2 Background and Related Work

Along the system evolution, different properties associated with the system quality are impacted. For instance, software systems grow in size and complexity as new modules and their functionalities are implemented in the existing program. The problem arises when architectural design decisions are not correctly performed in the descriptive architecture, and hence, the system maintenance can be compromised. When software maintenance activities are properly performed in the descriptive architecture along software evolution, architectural degradation symptoms will start to emerge. Examples of architecture degradation symptoms in the actual architectural of the implementation are complex component interfaces or scattered functionalities across components' implementation.

One of the main factors responsible for architecture degradation is the unavoidable and progressive insertion of code anomalies. Recent studies revealed that there is a strong correlation between the occurrences of instances of critical code anomalies and architectural problems (Macia *et al*., 2012a)(Macia *et al*., 2012b)(Macia *et al*., 2014)(Oizumi *et al*., 2014). Therefore, when critical code anomalies are not properly prioritized and removed as early as possible, the system maintainability can be compromised and, in some cases, software architecture have to be completely redesigned (Eick *et al*., 2001) (MacCormack *et al.*, 2006). For instance, when the interface of a component starts to bloat, it starts to blend several non-cohesive functionalities and expose information that should be hidden to the client components. Then, the implementation of all the client components increasingly gets artificially coupled to the elements of the bloated interface. Maintenance and evolution of the server and client component becomes increasingly complex, error-prone or even prohibitive over time.

In this chapter, we describe the basic terminology required to understand the development of our research (Section 2.1). The observation of architectural problems in the actual implementation is often only possible through the detection of code anomalies. Therefore, we discuss how the occurrences of code anomalies are related

with architectural problems (Section 2.2). Moreover, we introduce the existing strategies for detecting and/or prioritizing code anomalies, as well as the main limitations of those existing strategies (Section 2.3). We also present the concept of architecture blueprint used in this thesis and the properties used to characterize whether and how a given design model can be, in fact, classified as an architecture blueprint. The analysis of the architecture in the implementation is only possible if there is a projection (mapping) of the architecture elements in the source code. Therefore, we discuss how the mapping process between the architecture and source code elements is performed (Section 2.4). Even though it not the focus of our work to defining a new mapping approach, the mapping process between elements in both abstraction levels is essential for executing the architecture sensitive heuristics proposed in this work (see Chapter 5).

2.1. Basic Terminology

Henceforth, we discuss the basic terminology required to understand all the phases that permeate the research performed in this thesis. Firstly, we introduce the definitions associated with the concept of software architecture, as well as architecture degradation symptoms. Furthermore, we discuss how the occurrence of code anomalies can be associated with architecture degradation symptoms.

2.1.1. Software Architecture

Software Architecture can be defined as the structure of a software system, which comprises software components, (provided and required) interfaces of those modules, and the relationship among them (Bass *et al*., 2003). The architecture decomposition outlines the organization of a software system. That is, the architecture structure captures the architectural elements and their interaction (Gorton, 2006). *Architectural Component* can be defined as the architecture entity responsible for encapsulating a subset of the system's functionalities (Taylor *et al*., 2009). Architectural Components interact through *Architectural Connectors*, which in turn, allows the communication, coordination and data conversion between architectural

components (Meta *et al*., 2000). At the implementation level, an architecture connector can be realized by one or more simple method calls or by one or more classes realizing the protocols for communication, coordination or data conversion (Mehta *et al*., 2000).

In this context, the software architecture usually represents the key design decisions made in the early stages of software development (Jansen and Bosh, 2005). Therefore, the architecture decisions must preserve design and modularity principles (Martin, 2003), which are essential to the system evolution and longevity (Clements *et al*., 2002). Examples of these principles are simple interfaces and encapsulation, high cohesion and low coupling of components, separation of architectural concerns, and the like (Martin, 2003).

In this sense, the *prescriptive architecture* comprehends explicit design decisions made by system architects on the selection of components, interactions and their constraints (Taylor *et al*., 2009). On the other hand, the *descriptive architecture* describes how the system has been actually built (Taylor *et al*., 2009). The descriptive architecture should ideally resemble the prescriptive architecture. However, in real software development projects the descriptive and prescriptive architecture does not match. In these cases, the intended (prescriptive) architectures are available in models to guide developers on: (i) maintaining or evolving the system structure, and (ii) reasoning about the modularity and maintainability of the actual architecture implementation (Taylor *et al*., 2009)(Baltes and Diehl, 2014).

