Architecture-Sensitive Heuristics for Prioritizing Critical Code Anomalies

Everton Guimarães, Alessandro Garcia
Informatics Department
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
{eguimaraes, afgarcia}@inf.puc-rio.br

Yuanfang Cai
Department of Computer Science
Drexel University
Philadelphia, USA
yfcai@cs.drexel.edu

Abstract

The progressive insertion of code anomalies in evolving software systems might lead to architecture degradation symptoms. Code anomalies are particularly harmful when they contribute to the architecture degradation. Although several approaches have been proposed aiming to detect anomalies in the source code, most of them fail to assist developers when prioritizing code anomalies critical to the architectural design. Blueprints of the architecture design are artifacts often available in industry software projects. However, such blueprints are rarely explored to support the prioritization of code anomalies in terms of their architecture relevance. This paper proposes and evaluates 2 sets of blueprint-based heuristics for supporting the prioritization of critical code anomalies. The prioritization is based on their potential impact on revealing architectural drift problems. The heuristics allow developers prioritizing critical code anomalies by exploiting architectural information provided in the blueprint. The contributions of this paper include: (i) a set of architecture sensitive heuristics to support developers when prioritizing critical code anomalies; (ii) an evaluation of the proposed heuristics in terms of their prioritization accuracy in 3 systems; and (iii) an empirical analysis on how the blueprints' information might enhance the prioritization of critical code anomalies, as opposed to existing heuristics strictly based on source code analysis.

Categories and Subject Descriptors
D.2.10 [**Design**]: Quality Analysis and Evaluation, K.6.3 [**Software Management**]: Software Development,

Keywords Component; Code Anomalies; Blueprints; Software Architecture; Heuristics; Empirical Evaluation

1. Introduction

As the software system evolves, inappropriate changes in its implementation can lead to software architecture degradation. Architecture degradation [1] represents the progressive introduction of architectural problems in a software system. The actual software architecture of a system in realized by its implementation. Therefore, when architecture problems are not properly addressed in the source code, the continuity of the software project can be irreversibly compromised. However, it is hard to identify which program elements are realizing architecture problems.

Recent studies [3][5] revealed one of the main factors, which impact on software architecture degradation, is the presence of code anomalies – also popularly referred as *code smells* [2]. Code anomalies can be defined as symptoms in the source code that might indicate an architectural design problem [8]. In turn, an architectural problem is an anomalous structure affecting one or more components of software architecture. An example of architectural problem is the so-called concern overload [8], i.e. the occurrence of a single architecture component realizing several concerns. The aforementioned studies [3][5] revealed there is an intimate relationship between most of the architectural problems and code anomalies. They found around 80% of architectural design problems are related with the presence of well-known code anomalies [2], such as *God Class* and *Feature Envy*.

However, given the wide scope of an architectural problem [8][9], it tends to manifest in the implementation as a set of inter-related (rather than single) code anomalies [3][5][10]. Several studies [5][11][12] confirmed co-occurrences of code anomalies are effective indicators of architectural degradation symptoms. On the other hand, the occurrence of single anomalies in an isolated way is often not an indicator of an architecture design problem. In fact, only less than 40% of individual code anomalies present some relation to architectural problems [8]. Consequently, a high proportion of code anomaly instances, detected solely using automated source-code analysis [6][7], are not enough to support the location of architecture degradation symptoms [1][3][4] (Section 2). In other words, it is hard to locating the wide structure of an architectural problem in the source code.

When a code anomaly is related with an architectural problem, we state this code anomaly is *critical* to the software architecture design – we also use the term *architecturally-relevant code anomaly*. When developers perform architecture reviews of the source code, they need to prioritize critical code anomalies. Otherwise, developers tend to spend more time addressing problems that are not harmful to the architectural design. Prioritization can be understood as the process of distinguishing those code anomalies that are relevant to the architecture design from those that are not. Software developers should be provided with systematic means for prioritizing the most critical anomalies.

MODULARITY'15, March 16–19, 2015, Fort Collins, CO, USA
Copyright 2015 ACM 978-1-4503-3249-1/15/03...$15.00
http://dx.doi.org/10.1145/2724525.2724567

In this paper, we claim techniques for prioritizing code anomalies should consider information of architectural blueprints, typically available in industry software projects [26]. Blueprints [19] are herein defined as informal models, with high level of abstraction, usually created for communicating developers about the key architecture design decisions of a software system (Section 2.2). The use of blueprints would facilitate to determine the strength of the relationship between code anomalies and architectural design problems. These relationships would improve in turn the process of prioritizing code anomalies. In this context, we propose and evaluate two sets of heuristics that exploit architectural information represented in blueprints - we call them *architecture sensitive heuristics*. The goal is to improve the prioritization of critical code anomalies using architecture blueprints, typically archived and used in industry software projects [26]. As these critical anomalies are often related in the source code, we hypothesize that architecture blueprints can help to capture such anomalies' relationships (Section 3). The prioritization heuristics (Section 4) allow developers defining which anomalous code elements (e.g. classes, interfaces) are inter-related in different ways.

Our initial results (Section 5) demonstrated that, in average, the proposed heuristics were able to prioritize and rank, at least, 60% of critical code anomalies considering all the target applications under assessment. Therefore, we expect the heuristics can provide developers with means to improve the process of prioritizing critical code anomalies. When the prioritization process succeeds, architectural problems can be avoided, thereby preventing architecture degradation. As main contributions of this work (Section 6), we can mention:

- A set of heuristics to support developers when prioritizing critical code anomalies according to their architecture relevance.
- An evaluation of the proposed heuristics and indicators of their accuracy when supporting developers on the prioritization process; and
- An analysis on how the architecture information available on blueprints is relevant on the identification of critical code anomalies.

2. Background

This section discusses the key concepts, including the impact of code anomalies in the architecture design of software systems, as well as the limitations of existing detection and prioritization mechanisms. It also shows a running example for illustration the different types of architecture problems identified by the prioritization heuristic.

2.1 Detection and Prioritization of Critical Code Anomalies

Impact of Code Anomalies. Recent studies investigated the negative impact of code anomalies in the system's software architecture. One of these studies [13] shows how the architecture modularity of a larger communication system has degraded over the past seven years. Authors observed the coupling between the system's architecture components, which were increasingly hosting code anomalies, also increased over time. The authors also observed this architectural problem could not be revealed based only on conventional source code analysis, as those components were no longer aligned with the modular decomposition of the software architecture. Another study [15] reported the architecture decomposition of Mozilla's browser was overmuch complex and coupled, which hindered the system maintainability and evolvability. Developers spent around 5 years for completing all the reengineering process, which includes: (i) identification of the code anomalies associated with the architectural problems; and (ii) refactoring actions on more than 2 millions lines of code [23].

Detection of Code Anomalies. The most popular mechanisms for (semi-)automatic detection of code anomalies are based on the use of software metrics [6][7]. Developers are able to define their own detection strategies by relying on the use of a particular combination of metrics and thresholds. However, the main limitation of such detection strategies is they only explore automatically-collected measures from the source code. Hence, they are frequently agnostic to the design decisions performed by the system architect. As architecture decomposition is not explicit in the implementation, they fail to assist developers on the prioritization process [3][5]. That is, most of them disregard architecture information that could be exploited in combination with the source code measures. In addition, such detection strategies only consider individual occurrences of code anomalies, instead of analyzing the relationships between them. Most of the architectural problems are realized by several anomalies in the program. Thus, current detection strategies are not able to support developers when prioritizing critical code anomalies [4][5].

Prioritization and Ranking Systems. There are two main approaches that provide explicit capabilities for prioritizing critical code anomalies. The first approach is implemented by the InFusion tool, which can be used for analyzing Java, C and C++ systems. Besides the statistical analysis for calculating more than 60 code metrics, the tool provides numerical implementation-based scores to detect code anomalies. Those scores provide means to measure the negative impact of code anomalies in the software system. When combining the scores, a deficit index is calculated for the entire system – considering different source code metrics, such as code complexity, module coupling and cohesion metrics. The second approach is realized by the JSpIRIT tool [22], which supports semi-automated refactoring of the most critical design problems in a software system. The tool suggests a ranked list of the most critical code anomalies – based on a combination of different criteria – and intends to reduce the time spent on code anomaly prioritization. The main concern of using these tools is that their underlying techniques suffer from the same limitations: (i) it only considers the source code structure as input for detecting and prioritizing code anomalies; (ii) the prioritization system disregards how the architecture decomposition is realized by the code elements; and (iii) the user cannot define or customize their own criteria for prioritizing code anomalies.

2.2 Architecture Blueprints

The concept of architecture blueprint can be defined as a high-level model that represents the overall structure of a software system [19]. In addition, architecture blueprints are mainly used for communicating developers about the key architectural design decisions. Software developers are able to represent architecture design in blueprints with different levels of detail [19]. However, it is rarely the case there is a direct unitary mapping between architecture and source code elements. Mapping can be understood as the process of determining, for instance, which code elements are responsible for realizing each architecture component represented in the blueprint.

