface to load concerns, similarly to the one described for architectural components. Specifically, the developer must provide a concern name and select the system packages/classes the concern maps to. To calculate the ranking score of a code anomaly, we count the number of concerns it affects. At last, we normalize these values to obtain scores between [0...1]. For example, Figure 7 shows a Brain Method in class **AlbumController** and method **handleCommand** (from Mobile Media). The main class of this anomaly (**AlbumController**) is affected by 3 concerns, out of 7 defined in the application. Since the highest number of concerns related with a code anomaly is 5, the normalized score value for this anomaly is 3/5=0.6.

5.2.3. **Scenario criterion** This criterion analyzes the relationship of code anomalies with modifiability scenarios. A modifiability scenario describes a change-related property that is desirable in a system [16]. In Mobile Media, an example of a modifiability scenario is the following statement: "A developer wishes to add a new kind of screen for a given service (e.g., for displaying a picture of a video). This change should be made in the code at design time, and it should take less than one hour to make and test the change, with no side-effect changes other than the new screen in the behavior". That is, scenarios capture specific kinds of changes that the system must support. In terms of the blueprint, a scenario affects certain architectural components that are key for fulfilling the scenario. Thus, the rationale of this criterion is that those anomalies related with scenarios can be critical because they can directly compromise modifiability properties of the application.

Scenarios can be modeled as particular types of architectural concerns. Like in the case of concerns, each scenario is mapped to different packages/classes of a system using JSpIRIT. In particular, the importance of the scenarios is determined by the architect/developer, either based on domain knowledge or information from system requirements. Therefore, a different importance can be given to the scenarios, by assigning values between 0 and 1 to each scenario stating how critical its satisfaction is. For example, if the developer thinks that the satisfaction of the scenario is critical for the system, he/she will indicate an importance.
To calculate the ranking score of a code anomaly, we analyze if the main class of the code anomaly is affecting a scenario. This happens when the scenario is mapped to the code anomaly main class. In case that a code anomaly affects one or more scenarios, we sum the importance value of each scenario. We repeat this step for each code anomaly. Once this value was calculated for all the anomalies, we normalize the values. More formally, the normalized score is calculated as follows:

$$\text{Score}(\text{codeAnomaly}) = \sum \text{ImportanceValue(affectedScenario)}$$

$$\text{NormalizedScore}(\text{codeAnomaly}) = \frac{\text{Score}(\text{codeAnomaly})}{\text{highest Score for an anomaly}}$$

For example, let us consider the Dispersed Coupling anomaly found in class `BaseController` and method `goToPreviousScreen` (from MobileMedia), as depicted in Figure 8. In this case, `BaseController` is the main class of the anomaly and it is referenced by only one scenario with importance of 0.7. For this reason, the score value of the anomaly according to this criterion is 0.7. If we assume that the anomaly being most affected by scenarios has a score of 2.1, the normalized value of this Dispersed Coupling is $0.7/2.1=0.33$. In case this anomaly would be also related to another scenario with importance 0.2, the score value of the Dispersed Coupling instance would be $0.7+0.2=0.9$, with a normalized score value will be $0.9/2.1=0.43$.

5.2.4. Voting criterion The criteria described so far only consider a specific kind of information (e.g., components, scenarios, or concerns) for the prioritization. In this context, we also studied a combined criterion that implements a voting rule based on the three previous criteria. Since we are interested in filtering the most critical code anomalies from the rest, we defined individual thresholds for each of the criteria above. When the normalized score of a code anomaly is higher than its threshold, we consider the anomaly as potentially critical. The specific values of these thresholds can be configured by the developer in JSpIRIT. The values of the thresholds are project-specific. For example, consider a project with 100 code anomalies in which, after applying the scenario criterion, the top 10 anomalies have values between 0.6 and 0.8 and the rest of the anomalies have values between 0.5 and 0.59. In this case, it does not make sense to select a threshold of 0.5 for the scenario criterion because it will cover all the anomalies found. However, 0.5 could be an acceptable value for another project. For this reason, JSpIRIT automatically suggests these thresholds to the developer by calculating the third quartile (or upper quartile) of the ranking values for a criterion.