2.1.2. Architecture Degradation Symptoms

Architecture degradation symptoms represent mismatches between the prescriptive and descriptive architectures, as wells as modularity problems between those two architectures. It is important to highlight, in this thesis, we explore problems related with architectural drift. The motivation and justification was given in Chapter 1. *Architectural drift* symptoms occur when decisions introduced in the descriptive architecture violate modularity principles (Chapter 4). Therefore, architectural drift problems might impair the adaptability of the system architecture, and as consequence, the system evolution (Perry and Wolf, 1992). There are certain

cases where the drift problems were already introduced in the prescriptive architecture (also known as "congenital" architectural problems), being propagated to the implementation of the software architecture (prescriptive architecture) in the first versions of a program. Table 1 summarizes the set of six architectural drift problems considered in this thesis. We decided to focused on these different types of architectural problems because: (i) they represent all architecture degradation symptoms empirically observed in the systems analyzed in our research (and described in the next chapters), and (ii) they are representative of architectural problems affecting different types of architecture elements – i.e. interfaces, components, connectors, and other architectural relevant concerns not modularized in components or connectors. An architecturally-relevant concern is any functional or extra-functional feature of interest by the architects, which influences one or more architectural decisions. Some typical examples of such concerns are error handling, persistence, and GUI.

Table 1 - Architectural problems considered in this work

Architectural Problem	**Description**
Ambiguous Interface	The interfaces usually offer only a single, general entry-point into a component, reducing the system analyzability and understandability. The ambiguous interfaces handle more requests than it should actually process (Garcia *et al*., 2009b).
Component Concern Overload	An architectural component is responsible for realizing two or more unrelated system's concern (Stal *et al*., 2011).
Connector Envy	It occurs in components that encompass extensive interaction-related functionality that should be delegated to a different connector. That is, the architectural component realizes functionalities that should be assigned to another connector (Garcia *et al*., 2009b).
Overused Interface	Interface that exposes a lot of heterogeneous data and it is used by several other (required) interfaces of other components.
Redundant Interface	Interface that exposes the same information of the other interfaces.
Scattered Parasitic Functionality	It takes place when multiple components are responsible for realizing the same architectural high-level concern and, additionally, some of those components are responsible for independent concerns (Garcia *et al*., 2009b).

It is important to mention that the higher number of architecture violations and modularity problems, the higher is the chance of the software architecture suffers from degradation symptoms. As previously mentioned, the main factor that contributes to architecture degradation symptoms is the progressive and unavoidable insertion of code anomalies. *Code anomaly* is a term often used to define structural

problems in the source code, which may lead to severe maintenance problems in a software system (Fowler *et al*., 1999)(Eick *et al*., 2001). We can mention as examples of code anomalies *God Class*, *Shotgun Surgery*, *Feature Envy*, *Divergent Change* and *Long Method*. Thus, code anomalies can affect different source code structures or *code elements,* such classes, interfaces, attributes, constructors and methods. When instances of code anomalies related to architectural problems, we say that those code anomalies are critical to the architectural design. In this thesis, instances of code anomalies are related with architectural problems, either when they affect the communication between architectural components or they impact on the implementation of architectural concerns. For each type of architectural problems, an instance of code anomaly can contribute in different ways – i.e. a code element is affected by the *God Method* anomaly (Fowler *et al*., 1999), which is responsible for implementing many of the concerns realized by its enclosing component. In this case, the architectural component suffers from *Component Concern Overload* and the class is contributing for the realization of this architectural problems.

Whereupon, software developers are expected to be able when deciding which critical code anomaly should be refactored first. Thus, refactoring is commonly used to remove critical code anomalies that might be related with problems in the architecture design. The refactoring process (see also Section 2.2.1) consists of changing the design structure of a software system without changing its behavior, in order to improve the system maintainability (Fowler *et al*., 1999). Moreover, identifying architecture degradation symptoms is a challenging task particularly when the prescriptive architecture is not well documented. Therefore, in such scenarios the system implementation is one of the most reliable artifact when detection architecture degradation symptoms. It is important to mention the approach proposed in this thesis explores different artifacts (e.g. architecture blueprints, source code, metrics) in the process of prioritizing code anomalies related with architectural problems. The most innovative idea is the exploration of architecture blueprints as a way of improving: (i) the identification of which anomalous code elements are most likely to realize an architectural problem (and should be prioritized), and (ii) the ordering (ranking) of those elements according to their relevance to the realization of one or more architecture elements.

2.2. Code Anomalies as Architecture Degradation Symptoms

When software developers are performing architecture reviews (Kazman and Bass, 2002)(Starr and Zimmerman, 2002) of the source code, they are usually expected to choose which code anomalies should be refactored first to avoid architecture degradation symptoms. Time constraints associated with such reviews imply that finding the critical code anomalies in large systems cannot be performed without systematic prioritization support. Therefore, existing techniques for code anomaly detection should also consider architectural information, typically available in industry software projects (Baltes and Diehl, 2014), to explore the relationship between critical code anomalies and architectural problems.

