2014 IEEE 38th Annual International Computers, Software and Applications Conference

Exploring Blueprints on the Prioritization of Architecturally Relevant Code Anomalies

A Controlled Experiment

Everton Guimaraes, Alessandro Garcia
Informatics Department
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
{eguimaraes, afgarcia) at inf.puc-rio.br

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

***Abstract*—The progressive insertion of code anomalies in evolving programs may lead to architecture degradation symptoms. Several approaches have been proposed aiming to detect code anomalies in the source code, such as God Class and Shotgun Surgery. However, most of them fail to assist developers on prioritizing code anomalies harmful to the software architecture. These approaches often rely on source code analysis and do not provide developers with useful information to help the prioritization of those anomalies that impact on the architectural design. In this context, this paper presents a controlled experiment aiming at investigating how developers, when supported by architecture blueprints, are able to prioritize different types of code anomalies in terms of their architectural relevance. Our contributions include: (i) quantitative indicators on how the use of blueprints may improve process of prioritizing code anomalies; (ii) a discussion of how blueprints may help on the prioritization processes; (iii) an analysis of whether and to what extent the use of blueprints impacts on the time for revealing architecturally relevant code anomalies; and (iv) a discussion on the main characteristics of false positives and false negatives observed by the actual developers.**

Keywords — Software Architecture; Code Anomalies; Architecture Blueprints; Software Metrics;

I. Introduction

The unavoidable evolution of software systems results on the increase of its size and complexity, which in turn, may lead to architecture degradation. Architecture degradation [9] is often a direct consequence of the progressive insertion of code anomalies [9][14][15] in the source code. Hence, if these code anomalies are not systematically removed, the system's architecture may degrade. Some authors have claimed that code anomalies – popularly known as *code smells* [7], such as instances of God Class, Divergent Changes and Shotgun Surgery – are consistently reifications of architectural problems in a program [7][9]. In addition, it is an arduous task (if not impossible) to identify degradation symptoms directly on the architecture specification. The reason is that architectural decisions usually are not entirely specified in real software projects; usually, they are only partially represented as blueprints [10] – i.e. informal and high-level models of a system's architecture.

Recent studies confirmed, more than 80% of the architectural problems are usually related to well-known code anomalies [15]. If a code anomaly is related to an architectural problem [13][14], we call it *architecturally relevant code anomaly*. Examples of code anomalies are instances' of Shotgun Surgery and God Class affecting the implementation of a component interface. They might represent architectural problems, including bloated component interfaces, concern overload in components, scattered architectural functionality [8]. However, other studies [15][16] revealed a high proportion of code anomaly instances, detected using automated code analysis [11][18][19], are not architecturally relevant. According to these findings [14][15], only a limited proportion of code anomalies – around 40% – are related to software architecture problems [8].

In this sense, developers need to be provided with means to detect, prioritize and remove architecturally relevant code anomalies. When the anomalies are not prioritized and removed early in a software project, the cost to remove them later is usually high or prohibitive [15]. The most popular mechanism for detecting architecturally relevant code anomalies is the use of metrics [11]. Developers can define their own metrics-based detection strategy using a particular combination of measures and thresholds. Existing detection strategies provide (semi) automatic means for detecting code anomalies [18]. However, metrics-based detection strategies often fail to support automatic detection of architecture-relevant anomalies [11][18][19]. The reason is that automatically collected measures purely represent properties of the source code structure. The measures are often agnostic to the architectural design structure. That is, the architecture decomposition is not explicit in the source code; for instance, the package or class structure often does not reflect the software architecture decomposition. As consequence, developers often consider that all the modules and their respective measures have the same relevance to the architecture design [11][18][19]. The relevance could be otherwise captured by the analysis of blueprints [10] available is a software project.

Architecture blueprints [10] are often available in software projects from the design outset as they are used to communicate key architectural decisions. The use of blueprints has been exploited and assessed in different software engineering activities, including process evaluation [1], model transformation optimization [5] and test coverage analysis [2]. In our context, a research question that needs to be investigated

0730-3157/14 $31.00 © 2014 IEEE
DOI 10.1109/COMPSAC.2014.57

IEEE computer society

is to what extent the use of architecture blueprints would enhance the prioritization of architecturally relevant code anomalies (see Section III). Therefore, a controlled experiment was conducted in three different universities and the subjects are students with different working experience and technical knowledge. As main contributions (see Section V), we have evaluated: (i) whether the use of architecture blueprints may impact on precision and recall when prioritizing architecturally relevant code anomalies; (ii) what are the main characteristics of false positives and false negatives identified by the subjects during the controlled experiment; and (iii) whether the use of architecture blueprints may bring (or not) additional effort regarding the Time spent for prioritizing architecturally relevant anomalies; and (iv) a discussion about the impact of False Positives and False Negatives on the prioritization process (see Section VI).

II. BACKGROUND

This section describes the main concepts relevant to our study. Section II.A introduces the concept of code anomaly. Section II.B introduces the categories of software metrics, while Section II.C describes the concept of architecture blueprints explored in our study.

A. Code Anomalies and Architectural Problems

Code anomalies are implementation structures that possibly indicate deeper design problems [7], and hence, might hinder software maintenance tasks. A code anomaly is often and popularly referred as "code *smell"* [7]. Instances of code anomalies can contribute directly to the software architecture degradation [14][15]. Code anomalies, in fact, are more harmful when they directly contribute to the software architecture degradation [7][14][15]. The term architecture degradation is often used to refer to the continuous quality decay of architecture design when software systems evolve over time [9]. Thus, as the software architecture degrades, the maintainability of software systems can be compromised irreversibly. Classical examples of architectural problems are Ambiguous Interface [6], Component Envy [6], as well as cyclic dependencies between software modules [17].

In order to prevent architecture degradation, software development teams should progressively detect and remove architecturally relevant code anomalies. The code anomalies under investigation may not cause critical problems immediately, but their negative on the overall architecture decomposition may be observed in the long run [7][14][15]. Many researchers have investigated the impact of code anomalies on exerting undesirable modifications in the source code [13][17]. Recent studies revealed that the code structures affected by code anomalies suffer more changes during software maintenance [17]. However, as previously mentioned, it has been observed that metrics-based detection of code anomalies is not enough to support the prioritization of architecturally relevant ones [14].

