2
Background and Related Work

As software systems evolve, their size and complexity grow. In this context, the progressive manifestation of code anomalies is a key symptom of architecture quality decline. Furthermore, recent studies have shown that there is a strong correlation between code anomaly occurrences and architecture problems (MACIA *et al.*, 2012b). When those anomalies are not detected and removed early, the maintainability of software projects can be irreversibly compromised, and, eventually, a complete redesign is inevitable (MACCORMACK *et al*., 2006; EICK *et al*., 2001).

Although many studies have broadly investigated approaches for detecting code anomalies (WETTEL *et al*., 2008; LANZA *et al.*, 2006; MUNRO *et al.*, 2005; MARINESCU *et al.*, 2004), identifying those that are more likely to cause architecture problems is still a challenging task. In fact, recent studies have shown that automatically detected code anomalies are seldom good indicators of architectural problems (MACIA *et al.*, 2012a). Furthermore, even when there is tool support for detecting code anomalies, developers seem to invest more time removing those that are not related to architectural problems (MACIA *et al.*, 2012b; MURPHY-HILL *et al.*, 2009). Thus, developers tend to prioritize refactoring code structures that do not affect hierarchies or public interfaces, and, therefore, could not affect the architecture design. The identification and ranking of such code anomalies as soon as possible could improve refactoring effectiveness, by guiding developers into solving the right problems.

In this context, this chapter outlines the basic terminology we used throughout the development of our study (Section 2.1). Next, it outlines previous research on refactoring state-of-practice (Section 2.2). Section 2.3 provides a brief background on code anomalies and how they are related to architecture problems. Next, we describe approaches for both detecting code anomalies (Section 2.4) and recommending refactorings (Section 2.5). We decided to investigate refactoring recommendation systems as they could indicate relevant improvement

opportunities for the source code. Those indications often comprehend the detection of code anomalies. Finally, we also outline existing work on ranking code anomalies, although such mechanisms are still rather scarce (Section 2.6).

2.1. Basic Terminology

Software architecture is the structure of a system, which comprises software modules, the interfaces of those modules, and the relationship among them (BASS *et al.*, 2003). Architecture *components* are architectural entities which encapsulate a subset of the system's functionalities (TAYLOR et al., 2009). In the context of this dissertation, we also use the term *architecture role* to represent the architecture decisions realized by a code element. More than one code element might be required to realize an architecture role. For example, considering a system that implements the MVC pattern, a class could contribute to the implementation of the role Model, View or Controller (BUSCHMANN *et al.*, 2007).

The *intended architecture* comprehends explicit decisions made by the designers on the selection of components, their interactions, and their constraints (TAYLOR *et al.,* 2009). On the other hand, the *implemented architecture* describes how the system has been actually built (TAYLOR *et al.,* 2009). In software projects, the implemented architecture often does not match the intended architecture (TAYLOR *et al.,* 2009). Many prescribed architecture decisions can be undesirably violated by the actual implementation of a system. Those mismatches between the intended and the implemented architectures are called *violations.* As the number of violations and modularity problems increase, the architecture is known to *degrade* (HOCHSTEIN and LINDVALL, 2005).

A frequent symptom of *architecture degradation* (HOCHSTEIN and LINDVALL, 2005) is the progressive manifestation of code anomalies. Code anomaly, also referred to as *code smell*, is a symptom in source code of a deeper maintainability problem (FOWLER *et al.*,1999). Examples of code anomalies are Long Method and Inadequate Name. Code anomalies can affect different types of code structures – or *code elements.* In the context of this dissertation, code

elements refer to any implementation structure – such as classes, methods or constructors.

When code anomalies have a negative impact on the system's architecture design, contributing to architecture degradation, we call them *architecturally relevant* code anomalies. In this sense, *relevant* means it is harmful or related to architecture in some level. We chose to use the term architecturally relevant because it has been widely used on our previous studies (ARCOVERDE *et al.*, 2012; MACIA *et al.*, 2011; MACIA *et al.*, 2012a; MACIA *et al.*, 2012c).

Code anomalies are often removed through *refactorings*. Refactoring is the process of changing a system's design structure without changing its behavior, in order to improve its readability and maintainability (FOWLER *et al.*, 1999). When refactoring is performed on a publicly visible element, like public classes or method signatures, we call it an API-level refactoring (KIM *et al.*, 2011) or a *high-level* refactoring (MURPHY-HILL *et al.*, 2009). Low level refactorings, on the other hand, are those applied to internal code structures, such as method bodies.