As previously mentioned, architecture degradation (Hochstein and Lindvall, 2005) is frequently a direct consequence of the progressive insertion of anomalies (Macia *et al*., 2012a)(Macia *et al*., 2012b) in the source code. When critical code anomalies are not systematically prioritized and removed, the system's architecture might degrade. In addition, identifying degradation symptoms directly on the architecture specification can be an arduous task, when not impossible. The reason is that architecture design decisions are not entirely specified in real software projects, but they are partially represented as architecture blueprints. Thus, software developers need to be provided with means to detect, prioritize and remove critical code anomalies. When critical code anomalies are not prioritized and refactored early in a software project, the cost to perform this activity later is usually high or prohibitive (Macia *et al*., 2012a). For instance, many researchers have investigated the impact of code anomalies on exerting undesirable modifications in the source code (Macia *et al*., 2011)(Mantyla and Lassenius, 2003). Recent studies also revealed that the code structures affected by code anomalies suffer more changes during software maintenance (Mantyla and Lassenius, 2003).

Furthermore, recent studies investigated the negative impact of code anomalies in the system architecture. For example, a previous study (Eick *et al*., 2001) shows architecture modularity of a larger communication system has been degraded in 7 years. The key problem was the relationship between the architectural components, hosting several code anomalies, increased over time. Such anomalous modules were

not independent anymore and, consequently, further changes in the system structure were not possible. This problem could not be observed based on conventional source code analysis, as those architectural components were no longer aligned with module decomposition in the implementation. Another study (MacCormak *et al*., 2006) reported that the Mozilla's browser was overmuch complex and coupled, which hindering the system maintainability and its ability to evolve. This architecture problem was the cause of its complete reengineering and developers spent around 5 years to rewrite more than 2 millions lines of code (Godfrey and Lee, 2000). The study also indicated when refactoring operations are performed early in the counterpart groups of inter-related code anomalies, the architecture degradation could be avoided.

2.2.1. Refactoring Process and Removal of Code Anomalies

As aforementioned in this chapter, refactoring can be understood as the process of changing a software system in a way it does not alter the external behavior of the source code, but improve its structure (Fowler *et al*., 1999). In this sense, refactoring activities allow to restructure the source code, improving the effort required to make future modifications. Thus, when refactoring operations are performed in the source code, some benefits should be achieved, such as: reduction of duplicated code, reduction of coupling and improvement on other internal and external quality attributes (e.g. modularity, maintainability, evolvability and the like). In addition, the refactoring process comprises 6 main activities (Mens and Tourwe, 2004): (i) identify code fragments where refactoring is required; (ii) determine which type of refactoring should be applied; (iii) check whether the external behavior is preserved after the refactoring have been applied; (iv) apply the refactoring; (v) assess the effect of the refactoring on software quality attributes; and (vi) synchronize the refactored source and other software artifacts (e.g. architecture blueprints).

In this sense, instances of code anomalies can be seem as good opportunities for applying refactorings. According to Murphy-Hill *et al.* (2009), although several refactoring tools are available, software developers tend to limit their use on low-level refactorings. Arcoverde *et al*. (2011) also suggested refactorings usually are not

applied whether they are complex, time-consuming or there is no evidence of their effectiveness in maintaining the system modularity. Moreover, they pointed out a major factor hindering the proper use of refactorings to be applied is due to the co-occurrence of code anomalies (Liu *et al.*, 2011). Therefore, software developers need to evaluate how different code anomalies should be properly removed, before the refactoring could be properly applied. Thereby, refactoring recommendation systems could help developers on identifying good opportunities for removing code anomalies. For example, Xi *et al*. (2012) a refactoring recommendation mechanism based on the observation of manual refactoring steps. The proposed recommendation mechanism monitors common sequences of previous changes on code structures to detect the occurrence of refactorings, and dynamically recommend their automation while the software developer is programming. In turn, a technique for automatically identifying required refactorings in the source code, via static slicing, has been proposed in (Tsantalis and Chatzigeorgiou, 2009). Finally, Vidal and Marcos (2012) proposed an expert software agent (named *Refactoring Recommender*) to assist developers during the refactoring activities of a software system. The refactoring recommender system analyzes the user's interaction history for improving the agent's effectiveness over time, guiding developers to predict needed restructurings in the source code.

2.2.2. Co-occurrence of Critical Code Anomalies

When analyzing the state of art regarding code anomaly detection, we found researchers have mostly investigated only the impact of isolated instances of code anomalies. Moreover, their analysis rely on the investigation of specific code anomalies from different catalogues (Fowler *et al.,* 1999)(Brown *et al*., 1998)(Kerievsky, 2004)(Lanza and Marinescu, 2006). The empirical study developed by (Abbes *et al*., 2011) brings up the notion of interaction effects across code anomalies. Abbes and colleagues found that classes and methods identified as *God Classes* and *God Methods* in isolation had no effect on effort, but when appearing together those code anomalies led to a significant increase in the maintenance effort. However, their study is limited as it merely focuses on investigating the impact of co-

occurrences of only two types of code anomalies. In addition, they do not study their impact on the software architecture degradation process.

In this sense, none of the existing empirical studies has investigated to what extent the co-occurrence of different code anomalies, established by the exploration of blueprints, might be used as indicators for prioritizing and ranking architecture degradation symptoms. In addition, existing approaches for code anomaly detection do not support developers to detect co-occurrences of code anomalies, once they are focused on detecting isolated occurrences of code anomalies. Therefore, developers need to be provided with means that allow prioritizing and ranking the most critical code anomalies so that refactoring operations can be properly performed in order to complement remove those code anomalies.