The voting criterion essentially seeks to reinforce the notion of criticality. That is, if a given code anomaly gets critical under at least two of the three individual criteria, then the anomaly is marked as critical by the voting rule. In this case, the final score of the anomaly will be the average of the scores of the individual criteria over their thresholds. In case an anomaly is not identified as critical by the voting criterion, its score value will be 0.
Table VIII. GQM definition for research question 2 \((RQ_2)\)

<table>
<thead>
<tr>
<th>Goal / Question / Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analyze:</strong> The use of architecture information</td>
</tr>
<tr>
<td><strong>For the purpose of:</strong> Evaluating improvements provided by automated support</td>
</tr>
<tr>
<td><strong>With respect to:</strong> Prioritizing critical code anomalies according to 4 different criteria</td>
</tr>
<tr>
<td><strong>From the viewpoint of:</strong> Researchers and Developers</td>
</tr>
<tr>
<td><strong>In the context of:</strong> Two medium-size applications</td>
</tr>
</tbody>
</table>

Example, consider a code anomaly A whose score values are 0.6, 0.3, and 0.1 according to the component, concern and scenario criteria, respectively. If we assume the thresholds values were 0.4, 0.2 and 0.3 for such criteria, then the voting criterion will identify the code anomaly A as being critical anomaly, because it was already identified as potentially critical by the component \((0.6 > 0.4)\) and concern \((0.3 > 0.2)\) criteria. The final score of this code anomaly for the voting criterion will be \((0.6+0.3)/2=0.45\).

6. EMPIRICAL EVALUATION: ASSESSING THE ARCHITECTURE-BASED PRIORITIZATION CRITERIA

In this section, we present the empirical evaluation performed in the second part of our study, which explores how the use of architectural blueprints, with automated support, can help developers to distinguishing the most critical code anomalies. Since we are not constrained by the number of subjects and time limits imposed by the controlled experiment of Section 4, we conducted a broader exploratory analysis than in that experiment. Specifically, we use a broader set of types of code anomalies and also analyze all instances of anomalies found in two applications. Our research question is described in Section 6.1. The experimental procedures for our experiment are introduced in Section 6.2. Section 6.3 presents the target applications and lists their corresponding code anomalies. Section 6.4 presents an initial data analysis for the two applications, in the context of our research question.

6.1. Research Question

The main motivation of our second research question is to understand whether the automated support to the prioritization would reveal better results when compared to the results observed in the first part of our study (see Section 4). Specifically, our goal was to check if we can obtain precision improvements. In addition, we wanted to explore which kind of architecture information could bring benefits in terms of increasing the accuracy in the prioritization process. As means to clarify the goal of this investigation, we also used the GQM format (see Table VIII). Along this line, our second research question can be summarized as:

- \(RQ_2\): Does the use of architecture information, with automated support, help developers to improve the prioritization of critical code anomalies?

6.2. Experimental Procedures

We performed a set of procedures in order to conduct our second experiment. First, given a target application, we automatically detected its code anomalies using JSpIRIT. The metric thresholds were set for each application in such a way the number of false positives is minimized. Second, using each of our criteria, we ranked the code anomalies according to their score values. These two activities were executed by JSpIRIT in less than a minute...
for each application. In order to find the most critical classes related to the anomalies, we need to translate the ranking of anomalies (produced by the tool) to a ranking of classes.

The translation was done in 3 steps, namely: (i) we selected the subset of code anomalies whose score value is over a given threshold (e.g. threshold=0.5 would select all the anomalies whose score value is greater than 0.5); (ii) we replaced the code anomaly by its main class; (iii) we removed any repeated class from the list. Once we obtained the ranking of classes, we compared it against a reference ranking created by application experts to compute different metrics. In order to analyze the behavior of the criteria with different thresholds, we used different configurations of thresholds ranging from 0 to 0.9.