B. Studied Code Anomalies and Metrics

For our controlled experiment, we decided to focus on three types of code anomalies occur in software systems [17], and are often related to architectural problems [14][15]: God Class, Shotgun Surgery and Divergent Change. We also focused on a subset of anomaly types in order to be able to design and perform a controlled experiment that needs to be run in approximately one hour. Each of these code anomalies is briefly described in the following: (i) God Class - it represents a large class that has with too many responsibilities in the system [20]; (ii) Shotgun Surgery - it represents those cases when a class change leads to a lot of small changes in many different classes [7]; and (iii) Divergent Change - it is identified when the same class is commonly changed in different ways and for different reasons [7].

C. Software Metrics

Software metrics are the common mechanisms to support software quality evaluation, in particular for detecting code anomalies [11][18]. Table I summarizes the metrics used for detecting software anomalies in our experiment. We decided to focus on a subset of the most commonly used metrics because a larger set of metrics could make the subjects' analysis harder. The selected traditional metrics have been effectively used in previous work to detect the studied code anomalies [18] (Section B). In previous studies, the use of concern-based metrics was also required to detect these code anomalies [18] with reduced rate of mistakes. Therefore, we also selected a set of concern-based metrics. Those metrics overcome limitations of conventional metrics on the detection of code anomalies by allowing the quantification of concern properties in the source code [11]. For instance, they make it easier to detect if a class is realizing too many concerns (responsibilities) – i.e. God Classes. These metrics might also help to quantify the number of concerns are implemented in a class realizing a given component, as well as how the architectural concerns are spread through the source code.

TABLE I. TRADITIONAL AND CONCERN-BASED METRICS

Traditional Metrics	**Brief Definition**
Number of Attributes (NOA)	It counts the number of attributes in a class.
Number of Methods (NOM)	It counts the number of methods in a class.
Weighted Methods Per Class (WMC)	It counts the number of methods and their parameters in a class.
Lines of Code (LOC)	It counts the total number of lines of code.
Lack of Cohesion (LCOM)	It is measured by the number of pairs of methods that do not access attributes in common by the number of pairs of methods that access attributes in common.
Coupling Between Objects (CBO)	It counts the number of classes in which a given class either calls its methods or accesses its attributes.
Concern-based Metrics	**Brief Definition**
Number of Concerns per Component (NCC)	It counts the number of concerns in each class
Concern Diffusion over Components (CDC)	It counts the number of classes affected by the concern implementation.
Concern Diffusion over Operations (CDO)	It counts the number of methods affected by the concern implementation.
Concern Diffusion in LOC (CDLOC)	It counts the number of transition-points for each concern through the lines of code.

III. ARCHITECTURE BLUEPRINTS

The concept of blueprints [10] was introduced to describe each of the views proposed in the "4+1 View Model", which represents software architecture using five concurrent views. Architecture blueprints are informal models, with a high level of abstraction, initially created only to communicate the system architecture decomposition. Architecture blueprints are often available in software projects and are used to inform developers about the key design decisions defined by the system architect. Moreover, blueprints are considered to be of informal nature since they do not necessarily require a formal specification language. Our work focuses on architecture blueprints that capture only a structural view of the software architecture, once most part of the architecture blueprints in real software projects are not multi-view [10].

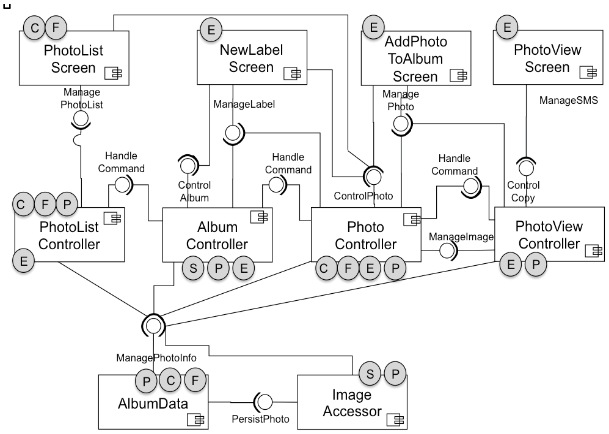

Fig. 1. Partial view of Mobile Media Architecture

Figure 1 presents an example of architecture blueprint; it depicts a partial view of an architecture blueprint representing the main components of the Mobile Media architecture. In this blueprint, each component is represented by a rectangle, and it can have provided and required interfaces. Figure 1 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. Moreover, architecture blueprints can be distinguished from other types of models as they simultaneously hold four different properties [1], which are: level of abstraction, incompleteness, inconsistency and bad-formedness. In the following, each of these properties are described in more details.

A. Level of Abstraction

Architecture *blueprint* is a high-level model that represents the overall structure of a software system. Even considering a "high-level" of abstraction, different developers will represent architecture design with different levels of details. However, when the mapping between the architecture blueprint and source code elements is performed, the relationship between the elements at these two levels are not unitary. Mapping is the process of determining elements in the source code that correspond to each element in the architecture *blueprint*. In summary, 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 are not 1 to 1. As architecture *blueprints* was firstly introduced in the "4+1 Model View" [10], in our study we used the development and logical views, which are here represented, respectively, by the component and class diagrams.

TABLE II. LEVEL OF ABSTRACTION OF ARCHITECTURAL BLUEPRINTS

$LOA_b = \Sigma\, LOA_c / NTC$ (1)	$LOA_c = 1 - (1/ T_{AC})$ (2)

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}) – see Table II. 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 $0 < LOA \leqslant 1$, 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.

B. Completeness

This property defines that an architecture *blueprint* is complete when it characterizes all the components involved on the representation of the implemented 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 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.

C. Consistency

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 exists in 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 an 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) by summing the total number of inconsistencies for each case described above.