Through the analysis of co-occurrences of code anomalies it is possible, for instance, detect the occurrence of a given code anomaly that might be related with other anomalies affecting the same code element. Thus, analyzing co-occurrence of critical code anomalies might also allow developers to identify the existence of architectural problems in the system implementation. Particularly, some architectural problems cannot be detected without analyzing relationships between code elements. Finally, the detection of co-occurrences of critical code anomaly help developers on properly applying refactoring techniques, and as consequence, reduces the maintainability costs during the system evolution. Some works revealed that usually co-occurrences of code anomalies provide better indicators of architectural degradation symptoms than single instances of code anomalies (Macia *et al*., 2014)(Oizumi *et al*., 2014). However, the main problem of existing techniques is that they do not support developers on distinguishing and analyze co-occurrences of inter-related code anomalies. It is also important to mention that the analysis of the relationship between co-occurrences of code anomalies is important to reveal how architecture components are implemented by several code elements. Thus, limitations of existing detection strategies can be overcome if they are able to analyze relationships among co-occurrences of critical code anomalies.

2.2.3. Classifying Occurrences of Code Anomalies

Several works have proposed different ways to classify co-occurrences of code anomalies by adopting different criteria. While some works have proposed a categorization of code anomalies according to their scope (e.g. intra-class, inter-class), others have categorized code anomalies according to their nature (e.g. structural, semantic). Moreover, in (Mantyla and Lassenius, 2003) the code anomalies are grouped according to the modularity property they affect in a software system. However, the aforementioned categorizations suffer from limitations, once they are based on exploring a particular characteristic of a code anomaly when they should rather consider the relationship between co-occurrences of instances of code anomalies. Moreover, those categorizations do not consider the architectural relevance of a code anomaly, that is, they do not take into consideration to what extent co-occurrences of code anomalies could negatively impact on the software architecture.

On the other hand, in (Macia *et al*., 2014)(Oizumi *et al*., 2014), they classified the co-occurrence of code anomalies and classified in 9 different patterns of code anomaly. Thus, they documented a catalogue of code anomaly patterns aiming to facilitate the analysis of co-occurrences of code anomalies. The proposed code anomaly patterns are classified into four categories: (i) *Intra-Component Patterns – it* comprises patterns that occur in a single component; (ii) *Inter- Component Patterns* – it is formed by patterns that are scattered over various components; (iii) *Inheritance-based Patterns* – it groups patterns that occur in inheritance trees; and (iv) *Concern-based Patterns* – it comprises patterns related with the inappropriate modularization of architectural concerns. It is important to mention that proposed code anomaly pattern aims at facilitating the analysis of co-occurrences of critical code anomalies that might contribute to the appearance of architecture degradation symptoms. Finally, some of those code anomaly patterns are used as insight in our work when proposing the architecture sensitive heuristics (see Chapter 4) for prioritizing critical code anomalies. We relied on those code anomaly patterns, which indicate specific scenarios that can be observed when exploring the mapping between the information represented in the architecture blueprints and source code.

2.3. Tool Support for Code Anomaly Detection

Many works has investigated and proposed different techniques and tools for detection code anomalies in software systems. The detection strategies proposed in (Marinescu, 2004) are the most common mechanism for code anomalies detection, as well as on of the most referred in the literature. Those detection strategies exploit source code information, based on the combination of static code metrics and thresholds into logical expressions. Moreover, each detection strategy can be understood as a heuristic responsible for identifying code elements that might be affected by instances of a particular code anomaly (Marinescu, 2004)(Lanza and Marinescu, 2006). Other approaches have also been proposed for detection code anomalies in Java systems, based on the analysis of structural properties of code elements (Emden and Moonen, 2002). On the other hand, other approaches also support the detection of specific types of code anomalies by examining change couplings (Ratzinger *et al*., 2005)(Wong *et al*., 2011).

Several approaches for static code analysis and visualization have also been proposed. Most part of those approaches relies on the detection strategies aforementioned. Ratiu *et al*., (2004) proposed an approach that explored the historical information to improve the accuracy on the automatic identification of code anomalies though the implementation of detection strategies. Marinescu *et al*., (2010) proposed a tool for supporting the automation of some detection strategies. Munro (2005) proposed heuristics for detecting code anomalies. In turn, Moha *et al*., (2010) proposed a domain specific language for improving the construction of code anomalies detection strategies. In summary, those works focused mostly on the identification of code anomalies, even though in some case they can detect instances of specific code anomalies. Thus, detection strategies have been widely adopted and an infinite number of tools are based on this techniques. Finally, several researchers use those well-defined detection strategies as means to investigate the presence of code anomalies during the system evolution, as well as to analyze the harmful impact of critical code anomalies on the overall software quality.

