4 Prioritizing and Ranking Code Anomalies with Blueprints

As mentioned in Chapter 1, our research work is focused on evaluating how the architecture information provided by blueprints can be efficiently used as means to improve the prioritization and ranking of critical code anomalies. In this context, we conducted controlled experiments aiming to investigate how architecture blueprints provided by software architects or developers could help to prioritize and rank critical code anomalies. These experiments are reported in this Chapter. The controlled experiments also allow us to evaluate how architecture blueprints could be useful on the prioritization and ranking process. Thus, we have applied controlled experiments in three different universities with the participation of 66 graduate students. The participants also provided qualitative feedback. For instance, they indicated examples of additional architecture information (e.g. dependency strength and mapping of architecturally-relevant concerns), which could be explored in the architecture blueprint to facilitate the prioritization and ranking of critical code anomalies. The additional architecture information would complement the source code information, already explored by the existing code anomaly detection strategies.

From our initial results, we inferred several types of information were useful to assist developers on the prioritization process. Thus, participants used different types of information to help them when deciding which code anomalies should be prioritized first. Our initial research results revealed architecture blueprints could be efficient in the prioritization process, in particular if the architecture information available on them is used in addition to the source code. The latter is already explored by existing strategies, while the former is not (Chapter 2). Moreover, we observed the use of architecture blueprints allow statistical significant improvements in terms of **Recall** and **Precision** related with the prioritization of critical code anomalies. The use of architecture blueprints did not bring any additional effort in terms of **Time Spent** for the prioritization process.

Finally, our investigation revealed other interesting research findings regarding the prioritization of code anomalies supported by architecture blueprints: (i) we discuss how architecture blueprints might influence in the prioritization of critical code anomalies; (ii) quantitative indicators on how the use of architecture blueprints could help to improve the prioritization process; (iii) an analysis in terms of time spent when using blueprints as additional artifacts for identifying critical instances of code anomalies; and (iv) what are the main characteristics of **False Positives** and **False Negatives** observed in the target applications under investigation. It is also important to mention that the results of this chapter were published at: (i) 5th International Workshop on Modeling in Software Engineering held in conjunction with at ICSE; and (ii) 38th Annual International Computer, Software & Applications Conference (COMPSAC).

4.1. Code Anomalies Studied

As mentioned in Chapter 1, the progressive insertion of code anomalies might contribute to architecture degradation symptoms, and therefore, might hinder the software maintenance and evolution tasks. In order to prevent architecture degradation symptoms, software development teams should progressively prioritize and rank the most critical code anomalies in order to remove them as early as possible in the development process. In this sense, our controlled experiments investigate how the presence of critical code anomalies might negatively impact on the overall software architecture during the system evolution (Fowler *et al.*, 1999)(Macia *et al.*, 2012a)(Macia *et al.*, 2012b). Thus, our study focused on three types of code anomalies, which have been also evaluated in other studies (Deligiannis *et al.*, 2004)(Lanza and Marinescu, 2006)(Li and Shatnawi, 2006). In the following, we briefly describe each of the code anomalies investigated in the controlled experiments.

- *God Class* is defined as a class that knows too much or does too much. That is, it represents a class that has grown beyond all logic to become the class that does almost everything in the system (Riel, 1996). In another perspective, *God Class* can be understood as a class that implements too many concerns and, so, have too many responsibilities. Therefore, it violates the idea that a class should capture only one key abstraction (Martin, 2003).

- *Shotgun Surgery* anomaly is somehow the opposite of the *Divergent Change*. We identify a *Shotgun Surgery* instance every time we make a kind of change that leads to a lot of little changes in many different classes (Fowler *et al.*, 1999).
- *Divergent Change* anomaly occurs when one class is commonly changed in different ways for different reasons (Fowler *et al.*, 1999). For example, we have to change three methods of a class every time we get a new database or we have to change other four methods every time there is a new financial instrument. Any change to handle a variation should change a single class, and all the new subtypes of this class should express the variation (Fowler *et al.*, 99).