D. *Well-formedness*

Each blueprint usually is not well formed. A well-formed model must follow rules and conventions of a modeling language. Otherwise, such occurrence of not well-formed model may reduce its comprehensibility. For instance, the UML specification [25] defines rules for well formedness of model elements. Most part of these rules are predefined and/or checked in modeling tools. As example of violations in the architecture blueprints occurs when components, interfaces and relationships have no name. Finally, the analysis of this property is performed only using a model that represents the high-level architecture design.

IV. STUDY METHODOLOGY

In our experiment, the added value of architecture blueprints is analyzed as a means to guide the prioritization of architecturally relevant code anomalies. The value of architecture blueprints has only been successfully assessed in other domains, including (i) implementing a visual approach for software process model evaluation based on architectural views [1]; and (ii) implementing visual aid to assess test coverage [2], where blueprints were used to find errors and problems in the specification. Our research question and study hypotheses are introduced in Section IV.A, while the experimental procedures of our controlled experiment is introduced in Section IV.B. The code anomalies reference list used as an oracle to compare the results is presented in Section IV.C. Finally, the target system used in our investigation is described in Section IV.D.

A. *Research Question and Study Hypotheses*

The main goal of this study is summarized by our research question (RQ): *Can the use of architectural blueprint, as additional artifact, improve the prioritization of architecturally relevant code anomalies?* The main motivation of our research question is due to the belief that early prioritization of code anomalies may prevent architecture degradation during the system evolution. In this work, the prioritization of architecturally relevant code anomalies can be understood as the process of distinguishing detected anomalies that are relevant to the architecture design. However, it is not clear that blueprints can improve the prioritization process due to its properties discussed in the previous section, such as high-level of abstraction and incompleteness.

In order to evaluate to what extent the use of architecture blueprints may improve the prioritization of architecturally relevant code anomalies, three measures were used: Precision, Recall and Time. We also analyzed the main characteristics of classes identified as False Positives or False Negatives (see Section VI). The analysis allows understanding why the sole use of metrics fails to correctly identify instances of architecturally relevant code anomalies. In this sense, our research question will be investigated through the execution of a controlled experiment. In this sense, 3 hypotheses (see Table II) were derived from the research question (RQ). For each hypothesis, the *null* ($H_{N\text{-}0}$) and *alternative* hypotheses ($H_{N\text{-}1}$) were defined. The N represents the number of the hypothesis under analysis.

TABLE III. RESEARCH QUESTION AND STUDY HYPOTHESES

Hypothesis		Definition
Precision	$H_{1\text{-}0}$	Precision (BP) ≤ Precision (NBP)
	$H_{1\text{-}1}$	Precision (BP) > Precision (NBP)
Recall	$H_{2\text{-}0}$	Recall (BP) ≤ Recall (NBP)
	$H_{2\text{-}1}$	Recall (BP) > Recall (NBP)
Time	$H_{3\text{-}0}$	Time (BP) = Time (NBP)
	$H_{3\text{-}1}$	Time (BP) ≤ Time (NBP)

Our first study hypothesis $H_{1.0}$ is concerned with the impact of the use of metrics and architecture blueprints on prioritizing software anomalies. The null hypothesis $H_{1.0}$ states that the use of architecture blueprints, as an additional artifact, does not provide any enhancement on the in terms of Precision of the prioritization process. In turn, the alternative hypothesis $H_{1.1}$ states that the Precision was higher when developers are provided with architecture blueprints as additional artifact for prioritizing architecturally relevant code anomalies. Our second study hypothesis $H_{2.0}$ is concerned with the impact of using architecture blueprints on Recall measures. Similarly to the first hypothesis, the null hypothesis $H_{2.0}$ states that the use of architecture blueprints to improve the prioritization of code anomalies does not impact on the Recall. The alternative hypothesis $H_{2.1}$ states that the Recall measures tend to be higher when developers are provided with architecture blueprints for prioritizing architecturally relevant code anomalies. Finally, the third hypothesis $H_{3.0}$ states that there is no different in terms of time spent on detecting code anomalies when subjects are provided with architecture blueprints. Furthermore, we also discuss whether the time could, for instance, influence in the number of False Positives and False Negatives observed in the prioritization and detection process.

B. *Experimental Steps*

A set of experimental procedures is defined in order to guide the subjects when conducting the controlled experiment. Firstly, a training session was performed to introduce the main concepts involved in the controlled experiment. During the training session, we introduced the concept of software anomalies, the code anomalies under investigation and some examples to help subjects on how to reason over the artifacts provided for prioritizing the code anomalies. Secondly, subjects were organized into two different groups: (i) the NBP[1] group that received software metrics and source code information; and (ii) the BP group that, in addition to software metrics and source-code information, also received architecture blueprints (e.g. component diagrams and class diagrams). It is important to mention that during the preparation of the artifacts provided in the controlled experiment, we have specifically

[1] NBP stands for "no blueprint".

evaluated the architecture blueprints – once it is the main artifact that should be analyzed by some subjects. In this way, we performed the mapping between the elements (components and interfaces) in the architecture blueprints and code elements in the system implementation. This mapping helped us to ensure that the architecture blueprints have a minimum quality, in terms of level of abstraction consistency, completeness and well formedness, so that they can be used on the prioritization process. Further details will be provided in Section VI.

All groups also received a document containing a partial view of the Mobile Media system (see Section IV.D), a brief explanation of its architectural design as well as a description of the system concerns. We presented a sequence of tasks that should be performed by each group before answering the experimental tasks. After reading the documents, the subjects were able to start the experimental tasks for prioritizing the code anomalies. Basically, for each subject were assigned 2 out of the three code anomalies. They should list the set of classes they have judged to have each anomaly, as well as explain what was the rationale they have used. That is, they should indicate what artifacts and which metrics they used on the detection process. After that, subjects were asked to provide an ordered list with the anomalous classes considering their architectural relevance. These activities should be performed for each code anomaly the subjects were assigned. The two last tasks as concerned with the use of architectural blueprints. Subjects should first provide details on how the architectural blueprints were used on the prioritization process by explaining. In other words, they should explain the rationale they used to interpret the information provided on the architecture blueprints. Finally, they were asked to indicate which information was useful on the prioritization process. In summary, the subjects should reason about the architecture blueprints, metrics values and source-code information when prioritizing classes that could possibly be candidates to exhibit a given code anomaly.