For example, let us consider the case presented in Figure 9, in which 6 code anomalies were detected and ranked using one of our criteria. Assuming a predefined threshold of 0.3, we retain only those anomalies whose score value is higher than 0.3, and a subset with the first four anomalies is obtained. Since the code anomalies ranked in positions 1 and 3 have the same main class, the final ranking of potential critical classes includes 3 classes, namely: A, B, and C.

### 6.3. Target Applications and Reference List of Code Anomalies

We selected two medium-size applications for this study. One of those systems is Mobile Media, which was described in Section 4.3. Our second application is SubscriberDB [16], which is a software product 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. We selected version 2.4 as it encompasses all the features implemented in the system.

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 (from 54 and 100 KLOC) are manageable for an in-depth analysis of code anomalies analysis; (ii) the applications have

---

**Table IX. Characteristics of Target Applications**

<table>
<thead>
<tr>
<th>Target Applications</th>
<th>Mobile Media</th>
<th>SubscriberDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Type</td>
<td>Software Product</td>
<td>Web</td>
</tr>
<tr>
<td>Programming Language</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td>Architecture Design</td>
<td>MVC</td>
<td>MVC</td>
</tr>
<tr>
<td>Selected Version</td>
<td>5</td>
<td>2.4</td>
</tr>
<tr>
<td>KLOC</td>
<td>54</td>
<td>100</td>
</tr>
<tr>
<td>Number of Architectural Elements</td>
<td>81</td>
<td>42</td>
</tr>
<tr>
<td>(Components and interfaces, concerns)</td>
<td>45</td>
<td>84</td>
</tr>
<tr>
<td>Number of Code Anomalies</td>
<td>90%</td>
<td>95%</td>
</tr>
<tr>
<td>Completeness (blueprint)</td>
<td>65%</td>
<td>82%</td>
</tr>
<tr>
<td>Consistency (blueprint)</td>
<td>70%</td>
<td>77%</td>
</tr>
</tbody>
</table>
Table X. Definition of f-measure

| f-measure = 2 * \(\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}\) |

been extensively evaluated in other studies [50, 51, 45, 41]; (iii) we needed to rely on the availability of the system’s developers to validate our identification of code anomalies; and (iv) the architecture blueprints, used to reason about change requests and new versions, were available for both applications. Table IX summarizes the main characteristics of Mobile Media and SubscriberDB. As previously mentioned, we used well-known strategies defined in other studies [10] for detecting code anomalies, which are implemented in the JSpIRIT tool (see Section 5).

The list of code anomalies outputted by the detection strategies was checked and validated by experts of each target application. The validation process is important to guarantee that the detection strategies actually revealed anomalous code elements. In MobileMedia, almost 50% of the classes were affected by at least one code anomaly, whereas in SubscriberDB around 40% of the classes are affected by at least one code anomaly. For the validation, we relied on the reference list of anomalies (ground truth) provided by the experts. The ground truth consists of a list of the most critical, anomalous code elements, according to their relevance to the architecture design of the system. Once the ground truth was provided for each target application (in the case of Mobile Media, the ground truth was the same list reported in Section 4.3), we are able to make comparisons with the critical code anomalies returned by each of the prioritization criteria.

We should mention that the system experts (i.e., developers and maintainer) only participated in the process of producing the ground truth required to compare the JSpIRIT results. These experts were asked to reason about the most critical code elements. For example, these code elements can be either classes realizing important architectural components, or classes responsible for implementing the key interfaces defined in the software architecture. As a result of this activity, developers of Mobile Media provided the top 10 critical classes, whereas developers of SubscriberDB provided a list with the top 12 critical classes. Most of these classes were part of the code anomalies detected by JSpIRIT and later validated by the experts. In both case-studies, the critical classes were 19-17% of the total system classes, and represented 35-38% of the main classes of the code anomalies.