In this work, we consider a simplification of the UML component diagram [18] to illustrate examples of architecture blueprint. Figure 1 depicts a blueprint capturing a partial view of the main components of the Mobile Media architecture. The Mobile Media is a software product line that provides support for

manipulation of media on mobile devices 0. Each component is represented by a rectangle. A component realizes provided and required interfaces, which are respectively represented by white circles and arcs connected to each other. In addition, the notation allows representing the concerns realized by each component. The association of an architectural component and its concerns is represented by decorating the component with small dark grey circles. Each of these circles represents a particular concern realized by the architectural component (Section 4).

Furthermore, an architecture blueprint is characterized by three properties, which are used to check whether an architecture design model would fit in the definition of blueprint defined in this paper. When those properties cannot be identified in a model, the blueprint will not convey the minimum architectural information required for computing our proposed heuristics (Section 4). Each of those properties is briefly described in the following.

Level of Abstraction. Level of abstraction (LoA) denotes "how much far" the architecture blueprint is from the architecture implementation. In order to calculate the LoA of an architectural component, it is required to compute the number of code elements realizing it. The computation of the level of abstraction of an architecture blueprint (LoA_B) requires two basic procedures: (i) compute the total number of elements represented therein 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. Thus, we can quantify the LoA not only for each component, but also for the entire set of architectural components (and other elements) represented in the blueprint. Moreover, the level of abstraction must assume a value $0 < LoA \leq 1$, once the elements in the blueprint should be mapped to at least one source code element.

Completeness. A blueprint is considered complete when it characterizes the components involved in the representation of the system's descriptive architecture [9]. For each architectural component represented in the blueprint, there must be at least one corresponding code element in its counterpart program. When the mapping between architecture and code elements is performed, each architectural component must be associated with at least one code element (e.g. a class) responsible for realizing it. To measure the completeness of an architecture blueprint, we have to consider 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: $C_B = 1 - (AC_{NM} / T_{AE})$, where AC_{NM} is the number of elements not mapped and T_{AE} is the total number of architectural elements.

Consistency. A *blueprint* is said fully consistent when it does not present any contradiction in the information represented on the mapping between architecture and source code elements. All cases of inconsistencies of the architecture blueprint considered in this paper are computed based on 5 measures: (i) dependencies not mapped – i.e. dependencies that exist between code elements realizing different components, but they were not prescribed in the architecture specification.; (ii) inverted dependency – i.e. there is a dependency from component A to component B represented in the architecture blueprint. However, when looking to the source code elements responsible for realizing those components, the dependency occurs in the inverse way; (iii) component with no interface; and (iv) different interfaces or components with the same name. Thus, the inconsistency of a blueprint is quantified by computing the total number of inconsistencies for each of the cases above. If a model is fully inconsistent, it does not serve the purpose of our study.

2.3 Running Example

This section presents the running example, which will serve the purpose of illustrating how architecture information can be used for enhancing the prioritization of critical code anomalies. Figure 1 depicts a blueprint representing a partial view of the Mobile Media architecture. In addition, it highlights in red color some architecture degradation symptoms. When analyzing the blueprint of Mobile Media, we could observe situations where architectural problems might be associated with anomalous code elements responsible for realizing specific components. For instance, some code elements realizing the PhotoController component might implement concerns (e.g. Counting and Persistence) that it should not be implementing – since other components are responsible for realizing it. This situation characterizes the problem of *Misplaced Concern* (Section 4.2.2) in the architecture decomposition.

For this scenario, the *PhotoController* component is realizing several concerns, such as *counting* (C), *features* (F), *exception handling* (E) and *persistence* (P). In particular, this component certainly should not be responsible to address the *persistence* concern, since the AlbumData and ImageAccessor components are the ones created with the purpose of realizing it. The PhotoController component suffers from *Concern Overload*, which also affects the maintainability of its interdependent components. Another example of *Concern Overload* is located in the PhotoListController component. The inner classes realizing this single component have to handle many different concerns. Those classes are instances of the God Class code anomaly as they are addressing multiple concerns in their code body. As illustrated in the blueprint, code elements implementing the component PhotoListController are forced to deal with many different concerns. Thus, these anomalous code elements are contributing to the violation of two design principles, i.e. separation of concerns and single responsibility [21].

Moreover, we observed the most critical code anomalies – and the architecture problem counterpart – tend to affect the communication between architectural components. For instance, in the Mobile Media's blueprint, we can observe an architectural problem characterized as *External Attractor* component (Section 4.1.2). In this scenario, the component AlbumData provides a single interface, which is used by many other components, including PhotoListController, BaseController, AlbumController, PhotoViewController and PhotoController. For the best of our knowledge, the AlbumData component represents the different abstractions of data recorded in the storage device. That is, the controlled classes are not aware on how the data is stored, because they only know the data media albums. On the other hand, components realizing any *controller* functionality are responsible for providing a controlled access to other system functionalities.

In this scenario, the client components are "addicted" to its service. Particularly, we also observe the occurrence of an architectural problem documented in the literature, called *Overused Interface* [8]. This architectural problem occurs when external interfaces require a lot of data from a single interface [8], which might indicate the provided interface is being used with too many purposes.

3. Study Settings

This section describes the research question addressed in this paper, as well as the study hypotheses defined in our investigation. In addition, we describe the target applications used for assessing the prioritization heuristics proposed in this study.

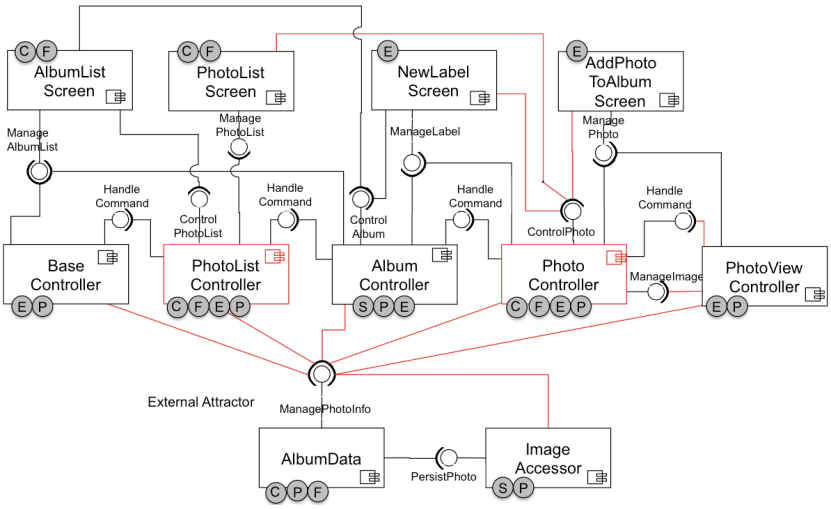

Figure 1. A Blueprint of Mobile Media Architecture

3.1 Research Question and Study Hypotheses

To conduct our investigation and evaluate the proposed heuristics, we firstly defined the research question to be addressed, as well as the study hypotheses under test. Our research question (RQ) aims at investigating:

RQ. *To what extent critical code anomalies are accurately prioritized with the blueprint-based heuristics?*

The expectation is that the proposed heuristics can assist developers on the prioritization of critical code anomalies, therefore facilitating the location of architectural degradation symptoms. However, the answer to this research question is far from being obvious. As mentioned in Section, 2.2, architecture blueprints are very often high-level, incomplete and inconsistent with respect to the system's descriptive architecture [9]. Consequently, their use in the heuristics might lead to inaccurate prioritization results. To make clear the purpose of this study, Table 1 defines it using the GQM format [15].

Table 1 – Study Definition using GQM Format

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

As observed in Table 2, we proposed and tested two sets of heuristics (see Section 4). Our main goal is to evaluate the accuracy of the proposed heuristics for prioritizing critical code anomalies based on their architecture relevance. In this sense, Table 2 summarizes the study hypotheses required for testing the accuracy of the proposed heuristics. In order to analyze the accuracy (acc) of the proposed heuristics on the prioritization process, three different values are considered: low ($0 > acc < 30\%$), acceptable ($30 \geq acc < 80\%$) and high ($80 > acc \leq 100\%$). The thresholds are similar to those commonly adopted when performing experimentations in software engineering [15]. Thus, we analyzed three levels of accuracy aiming to investigate to what extent the proposed heuristics would assist developers in the prioritization process. For instance, an accuracy level of 50% means the heuristic should be able to correctly prioritize at least half of the critical code anomalies.