C. Code Anomalies Reference List and Target Application

We have counted on the collaboration of the system experts to build the code anomalies reference list. They helped us to identify instances of each code anomaly provided on the reference list. A systematic analysis was performed on the target system to identify classes affected by relevant code anomalies. The experts were involved during the development, maintenance or assessment of the target application. Table IV shows the reference list of code anomalies identified in the Mobile Media system. In order to get the code anomalies reference list, we asked the experts to apply their own strategy to detect the code anomalies in the target application. One of the experts focused on code inspection, while another expert used a set of detection strategies [13], as a complimentary approach to code inspection. The results indicated that for each code anomaly a set of potential instances were not exactly the same. That is, around 75% of the code anomalies detected by all experts achieved the same result. Thus, the reference list was a result of a joint decision.

Moreover, the Mobile Media system [24] was selected as the target application for conducting the controlled experiment. Mobile Media is a software product line that provides support for manipulation of photos, music and videos on mobile devices. The main reasons why Mobile Media was selected as our target application are: (i) original developers produced the architecture blueprints; (ii) architecture blueprints were the artifacts used to reason about changes requests and derive new products during the system evolution; (iii) it is a medium-sized open source project; (iv) different types of change were performed in each release, including refinements on the architecture style employed; and (v) the system has been successfully used in other studies involving empirical evaluation [4]. Mobile Media evolution scenarios range from changes in heterogeneous mobile platforms to additions of many alternatives and optional features. For the controlled experiment, we selected the release R7 of Mobile Media once the main changes performed during the system evolution are presented in this release. Moreover, the selected release has a more stable architecture when compared to the previous releases. More details about the system evolution and its characteristics can be found at [5].

TABLE IV. CODE ANOMALIES REFERENCE LIST

Code Anomaly	**Classes**
God Class	MediaAccessor, MediaController
Shotgun Surgery	AlbumController, MainUIMidlet, MediaAccessor, MediaController, MediaListController, SmsMessaging.
Divergent Changes	ImageMediaAccessor, MediaAccessor, MediaController, AlbumController, VideoCaptureController, MainUIMidlet, MediaListController, MusicPlayController, PhotoViewController, PlayVideoController, SelectMediaController

V. DATA ANALYSIS

This section presents the analysis results from the data collected through the controlled experiments. To perform a comparative analysis on the efficiency of using architecture blueprints, we used three measures: Precision, Recall and Time. Precision and Recall leverage other three metrics: True Positives (TP) to measure the number of correctly identified code anomalies; False Positives (FP) to measure the number of wrongly identified code anomalies; and False Negatives (FN) to measure the number of missing code anomalies.

Precision (see Table V, Equation 1) is defined as the ratio of architecturally relevant code anomalies that were correctly identified by the subjects. A high Precision implies that the subject identified more relevant code anomalies than irrelevant ones. On the other hand, Recall (see Table V, Equation 2) can be defined as the fraction of architecturally relevant code anomalies identified by the subjects to the total number of anomalies presented in the code anomalies reference list. For instance, a high Recall implies that a subject have identified most of the architecturally relevant anomalies. In this way, our main focus is on Recall because it is more important not missing many architecturally relevant anomalies. Finally, we evaluate to what extent the use of architecture blueprints impacts on the time spent by the subjects to identify and prioritize each code anomaly. Basically, we measure the time each subject took to perform the experimental tasks when prioritizing each code anomaly (see Section IV.C).

TABLE V. DEFINITINO OF PRECISION AND RECALL

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

It is important to emphasize that the controlled experiment was performed in three different institutions. The first two replications of the controlled experiment were performed in two universities in Brazil, UFBA (8 subjects) and UFMG (42 subjects), with undergraduate and graduate students. The last replication was performed at Drexel University (16 subjects), USA, with only Master and PhD students. In total 66 subjects participated on the controlled experiment. In this way, it was possible to observe how the subjects' technical knowledge and experience might impact on the conclusions when comparing the results achieved by the undergraduate and graduate students. In the following, we present the descriptive statistics and the hypothesis testing for the three measures collected during the controlled experiment. Finally, when testing the study hypotheses we have used the R language and its environment. To verify whether the collected data is normally distributed, Shapiro-Wilk test [21] was applied. As the collected data are not normalized, we applied a Mann-Whitney non-parametric method to test our study hypotheses [21]. This test was chosen because it is designed to perform a non-paired comparison of two independent samples (that do not necessarily have the same size). Moreover, aiming to guarantee a statistical significance of our test, we used as default a confidence level of 95% (p-value = 0.05).

A. Architecture Bluerpints vs Precision

Our first study hypothesis (H_1) investigates whether higher Precision measures can be achieved on prioritizing architecturally relevant code anomalies when subjects are provided with architecture blueprints. Firstly, we performed a comparative analysis of the Precision measures achieved by the BP and NBP groups. We observed that the higher Precision was achieved on detecting the Divergent Change anomaly (see Table VI). For this anomaly, we observed an increase of Precision around 12% in favor of the BP group when comparing the mean values. For the Shotgun Surgery and God Class anomalies, the NBP group achieved better results. More specifically, for the case of the God Class anomaly, we noticed that some subjects were not able to build an interpretation based on the information available from the architecture blueprint. This preliminary observation was based on the feedback provided by some subjects, when asked about the usefulness of architecture blueprint. For those cases, the subjects have only used the metrics provided in the controlled experiment. The problem is that the misinterpretation of metrics values may lead to False Positives, which in turn, directly impacts on Precision measures.

After collecting the descriptive statistics, statistical tests were applied in order to confirm or refute the null hypothesis $H_{1.0}$. Assuming the default level of significance adopted for testing the study hypotheses, and a calculated p-value = 0.09, the null hypothesis could be rejected assuming a marginally statistical significance. Usually, it is desirable that the calculated p-value is lower than the level of significance in order to reach a very significant statistical result. In summary, the statistical results showed that there is weak evidence that the use of blueprints on the prioritization process can improve the Precision measures (for these three anomalies under investigation). In this sense, further empirical evaluation with systems in different domains are still required in order to reach a better statistical significance.