Despite the existence of several works proposing different techniques and tools for code anomalies detection (Lanza and Marinescu, 2006)(Marinescu *et al*., 2004)(Ratiu, 2004)(Wettel and Lanza, 2007), it is still a challenging task to identify which code anomalies are critical to the architecture design. As previously mentioned, most part of those techniques and tools fail to assist developers when distinguishing those code anomalies that are critical to the architecture design, and those that are not. The problem is those detection strategies disregard other software factors (i.e. architecture specification, architecture blueprints) that might indicate the architectural relevance of instances of code anomalies. In this sense, code anomalies detection strategies should also: (i) exploit other information available in software projects during the development process in combination to source code artifacts; and (ii) support the co-occurrence of instances of code anomalies, instead of analyzing the occurrence of individual occurrence of code anomalies. Other studies (Macia *et al*., 2012a) (Macia *et al*., 2012b) observed that automatic detection strategies could eventually be good indicators of architectural design problems. However, software developers still invest more effort on removing code anomalies that are not critical to the architectural design problems (Arcoverde *et al*., 2011)(Macia *et al*., 2012a). That is, software developers tend to prioritize code structures that have no impact, for example, either on the communication between architectural components or on the definition of the provided/required interfaces. The prioritization of critical code anomaly as early as possible might be seen as means to guide developers correctly solving problems, and hence, avoid more severe problems related with architecture degradation symptoms.

2.3.1. Prioritization and Ranking of Code Anomalies

As aforementioned, there are many available tools and techniques for detecting code anomalies. However, as the system evolves the number of code anomalies tend to increase, and when they are not properly addressed, those instances can became unmanageable. In this case, software developers are expected to decide which code anomalies should the refactored first in order to prevent more severe problems. Additionally, developers should be able to prioritize critical code anomalies either due

to time constraints or as attempt to find the correctly solution when restructuring a large software system. Therefore, prioritizing critical code anomalies could be an important activity for increasing the accuracy and effectiveness of existing strategies when refactoring problems in a software system. Nevertheless, existing approaches mostly do not focus on the prioritization of critical code anomalies, but they rather focus on the extraction and combination of static source code measures. On the other hand, researchers also propose the use of software project repositories and explore their additional information, such as bug reports and change density – i.e. (Eick *et al*., 2001)(MacCormak *et al*., 2006)(Vidal and Marcos, 2012)(Olbrich *et al*., 2010)(Baltes and Diehl, 2014)(Oizumi *et al*., 2014). However, they do not aim at detecting architectural problems in early software development stage. They only perform retrospective analysis of software history data. Moreover, those techniques are not concerned in supporting the prioritization of critical code anomalies. When investigating the state of art, we have been able to identify 3 existing tools that provide prioritization capabilities in different development platforms.

The first tool is the InFusion (2014), which can be used for analyzing Java, C and C++ systems. Besides the statistical analysis for calculating code metrics (more than 60 metrics, the tool also provides numerical 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. The deficit index considers different measures such as size, encapsulation, complexity, coupling and cohesion metrics. The second tool is Code Metrics tool (2014), which is add-in for visual studio based on the .NET platform. It allows calculating a limited set of metrics, over which the tool will assign a score – namely *maintainability index* - to each of the analyzed code element. For all the metrics used in this tool, there is a pre-defined threshold. For this reason, the criteria using when ranking code anomalies for a given code element is based on the number of measures that are great than the threshold. Finally, the third tool is JSpIRIT (Vidal and Marcos, 2012), which is a semi-automated refactoring tool that helps developers focusing on the most critical problems in a software system. Therefore, the tool suggests a ranking of code anomalies, based on a combination of different criteria. The JSpIRIT tool ranks the most critical code anomalies and allows the developer to spend less time during the code anomaly refactoring process.

The main concern of using these tools is that the techniques they implement have some limitations: (i) they usually only consider the source code structure as input for detecting and prioritizing code anomalies; (ii) the ranking system disregards the architecture information of the code elements; and (iii) the user cannot define or customize their own criteria for prioritizing code anomalies. Our study intends to improve those results by proposing a model-based approach for identifying which code anomalies should be prioritized based on their architecture relevance. In this sense, we analyze different properties of the code elements that affect the architecture elements. Thus, we are able to support developers on the identification and ranking of those code anomalies that are more harmful to the overall software quality, and hence, should be refactored first.

2.4. Prioritization of Code Anomalies Supported by Blueprints

As mentioned in Section 2.3, the most popular mechanism for detecting critical code anomalies is the use of metrics (Lanza and Marinescu, 2006), once developers are able to define their own metrics-based detection strategy using a particular combination of measures and thresholds. The problem is that automatically collected measures purely represent properties of the source code structure. Therefore, the measures are often agnostic to the architectural design structure. In other words, the architecture decomposition is not explicit in the source code, the package or class structure often does not reflect the software architecture decomposition. As a consequence, developers often consider that all the modules and the respective measures have the same relevance to the software architecture design (Lanza and Marinescu, 2006)(Moha *et al*., 2010). The relevance could be otherwise captured by the analysis of architecture blueprints (Kruchten, 1995) usually available in software projects.