6.4. Initial Data Analysis

In order to analyze the results of the criteria, we used three standard metrics: precision, recall, and f-measure. The first two metrics were computed as explained in Section 4.4. F-measure was computed as a harmonic mean of precision and recall (Table X) [52]. The f-measure is a standard metric for systems in which both precision and recall are considered. We report f-measure here to facilitate the analysis of the results since we experimented with several thresholds configurations. First, we analyzed values of precision, recall and f-measure for the complete set of code anomalies found in the applications (that is, the main classes of every anomaly). Our intention was to create a “baseline” for comparison purposes. That is, we want to apply our criteria to filter the most critical classes, compared to the whole list of anomalies identified by JSpIRIT. For this reason, the baseline values of precision, recall, and f-measure are calculated from all the anomalies found in a system. Note that this calculation differs from the controlled experiment presented on Section 4, in which precision and recall were computed based on the values of a control (subject-based) group. Thus, while our criteria could match the recall of the baseline, these values cannot be improved. This is because the number of critical classes is determined by the code anomalies that are automatically detected by JSpIRIT. Then, the anomalies are filtered by the criteria, which could increase the precision and decrease the recall.
6.4.1. Results for Mobile Media  In the case of Mobile Media, we obtained a precision of 0.35, a recall of 0.9, and an f-measure of 0.5 for the baseline. Since 10 classes were identified as critical in this application, these results mean that: (i) 9 of these classes were main classes of at least one of the anomalies (recall), and (ii) only the 35% of the anomalies have a critical class as its main class (precision).

After applying our criteria to filter the most important code anomalies, we found several improvements in the precision values. Figure 10 shows the precision (10a), recall (10b), and f-measure (10c) values of the scenarios, concerns and component criteria for different thresholds values (the dotted line depicts the reference value given by the baseline). Remember that the threshold in the X axes means that we are building the ranking of critical classes from those code anomalies with a score value higher than the threshold (see Section 6.2).

Figure 10. Results of automated criteria for Mobile Media
The highest values of f-measure (0.67) were obtained by the concerns criterion when the criterion value was higher than 0 and 0.1, and also by the component criterion when the anomalies with scores higher than 0.5 and 0.6 were considered. For example, in the case of the component criterion, when the threshold is 0.6, the precision is equal to 1 (meaning that all the retrieved anomalies have a critical class as main class), and the recall is 0.5 (meaning that half of the critical classes are found by the retrieved anomalies). If we compare both precision and recall (i.e., f-measure) of the three criteria, we found that the component criterion was the one with the best performance. This is so because this criterion obtained higher f-measure values than the other two criteria in 7 out of 10 threshold configurations. Moreover, 6 of these 7 values were higher than 0.5, which is the f-measure value calculated for the complete set of code anomalies. Also, it is interesting to note that the component criterion had better f-measure results with “intermediate” thresholds values. Intermediate thresholds values could be helpful to filter a number of not critical anomalies at the same time that they keep most critical ones.

Regarding precision, we noticed that the scenario-based criterion reaches a precision of 1 when the threshold is equal or higher than 0.4. This happened because only 8 anomalies (from a total of 45 automatically detected by JSpIRIT) have a score value higher than 0.4. In addition, these 8 anomalies have as main classes only 3 different classes (e.g. the first four anomalies have the same main class), which were identified as critical by the experts. For this reason, the criterion obtained low values of recall. In this case only 3 of 10 critical classes were identified. A similar situation occurs with the components criterion, where the results reached a “perfect” precision when the threshold is equal or higher than 0.6, while at the same time the recall drops steadily. Only 6 code anomalies had a score value higher than 0.6 for this criterion, and they relate to 5 different main classes. Overall, the average precision (for all anomalies) was 0.84, 0.65 and 0.75 for the scenarios, concerns and components criteria, respectively. When considering only the configurations with a recall equal or higher than 0.4, the average precision gets in general lower, with value of 0.44, 0.61, and 0.69 for the three criteria.