2.2. Empirical Studies on Refactoring

The first part of our research was focused on understanding the longevity of code anomalies; for doing so, we conducted an empirical study on refactoring habits, and their prioritization (ARCOVERDE *et al.*, 2011).

Previous studies were dedicated to understanding common refactoring practices, as well as identifying how and when they are routinely applied. Murphy-Hill has recently presented an extensive study on how programmers refactor, identifying several common refactoring habits (MURPHY-HILL *et al.*, 2009). They found that refactorings are performed frequently, and that about half of them are not high-level. Dig *et al*. (2005) investigated the role of refactoring on APIs evolution, and found that 80% of the changes that could break client applications are high-level refactorings. Such studies motivated us into investigating refactoring habits, looking for reasons why some refactorings – in this case, high-level refactorings – are postponed or neglected.

Weißgerber and Diehl (2006) found that, following some revisions where refactoring took place, there was an increasing ratio of bugs reported. Those results were confirmed by Kim *et al*. (2011), who found an increase in the number of bug fixes after refactorings. Xing and Stroulia (2006) analyzed the Eclipse code evolution and found that 70% of the observed code changes were refactorings. Moreover, they found that state-of-art IDE's support only a subset of commonly applied low-level refactorings, lacking support for more complex ones. Those results helped us in understanding why architecturally relevant code anomalies are seldom removed, as those anomalies intuitively require high-level refactorings (MACIA *et al.*, 2012b).

2.3. Code Anomalies and Architecture Problems

The negative impact of code anomalies on the system's architecture has been analyzed by several studies documented in the literature. MacCormack *et al.* (2006) reported that the Mozilla browser's code was overly complex and tightly coupled, therefore hindering its maintainability and ability to evolve. Such problems were the main causes for its complete re-engineering in 1998. This effort consumed about five years to rewrite over seven thousand source files and two million source lines of code (Godfrey and Lee, 2000).

Eick *et al.* (2001) described how the modularity of the architecture of a large telecommunication system degraded between 1989 and 1996. In particular, the relationship among the architectural modules increased over time. This was the main cause why the system's architectural modules were not independent anymore and, consequently, further changes were not possible.

Hochstein and Lindvall (2005) investigated the main causes for architecture degradation, indicating that refactoring specific code anomalies could help to avoid it. Wong et al. (2011) also identified that duplicated code was related to design defects – more specifically, design violations.

2.4. Detection of Code Anomalies

Many authors have proposed techniques and tools for automatically detecting code anomalies (WETTEL *et al.*, 2008; LANZA and MARINESCU, 2006; MUNRO *et al.*, 2005; MARINESCU, 2004). Most of them are based on exploiting information that is extracted from the source code structure, relying on the combination of static code metrics and thresholds into logical expressions. Those mechanisms are known as *detection strategies* (MARINESCU, 2004). The example below illustrates a well-known detection strategy (LANZA and MARINESCU, 2006) for identifying *God Classes.* This strategy and its thresholds have also been used in previous studies (OLBRICH *et al.*, 2009; 2010).

GodClass(c) = (WMC(c) >= 47) ^ (TCC(c) < 0.3) ^ (ATFD(c) > 5)

In this detection strategy:

- *c* is the class under analysis
- WMC is the Weighted Method Count, which is the sum of the cyclomatic complexity of all methods within *c*
- TCC is the Tight Class Cohesion, representing the number of connected methods, i.e., methods that access the same instance variables (McCABE, 1976)
- ATFD, or Access to Foreign Data, is the number of attributes in foreign classes accessed by class *c*

The main limitation of detection strategies for identifying relevant code anomalies is that they are solely based on information that emerge from the source code structure. That is, they disregard other kinds of information (e.g. architectural information) that could be exploited with the source code in order to reveal architecturally relevant code anomalies. Moreover, they only consider individual occurrences of code anomalies, rather than analyzing the relationships between them. These limitations are the main reasons why current mechanisms are unable to support the detection of code anomalies responsible for introducing architectural problems (MACIA *et al.*, 2012a).

Moreover, the effectiveness of automatically-detected code anomalies using detection strategies has been recently studied under different perspectives: Mantyla and Lassensius (2006), for instance, investigate to what extent

automatically-detected code anomalies can be used as a basis for subjective evaluation of code evolvability. Olbrich *et al.* (2009;2010) and Khomh *et al.* (2009) analyze whether the number of code anomalies increases over time and to what extent the anomalies influence the frequency of changes on code elements. More recently, Macia et al. (2012a) investigated to what extent detection strategies accurately localize code anomalies related to architecture problems. This study in particular has shown that more than 50% of the automatically detected code anomalies were not correlated to architecture problems, while more than 50% of the relevant code anomalies were not detected. Those results are directly related to our research, as they motivated our search for characteristics that could help to identify architecturally relevant code anomalies.