Our study focused on architectural drift problems [9]. We investigate two categories of architectural problems: (i) problems directly detected by the proposed heuristics, which might be related to bad design practices on the implementation of architecturally-relevant concerns (see Section 4.2), and (ii) problems with inter-component communication (see Section 4.1). In addition, we focused on problems already catalogued in the literature [8]. In total, we investigated 4 types of architecture problems observed in the target applications, namely *Ambiguous Interface*, *Connector Envy*, *Overused Interface* and *Scattered Parasitic Functionality*. The latter falls in category (i), while the others fall in category (ii). However, the problems in category (i) can also be a sign of poor modularization of architectural concerns. Further details about those architectural problems can be found in the literature [8].

Table 2 – Study Hypotheses

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

3.2 Target Applications

Aiming to evaluate the proposed heuristics, we selected 3 medium-size target applications. The first application is the Mobile Media (MM) [14], which have been previously introduced in Section 2.2. The second application is the Health Watcher (HW) [28], which is a real web-based application that allows citizens to register complaints about health issues in public institutions. We selected the last version of both applications because it comprises many changes performed during the system evolution. In addition, the last version of both systems realizes a more stable architecture. For instance, in Health Watcher the changes range from functionality increments and enhancements on error handling policies to the incorporation of design patterns as means to improve the system modularity. Our third application is the Subscribers DB [22], which is large software of a publishing house. The Subscribers DB system manages data related with the subscribers of its publications and it supports

complex queries on several types of data. We selected version 2.4 since it encompasses all the features implemented in the system, as well as it has a more stable version of the system architecture. Table 3 summarizes the main characteristics of the all target applications evaluated in our study.

Furthermore, we should mention these systems were selected because they met a number of relevant criteria for our study: (i) the target applications are representative of several types of medium to large-size applications (varying from 54 to 100 KLOC). Thus, those applications are manageable for an in-depth analysis of code anomalies as required in our study; (ii) the applications have been extensively and successfully evaluated in other studies [4]00; (iii) we needed to rely on the availability of the system's experts to validate the detection of code anomalies instances; and (iv) the blueprints, used to reason about changes requests and guide the software evolution, were available for the target applications.

Table 3 – Characteristics of the Target Applications

Target Application	MM	HW	S_{DB}
System Type	SPL	Web	Web
Programming Language	Java	Java	Java
Architecture Design	MVC	Layers	MVC
Selected Version	5	8	2.4
KLOC	54	49	100
# Architectural Elements	81	48	42
# of Code Anomalies	260	497	582

4. Architecture Sensitive Heuristics

This section introduces the proposed heuristics for supporting developers on the prioritization of critical code anomalies according to their architecture relevance. Our goal was not to have a *complete* set of heuristics – i.e. a set of heuristics that prioritizes *all* instances of code anomalies relevant to the software architecture design. Instead, we aim at conceiving heuristics that creates a list of the most critical code anomalies given a set of well-known architectural problems [8]. The former goal is certainly part of a much longer-term research project, as software systems would suffer from different architectural problems. We should mention the heuristics have been chosen based on the "patterns of code anomalies" documented at [11][30]. The results showed certain critical code anomalies frequently co-occur and are associated mainly with different drift problems. In this select, we decided to focus on two critical patterns of code anomalies [30] associated to problems in the communication between architectural components (Section 4.1), as well as problems associated with the implementation of concerns (Section 4.2).

4.1 Inter-Component Heuristics

Our first set of heuristics exploits information about two or more occurrences of code anomalies affecting the communication between architectural components. The investigation of co-occurrences of these code anomalies is likely to affect software maintenance, as they are spread through inter-connected architectural elements. Occurrences of inter-related code anomalies might also be associated with the violation of design principles in the architecture decomposition, such as the *Interface Segregation Principle* and *Single Responsibility Principle* [21].

To execute the proposed heuristics, a set of software artifacts might be required as input: (i) a set of source code metrics [24][25]; (ii) architecture blueprints representing the overall structure of the software system (e.g. components and interfaces); and (iii) mapping between artifacts of both levels of abstraction - the architectural components should already have been mapped to the corresponding source code elements. The mapping process is out of the scope of our research. This is a topic of research on its own, and there are many approaches in the state of the art that support this process (e.g. [27][28][29][30]) with 90%-100% of accuracy for architecturally-relevant concerns [27][30].

4.1.1 Heuristic for External Attractor Components

The first heuristic identifies code elements implementing a given component, whose provided interface is used by too many external anomalous code elements. This scenario characterizes an occurrence of the architectural problem called *External Attractor* component. A code element is considered external if it is located in another architectural component than the one under analysis. Moreover, the heuristic also helps identifying components are affected by other architectural problems documented in the literature [8]. For instance, occurrences of *External Attractor* component are often associated with an architectural problem called *Overused Interface* [8].

Code elements realizing the overused (provided) interface are accessed by its client code. When code elements are accessed by external code elements, this situation might favor the insertion of code anomalies in those architectural components. Thus, when an accessed code element implements many different concerns, its client components are forced to realize the concerns they should not be addressing. In this context, the *Interface Separation Principle* [21] is neglected, and therefore, the internal complexity of the architectural component is increased. Consequently, the maintainability of the provided interface overused by other external components decreases. Whenever a code element in the interface needs to be changed, the client's component might also be updated.

Definition 1.1. The set of occurrences of the *External Attractor (*EAt) component in a system *S* is denoted by EAt_S. Considering an architectural component $AC_i \in AC_S$ *(*set of architectural components in the system S*)*, a code element $CE_j \in CE_{ACi}$ (set of anomalous code elements in the architectural component AC_i), a set of architectural components $AC_{i+1} \in AC_S$ and a set of code elements $CE_{j+1} \in CE_{ACj+1}$, the formal definition of EAt_S is:

- $EAt_S = \{CE_j \cup CE_{j+1} \mid (CE_j, CE_{j+1}) \in D\ (CE_j, CEj_{+1})$ AND $|CE_j| > th1$ AND $|AC_i| > th2\}$, where:
- $D(CE_j, CE_{j+1})$ represents a dependency from the code element CE_j to the code element CE_{j+1}.

The generic thresholds *th1* and *th2* can be chosen depending on the characteristics of the software system under analysis and the design decisions defined by the software architect.

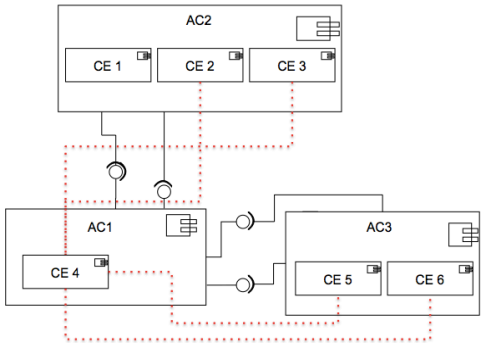

Figure 2. External Attractor component

Abstract Example. Figure 2 illustrates an abstract representation of the *External Attractor* component. As we can observe, three architectural components (AC1, AC2 and AC3) are defined in the blueprint. The architectural component AC1 has an anomalous code element CE4, which is accessed by other code elements – CE2, CE3, CE5 and CE6 – belonging to the components AC2 and AC3.

The interfaces provided by AC1 are being overused by code elements realizing AC2 and AC3. Therefore, the code element CE4 participates in the occurrence of an architectural problem, called *Overused Interface*, since it exposed methods called by many classes. Thus, its implementation is neglecting the *Single Responsibility Principle*. In particular, the public methods of the provided interface defined in this component are called by different client classes. This behavior might indicate the inappropriate declaration of those methods. The code elements realizing the components AC2 and AC3 also suffer from the *Long Method* code anomaly. Those code elements might also be affected by several changes due to modifications performed in the code element CE4.

Concrete Examples. Now we discuss instances of architecture problems and the code anomaly counterparts prioritized with the proposed heuristic. We illustrate the manifestation of *External Attractor* in Figure 1, which represents the Mobile Media architecture. An analysis of this case reveals occurrences of *External Attractor* problem are mostly related to instances of the code anomalies "*Data class"*. These classes are accessed by anomalous code elements located externally to the *External Attractor* component. Thus, occurrences of such "*Data classes*" are directly harmful to the architectural design. They are frequently under maintenance, and hence, all their client classes (located in other components) suffer from many changes. Nonetheless, occurrences of *External Attractor* problem might favor the insertion of code anomalies in the classes of every client component. For example, several client components of AlbumData, hosting the *External Attractor* problem, are forced to deal with these concerns even when they are not interested in them. Thus, occurrences of the *External Attractor* component reduces the maintainability of the dependent components because whenever the server code element needs to be changed, the client components might also need to be updated.

On the other hand, when analyzing the Health Watcher system (Section 3.2) we found the most critical case of *External Attractor* is located in the IFacade interface of the GUI component. This interface is also affected by another architectural problem, called *Overused Interface*. That is, the interface is large and exposes a non-cohesive set of methods called by many external code elements – which indicate an inappropriate declaration of those semantically-disjoint methods in a single interface. The external classes Complaint, HealthUnit and Employee, which are responsible for realizing the architectural component DataManager, are infected by the code anomaly *Long Method*. Thus, it deals with different types of information exposed by the IFacade interface (e.g. Persistence and Transaction). In fact, those classes are always densely affected by several changes occurring in the IFacade interface. All code anomalies comprising the cases of *External Attractor* are prioritized with the proposed heuristic.