Regarding the performance of the voting criterion, its analysis is more complex because the results involved the threshold values given for the three individual criteria. In general, we found that from 1000 possible combinations of these thresholds (this is why Figure 10 does not show the results of this criterion), 532 combinations obtained better f-measure values than the one calculated for the complete set of code anomalies. However, we obtained the best f-measure results (ranging from 0.7 to 0.82) for low values of the concern criterion (0 to 0.1). and scenarios and component thresholds ranging from 0.1 to 0.6 and from 0.2 to 0.6, respectively. These values are higher than the ones obtained individually for each criterion. Since the thresholds values are generally low, this trend could indicate that combining criteria is a good approach to prioritize critical code anomalies. In other words, the low threshold values indicate that if an anomaly is related, for example, with at least one concern and one scenario, or one concern and one component, then the main class of the anomaly turns out to be critical.

6.4.2. Results for SubscriberDB When it comes to SubscriberDB, we obtained similar results as in Mobile Media. We analyzed the baseline for the complete set of code anomalies found in SubscriberDB. In this case, the precision was 0.37, the recall 0.92 and the f-measure 0.52. After applying our criteria, we found improvements regarding the baseline values for both precision and f-measure. Figure 11, shows the precision (11a), recall (11b), and f-measure (11c) values of the criteria for the threshold values.

While the precision values improved with any configuration (compared to the baseline), the improvements in f-measure are mainly concentrated in low threshold values (0 to 0.3). The reason is that applying the criteria rapidly reduces the number of critical classes found in the retrieved code anomalies (recall). In other words, the classes identified as critical by the criteria are not always the same identified as critical by the experts. For example, the
Figure 11. Results of automated criteria for SubscriberDB

precision value of the scenarios criterion was 1 regardless of the threshold, because only 27% of the code anomalies (23 of 84) have a score value higher than 0 for this criterion. That is, only 23 anomalies are at least related with a modifiability scenario. These 23 anomalies are related with only 5 classes that, of course, are critical classes. It is also interesting to analyze the case of the concerns criterion where the precision is 1 with any threshold value being equal or higher than 0.5. Only 4 code anomalies have a score value higher than 0.5 and all of them have the same main class. This is also the reason why this criterion has such low recall values for thresholds greater than 0.5. Thus, only 1 critical class was detected from 12 identified by the experts. This sudden decrease in recall is also reflected in the f-measure (11c). In general, the average precision (for all anomalies) was 1, 0.78 and 0.65 for the scenarios, concerns and components criteria. Considering only the configurations with a recall equal or higher than 0.4, the average precision was 1, 0.55, and 0.51 for the three criteria, respectively.
Regarding the voting criterion, the sets of code anomalies that improved the SubscriberDB baseline were fewer than the ones of MobileMedia. In SubscriberDB, from 1000 possible combinations of threshold values, only 294 combinations obtained better f-measure values than the baseline. However, these f-measure values are, in general, higher than the ones obtained individually for each criterion. Specifically, the best results ranged between 0.63 and 0.7, and they were obtained for thresholds ranging from 0 to 0.5 for the scenarios criterion, from 0 to 0.9 for concerns criterion, and from 0 to 0.1 for the blueprint criterion. Since the thresholds of the concern criterion were so widely spread, we can infer that the most critical classes are found by applying a combination of criteria based on scenarios and components.