From our preliminary study (Guimarães *et al*., 2013), we gathered insights about which information provided in architecture blueprints could be used to improve the process of prioritizing critical code anomalies. For instance, we observed that the most critical code anomalies – and the architecture problem counterpart – tend to affect the communication between the architectural components. In addition, some

critical code anomalies are also related with bad architecture practices when realizing the architecture concerns.

For the best of our knowledge, the concept of blueprint (Kruchten, 1995) was firstly introduced to describe different views proposed in the "4+1 View Model", which represents software architecture according to five concurrent views. The use of blueprints has been exploited and assessed in different software engineering activities such as including process evaluation (Alegría et al., 2010) and test coverage analysis (Araya, 2011). In this thesis, we define architecture blueprints as informal models, with a high level of abstraction, initially created for communicating the architecture decomposition of a software system. In addition, architecture blueprints are: (i) often available in software projects; (ii) used to inform developers about the key design decisions defined by the system architect; and (iii) considered to be of informal nature since they do not necessarily require a formal specification language. It is important to mention that our work focuses on architecture blueprints that only capture a structural view of the software architecture, once most part of the architecture blueprints in real software projects are not multi-view (Kruchten, 1995). For instance, **Figure 2** presents an example of architecture blueprint that depicts a partial view of an architecture blueprint representing the main components of the Mobile Media system (see Chapter 3).

In this architecture blueprint, each component is represented by a rectangle, and it can have provided and required interfaces. Figure 2 also shows the system features each component is realizing. If a component is responsible for implementing one or more system features, it is decorated with a small circle that represents the concern that this component is responsible to handle. For instance, we can observe 5 concerns represented in the architecture blueprint, which are *Counting* (C), *Sorting* (S), *Exception Handling* (E), *Persistence* (P) and Favorites (F). Moreover, architecture blueprints can be distinguished from other types of models as they simultaneously hold three different properties (Lange and Chaudron, 2010)(Guimarães *et al*., 2014): level of abstraction, incompleteness and consistency. In the following, each of these properties will be described in more details.

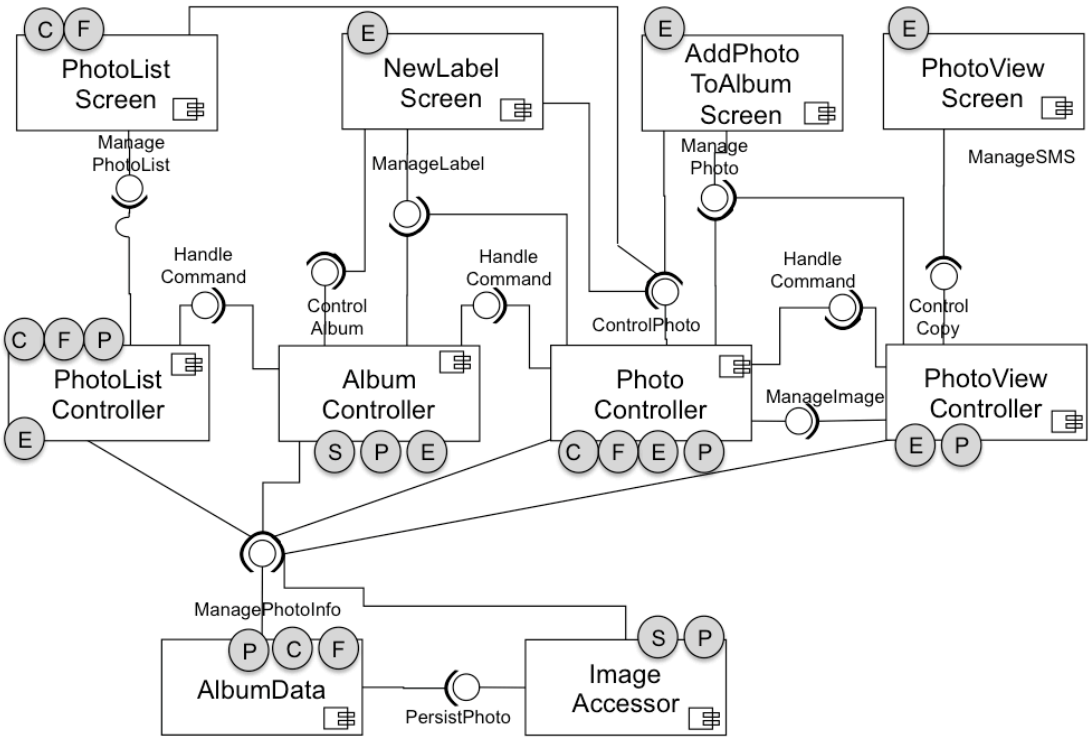

Figure 2 – Partial Architecture Blueprint of Mobile Media SPL

2.4.1. Level of Abstraction

As previously mentioned, architecture *blueprint* is a high-level model that represents the overall structure of a software system. Even considering a "high-level" of abstraction, different software developers will represent architecture design with different levels of details. However, when performing the mapping of information between architecture and source code elements, the relationship between elements in those two levels of abstraction might not be unitary. The mapping process consists in determining which elements in the source code correspond to each architecture element represented in the architecture blueprint. Therefore, independently of the way that different developers create blueprints on the development process, the high level of abstraction means that the mapping between architectural components and code elements is not unitary.