4.1.2 Heuristic for External Addictor Components

The second heuristic investigates occurrences of code elements implementing a given architectural component, which has several dependencies with code elements realizing external components. This situation characterizes an occurrence of *External Addictor* component, which is associated with the violation of the *Single Responsibility Principle* and *Interface Segregation Principle* [21]. A code element is considered to be external when it is located in other architectural component than the one under evaluation. Moreover, the heuristic assists developers on the identification of anomalous code elements addicted to other code elements realizing external components functionalities.

Definition 1.2. The set of occurrences of the *External Addictor* (EAd) in a system S is denoted by EAd_S. Considering an architectural component $AC_i \in AC_S$ (set of architectural components in the system S), a code element $CE_j \in ACE_{ACi}$ (set of anomalous code elements in the architectural component AC_i), a set of architectural component $AC_{i+1} \in AC_S$ and a set of anomalous code elements $CE_{j+1} \in ACE_{ACj+1}$, the formal definition of EAd_S is:

- $EAd_S = \{CE_j \cup CE_{j+1} \mid (CE_j, CE_{j+1}) \in D\ (CE_j, CE_{j+1})$ AND $|CE_{j+1}| > th_1$ AND $|AC_{i+1}| > th_2\}$, where,
- D (CE_j, CE_{j+1}) represents a dependency from the code element CE_j to the code element CE_{j+1}.

Similarly to the definition of the *External Addictor component, th1* and *th2* represent thresholds that can be chosen depending on the characteristics of the software system under analysis and the design decisions defined by the software architect.

Abstract Example. An abstract representation characterizing an *External Addictor component* (EAd) is illustrated in Figure 3. The architectural component AC2 have an anomalous code element CE4, which access several code elements (CE2, CE3, CE5 and CE6) belonging to the other 2 external components AC1 and AC3. The architectural component AC2 is more interested in accessing the interfaces provided by other architectural components (AC1 and AC3) than realizing the functionality it was initially designed to accomplish. Moreover, components AC1 and AC3 are implemented by the anomalous code elements CE2, CE3 CE5 and CE6, which in turn, are infected with *DataClass* anomaly [2]. Thus, the code element CE4 is infected by the *God Class* anomaly [2], and hence, it defines several non-cohesive methods. Consequently, the code element CE4 propagates several concerns that should be treated internally. The propagation of concerns forces the code element CE4 to deal with those concerns that are not properly addressed.

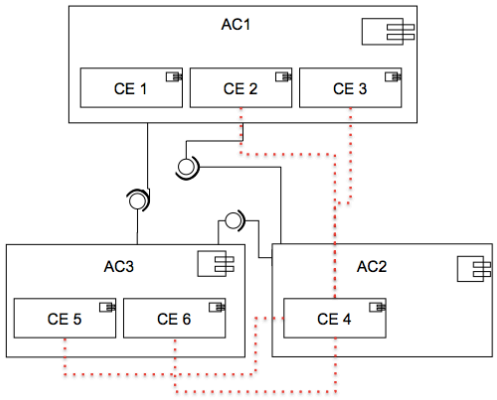

Figure 3. External Addictor component

Concrete Examples. The heuristic above helps developers detecting scenarios where there is a tight coupling between the anomalous code elements realizing a give architectural component (e.g. one or more elements centralize the communication between its own component and the adjacent ones). In other words, the heuristic helps to prioritize anomalous code elements that realize occurrences of *External Addictor* components in the target applications. We have previously illustrated an instance of *External Addictor* in the Mobile Media (Section 3.2). Nonetheless, the heuristic have also been able to prioritize code elements associated with the most critical instance of *External Addictor* in Health Watcher (Section 3.2).

As previously mentioned, the IFacade interface hosts critical code anomalies because it defines a significant number of non-cohesive methods – i.e. it is affected by the *God Class* and *Long Method* anomalies. Moreover, the IFacade breaks encapsulation of the component as it propagates several exceptions that should be

addressed internally (or, at least, should be remapped to other exception types). Therefore, classes such as *InsertHealthUnit*, which depends on the IFacade interface, need to address concerns associated with *Persistence* and *Transaction* exceptions. The problem occurs because those classes should not be in charge of performing persistence and transaction-related actions. The most critical scenario would be whether the implementation of those concerns violates design decisions initially defined by the system's architect (e.g. architectural layers that should not communicate with each other).

4.2 Concern-Based Heuristics

The second set of heuristics is focused on revealing anomalous implementation of architectural concerns. An architectural concern is defined as an architect's interest that significantly influences the architecture design decisions. The concerns can be modularized in one or more architectural components. Thus, the proposed heuristics investigate sources of critical code anomalies that violate the principle of *Separation of Concerns* [20]. The maintenance of each component violating this principle is also impaired as it is responsible for implementing several architectural concerns. When code elements realize many architectural concerns, they might lead to tight coupling, which is a factor that delimitates the prioritization of specific anomalous code elements.

4.2.1 Heuristic for Concern Overload in Components

The third heuristic aims at identifying architectural components realizing many different architectural concerns. In particular, the heuristic investigates instances of anomalous code elements that: (i) realize the same architectural component, and (ii) modularize several independent concerns. Architectural concerns are considered independent when each of them should be modularized by different components. In this context, the instances of Concern Overload might imply on a violation of two the principle: *Separation of Concerns* [20] and *Single Responsibility Principle* [21]. When an architectural component implements several concerns, it centralizes more than it should actually implement. Thus, the component will present architecture degradation symptoms and its maintainability can be compromised.

For detecting instances of *Concern Overload*, we first identify all the architecture concerns realized by each architectural component using existing techniques to this purpose (e.g. [27][28]). In addition, we can check and validate this information with the system's expert. After this step, the heuristic validates whether the number of concerns modularized by a component respect the thresholds defined by the system architect. If the component violates the threshold, the anomalous code elements within this component must be characterized as an instance of *Concern Overload*. We should also mention that verification of this architectural problem is performed for all the components represented in the blueprint.

Definition 1.3. The set of occurrences of *Concern Overload* (CnO) in a system S is denoted by CnO_S. Before formally defining the CnO_S, we should have the list of concerns realized by each code element in a software system. Therefore, the list of all concerns in the system S (CnO_S) realized by a code element $CE_{j,c} \in CE_S$ (where CE_S represents all the code elements realized in the system S). Thus, considering an architectural component $AC_i \in AC_S$ (set of architectural components in the system S) and a code element $CE_j \in CE_S$, the formal definition for instances of CnO is represented as:

- $CnO_S = \{CE_j \mid CE_j \in CE_{ACi}$ AND $\mid CnO_S(CE_j) \mid > th1$ AND $\mid CE_j \mid > th2\}$, where:
- CE_{ACi} represents all the anomalous code elements realizing the architectural component AC_i.

Moreover, the threshold *th1* represents the maximum number of concerns that a given code element should realize. On the other hand, the threshold *th2* represents the number of code elements that should be considered in an occurrence of *Concern Overload*.

Abstract Example. Figure 4 illustrates an abstract example of a component suffering from *Concern Overload*. As we can observe, the architectural component AC1 is realized by four different code elements CE1, CE2, CE3 and CE4. Those code elements are responsible for realizing tree different concerns in a given software system. The code elements CE3 and CE4 are each responsible for realizing a specific concern. In turn, code elements CE1 and CE2 are responsible for modularizing 3 different concerns, which should be addressed only by the code elements CE3 and CE4. In this context, CE1 and CE2 are dealing with several concerns, and therefore, the principles of *Separation of Concerns* and *Single Responsibility* are being violated. This scenario characterizes an occurrence of *Concern Overload* in the component AC1, and more specifically, in the code elements CE1 and CE2.

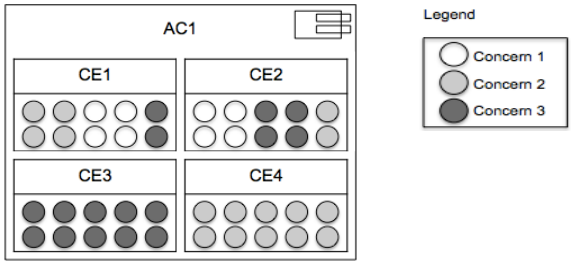

Figure 4. Architectural Component with Concern Overload

Concrete Examples. As observed in the running example (see Figure 1), the components PhotoController and PhotoListController suffer from *Concern Overload*. Those architectural components are also infected with an architectural problem called *Scattered Parasitic Functionality* [8]. Occurrences of this architectural problem imply multiple components are responsible for realizing the same architectural concern. In addition, some architectural components might be responsible for independent concerns. In particular, the architectural concerns, *Exception Handling* and *Persistence*, are scattered across several components, including the ones mentioned above. We should mention at least one architectural component modularizes more than one independent concern. Thus, when anomalous code elements in the same component realize independent concerns, it might indicate the architectural component is not cohesive. Consequently, the architectural component can be decomposed into smaller ones that are more cohesive than the original structure. In summary, code anomalies comprising *Concern Overload* are prioritized by the heuristic when the architectural component realizes more concerns that it should do, thereby harming the component maintainability.