6.4.3. Accuracy of the Automated Prioritization Criteria Taking into account the results of both applications, we can answer RQ2 by saying that the use of architecture information, with automated support, showed considerable improvements in precision to identify critical classes linked to code anomalies. For a precision baseline of ~0.35, we obtained precision values of up to 0.65 with the component and concern criteria used in isolation (considering a recall equal or greater than 0.4). However, as it was expected, the recall tends to decrease for high threshold values as the precision increases. Furthermore, in the case of the voting criterion, we observed that the combination of the scenario criterion with either the component or the concern criteria can lead to a good prioritization in terms of precision.

Along this line, the precision and f-measure results observed in the experiment encouraged the use of the automated approach over a manual approach. Specifically, we achieved better precision values than those of the baseline, in spite of analyzing a larger number of anomaly types, as well as analyzing a larger number of anomaly instances than in the controlled experiment of Section 4. Moreover, using the automated approach we took into account blueprint information (such as concerns and modifiability scenarios) that goes beyond architectural components, which would have been difficult to consider in a manual approach. For instance, developers can do reasonably well when (manually) analyzing certain architectural components vis-à-vis with the code, but if they would have to cover more artifacts in such an analysis, the task would be much more time-consuming and error-prone.

6.4.4. Investigated Code Anomalies We analyzed a sample of code anomalies in our study. The subset of code anomaly types we considered is not different from what previous experimental studies did in the field. According to the literature of empirical software engineering (e.g. [49]), each experiment should not last longer than one hour. Nevertheless, we focused on a subset of code anomaly types that were usually correlated with architectural problems [3], such as God Classes, Shotgun Surgery and Divergent Change. These anomaly types tend to involve many code elements and, therefore, are likely to be related to architecture problems. In any case, our proposed automated approach supports a significant list of code anomaly types.

Moreover, we used Marinescu’s catalog in our work, as this is a well-known model of types of code anomalies and is also equipped with detection rules for the anomalies (which were implemented as part of JSpIRIT). Certainly, not all the anomalies of Marinescu’s catalog were present in our case studies. On the other hand, the subsets of code anomalies employed in each experiment were different (although overlapping) due to the characteristics of each experiment.

6.4.5. Practicality of the Proposed Approach All the steps of our approach do not require significant manual efforts, including the generation of the architecture-implementation mappings. Moreover, we only relied on blueprints that were already defined and available in those projects. Based on the results of the second experiment, the main benefit was about savings in the time spent for performing the analysis of the code anomalies when using...
JSpIRIT. This is so because a developer requires less manual inspections of the source code vis-à-vis with the architecture blueprint in order to flat relevant anomalies. Although we did not replicate the first experiment using JSpIRIT in order to assess time reductions, we informally compared the times of Table IV against the performance of JSpIRIT, which normally takes less than 1 minute to run its prioritization criteria.

In addition, our results were validated with 2 different systems, showing that the automated approach is practical. Admittedly, there are some manual tasks involved in JSpIRIT regarding the provision of “good” mappings between blueprint and code. Certainly, the outputs of the criteria depend on the quality of those mappings. Further discussion about these aspects is in the Threats to Validity section.

Furthermore, we would like to reinforce the fact that the sample of code anomalies selected for the experiments is representative of real medium-size systems, which allows us to conjecture that our approach is applicable to other systems (beyond MobileMedia and SubscriberDB). Therefore, the our approach is very practical because: (i) there is no need for producing additional artifacts, (ii) the major steps of the approach can be automated, (iii) we reduce the effort of developers as they need to spend less time on ranking architecturally-relevant code anomalies, and (iv) our approach leads to precision improvements.

7. THREATS TO VALIDITY

This section describes the threats to be considered in our work. Thus, we have organized this section according to 4 different threats to validity: internal, external, construct and conclusion.

**Internal Validity.** The internal threat is associated to the mapping between elements in both levels of abstraction: architecture blueprints and source code. Aiming to mitigate this threat, we validated the mappings for Mobile Media and SubscriberDB with the system experts (e.g. architects, maintainer, developers). However, we should point out that, as a software system evolves, it is hard to synchronize changes in the architecture with the system implementation. In addition, we validated with experts all the responsibilities, concerns, scenarios, and architectural components realized by the code elements in the different system versions. Furthermore, complementary approaches exist that already address the problem of automated mappings (of architectural concerns in the source code) with high accuracy [53, 54], such as Mallet and XScan. These tools can be easily integrated with JSpIRIT as future work.