$LOA_B = \Sigma LO_{AC,I} / T_{AC}$, where:

$LO_{AC,I}$ = level of abstraction of an architectural component

T_{AC} = Total number of architectural components in the blueprint

In this sense, the level of abstraction of an architecture *blueprint* (LOA_B) is defined by the sum of the level of abstraction of each component (LOA_C) represented therein, divided by the total number of architectural components (T_{AC}). In turn, to calculate the level of abstraction of an architecture component it is necessary to compute the number of elements implemented to realize its functionalities. From those metrics, we can quantify the level of abstraction not only for each component, but also the level of abstraction of the set of architectural components represented in the architecture *blueprint*. It is important to mention that the level of abstraction must assume a value $0 < LOA \leq 1$, once the elements in the architecture *blueprint* should be mapped to at least one element on the software artifact in which the mapping was performed. If there is a component not mapped to a code element, this blueprint characteristic is captured by another properties being discussing below.

2.4.2. Completeness

This property defines that an architecture *blueprint* is complete when it characterizes all the components involved on the representation of the descriptive architecture. In this way, for each architectural element in the *blueprint* there must be at least one corresponding element in its counterpart mapping. For instance, when a mapping between the architecture blueprint and the source-code elements is performed, each architectural component must be associated with at least one code element (class or interface) responsible for realizing this component. Thus, in order to measure the completeness of the *blueprint* regarding the mapping with other software artifacts, 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. At the end, we can quantify the completeness of an architecture blueprints using the following equation:

$CB = 1 - (AE_{NM} / T_{AE})$, where:

AC_{NM} = *Architectural Component Not Mapped*

T_{AE} = *Total Number of Architectural Elements*

2.4.3. Inconsistencies

An architecture *blueprint* is said consistent when there is not any contradiction on the common information represented on the mapping between elements in different software artifacts. That is, the consistency occurs when the information represented in different software artifacts is well aligned. For instance, when analyzing the system implementation there might be an inconsistency between components that should be related with each other. However, a dependency relationship between the classes implementing those components could be observed when analyzing the source code.

In order to measure the inconsistencies of an architecture *blueprints*, we compute occurrences of: (i) *dependencies not mapped* – for example, classes that have dependencies with each other, but this communication should not exist according to the architecture blueprints; (ii) *inverted dependency* – cases where a architectural component AC_X uses a provided interface from component AC_Y, but the dependencies between the classes responsible for realizing those components are in the opposite way; (iii) *component with no interface* – cases where two classes in the source code have any kind of dependency, but the components they are realizing have no communication in the architecture blueprint; (iv) *interface with the same name* - two (or more) interfaces may not have the same name in the blueprint - even worse is when those interfaces belong to the same component; and (v) *components with the same name* – two (or more) components may not have the same name in the blueprint. After collecting those measures, we quantify the inconsistency rate (IR) according to the following equation:

- $IC_{BP} = \Sigma (NM_D + NI_D + NC_{WI} + NI_{SN} + NAC_{SM}) / T_{AE}$, where:
    - *Dependencies not mapped (NM_D)*
    - *Inverted dependency (NI_D)*
    - *Component with no interface (NC_{WI})*
    - *Interface with the same name (NI_{SN})*
    - *Components with the same name (NAC_{SM})*
    - *T_{AE} = Total number of architectural elements*

2.4.4. Mapping Architecture Blueprints to Source Code

This section illustrates the usefulness of mapping architectural elements (André *et al*., 2010)(Langhammer, 2013) into the source code. This section provides an example on how an anomalous code element can be related with architectural drift problems by analyzing the mapping of architectural elements in the source code. Taking as example the partial view of Mobile Media architecture, consider the *AlbumData* component. We will use this component as an example to illustrate the mapping of the architecture elements and the corresponding source code elements. The *AlbumData* component is realized by 5 code elements: *ImageAlbumData*, *MusicAlbumData*, *VideoAlbumData*, *MediaData* and *AlbumData*. In addition, 3 other code elements *extend* the abstract class *AlbumData*, namely *ImageAlbumData, MusicAlbumData* and *VideoAlbumData*..

Still considering this example, we can observe two architectural problems in which the *AlbumData* class might be involved. The first architectural problem is related to the *Overused Interface*, which characterizes an interface that exposes a non-cohesive heterogeneous data and is used by several other required interfaces from other components. In this case, the interface *ManageInfo* is used by 5 different architectural components, which are: *ImageAccessor*, *PhotoListController*, *AlbumController*, *PhotoController* and *PhotoViewController*. The problem is that the abstract class *AlbumData* is mainly responsible for realizing the *ManageInfo* interface since it defines many different methods for addition, deletion and update of different information.