TABLE VI. DESCRIPTIVE STATISTICS FOR PRECISION

Measure	Anomaly	Mean (%)		Median (%)		Diff.
		BP	***NBP***	***BP***	***NBP***	
Precision	DC	82.8	70.8	100.0	70.8	11.9
	SS	48.8	54.72	33.3	50.00	6.0
	GC	58.0	61.9	50.0	58.34	3.8

B. Architecture Blueprints vs Recall

Our second study hypothesis (H_2) aims at investigating whether subjects, when provided with architecture blueprints, could achieve higher Recall. As previously mentioned, Recall indicates the proportion of real positive cases that are correctly predicted as positive by using the software artifacts provided in the controlled experiment. When analyzing the collected data (see Table VII), we observed that, in general, the BP group achieved better results. An increase of around 7% could be observed in detecting the anomalies Divergent Changes and Shotgun Surgery. For the case of the God Class anomaly, the result was even better with an increase of 20% for Recall measure. Even observing a better result for the prioritization of all the three code anomalies, we still need to apply a statistical method to confirm or refute our second null hypothesis $H_{2.0}$.

TABLE VII. DESCRIPTIVE STATISTICS FOR RECALL

Measure	Anomaly	Mean (%)		Median (%)		Diff.
		BP	***NBP***	***BP***	***NBP***	
Recall	DC	37.7	30.6	27.2	27.2	7.0
	SS	32.6	25.7	33.3	33.3	6.9
	GC	85.9	65.3	100.0	100.0	20.6

From the data collected, we could observe that, in average, the Recall was higher when subjects were provided with architecture blueprints. Thus, we can say that the effectiveness of the prioritization process was improved in terms of Recall. It can be explained by the fact that lower rate of False Negatives was observed for the BP group. The lower the number of False Negatives, the higher is the Recall measures. Furthermore, when applying the statistical test, the results showed a calculated p-value = 0.02. Since the significance level adopted for all the hypotheses is 0.05, and the calculated p-value is lower, we can conclude that the second null hypothesis $H_{2.0}$ is rejected with a strong statistical significance. That is, the prioritization of these three code anomalies can be improved in terms of Recall when architecture blueprints are used as complementary artifacts on the prioritization process.

C. Time Spent for Prioritizing Code Anomalies

Our third hypothesis aims to investigate whether the use of blueprints may increases the effort in terms of time spent for prioritizing architecturally relevant code anomalies. Firstly, we analyzed the mean time spent by the subjects (see Table VIII), when prioritizing each code anomaly. For the Divergent Changes anomaly, the BP group spent 4 minutes less, when compared to the NBP group. For the God class anomaly detection, the difference, in favor of BP group, was higher. On the other hand, for the Shotgun Surgery anomaly, we observed a very small difference regarding the time spent for each group. Moreover, the NBP group achieved a better time for prioritizing this anomaly. When evaluating the median values, we observed that the results are close to each other. Subjects of BP group spent less time for prioritizing the anomalies Divergent Change and God Class, which was not the case for the Shotgun Surgery, where the NBP group spent less time.

TABLE VIII. DESCRIPTIVE STATISTICS FOR TIME

Measure	Anomaly	Mean (min)		Median (min)		Diff
		BP	NBP	BP	NBP	
Time	DC	15	19	13	15	4
	SS	10	8	8	6	2
	GC	16	30	14	25	14

In order to test our third hypothesis we have also applied a two-tailed Mann-Whitney U test. We analysed the time spent by the subjects for prioritizing each of the tree code anomalies. For each anomaly, we have a pair of tasks where subjects should identify and prioritize classes containing a specific code anomaly. Subjects should inform the start and end-time for each pair of tasks. After that, we recorded the time spent to compute all the tasks. For the sake of simplicity, we decided to organize the data this way in order to apply the statistical test. After applying the statistical test, we observed a calculated p-value = 0.8, which means that our third hypothesis ($H_{3.0}$) cannot be rejected. In other words, we conclude that the use of architecture blueprints does not bring extra effort regarding the time for prioritizing the code anomalies under investigation.

VI. DISCUSSION

This section discusses the main characteristics of classes identified as False Positives and False Negatives (see Section IV.B), as well as the properties of the architectural blueprint most used in the prioritization process. Furthermore, we also discuss how the subjects' technical knowledge may impact on the results of the controlled experiment. Firstly, in order to better organize the classes identified as False Positives, the same package structure recovered from Mobile Media implementation is used. We decide to use the same structure because each package contains classes responsible for implementing the same functionality and/or have similar characteristics. In this way, we are able to identify the classes responsible for a higher number of False Positives, which directly impact on Precision.

A. Subjects Technical Knowledge

Aiming to provide discussions about the results found through the controlled experiment, we decided to investigate how experienced our subjects are. The analysis of subjects' technical knowledge is important to understand whether they were able to interpret the information presented in the architecture blueprints. Even having architecture blueprints with different level of details and tailored with additional information, subjects should be capable to use all the artifacts to prioritize the architecturally relevant code anomalies. The technical knowledge was assessed considering three main topics required for the execution of the controlled experiment: software architecture, software evolution and code anomalies. For the sake of simplicity, the scores are computed considering their knowledge as being none, moderate and advanced. The questionnaire was applied for all the subjects. When analyzing the knowledge of subjects in the BP group, we observed that 14,4% of subjects have no knowledge on software anomalies before the training session. In addition, around 76,2% and 9,5% of them have, respectively a moderate and advanced knowledge in code anomalies. That is, more than 80% of the total of subjects have a moderate or advanced knowledge on detecting and prioritizing software anomalies. In addition, we also observed that around 85% of subjects have moderate or advanced knowledge on software architecture. On the other hand, when analyzing the results of the NBP group, we observed that 93% of subjects have moderate knowledge on software anomalies. Moreover, all the subjects in this group have experience with software architecture. Once there are groups of subjects with different technical knowledge and working experience, it was possible to balance the groups according to their experience and knowledge.