**External Validity.** The first threat is associated with possible errors in the detection of anomalies. In order to minimize the risk of imprecision in such a process, we involved the original developers and architects. Also, we employed well-known metrics and thresholds for the detection strategies, which were implemented in the JSpIRIT tool. The second threat is associated with the use of the ground truth of code anomalies. Each system expert used their own strategies to identify the most critical anomalies. For instance, in Mobile Media around 75% of code anomalies were equally identified. In this sense, the final ground truth of code anomalies was produced as a joint decision. A similar process took place for SubscriberDB, where the ground truth of code anomalies was validated and refined with the help of system experts. Another related threat is the fact that we only used a subset of 10 code anomalies. Other types of anomalies exist, and they were out of the scope of our study. However, we think that the analyzed anomalies cover a wide range of possible problems.

**Construct Validity.** A first threat can be associated with the subjects of the controlled experiment. In particular, the way these subjects tried to figure out what the goal of an experiment was, and their understanding about what they thought was being studied, might had affected their responses during the experiment and thus might bias the results. In order to mitigate this threat, we performed a training section to explain the main goal of the controlled experiment, as well as how subjects should use the software artifacts on the prioritization tasks. Despite this training section, some subjects might have not
been able to correctly analyze/interpret all the information provided in the controlled experiment. As reported in Section 4, for those cases, the prioritization was impaired in terms of the precision achieved by the subjects when prioritizing critical code anomalies. A second threat refers to the technical knowledge of the subjects when using all the provided information in the prioritization tasks. Aiming to mitigate this threat, we asked the subjects to fill a technical questionnaire, as a means to balance the pairs of subjects that should be in each experimental group.

**Conclusion Validity.** The first threat is associated to the choice of the target applications. We know that a higher number of target applications are needed in order to generalize the results reported in this article. One of the main problems is that the information required to conduct this type of investigation can be difficult to obtain. In order to minimize this threat, we selected systems developed by different programmers, and with different domains. Moreover, we tried to make our best to describe carefully the different parts of our experiments/study, so that others can replicate it with other software systems. The second threat comes from selecting only applications implemented in Java. Further investigations are still required in order to figure out whether our findings hold for systems implemented in other languages (e.g., C++, C#).

8. **FINAL REMARKS AND FUTURE WORK**

In this article, we performed an empirical investigation to: (i) evaluate the impact of using architecture blueprints on the process of prioritizing critical code anomalies; and (ii) provide evidence that an automated process can lead to better prioritization results, for example, in terms of precision in the ranking of code anomalies. Along this line, our work was organized into two different steps, as depicted in Figure 12. In a first step, we performed a controlled experiment to investigate the impact of using architecture information, represented as blueprints, on the prioritization of critical code anomalies. Given this context, we used three different measures (precision, recall and time) to quantify potential benefits of using architectural blueprints by the subjects.

Our initial findings showed that, to some extent, the architecture blueprints can improve both precision and recall. For example, we observed high precision on the prioritization of Divergent Change and Shotgun Surgery anomalies. However, in the case of God Class anomalies, the results were not conclusive and the architecture blueprints did not substantially improve the prioritization process. For the cases where blueprints were helpful, the results showed an improvement of up to 20%.