This section described the most common mechanisms for detection of code anomalies. Next, we analyze some tools aimed at detecting refactoring opportunities – or refactoring recommendation systems – and some specific code anomalies detectors.

2.4.1. Refactoring Recommendation Systems

Even though refactoring tools are available for most development environments, developers seem to limit their use on low level refactorings (MURPHY-HILL *et al.*, 2009). Therefore, as public interfaces and hierarchies are not changed as a result of low level refactoring, code anomalies related to architecture problems tend to linger. In fact, we have observed in recent studies that, for 8 analyzed systems, only 40% of all code anomalies causing architecture problems were refactored in some point of the software evolution.

Refactoring recommendation systems could help developers to identify code anomalies removal opportunities. Previous studies have proposed a number of techniques for refactoring recommendation. Vidal *et al.* (2012) propose an expert software agent that assists developers when refactoring an object-oriented system into an aspect-oriented one. It analyzes the user's interaction history for improving the agent's effectiveness over time, guiding developers through the steps they should take. Xi *et al.* (2012) also propose a refactoring recommendation mechanism based on the observation of manual refactoring steps. Their goal is to

monitor common sequences of previous changes on code structures in order to detect the occurrence of refactorings, and recommending their automation on-the-fly, while the developer is programming. The recommendation is based on previously observed steps needed for performing the refactorings supported.

Both approaches can be used for assisting developers into safer refactorings, minimizing the risks of introducing breaking changes. However, they are not aimed at maximizing their gains, prioritizing those anomalies that harm the architecture the most.

2.4.2. Detection Tools for Code Anomalies

There are currently many tools dedicated to the identification of code anomalies, targeting many different development environments and languages. We analyzed three of them: Hist-Inspect (MARA *et al.*, 2010), Semmle Code (SEMMLE CODE, 2012) and NDepend (NDEPEND, 2012). The main reason why we chose to analyze those detection tools is that they all consider other sources of information in their detection mechanisms, in addition to the source code structure.

Hist-inspect. Mara *et al.* (2011) proposed a tool called *Hist-Inspect* to support both the definition and the automatic application of history-sensitive detection strategies. The tool supports conventional metrics, such as coupling (CBO) and lines of code (LOC) (LANZA and MARINESCU, 2006), and history-sensitive metrics (MARA *et al.*, 2011). Those metrics are calculated by evaluating conventional metrics through the system's evolution, and computing their values in each revision. The resulting metric represents how the measured characteristic evolved. For example, Hist-Inspect calculates the *rpiLOC* metric (MARA *et al.*, 2010b), which calculates the average variation for the number of lines of code throughout the software evolution.

Hist-Inspect detection strategies, or rules, are defined in XML, as illustrated below.

Listing 1

```
01:   <?xml version="1.0" enconding="UTF-8"?>
02:   <rule-catalog>
03:         <rule id="sampleRule"
04:               anomaly="unexpectedComplexGUI"
05:               expression="LOC >= 100 || CC >= 5"/>
06:   </rule-catalog>
07:   <anonaly-catalog>
08:         <anomaly id="unexpectedComplexGUI">
09:               applyTo="class"/>
10:   </anomaly-catalog>
```

Listing 2 declares a strategy for detecting complex GUI classes, which may implement more responsibilities than desired (lines 03-05). It also defines an anomaly named `unexpectedComplexGUI` that may be manifesting in classes (lines 07-09). This strategy checks all system classes and retrieves the ones who have 100 or more lines of code or whose cyclomatic complexity is greater or equals to 5. These metrics were selected for illustrative purpose, and other metrics could be used for detecting similar or different anomalies, including the aforementioned history-sensitive metrics.

Hist-Inspect uses history information for improving the accuracy of code anomalies detection strategies – in terms of decreasing their number of false negatives and false positives. However, is suffers from two main limitations: first, as it is solely based on software evolution, it can only be used over systems that present a reasonable number of distinct revisions. Although the *change-proneness prioritization heuristic* (Section 3.1.1) we proposed also depends on the existence of different software versions, it can be combined with other heuristics that do not present such requirement. Moreover, Hist-Inspect was based on the analysis of different software releases; that means it ignores the intermediary changes that might have occurred between two major releases. On the other hand, the *change-proneness prioritization heuristic* operates over the *commits* executed by developers, taking every change made to the system's code into consideration.