B. Blueprints Properties and Mapping elements

After the prioritization tasks, subjects were asked to inform how (and whether) architecture blueprints are useful on the prioritization process. It is important to mention that most part of them used the component diagram instead of the class diagram. They argued that although the former have a higher level of abstraction than the latter, it was representing more useful architectural information according to subjects, such as the main interfaces between components as well as the system concerns that each component is responsible for realizing. We decided then analyzing the properties of the *blueprint* used in the controlled experiment. That is, we have specifically analyzed the architecture blueprints in terms of consistency, completeness and level of abstraction. Moreover, the well formedness of our blueprints is guaranteed once the notation used in our study is an extension of the UML component diagram. Therefore, it follows the same well-formedness rules for model elements defined in the UML specification [1].

TABLE IX. BLUEPRINT CONSISTENCY AND COMPLETENESS

Consistency	Value	Completeness	Value
Dependency not mapped	16	Components mapped	100%
Inverted Dependency	2	Provided interfaces mapped	100%
Provided Interface with same name	6	Required interfaces mapped	55%
Required Interface with same name	2	Concerns mapped	100%

Table IX summarizes consistency and completeness measures. For the consistency measures, we identified occurrences of each type of inconsistency. For example, no instances of architectural components without interface and components with the same name were found. In this sense, given the number of inconsistencies observed in the blueprints and the number implemented elements participating in the mapping, the architecture blueprint is around 70% consistent. The consistent elements are the core elements of Mobile Media architecture. However, the 30% of inconsistencies might somehow impact on the effectiveness of the blueprints when guiding subjects on the prioritization of code anomalies. On the other hand, when analyzing the completeness, we observed that the only element in the architectural blueprint that not reached at least 60% of completeness is the set of required interfaces. The fact is that only 22 out of 40 interfaces in the architecture blueprint could be directly mapped to the source code. Other elements (18 architectural components, 23 provided interfaces and 5 concerns) were successfully and directly mapped to several elements in the source code. In this way, the architecture blueprint reached a completeness of around 90% in the mapping.

It is important to mention that both consistency and completeness measures achieved in our study were similar in many other real software projects. Therefore, our results might also be observed in software systems where architecture blueprints have similar consistency and completeness measures. Furthermore, we also analyzed the level of abstraction for the component diagram. When analyzing the mapping, we observed that 34 out of 50 classes implemented were mapped. It is interesting to observe that 8 classes are specifically implementing exception handling and 3 are utilitarian classes. The remaining classes are related to controller, datamodel or screen functionalities. In this sense, we were able to identify the mapping between 18 components represented in the architecture blueprint and 34 code elements (i.e. classes, interfaces) on the system implementation. At the end, the level of abstraction of the architecture blueprint was calculated considering the sum of level of abstraction in all components and the total number of components. The level of abstraction of the component diagram used in the controlled experiment is around 65%. It means, that in average a component is mapped to 2 or 3 classes on the system implementation. The only exception was the architectural component *SmsController* that is implemented by 5 classes.

C. Software Anomalies and False Positives

We now discuss what is the proportion of False Positives found for each code anomaly in the controlled experiment (see Table X). Firstly, when prioritizing the God Class anomaly, the BP group identified False Positives in classes contained in 5 different packages, while the NBP group identified classes in only 4 packages. The God Class anomaly presented the highest number of instances when compared to the other two anomalies under investigation. Moreover, most part of classes of *Controller* and *Datamodel* packages are responsible for more than 80% of instances of False Positives.

TABLE X. CHARACTERISTICS OF FALSE POSITIVES

Group	Package	#Instances	GC	SS	DC
BP	Controller	41	30.61%	54.35%	12.29%
	Datamodel	43	42.86%	34.35%	85.71%
	Screens	10	10.20%	10.87%	--
NBP	Controller	33	40.00%	63.16%	25.00%
	Datamodel	23	40.00%	29.73%	50.00%
	Screens	8	15.00%	8.11%	25.00%

More specifically, the results for the BP group showed that classes in the *Controller* package are responsible for 30.61% of False Positives, while classes in the *Datamodel* package are responsible for 42.86%. Based on the feeback provided by the subjects, we observed that a higher number of False Positives for the God Class anomaly was observed when the class diagram was used.On the other hand, when analyzing the False Positives for the NBP we observed that classes contained in the *Datamodel* and *Controller* packages were responsible for 80% of False Positives. Each of those packages presented 8 instances of False Positives (around 40% of the total), with 5 different classes. Due the fact that those packages have classes representing a high number of False Positives, we organized all the classes in descending order and listed the three classes most frequently identified as False Positives for each group. In the BP group, the 3 classes with more instances of False Positives are: MediaData (14 instances), AbstractController (6 instances) e SelectMediaController (6 instances). For the NBP group, those classes were also identified with a high number of False Positives, but with different instances: MediaData (6 times), AbstractController (3 times) e SelectMediaController (4 times).

When analyzing the Shotgun Surgery anomaly, we identified classes in 3 different packages: *Datamodel*, *Controller* and *Screens* - which are responsible, respectively for 54.35%, 34.78% and 10.87% of instances of False Positives identified for the BP group. The classes responsible for more instances of False Positives are MediaData, AlbumData and AbstractController (each one with 7 instances). On the other hand, classes in the same three packages were responsible for a high number of False Positives: *Datamodel* - 62.16%, *Controller* – 29.73% and *Screens* – 8.11%. In the NBP group, the classes MediaData and AlbumData were also tied regarding the number of instances of False Positives, where each had at least 9 instances. Considering instances of all code anomalies, the Divergent Change was the one with a lower number of False Positives. When analyzing the data collected by the *BP* group, we found that the set of classes responsible for more than 85% of False Positives are contained in the *Datamodel* package. In turn, the NBP group had False Positives in three different packages, but only one of them, *Datamodel* package, contain classes responsible for 50% of False Positives related to the prioritization of the Divergent Change anomaly.