4.2.2 Heuristic for Misplaced Concerns

This heuristic investigates occurrences of anomalous code elements responsible for modularizing an architectural concern that is not the predominant one of their enclosing component. A concern is considered predominant whether most of the code elements within the architectural component are dedicated to modularize it. Instances of *Misplaced Concern* might violate either *Separation of Concerns* or *Single Responsibility* principles. The fact is the dispersed anomalous code elements reify the scattering of a concern in the architectural design. In addition, violations to the *Separation of Concerns* principle affect the system maintainability since changes in specific concerns can spread over many other components.

Definition 1.4. The set of occurrences of *Misplaced Concern* (MC) in a system S is denoted by MC_S. Consider an architecture concern $CN_Z \in CN_S$ (it represents the list of all concerns realized in a system S*)*, and two different architectural components $AC_i \in AC_S$ and $AC_{i+1} \in AC_S$ (set of all architectural components in a system S). The formal definition for instances of MC is represented as:

- $MC_S = \{CE_j \mid CE_j \in CE_{ACi,\ COz}$ AND $|CE_{ACi,\ COz}| < th1$ AND $|CE_{ACi+1,COz}| > th2\}$, where:
- $CE_{ACi,\ COz}$ represents a code element responsible for realizing the architectural component AC_i and implementing the system concern CO_z.
- $CE_{ACi+1,COz}$ represents a code element responsible for realizing the architectural component AC_{i+1} and implementing the system concern CO_z.

That is, this heuristic verifies whether code elements (CE_{ACi} and CE_{ACi+1}), belonging to different components (AC), are responsible for implementing the same concern CO_z - which should predominantly be implemented by only one of the architectural components. Thus, the thresholds *th1* and *th2* represent, respectively, the number of concerns realized in a software system and the maximum number of concerns a given code element should realize. In summary, those values indicate the acceptable measures of which the system concerns are scattered. In this sense, the thresholds th_1 and th_2 must respect the values $0 \leq th1 \leq 1$ and $0 \leq th2 \leq 1$, respectively.

Abstract Example. Figure 5 provides an abstract representation of a component suffering from *Misplaced Concern*. As observed, code elements realizing the architectural components AC1, AC3 and AC4 are responsible to deal with only one specific architectural concern, which is predominant for each of these components. Moreover, the architectural component AC2 should be implemented by code elements responsible for addressing only one specific concern. Nevertheless, code elements CE3 and CE4 are forced to deal with two different concerns, which should be implemented by the external code elements CE1 and CE2. Therefore, the situation characterizes an occurrence of *Misplaced Concern* in CE3 and CE4, which are responsible for realizing the component AC2.

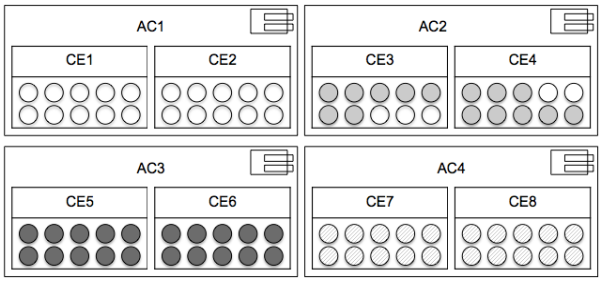

Figure 5. Architectural Component with Misplaced Concern

Concrete Examples. As previously mentioned, the architectural components PhotoController and PhotoViewController implement many different and independent concerns. Those components are responsible for addressing the Exception Handling (EH) concern. However, the EH concern is not predominant in both components, and even worse, it is scattered across many other components in the Mobile Media architecture. Therefore, anomalous code elements realizing those components - which participate in occurrences of *Misplaced Concern* - should be defined in another component. In this context, the proposed heuristic prioritizes each of these code anomalies realizing parts of the *Misplaced Concern*.

In addition, instances of *Misplaced Concern* can be more remarkable when other components modularize the concern that is misplaced. In other words, disperse anomalous code elements might favor the scattering of concerns in the architectural design. Thus, they all neglect the principle of *Separation of Concerns*, and consequently, they will often affect the architecture maintainability – e.g. changes in specific concerns might affect other components in the architecture.

5. Evaluating Architecture Sensitive Heuristics

This section evaluates the proposed heuristics for prioritizing critical code anomalies (see Section 5.2 and 5.3). Before evaluating the heuristics, we firstly described how the detection of individual instances code anomalies was performed.

5.1 Preparing the Study Environment

In order to execute the proposed heuristics, we need first to perform a set of tasks in order to prepare the study environment. As the prioritization heuristics required a set of architecture and source code information, all the systems are required to have a minimum set of artifacts: (i) architecture blueprints; (ii) architecture specification with the description of the components, as well as the concerns realized in the system; and (iii) source code and quality metrics. In order to apply the proposed heuristics, the target application should also have a stable version released. Given this context, we have organized the study into 3 steps.

Step1 - Mapping Between Architecture and Source Code. This steps consists on establishing the correspondence between architecture and the source code elements. For doing so, we first evaluate whether the architecture design satisfy the properties defined in Section 2.2, so that it can be characterized as an blueprint. Aiming to allow developers to trace the mapping between elements in both levels of abstraction, we used the ConcernMapper tool [29]. Thus, the mapping between the architecture information (e.g. components, interfaces and concerns) and source code elements are performed in a semi-automated way. Before applying the proposed heuristics, each component represented in the blueprint should be mapped to the corresponding code elements responsible for realizing it. Even when the architecture *blueprint* is incomplete, all the components must be mapped to at least one element in the system implementation. The mapping process has been performed before the detection and prioritization process take place.

Step 2 - Detection Strategies and *Ground Truth* of Code Anomalies. We used well-known strategies and thresholds defined in other studies [3][5] for detecting code anomalies. Those detection strategies are implemented by SCOOP [3][4]. It is not the purpose of this paper evaluating these strategies for detecting individual code anomalies. Moreover, we investigated types of code anomalies already catalogued in the literature and extensively investigated in other studies [3][5]. Our study included the most critical code anomalies found in the three target applications [2], namely: Divergent Change, Shotgun Surgery, Duplicated Code, God Class, Feature Envy, Data Class, and Deep Inheritance Tree.

The list of code anomalies provided by the detection strategies was checked and validated by experts of each target application. The validation process is important to guarantee the detection strategies have, in fact, revealed anomalous code elements. For validating the list of code anomalies, we relied on the *ground truth* provided by the experts. The *ground truth* consists on a list of the most critical anomalous code elements, according to their architectural relevance, provided by developers of each target application. Thus, we are able to compare with the top-*k* code anomalies provided by the each heuristics.

Our main goal was to evaluate how critical code anomalies could be prioritized earlier in the software development, according to their relation to architectural problems. Therefore, the *ground truth* was used as baseline for comparing the results achieved by the

heuristics. We should also mention the system experts only participated in the process of producing a trustworthy ground truth required to compare our research results. Thus, experts (i.e. developers or maintainers) of the target applications were asked to reason about the most critical code elements. The critical code elements should be, for example, those classes that realize the most important architectural components, as well as those classes responsible for implementing the key *provided* and *required* interfaces. Therefore, the anomalous code elements listed by the system's experts, as the most critical ones, should be urgently refactored so that severe maintainability problems could be avoided. As a result of this step, developers of the Mobile Media application provided the top 10 critical code elements, whereas developers of the Health Watcher system indicated 30 code elements as being the most critical. Developers of the SubscriberDB application provided a list with the top 15 anomalous code elements.

Step 3 - Evaluating the Prioritization Heuristics. Once Steps 1 and 2 are performed, we apply the prioritization heuristics for all the target applications. Each heuristic might require different information in order to produce the ranked list with the most critical code anomalies. When a heuristic is applied, it identifies all the instances of the architectural problems it was design to address. Thus, all the code elements are ranked according to their architecture relevance. In addition, when a code element is participating in more than one instance of the same type of architectural problems, it receives a high priority. After a list with the most critical code anomalies has been produced, we compare the rankings with the *ground truth* provided for the *target application*. The main reasons for this analysis are: (i) if we asked developers/experts to produce a ranked list containing all the code elements that could impact on the architecture design, our analysis would be unviable; and (ii) our goal was to evaluate the proposed heuristics in terms of code elements that could represent deeper maintainability problems. Thus, the critical code elements are the ones that should refactored earlier in the software development, so that deeper maintainability problems could be avoided.