Furthermore, several studies have investigated the impact of those three code anomalies in different software activities (e.g. software maintenance). For instance, *Shotgun Surgery* was positively associated with software faults (Lanza and Marinescu, 2006). Other work also reported (Tsantalis *e al.*, 2008) this anomaly has been consistently correlated with defects across systems. Different studies investigated the effect of code anomalies from the perspective of system defects. Other work (Li and Shatnawi, 2007) 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. In turn, the work developed by Deligiannis et al. (2004) 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 provided evidence a software design without *God Class* is better with respect to completeness, correctness and consistency. Finally, an empirical study performed by Abbes *et al.* (2011) brings up the notion of interaction effects across code anomalies. The study concluded 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.

4.2. Experimental Evaluation

Aiming to address our second research question (**RQ$_2$**), we derived two auxiliary research questions (ARQs). The main motivation of this investigation is due to the belief that early prioritization of code anomalies may prevent architecture degradation during the system evolution. Recalling the prioritization of critical code anomalies can be understood as the process of distinguishing detected code anomalies harmful to the architecture design. In this sense, the auxiliary research questions are defined as:

- **ARQ[3]** - *How can architecture blueprints help the prioritization of relevant code anomalies?*
- **ARQ[4]** - *Can the use of architecture blueprint, as an additional artifact, to improve the prioritization of architecturally relevant code anomalies?*

To evaluate to what extent the use of architecture blueprints may improve the prioritization of critical 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**. This analysis allows us understanding why the existing strategies (based on the sole use of source-code metrics) fail to correctly identify instances of critical code anomalies. In this sense, we defined 3 study hypotheses (see **Table 6**) to be tested based on the data collected with the participants in the experiment. For each hypothesis, the *null* ($H_{N.0}$) and *alternative* hypotheses ($H_{N.1}$) were defined (N represents the number of the hypothesis under analysis). It is important to remember that the controlled experiments have been executed in three different universities. Therefore, we counted on the collaboration of 66 participants with different working experience. We could assess their working experience through a questionnaire applied before the training session (see Section 4.2.1).

Table 6 – Study hypotheses definition

Study Hypothesis	**Description**
Hypothesis $H_{1.0}$	Precision (blueprints) ≥ Precision (non-blueprint)
Hypothesis $H_{2.0}$	Recall (blueprints) ≥ Recall (non-blueprint)
Hypothesis $H_{3.0}$	TimeSpent (blueprints) ≠ TimeSpent (non-blueprint)

Our first study hypothesis $H_{1.0}$ is concerned with the impact of the use of metrics and architecture blueprints on prioritizing and ranking critical code 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. It is important to remember that, similarly to the study presented in the Chapter 3, the blueprints represent the descriptive architecture of the target application under assessment.

4.2.1. Experimental Steps

In order to perform the controlled experiments, we have defined a set of experimental steps. As a first step, we performed a training session to introduce the main concepts involved in the controlled experiments. For example, we introduced the concept of code anomaly, software architecture and detection strategies. We have also provided examples on how to reason over the artifacts provided for prioritizing and ranking the code anomalies.

The second step consists in organizing participants into two different groups, based on the set of artifacts they were provided. The first group received only software metrics and source code information for prioritizing critical anomalies. That is, participants in this group have not received any architecture blueprints or additional architecture information. In turn, participants in the second group have received blueprints representing information of the descriptive architecture (e.g. components, interfaces, deign decisions), in addition to software metrics and source-code information. For the sake of simplicity, we call both groups as *non-blueprint*

(**NBP**) and *blueprint* (**BP**), respectively. In addition, the preparation of the artifacts provided in the controlled experiments, we specifically evaluated the architecture blueprints – once this is the main artifact analyzed by the participants in the **BP** group. Thus, we performed the mapping between the elements (components and interfaces) in the architecture blueprints and code elements in the system implementation. The mapping process (see Chapter 2) helped us to guarantee architecture blueprints all architecture design models could be classified in the concept of blueprints defined in this thesis. Moreover, the mapping process helped us to evaluate whether architecture blueprints have a minimum quality, in terms of level of abstraction, consistency and completeness, so that they can be properly used in the prioritization process.