Semmle Code. Semmle Code (SEMMLE CODE, 2012) is a tool that provides a source code query language (VERBAERE *et al.,* 2008), allowing maintainers to define their own custom design violations. Those queries take into consideration several properties of source code elements, such as dependencies, lines of code and depth of inheritance tree. Therefore, maintainers are able to

define design rules and constraints and to verify them automatically, in order to find architecture violations.

The listing below shows an example that measures the depth of inheritance tree using the tool's source query language:

Listing 2

```
01:   from MetricRefType t, int d
02:    where t.fromSource() and d = t.getInheritanceDepth() and d > 6
03:   select t, d order by d desc
```

Semmle Code has two main limitations for detecting architecturally relevant code anomalies: first, it requires explicit documentation for the system's design rules on its own DSL. Therefore, when this information is absent, outdated or incorrect, its ability to detect design violations will be compromised. Second, it only considers the source code structure as a source of information for detecting violations. Therefore, it cannot identify architecture violations – as it is not possible to define queries that explore architecture information. For example, one cannot define rules that detect God Classes based on the number of responsibilities (or concerns) implemented by a given class, as that information is not supported.

NDepend. Targeting the .NET platform, NDepend also provides a flexible code query language for defining detection strategies, as well as many different visualization features. It also allows developers and architects to enforce software quality through standard and custom rules that can be integrated to the development environment. For example, the rules mechanism can be used to identify refactorings that could possibly introduce breaking changes, by analyzing recent changes and test coverage reports. This situation is illustrated in the example below:

```
warnif count > 0 from t in codeBase.OlderVersion().Application.Typ
es
where t.IsPubliclyVisible &&

  // The type has been removed and its parent assembly hasn't been
 removed ...
     ( (t.WasRemoved() && !t.ParentAssembly.WasRemoved()) ||

     // ... or the type is not publicly visible anymore
       !t.WasRemoved() && !t.NewerVersion().IsPubliclyVisible)

select new { t,
             NewVisibility = (t.WasRemoved() ? " " : t.NewerVersio
n().Visibility.ToString()) }
```

This query warns if a publicly visible type is not publicly visible anymore or if it has been removed. For doing so, NDepend is able to connect to software version control and issue tracking systems – from where code coverage and test reports information can be retrieved. The integration of such properties allows NDepend to run queries that explore not only the source code structure but also history-sensitive information – as exemplified by the *WasRemoved* constraint. By exploring evolution information, NDepend is also able to detect code anomalies that have been postponed for many revisions. Additionally, two of our prioritization heuristics also consider information retrieved from source version control and issue tracking systems. However, they apply such information for ranking code anomalies, while NDepend uses it for inspecting the source code and possibly detecting design problems.

It is not possible, however, to input architecture information into NDepend, in order to aid relevant anomalies detection. Moreover, once detected, all the violations and code anomalies are considered equally harmful. Thus, as the amount of problems increases, it becomes harder to identify which ones should be prioritized.

Finally, conventional mechanisms for detecting code anomalies do not support the ranking of code anomalies according to their harmful degree on systems' architecture. Consequently, developers have to manually inspect each suspect reported by those mechanisms. For each code anomaly detected, they must determine whether it really represents threats to the system architecture and, then, decide *ad hoc* which one should be prioritized. This process requires a huge effort when the list of reported suspects is large and covers many parts of the system, as it usually occurs.

However, some specific detection tools offer support for ranking code anomalies. Next section describes two of them.

2.5. Ranking Systems for Code Anomalies

As shown in the previous sections, there are many tools and techniques available for detecting code anomalies. However, as systems grow, the number of detected code anomalies also tends to increase (OLBRICH *et al.*, 2009), and can eventually become unmanageable. Furthermore, maintainers are expected to choose which refactorings they are performing first. Some of the reasons why that choice is necessary are (i) time constraints and (ii) attempts to find the correct solution when restructuring a large system. In this context, ranking code anomalies could be an asset for increasing the effectiveness of such refactoring efforts.

However, most of the aforementioned works do not focus on ranking or prioritizing code anomalies. This section describes two well-known tools that provide ranking capabilities, for different development platforms.