Aiming to analyze the code elements classified as most critical by the heuristics, we used the *Size of Overlap* between the different prioritization lists. The reason why we select this measure is that it is simple to calculate, and it allows us to identify whether the prioritization heuristics have accurately distinguished the top-*k* code elements. The *Size of Overlap* indicates the accuracy of the heuristic when identifying the most critical code elements. Moreover, this measure might be a good indicator of the heuristics capability when identifying the critical code elements that should be refactored first.

Breaking ties between Code Anomalies. Once the list code anomalies has been provided by each heuristic, different criteria might be applied for breaking ties between code anomalies with the same architectural relevance.

Table 4 – Architecture Sensitive Metrics for Breaking Ties

Metric	Description
Density of Code Anomalies	It calculates the density of code anomalies in a code element realizing a given component.
Concerns per Code Element	It counts the number of concerns a measured code element implements.
# of Concerns per Architecture Element	It counts the number of concerns a measured architectural component realizes
Concern Diffusion over Components	It counts the number of code elements affected by the implementation of a concern
Concern Diffusion over Operations	It counts the number of methods and constructors affected by the implementation of an architectural concern

In this context, we introduced different metrics and source code information that might also be used during the prioritization process. Depending on which heuristic developers decide to use, different metrics can also be used according to criteria defined by the system architect. Table 4 shows the architecture sensitive metrics used as additional measures on the prioritization process. Those metrics can assist developers when identifying architecture degradation symptoms in software systems, regardless of the modularization technique employed.

5.2 Inter-Component Heuristics

In this section, we discuss the main results when applying the prioritization heuristics related to problems on the communication between architectural components.

5.2.1 External Attractor Component

Our first heuristic is based on the assessment of anomalous code elements used by several code elements belonging to other architectural components. Occurrences of *External Attractor* component might lead to the introduction of code anomalies in the elements using the architectural component under assessment. If the anomalous code element under investigation implements different concerns, others external code elements depending on it might be forced to deal with concerns they are not interested. When applying the prioritization heuristics, we first identify the anomalous code elements responsible for realizing each component. After that, the heuristic computes the number of anomalous code elements realizing the external components, which depend on the component under assessment. Finally, we produce the ranked list with the most critical anomalous code elements identified as instance of *External Attractor* component. The heuristic have been applied for all target applications selected in our study. The results revealed an acceptable accuracy in terms of prioritizing the critical code anomalies. Table 5 shows the results of applying the heuristic for identifying *External Attractor* components.

Table 5 – Results for External Attractor Heuristic

Name	N-ranked	Overlap	
		Value	*Accuracy*
Mobile Media	10	5	50%
Health Watcher	30	12	40%
Subscribers DB	15	10	67%

For the Mobile Media, we observed that 5 out of 10 measures had low accuracy when compared to the anomalous code elements presented in the *ground truth.* For the Health Watcher, the results revealed an accuracy level of 40% on the prioritization process. A recurrent problem is that most part of anomalous code elements identified by the heuristic had the same number of code anomalies. The problems are related with the fact those anomalous code elements are all implementing the GUI concern. Although the GUI concern is represented in the blueprint by two architectural components, there are 47 code elements in the source-code responsible for realizing it. Finally, the heuristics performed better when prioritizing the anomalous code elements in the Subscribers DB, if compared to the results achieved in the other two applications. Although the prioritization heuristics achieved 67% of accuracy, we observed in the Subscribers DB the number of instances of anomalies in many code elements is the same.

5.2.2 External Addictor Component

Our second heuristic identifies groups of anomalous code elements that depend (or are addicted) on anomalous code elements belonging to external components. In order to identify the most critical code anomalies, we firstly detected anomalous code elements realizing the architectural components under assessment. Furthermore, we

need to identify the anomalous code elements - belonging to an external component – from which the code element under assessment depends. Thus, a higher score should be assigned to components with: (i) a high number of anomalous code elements; and (ii) a high of number of external dependencies of external components associated to the architectural component under assessment. Moreover, code elements realizing a given architectural component should be ranked according to the architectural relevance of the system concern they implement.

When analyzing blueprints of each target application, we observed the Mobile Media represents 18 components, while Health Watcher and Subscribers DB models have respectively 6 and 8 components. Even with different levels of abstraction, we could observe instances of *External Addictor component* in 2 out of 3 target applications. In the Mobile Media, 6 architectural components have been involved in occurrences of *External Addictor component*, while in the Health Watcher we observed only 3 components. Table 6 shows the results of applying this heuristic when prioritizing critical code anomalies. For Mobile Media, we observed that 5 out of 10 code elements were correctly prioritized and ranked as being the most critical code elements - which indicates an accuracy of 50%. For the Health Watcher, we observed 11 out of 30 code elements are correctly prioritized and ranked as being critical to the architectural design. Thus, the heuristic achieved an acceptable accuracy when prioritizing critical code elements. Since we could not identify instances of *External Addictor* component in the Subscribers DB, the heuristic have not been applied for this system.

Table 6 – Results for External Addictor Heuristic

Name	N-ranked	Overlap	
		Value	*Accuracy*
Mobile Media	10	5	50%
Health Watcher	30	11	37%

5.3 Concern-Based Heuristics

Our second set of heuristics is directly related with problems on the implementation of architectural concerns. For instance, anomalous code elements responsible for realizing a high number of concerns should be prioritized, since they violate the *Single Responsibility Principle.* Different weights (or level of importance) must be assigned to architectural concerns according to the system architect. Thus, an architectural concern can have high priority when it is implemented by many anomalous code elements. Moreover**,** the heuristics also identify architectural problems caused by the violation of the principle of *Separation of Concerns* (SoC). This type of architectural problem occurs when anomalous code elements within an architectural component contains several concerns (e.g. crosscutting concerns).

In order to identify the critical components, we detected the number of anomalous code elements within each component that are responsible for realizing the same concern. For each group of anomalous code elements, we verify the number of concerns modularized, as well as the number of code elements contained in this group. In addition, architectural components can also be classified depending on how many concerns it is realizing. An architectural component is classified as weak when it realizes a high number of concerns, while components with lower number of concerns are classified as strong. Thus, when prioritizing the most critical code elements, the heuristics consider strong architectural components as high priority.

5.3.1 Concern Overload

Our third heuristic aims at identifying code elements belonging to the same component, and that are responsible for modularizing several independent architectural concerns. A concern is considered to be independent when it should be modularized by a different component. Thus, we firstly identify the number of anomalous code elements responsible for implementing the same concern. In addition, we consider the number of concerns modularized for each architectural component. For example, in the Health Watcher system we can identify architectural components that implement at least2 different concerns: GUI (4 concerns), Business Rules (2 concerns), Distribution Manager (4 concerns) and Data Manager (2 concerns). Similarly, the SubscribersDB application implements 8 architectural components (AddSubscribersUI, SubsriberController, MailingUI, SearchUI, MailingController, SearchController, Persistence and EditSubscribersUI), and each component realizes 2 concerns.

Code elements implementing one of those components are likely to suffer from *Concern Overload*, since they have to deal with most part of the concerns realized in the target application. Moreover, we relied on two additional measures when producing the results for this heuristic, namely: Density of Code Anomalies and Concerns Per Code Element. Those measures are described in Table 4.. We observed that, in general, this heuristic performed well for both target applications (see Table 7).

Table 7 – Results for Concern Overload Heuristic

Name	N-ranked	Overlap	
		Value	*Accuracy*
Mobile Media	10	6	60%
Health Watcher	30	26	87%
Subscribers DB	15	11	73%

For the Mobile Media, 6 out of 10 (60% accuracy) anomalous code elements are correctly prioritized when compared to the *ground truth*. The performance for this heuristic is even better for Health Watcher and Subscribers DB systems. While Subscribers DB achieved 73% of accuracy, in the case of Health Watcher the heuristics achieved an accuracy of 87%. In addition, we observed 9 out of 10 anomalous code elements were correctly prioritized and ranked by the heuristic when compared with the *ground truth*. Similarly, the proportion for the Subscribers DB system was 8 out of 10 anomalous code elements have been correctly prioritized and ranked as critical to the architectural design.

5.3.2 Misplaced Concern

Our fourth heuristic identifies groups of anomalous code elements responsible for modularizing an architectural concern that is not the predominant one of their enclosing component. Although all the architectural components in the Subscribers DB implement at least two concerns, developers have not provided information of which concern is predominantly addressed by each component. Thus, we have not applied the heuristic for identifying instances of *Misplaced Concern* in this system. On the other hand, the concerns of Mobile Media and Health Watcher have already been well documented, respectively, in [14] and [17].