D. Characteristics of False Positives vs. Software Metrics

As discussed above, classes contained in the *Datamodel* and *Controller* packages are responsible for most part of False Positives. Now we discuss what are the main characteristics of those classes in terms of the metrics used on the prioritization process. For the sake of simplicity, we selected the three most recurrent classes identified as False Positives for each of the three architecturally relevant code anomalies. Since those classes implements either *Controller* or *Data* functionalities, they are likely to have similar characteristics. Thus, on the prioritization of God Class and Shotgun Surgery anomalies, the three main classes with more instances of False Positives are, MediaData, AbstractController and SelectMediaController, respectively. On the other hand, the three main classes identified as False Positives for the Divergent Changes anomaly are, respectively, MediaData, AbstractController and AlbumData.

Table XI summarizes the metrics for those 4 classes. We are only showing the metrics that the subjects have used to detect those anomalies and that lead to False Positives, for example, due to the misinterpretation of the values. It is also important to mention that other classes from both packages were also responsible for the occurrence of False Positives, but they had less than 2 instances for most part of the classes. We identified 11 classes implementing the *Controller* functionality and 6 classes implementing *Data* functionality, totaling 16 classes identified as False Positives. As we can observe, classes implementing the *Controller* functionality have high coupling (CBC) and low cohesion (LCOM). Moreover, the number of attributes (NOA) and number of methods of those classes cannot be considered high. On the other hand, when analysing the data classes, we can observe that the MediaData have a high coupling (CBC) and very low cohesion (LCOM). Furthermore, the number of attributes (NOA) and number of methods (NOM) are not considered high.

Based on the answer that the subjects feedback, we observed that usually the classes they selected as candidates for a given anomaly have similar characteristics, but not specifically the same. When subjects have not correctly used the information provided on the architecture blueprint, they have judged only based on metrics value and selected classes with similar characteristics. From the feedback provided by some subjects have not considered, for example, the number of dependencies or the number of concerns implemented by each class as shown in the architecture blueprint. The solely use of metrics is not sufficient to indicate that a class is a possible candidate for a given code anomaly.

TABLE XI. CHARACTERISTICS OF FALSE POSITIVES

Class	LCOM	CBC	NOA	NOM	WMC
AlbumData	6	0	1	14	4
MediaData	0	98	10	17	3
AbstractController	2	54	4	12	1
SelectMediaController	1	42	6	14	3

E. Blueprints and False Negatives

In this section, we discuss the characteristics of False Negatives, that is, classes that should be identified as possible candidate for having code anomalies, but were not. Table XII summarizes the list of False Positives considering all the instances found for each code anomaly. The percentage is based on the total number of False Negatives that each class were responsible considering the three code anomalies under investigation. Surprisingly, in both groups the four most recurrent classes identified as instances of False Negatives are AlbumController, MediaListController, MainUIMidlet and SmsMessaging, respectively.

TABLE XII. INSTANCES OF CODE ANOMALIES BY GROUP

Class	#Instances		% Instances	
	BP	NBP	BP	NBP
AlbumController	23	24	15.7%	18.8%
ImageMediaAccessor	9	8	6.2%	6.3%
MainUIMidlet	19	16	13.0%	12.5%
MediaAccessor	11	10	7.5%	7.8%
MediaController	9	4	6.2%	3.1%
MediaListController	23	23	15,8%	18.0%
MusicPlayController	7	6	4.8%	4.7%
PhotoViewController	6	5	4.1%	3.9%
PlayVideoController	7	6	4.8%	4.7%
SelectMediaController	7	6	4.8%	4.7%
SmsMessaging	17	15	11.6%	11.7%
VideoCaptureController	8	5	5.5%	3.9%

In general, those classes are still very difficult to be detected as candidates of having any of the three code anomalies under investigation – even when subjects are provided with architecture blueprints. Each of the other classes presented in the code anomalies reference list were responsible for 4 – 8% of False Positives. At a first moment, it seems that the use of blueprints has not improved prioritization of code when we compare the number of False Negatives, which has direct impact on Recall. Although that those four classes had similar instances of False Negatives, we have to consider that the blueprint group had 27 replications of the controlled experiment, while the NBP-group had only 16. Thus, if we look to the total number of instances of False Negatives and split by the total number of replications, we have a density of False Negatives equals to 5 and 8, respectively, for the BP and NBP groups. That is, considering 12 classes in the code anomalies reference list, around 42% and 67% of them were not identified, respectively, in the BP and NBP group.

VII. RELATED WORK

Many works investigated the detection and prioritization of code anomalies, supported by the use software metrics. For instance, Marinescu [18] proposed the use of traditional metrics (e.g. WMC, TCC, AFTD) for detecting code smells. One of his findings was that multiple metrics are required to capture factors in the anomaly definition. In addition, he reported an accuracy of 60% for the anomalies investigated. In our work, we provided subjects with a set of traditional and concern-based metrics and architecture blueprints to help them on the anomaly detection process. Furthermore, some works have investigated the Shotgun Surgery and God Class anomalies and their impact on software maintenance. For example, Shotgun Surgery was positively associated with software faults [12], as well as there are reports [10] that this anomaly was consistently correlated with defects across systems. Different studies have also investigated the effect of code anomalies from the perspective of system defects. Other work [23] investigated the relationship between the class error probability and code anomalies, based on three versions of the Eclipse project. Their result showed that classes with code anomalies (e.g. *Shotgun Surgery*, *God Class* or *God Methods*) are more likely to present errors than non-infected classes.

D'Ambros [22] has investigated the influence of code anomalies on software defects in six open-source systems. Their investigation suggested that an increase in the number of code anomalies is likely to generate software defects. However, there is no a single code anomaly that was consistently correlated to errors more than others across the totally of the systems. Deligiannis et al. [3] showed that a design (not code) without a *God Class* was judged and measured to be better (in terms of time and quality) than a design for the same system with a *God Class*. The results provide evidence that a software design without God Class is better with respect to completeness, correctness and consistency. From the empirical studies identified, only the study by Abbes *et al.* (2011) brings up the notion of interaction effects across code anomalies. They concluded that classes and methods identified as *God Classes* and *God Methods* in isolation had no effect on effort, but when appearing together, they led to a significant increase in the maintenance effort.