Code Metrics is a .NET based add-in to the Visual Studio development environment. It is able to calculate a limited set of metrics – lines of code, cyclomatic complexity and afferent coupling. Once those metrics are calculated, Code Metrics assigns a "Maintainability Index" score to each of the analyzed code elements. This maintainability score is based on the combination of the metrics calculated for that code element. For each supported metric, there is a pre-defined threshold – which cannot be configured. In that sense, the ranking criteria is based on the number of measures for a given code element that are greater than the thresholds.

Some limitations of this work are: (i) it only takes into consideration the source code structure as input for identifying code anomalies, (ii) the ranking system disregards the architecture role of each analyzed code element and (iii) users cannot define their own ranking criteria for prioritizing code anomalies.

Infusion is a tool for analyzing Java, C and C++ software projects. It is able to calculate over 60 different code metrics, and to detect code anomalies such as Data Classes (FOWLER *et al.*, 1999). Besides providing static analysis features for calculating code metrics, it also associates numerical scores to all detected anomalies. Those scores measure the negative impact a given anomaly has on the overall systems' quality. By combining those scores, a quality deficit index is

calculated for the entire system. That index takes into consideration size, complexity, encapsulation, coupling and cohesion metrics. Figure 2 illustrates a report generated by InFusion.

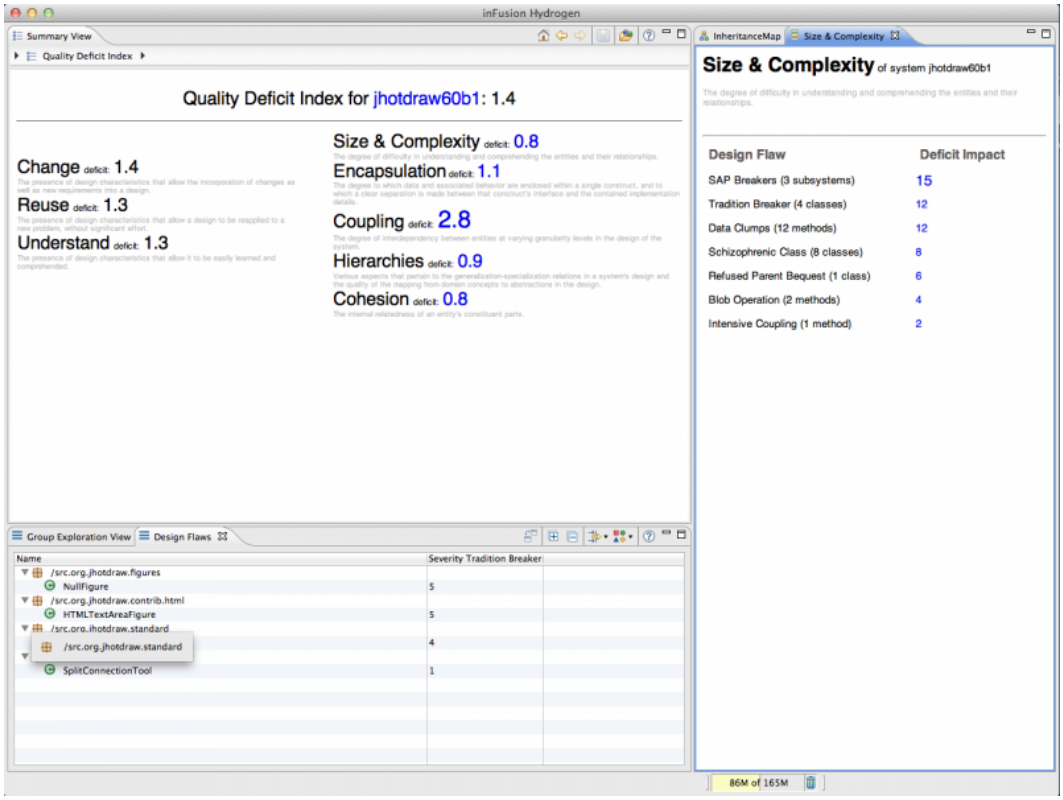

Figure 2 - InFusion report example

As InFusion is a proprietary tool, their ranking criteria are not clear. However, as it can be noticed in the report shown in Figure 2, three different maintainability indexes are taken into consideration (change, reuse and understand). Furthermore, as only the source code structure is taken into consideration as a source of information, the detection mechanism is not enriched by actual architecture information.

Our study intends to complement such results by proposing an approach for identifying which anomalies should be prioritized – or more promptly refactored – based on their architecture relevance. We analyze different properties of the code elements they affect, such as change-proneness and error-proneness for identifying which anomalies are more harmful to the overall architecture and should be removed first.