Table 8 – Concerns for Health Watcher and Mobile Media

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

In addition, there is information available about the diffusion of concerns over components (CDC) and concerns over operation (CDO) as illustrated in Table 8. As observed, Health Watcher and Mobile Media implement, respectively, 6 and 5 concerns. To define which concern would be more relevant for the prioritization heuristic, we analysed the metrics CDC and CDO [28]. Those metrics together quantify the degree of diffusion of the architecture concerns in the system. A higher diffusion value means more code elements implement the same high-level concerns.

Table 9 indicates the results produced for this heuristic. For Mobile Media, we observed 5 out of 10 code elements were correctly prioritized and ranked. Furthermore, when comparing the list of code anomalies provided by the heuristic and the *ground truth*, we observed some anomalous code elements were equally prioritized – considering their position as ranked elements. In this case, we can use different measures (e.g. density of code anomalies, number of concerns) as means for breaking ties. Moreover, for this heuristic we only considered the top 10 elements for the Mobile Media (after the ties have been solved). On the other hand, we observed 25 out of 30 anomalous code elements in the Health Watcher reached high accuracy. Therefore, the heuristic could produce a highly accurate list of the most critical code anomalies. In addition, the results revealed most part of the code elements affected by multiple anomalies are often ranked with a high priority – i.e. the heuristic achieved 84% of accuracy for Health Watcher system.

Table 9 – Results for the Misplaced Concern Heuristic

Name	N-ranked	Overlap	
		Value	*Accuracy*
Mobile Media	10	5	50%
Health Watcher	30	25	84%

5.4 Accuracy of the Architecture Sensitive Heuristics

After applying the prioritization heuristics, we relied on the initial data analysis in order to provide interesting research findings. For instance, we observed occurrences of *External Attractor* and *External Addictor components* often occur for most of the target applications. Those occurrences concentrate more than 50% of dependencies between components, which indicates a tight coupling between them. The strong coupling is likely to be related with anomalous code elements realizing the communication between the architectural components in the architecture blueprint. In addition, the results revealed that, in general, the inter-component heuristics achieved (at least) an acceptable accuracy for all the target applications. That is, results indicated code elements infected by multiple anomalies are often perceived as high priority. Thus, when applying the statistical test we could reject the *null hypothesis* $H_{1.0}$ – as the inter-component heuristics achieve an acceptable accuracy when prioritizing critical code anomalies for all applications.

In turn, the number of occurrences of *Misplaced Concern* and *Concern Overload* indicates a high proportion of anomalous code elements related with problems on the implementation of concerns. For both applications, we observed some architectural concerns are crosscutting several code elements – which implies they are scattered through the source code. In this sense, the mapping of concerns was clearly useful for prioritizing a significant number of code elements harmful to the software architecture. Moreover, the results showed that, in general, the concern-based heuristics achieved acceptable to high accuracy for all applications. The results indicated the *null hypothesis* $H_{2.0}$ is rejected, as the *concern-based heuristics* were able to identify code elements containing critical anomalies for most target applications.

6. Discussion

This section discusses other findings observed when analyzing the overall results produced by the proposed heuristics.

6.1 Prioritization Heuristics vs. False Positives/Negatives

Our first discussion is associated to the proportion of False Positives (FP) and False Negatives (FN) observed when applying the heuristics for all target applications. The proportion of FP and FN was computed after we have applied four proposed heuristics, and the list of most critical code elements have been produced for each tem. Table 10 summarizes the number of FP and FN considering all the target applications. We should mention only the top *k* anomalous code elements (ACE) prioritized and ranked by each heuristic have been selected. When analysing results for Mobile Media (MM), we observed the heuristics identified FP mostly related with the implementation of Data (10 instances), View (6 instances) and Controller (2 instances) functionalities. In addition, all code elements prioritized by the heuristics as FN are responsible for implementing the Controller functionality.

Table 10 –Relevance Based on Architecture Information

System	Measure	ACE	Prioritization Heuristic			
			EAt	*EAd*	*MC*	*CO*
MM	FP	10	5	5	4	4
	FN	10	5	5	4	4
HW	FP	30	18	19	3	4
	FN	30	12	11	3	4
S_{DB}	FP	15	5	**N/A**	**N/A**	5
	FN	15	5	**N/A**	**N/A**	5

Furthermore, we analyzed the anomalous code elements ranked by the heuristics for the Health Watcher and Subscribers DB. For Health Watcher (HW), we observed the number of FP is mainly associated with code elements implementing Data, Concurrency and Distribution functionalities. However, one of the criteria used for classifying those anomalous code elements as FP is the fact they implement a high number of code anomalies when compared to the other ranked code elements. This result is specially observed when the *External Attractor* (Eat) and *External Addictor* (Ead) heuristics are applied.

For the concern-based heuristics MC and CO, code elements identified as FP are mostly related with the realization of the GUI component. On the other hand, code elements identified as FN are mostly related with GUI (19 cases) and Business Rules (3 cases), which represent the core components of this application. For instance, the code elements responsible for realizing the GUI are responsible for implementing the key interfaces that provide access to all the services available in the system. Although those code elements implements one of the most important components in the Health Watcher architectural design, they do not have a high number of code anomalies – and therefore the heuristic have not ranked them with high priority. Finally, for the Subscribers DB (S_{DB}) we observed instances of FP associated with code elements responsible for implementing different functionalities, namely: Model (1 instance), View (2 instances) and Controller (2 instances). We should also mention that many ties when the prioritization heuristics are applied in the SubscriberDB. The problem was that the anomalous elements are similarly distributed through the components represented in the architecture blueprint.

6.2 Comparing Rankings Provided by Different Heuristics

Our previous work [4] proposed heuristics for prioritizing critical code anomalies based on the evolution history of 4 target

applications – also known as history-sensitive heuristics. Those heuristics strictly extract source code information (e.g. bugs, error report and density of code anomalies). Moreover, the heuristics collect this information considering the evolution history of a software system. Table 11 summarizes the results of applying the history-sensitive heuristics. As observes, the heuristics performed well in most part of the cases achieving from acceptable to high accuracy. The history-sensitive heuristics also prioritized code anomalies associated with architectural erosion problems. For each prioritization heuristic, it was ranked the top *k* anomalous code elements according to their architectural relevance. The four heuristics performed well when prioritizing the critical code elements associated with architecture erosion symptoms, since all heuristics achieved accuracy higher than 70%.

Table 11 – Relevance Based on the System History Evolution

Heuristic	Name	N-Ranked	ArchRel	% ArchRel
Change-Proneness	HW	14	10	71%
	MM	10	7	70%
	PDP	**10**	**10**	**100%**
Error Proneness	HW	14	10	85%
	MM	**10**	**8**	**80%**
	PDP	**10**	**8**	**80%**
Architecture Role	HW	10	4	40%
	MM	**10**	**9**	**90%**
	PDP	**10**	**10**	**100%**
Anomalies density	HW	10	5	50%
	MM	**10**	**9**	**90%**
	PDP	**10**	**8**	**80%**
	MIDAS	10	6	60%

On the other hand, in this work we proposed heuristics that exploit architecture information provided in the architecture blueprints produced by developers and maintainers - also known as *architecture sensitive heuristics*. Although the proposed heuristics have not presented much superior results when compared with the history-sensitive heuristics, most of anomalous code elements are also related with architectural problems. We also performed an analysis to evaluate how accurate the architecture--sensitive heuristics performed in terms of prioritizing anomalous code elements associated with architectural drift symptoms. The collected data (see Table 12) revealed most part of the anomalous code elements are related with architectural drift problems when analysing the three target applications. In average more than 75% of the anomalous code elements identified by the heuristics are harmful to the software architecture design.

Table 12 –Relevance Based on Architecture Information

Heuristic	Name	N-Ranked	ArchRel	% ArchRel
External Attractor	HW	30	21	70%
	MM	10	9	90%
	S_{DB}	15	10	67%
External Addictor	HW	30	22	74%
	MM	10	5	50%
Misplaced Concern	HW	30	24	80%
	MM	10	8	80%
Concern Overload	HW	30	24	80%
	MM	10	9	90%
	S_{DB}	15	11	74%

We should recall the history-sensitive and architecture sensitive heuristics are used for different purposes. The architecture sensitive heuristics aim at prioritizing critical code elements based on their relation with architecture drift symptoms [21]. Architectural drift is usually manifested in early versions of software systems. That is, the reason why we selected specific versions of those systems when evaluated the proposed heuristics. Once these architectural drift symptoms are identified and the corresponding anomalous code elements are refactored, more severe problems related with architecture erosion symptoms [21] might be avoided. If architecture erosion symptoms are manifested in later versions of each software systems, we can detect those problems by applying the history-sensitive heuristics. Thus, we provide developers with means to address different architecture degradation symptoms, and solve them as early as possible in the software development process. In this way, we provide developers with means for avoiding the system architecture to degrade.