On the other hand, recall increased for the three anomalies in the group that used blueprints. Independently of whether the subjects used the architecture blueprint correctly, the results showed that this artifact reduced the occurrences of *false positives* and *false negatives*. In the statistical tests for recall, we were able to confirm the hypothesis that the use of blueprints improves the prioritization process, when compared to the sole use of metric-based strategies. Finally, we evaluated the time spent by the subjects when prioritizing the
three code anomalies under investigation. The results showed a slight difference (not higher than 10 minutes) when both groups performed the prioritization. We could not corroborate that architecture blueprints would bring any additional effort in terms of time spent on the prioritization. Therefore, this finding motivated the proposition of our automated approach for prioritizing architecturally-relevant code anomalies.

In the second step, we investigated whether developers, when provided with tool support, would better prioritize critical code anomalies. To do so, we proposed three different criteria for prioritizing anomalies, as well as a voting criterion that combines information from the individual criteria. In addition to precision and recall, we used the f-measure metric to evaluate the performance of the four criteria. The automation of the prioritization process had positive results in terms of precision and f-measure, when analyzing a larger number of anomaly types and anomaly instances than in the first study. If we take into account the results of both applications (Mobile Media and SubscriberDB), we can say that the main improvements were in the precision of both the component and concern criteria for identifying critical system classes. In such cases, the precision results doubled the baseline values of the applications. However, as it was expected, the recall decreased with high threshold values, as the precision increased. Moreover, our experiments with the voting criterion revealed that some combinations of different types of architecture information (e.g., components, concerns, scenarios) can enhance the prioritization in terms of precision and recall, when compared to using any of the three criteria in isolation.

From a practitioners’ perspective, JSpIRIT can be applied in small- to medium-size Java projects, provided that the developer is willing to invest some efforts in the mapping of architectural information (e.g., concerns, components, scenarios) to package and classes in the implementation. In our experience, entering mappings at the package level or for particular classes is straightforward using the GUIs provided by JSpIRIT. For instance, we applied JSpIRIT to an industrial project with an 83 KLOC code base and around 800 code anomalies, and the mapping task took less than 1 hour (including the configuration of the prioritization criteria). Nonetheless, if the set of mappings is detailed the current GUIs are not very user-friendly for the task. A similar situation happens if the developer needs to modify the existing mappings at a later time (e.g., because of code changes). We expect to improve this functionality with some kind of wizard able to assist developer in the completion of the mappings, once a small set is defined, and also to maintain its consistency over time. Although it is not a definitive factor, having a satisfactory accuracy/granularity in the blueprint mapping is required for JSpIRIT to rank the anomalies with good precision. The default configuration of anomaly detection rules suffices in most settings, but the developer can still tune (easily) the thresholds of the rules for the different types of anomalies, if necessary in the project under analysis. A recent evaluation of JSpIRIT (without using architectural information) as part of a comparison of smell detection tools is reported in [55]. In fact, MobileMedia is one of the systems addressed in the evaluation. The authors highlight some usability features of JSpIRIT, but they also mention that there is no support for filtering particular smells in the rankings (e.g., in the case of false positives) or for visualizations. Currently, we have developed a basic visualization that helps developers to identify the parts of a system being most affected by anomalies.

As future work, we will further evaluate our existing criteria and explore extensions to them. In particular, we aim at defining criteria for prioritizing groups of inter-related code anomalies that might be better indicators of architectural problems. Specifically, we are interested in comparing the anomalies identified as critical by JSpIRIT with other tools for ranking anomalies, such as JCodeOdor [13]. Also, we will investigate improvements to the ranking process. In particular, we are interested in new strategies to generate the ranking based on multiple criteria. In addition, we will continue developing JSpIRIT and including new features. For instance, we plan to provide developers with support for different detection strategies, and allow them to customize those strategies by using different
thresholds. Another interesting feature is the provision of visualization capabilities (e.g., heat maps) for code anomalies and related architectural problems. Furthermore, we would like to investigate a “direct” approach for identifying architectural smells in an automated fashion [56] (or rely on tools such as Hotspot Detector or Arcan), and assess their pros and cons when compared to the “indirect” approach of JSpIRIT driven by code anomalies.

REFERENCES