In addition, the class is infected by two different code anomalies, namely *shotgun surgery* and *duplicated code*. Moreover, the code elements *ImageAlbumData, MusicAlbumData*, and *VideoAlbumData* are also related to problems on the realization of this interface, since they are infected with the code anomalies *duplicated code* and *small class*. This scenario also characterizes an architectural drift problem related to *external attractor component* (Section 5.3.1.1), which occurs when a provided interface is extensively used by other several external components. However, when detecting this type of architectural problems we should also take into

consideration the elements within external components that depend of one or more classes in the *AlbumData* Component.

```
public abstract class AlbumData {
(...)
try {
mediaAccessor.loadAlbums();
                } catch (InvalidImageDataException e) {
                                e.printStackTrace();
                } catch (PersistenceMechanismException e) {
                                e.printStackTrace();
                }
return mediaAccessor.getAlbumNames();
}

public MediaData[] getMedias(String recordName) throws UnavailablePhotoAlbumException
MediaData[] result;
try {
result = mediaAccessor.loadMediaDataFromRMS(recordName);
} catch (PersistenceMechanismException e) {
                throw new UnavailablePhotoAlbumException(e);
(...)
return result;

}
public void createNewAlbum(String albumName)
                throws PersistenceMechanismException, InvalidPhotoAlbumNameException {
mediaAccessor.createNewAlbum(albumName);
}

public void deleteAlbum(String albumName)
                throws PersistenceMechanismException{
mediaAccessor.deleteAlbum(albumName);
}

public void addNewMediaToAlbum(String label, String path, String album)
                throws InvalidImageDataException, PersistenceMechanismException{
mediaAccessor.addMediaData(label, path, album);

}
```

Figure 3 - Mapping Component to Source-Code

Another architectural problem associated with the implementation of the abstract class *AlbumData* is the fact that it realizes two different concerns in the system. For instance, Figure 3 illustrates the lines of code responsible for realizing the *exception handling* (dark grey) and *persistence* (light grey) concerns. Classes within external components should address the exception handling, since the *AlbumData* architectural component mainly responsible for addressing the *persistence* concern. In

this case, the mapping of the software architecture into the source code could be used to support the detection of, at least, two architectural problems. The first is related to the problem *Misplaced Concern*, since the *AlbumData* class is implementing the *exception handling* concern, which is not predominant on its enclosing component (see Section 5.3.2.2). The other problem is related with the problem *Concern Overload* (see Section 5.3.2.1), as the *AlbumData* class and its subclasses (*ImageAlbumData*, *MusicAlbumData* and *VideoAlbumData*) are responsible for dealing with too many concerns or responsibilities in the Mobile Media system. It is also important to mention that the aforementioned subclasses are all infected with the same instances of code anomalies, which are, respectively, *duplicated code* and *small class*.

The examples above show how the reasoning about the mapping of architectural concerns into the source code can help to reveal certain types of critical code anomalies (and their architectural problem counterparts). These anomalies and problems could not be easily revealed if there is no mapping of the software architecture into the program.

2.5. Summary

This chapter presented the main concepts addressed in this thesis. It also presented an overview of existing studies and a critical discussion of their limitations. Section 2.1 presented the definitions of the main terms discussed throughout this research work, such as software architecture, prescriptive architecture and descriptive architecture. In addition, we also presented the definition of architectural degradation and discussed the symptoms of architectural drift.

After that, we discussed how critical instances of code anomalies might be related to architectural drift problems (see Section 2.2). Code anomalies might occur in an isolated way. However, instances of code anomalies related to architectural problems might co-exist in code elements implementing architectural components. The co-occurrence of critical instances of code anomalies has been investigated as one of the factors associated to architecture degradation symptoms (Section 2.2.2).

Nevertheless, software developers are usually expected to decide which code anomalies should be refactored first. Although several detection strategies have been proposed in the literature, they do not provide means for prioritizing and ranking code anomalies (Section 2.3). The fact is that existing strategies only focus on source code analysis and metrics, and therefore, they do not consider the system's descriptive architecture decomposition. In this sense, there is a lack of support on how prioritizing and ranking critical code anomalies based on their architectural relevance. In addition, there is no empirical study that investigates how the system's descriptive architecture could be exploited as means to assist developers when properly prioritizing and ranking critical code anomalies (Section 2.3.1).

In this sense, we investigated how the use of blueprints, representing the systems' descriptive architecture, can be used as means for prioritizing and ranking critical code anomalies according to their architecture relevance (Section 2.4). In addition, we introduced a set of properties that should be applied in architecture design models in order to check if they could be classified as blueprint. In addition, the proposed properties can be used to guarantee that architecture blueprints have a minimum quality required so that it can be used when prioritizing and ranking critical code anomalies. Finally, we provide an example on how architectural components can be mapped to anomalous code elements, as well as we describe what type of architectural drift problems they can be associated.