7. Threats to Validity

Internal Validity. Our first internal validity threat is related with the quality of the architecture blueprints. Three properties have been presented in Section 2.2 to make it clear how we selected blueprints that reach a minimum quality, so that it can be used on the prioritization process. However, as the system evolves it is hard to synchronize changes in the system architecture, and the code elements in the system implementation. Our second internal threat is related with the mapping between architecture blueprints and source code elements. In order to mitigate this threat, we have validated the mappings with the system architects and developers.

External Validity. Our first external threat is related with possible errors on the detection of anomalies. To avoid the risk of imprecision on the detection process: (i) the original developers and architects have been involved in this process; and (ii) well-known metrics and thresholds were used by the detection strategies. The second external threat is related with the use of the ground truth. Although each system expert has used their strategies to identify the most critical anomalies, around 75% of code anomalies were equally identified. The final ground truth was produced as a joint decision.

Conclusion Validity Our external threat is related with the choice of the target applications. The results found in our study are limited to 3 target applications. In order to minimize this threat, we selected systems developed by different programmers, with different domains, programming languages and architectural styles. However, to generalize our results, further empirical investigation is required. Moreover, we have tried to make our best to describe carefully our study so that others can replicate it using other software systems.

8. Final Remarks

This paper proposes and evaluates architecture sensitive heuristics that exploits information available in software project blueprints. To the best of our knowledge, this is the first work exploring blueprints to (semi)-automate the prioritization of critical code anomalies. Our main contributions encompasses: (i) prioritization heuristics based on different criteria for prioritizing the most critical code elements based on the architectural relevance; (ii) the evaluation of the proposed architecture sensitive heuristics regarding the architectural relevance of the code anomalies; (iii) a discussion on how the architecture sensitive heuristics impact on the prioritization of critical code anomalies for all the 3 applications under analysis.

As main findings observed in the empirical evaluation of the proposed heuristics, we can mention: (i) there are architectural problems involving groups of classes that realize architectural components, which are intended to implement a specific functionality in the system architecture; (ii) there are several symptoms of degradation involving architectural components infected by multiple anomalies; and (iii) even for the architectural concerns well defined and relevant to a software system, the prioritization heuristics were efficient to pinpoint architectural problems. Finally, we systematically evaluated each of the architecture-sensitive heuristics proposed in this paper.

Acknowledgements

This work was funded by CNPq (productivity grant 305526/2009-0 and Universal Project grant number 485348/2011-0), and FAPERJ (distinguished scientist grant E-26/102.211/2009, project grant number E-26/111.152/2011).

References

[1] L. Hochstein and M. Lindvall. Combating Architectural degenerations: A Survey. Journal of Information and Software Technology, Vol. 47, Issue 10, pp. 643-656, July 2005.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts. Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.

[3] I. Macia, R. Arcoverde, A. Garcia, C. Chavez and A. von Staa. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In Proc. of 16th European Conf. on Software Maintenance and Reengineering, pp. 277-286, Szeged, Hungary, March 2012.

[4] Arcoverde, R.; Macia, I.; Garcia, A.; Staa, A. Automatically Detecting Architecturally-Relevant Code Anomalies. In Proc. of the 3rd RSSE in conjunction with Int'l Conf. on Soft. Engineering, pp. 90-91, Zurich, Switzerland, June 2012.

[5] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic and A. von Staa. Are Automatically-Detected Code Anomalies Relevant Architectural Modularity? – An Exploratory Analysis of Evolving Systems. In Proc. of 11th Int'l Conf. on Aspect-Oriented Software Development, pp. 167-178, USA, March 2012.

[6] M. Lanza and R. Marinescu. Object-Oriented Metrics in Practice – Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems, Springer-Verlag, 2006.

[7] R. Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In Proc. of 20th IEEE Int'l Conf. on Soft. Maintenance, pp. 350-359, Chicago, USA, Septemver 2004.

[8] J. Garcia, D. Popescu, G. Edwards and N. Medvidovic, Identifying Architectural Bad Smells. In Proc. of 13th European Conf. on Soft. Maintenance and Reengineering, pp. 255-258, Germany, March 2009.

[9] R. N. Taylor, N. Medvidovic and E. M. Dashofy. Software Architecture - Foundations, Theory, and Practice. Wiley 2010,.

[10] N. Moha, Y. Gueneheuc and P. Leduc. Automatic Generation of Detection Algorithms for Design Defects. In Proc. IEEE/ACM Int'l Conf. on Automated Software Engineering, pp. 297-300, Japan, September 2006.

[11] I. Macía, F. Dantas, A. Garcia and A Staa. Are Code Anomaly Patterns relevant to Architectural Problems? IEEE Trans.on Soft. Engineering, 2014 (Submitted).

[12] W. Oizumi, A. Garcia, M. Ferreira, A. Staa and T. E. Colanzi. When Code-Anomaly Agglomerations Represent Architectural Problems? An Exploratory Study. In Proc. of 27th Brazilian Symposium on Soft. Engineering (SBES'14), pp. 91-100, Brazil, September 2014 .

[13] A. S. Eick, T. Graves and A. Karr, "Does Code Decay? Assessing the Evidence from Change Management Data", IEEE Transactions on Soft. Eng., Vol. 27, Issue 1, pp. 1-12, 2001.

E. Figueiredo Figueiredo *et al*. Evolving Software Product Lines with Aspects: An empirical Study on Design Stability, In Proc. of 30th Int'l Conf. on Soft. Engineering, pp. 261-270, Germany, May 2008.

[14] S. Soares, E. laureano and P. Borba. Implementing Distribution and Persistence aspects with AspectJ. In Proc; of 17th ACM SIGPLAN Object Oriented Programming, Systems, Languages and Applications, Vol. 37, Issue 11, pp. 174-190, November 2002.

[15] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell and A. Wesslen. Experimentation in Software Engineering – An Introduction, Kluwer Academic Publisher, 2000.

[16] A. MacCormak, J. Rusnak and C.Y. Baldwin. Exploring the Structure of Complex Software Designs – An Empirical Study of Open Source and Proprietary Code. Journal of Management Science, Vol. 52, Issue 7, pp. 1015-1030, 2006.

[17] P. Greenwood *et al*. On the Contributions of an End-to-End AOSD Testbed. In Proc. of Workshop in Aspect-Oriented Requirements Engineering and Architectura Design @ Int'l Conf. on Soft. Engineering (ICSE) 7th Int'l Conf. on Aspect-Oriented Soft. Development, pp. 8-16 Minneapolis, May 2007.

[18] Object Management Group. Unified Modeling Language, specification, 1.5. formal/2003-03-01, March 2003.

[19] Lange, L. and Chaudron, M. An Empirical Assessment of Completeness in UML Designs. In Proc. of 14th Int'l Conf. on Empirical Assessment in Soft. Engineering, pp. 111-121, U.K., Abril, 2010.

[20] G. Kiczales, J. Lamping, A, Mendhekar, C. Lopes, J. Loinftier and J. Irwin. Aspect-Oriented Programming. In Proc. of 11th European Conf. on Object-Oriented Programming, pp. 220-242, Finland, May 1997.

[21] R.C. Martin. Design Principles and Design Patterns. Object Mentor, 2003.

[22] S. Vidal and C. A. Marcos. Building an Expert System to Assist System Refactorization. Journal on Expert Systems with Applications, Vol. 39, pp; 3810-3816, 2012.

[23] M. Godfrey and E. Lee. Secrets from the Monster: Extracting Mozilla´s Software Architecture. In Proc. of 2nd Symposium on Constructing Soft. Engineering Tools (CoSET), pp. 15-23, Ireland, June 2000.

[24] NDepend. < http://www.ndepend.com> Accessed in December, 2014.

[25] Understand < http://www.scitools.com > Accessed in December, 2014.

[26] S. W. Thomas, B. Adams, A.E. Hassan and D. Blostein: Studying Software Evolution Using Topic Models. Journal Science of Computer Programming, Vol. 80, pp. 457-479, 2014.

[27] C. Nunes, A. Garcia, C. Lucena and J. Lee. Heuristic Expansion of Feature Mappings in Evolving Program Families. Journal on Software: Practice and Experience, Vol. 44, Issue 11, pp. 1315-1349, May, 2013.

[28] Figueiredo, E., Whittle, J. and Garcia, A. Concern-Morph: Metrics-based Detection of Crosscutting Patterns. In Proc. of the 7th joint meeting of the European Soft. Engineering Conference and ACM SIGSOFT Foundations on Soft. Engineering, pp. 299-300, 2009.

[29] Robilard, M.P and Warr, F. W. ConcernMapper: Simple View-Based Separation of Scattered Concerns. In Proc. of the Workshop on Eclipse Technology eXchange, in conjunction with the 20th Object-Oriented Programming, Systems, Languages and Applications (OPSLA), pp. 65-69, San Diego, USA, October 2005.

[30] Macia, I. On the Detection ofArchitecturally-Relevant Code Anomalies in Software Systems. Ph.D. Thesis. Pontifical Catholic Unviersity of Rio de Janeiro, 2013.