VIII. Final Remarks and Future Works

Our findings have shown that to some extent, the architecture blueprints has improved both Precision and Recall. For example, we observed, that the Precision was higher for Divergent Change and Shotgun Surgery anomalies. However, for God Class anomaly, the results were not expressive and the architecture blueprints have not substantially improved the prioritization process. For the cases where the use of architecture blueprints improved the Precision, the results showed that the improvement was around 20 %. The test of our first hypothesis $H_{1.0}$ showed that it could not be rejected because the calculated p-value was higher than the acceptable significance level. Therefore, even though we observed that the architecture blueprints could somehow improve Precision measures, the tests do not indicate a high statistical significance. On the other hand, Recall has increased in the BP for all the three anomalies. That is, As the Recall measures indicate the sensitivity of subjects on detecting real positive cases that are correctly predicted as positive, we observed a lower number of False Negatives – the lower number of False Negatives, the higher is the higher Recall. Independently whether the subjects have used correctly the architecture blueprints, the results showed that the use of this artifact has somehow improved occurrence of False Positives and False Negatives. The test of our second hypothesis $H_{2.0}$ revealed that the Recall was affected by the used of architecture blueprints on the detection process. Since the calculated p-value was lower than the significance used on the hypothesis test, the hypothesis $H_{2.0}$ can be rejected.

Finally, we evaluated the time spent by the subjects when prioritizing the three code anomalies under investigation. The results showed that there is not a difference higher than 10 minutes for prioritizing the architecturally relevant anomaly, when comparing the results between the BP and NBP groups. The results of testing our third hypothesis $H_{3.0}$ indicated p-value = 0.08, which means that it cannot be accepted. Therefore, the use of architecture blueprints did not bring any additional effort in terms of time spent. As a future work, we intend to invite experienced developers to participate in our experiment. The controlled experiment with experienced developers may provide more interesting results regarding the usefulness of architecture blueprints on the detection process. Furthermore, we also want to investigate more architecturally relevant anomalies that have not been considered in this paper.

References

[1] J. A. Alegría, A. Lagos, A. Bergel and M. C. Bastarrica. Process Model Blueprints. In Proc. of 4th ICSP, Springer Verlag, pp. 273-284, 2010.

[2] V. A. Araya. Test Blueprint: An Effective Visual Support to Test Coverage. In Proc. of 33rd ICSE, pp. 1140-1145, 2011.

[3] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis and M. Shepperd. A Controlled Experiment Investigation of an Object-Oriented Design Heuristics for Maintainability. Journal of Systems and Software, Vol. 72, Issue 2, pp. 129-143, 2004.

[4] E. Figueiredo, C. Sant'Anna, A. Garcia, T. Bartolomei, W. Cazzola and A. Marchetto. On the Maintainability of Aspect-Oriented Software: a Concern-Oriented Measurement Framework. In Proc. of 12th CSMR, pp. 183-192, 2008.

[5] E. Figueiredo, et al. Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability. In Proc. of 30th Int'l Conf. Soft. Engineering, 261-270, 2008

[6] E. Figueiredo, A. Garcia, M. Maia, C. Nunes, and J. Whittle. On the Impact of Crosscutting Concern Projection on Code Measurement. In Proc. of 10th AOSD, pp. 81-92, 2011.

[7] M. Fowler and K. Beck. Refactoring: Improving the Design of Existing Code, Addison-Wesley, USA 1999.

[8] J. Garcia, D. Popescu, G. Edwards and N. Medvidovic, Identifying Architectural Bad Smells. In Proc. of 13th CSMR, March 2009.

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

[10] P. B. Kutchen. Architectural Blueprint: The "4+1" View Model of Architecture. IEEE Software, Vol. 12, Issue 6, pp. 42-50, 1995.

[11] 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.

[12] W. Li and R. Shatnawi. An Empirical Study of Bad Smells and Class Error Probability in the Post-release Object-Oriented System Evolution. Journal of Systems and Software, Vol. 80, pp. 1120-1128, July 2007.

[13] I. Macia, A. Garcia and A. von Staa. An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In Proc. of 10th AOSD, pp. 203-214, Recife, March 2011.

[14] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic and A. von Staa. Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity? – An Exploratory Analysis of Evolving Systems. In Proc. of 11th AOSD, pp. 167-178, USA, March 2012.

[15] 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 CSMR, Szeged, Hungary, March 2012.

[16] Arcoverde, R.; Macia, I.; Garcia, A.; Staa, A. Automatically Detecting Architecturally-Relevant Code Anomalies. In Proc. of the 3rd RSSE @ ICSE'12, Zurich, Switzerland, June 2012.

[17] M. Mantyla, J. Vanhanen and C. Lassenius. A Taxonomy and an Initial Empirical Study of bad Smells in Code. In Proc. of 19th ICSM, Amsterdam, Netherland, September 2003.

[18] R. Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In Proc. of 20th IEEE ICSM, pp. 350-359, 2004.

[19] N. Moha, Y. Guéhéneuc, L. Duchien and A. F. Meur. DECOR: a Method for the Specification and Detection of Code and Design Smells. In Proc. of IEEE TSE, Vol. 36, Issue 1, pp. 20-36, February 2010.

[20] A. Riel. Object-Oriented Design Heuristics. Addison-Wesley Longman Publishing Co. Boston, MA, April 1996.

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

[22] M. D'Ambros, A. Bacchelli and M. Lanza. On the Impact of Design Flaws on Software Defects. In Proc. of 10th ICSQ, pp.23-31, 2010.

[23] W. Li and R. Shatnawi. An Empirical Study of Bad Smells and Class Error Probability in the Post-release Object-Oriented System Evolution. Journal of Systems and Software, Vol. 80, pp. 1120-1128, July 2007.

[24] E. Figueiredo *et al*. Evolving Software Product Lines with Aspects: An empirical Study on Design Stability. In Proc. of 30th ICSE, pp. 261-270, Germany, 2008.

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

[26] Lange, L. and Chaudron, M. An Empirical Assessment of Completeness in UML Designs. In Proc. of 8th Int'l Conf. on Empirical Assessment in Software Engineering, United Kingdom, 2010.