After the training section has been performed and participants organized into groups, we provided to all groups a document containing: (i) a partial view of the Mobile Media architecture, including a description of the architectural design, as well as the description of the system concerns. It is important to mention only the Mobile Media has been selected as target application, since we performed a preliminary investigation regarding the impact of architecture blueprints in the prioritization process. In this phase, we presented a sequence of tasks should be performed by each group before answering the experimental tasks. After reading the documents, the participants were able to start the experimental tasks for prioritizing the code anomalies. Basically, for each subject were assigned 2 out of the 3 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. In addition, they should indicate what artifacts and which metrics they used on the prioritization process.

After that, participants 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 architecture blueprints. Participants should explain the rationale used to interpret the information provided on the architecture blueprints. Finally, they were asked to indicate which information was useful on the prioritization and ranking process. Participants should reason about the architecture blueprints, metrics values and source-code information when performing the experimental tasks.

4.2.2. Ground Truth of Code Anomalies

We have counted on the collaboration of the system experts to build the code anomalies reference list – also referred to as *ground truth*. The experts helped us to identify instances of each code anomaly provided in the reference list. A systematic analysis was performed on the target system to identify classes affected by critical code anomalies. In addition, the experts were involved during the development, maintenance or assessment of the target application. **Table 7** shows the reference list of code anomalies identified in the Mobile Media system.

Table 7 – Code anomalies reference list for Mobile Media

Code Anomaly	Code Element
God Class	MediaAccessor, MediaController
Shotgun Surgery	AlbumController, MainUIMidlet, MediaAccessor, MediaController, MediaListController, SmsMessaging
Divergent Change	ImageMediaAccessor, MediaAccessor, MediaController, AlbumController, VideoCaptureController, MainUIMidlet, MediaListController, MusicPlayController, PhotoViewController, PlayVideoController, SelectMediaController

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. For instance, one of the experts focused on code inspection, while another expert used a set of detection strategies, 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. Thus, 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.

4.3. Hypotheses Testing and Data Analysis

This section presents the data analysis from the results observed through the controlled experiments. To perform a comparative analysis on the efficiency of using architecture blueprints representing the descriptive architecture of the target system, we used three measures: **Precision**, **Recall** and **Time Spent**. **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 8**, Equation 1) is defined as the ratio of critical code anomalies correctly identified by the subjects. A high **Precision** implies the participants identified more relevant code anomalies than irrelevant ones. On the other hand, **Recall** (see **Table 8**, Equation 2) can be defined as the fraction of critical code anomalies identified by the participants to the total number of anomalies presented in the code anomalies reference list. Therefore, a high **Recall** implies the participants identified most of the critical code anomalies. In this way, our main focus is on **Recall** because it is more important not missing many critical code anomalies. Finally, we evaluate to what extent the use of architecture blueprints impacts on the **Time Spent** by the participants when prioritize and ranking each code anomaly under investigations. Basically, we measure the time each subject took to perform each experimental task.

Table 8 - Definition of Precision and Recall

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

As previously mentioned, the controlled experiments have been performed in three different institutions. The first two replications of the controlled experiments 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 we had 66 participants in the controlled experiments. In this way, it was possible to observe how the participants' technical knowledge and working experience might impact on the conclusions when comparing the results achieved by the undergraduate and graduate students. It is important to mention the hypotheses test was performed using the R language and its environment. To verify whether the collected data is normally distributed, *Shapiro-Wilk test* (Wohlin *et al.*, 2000) was applied. As the collected data are not normalized, we applied a *Mann-Whitney* non-parametric method to test our study hypotheses (Wohlin *et al.*, 2000). This test was chosen because it is designed to perform a non-paired comparison of two independent samples, which do not necessarily have the same size. Moreover, aiming to guarantee a statistical significance of our tests, we used as default a confidence level of 95% (*p-value* = 0.05) for testing all the study hypotheses.

4.3.1. Impact of Architecture Blueprints On Precision and Recall

Our first study hypothesis (H_1) investigates whether higher **Precision** measures can be achieved on prioritizing and ranking critical code anomalies when participants are provided with architecture blueprints. Firstly, we performed a comparative analysis of the **Precision** measures achieved by the *blueprint* (BP) and *non-blueprint* (NBP) groups. We observed the higher **Precision** was achieved on the prioritization and ranking of the *Divergent Change* anomaly (see **Table 9**). For this code 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 some participants 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 participants, when asked about the usefulness of architecture blueprint. For those cases, the participants have only used the metrics provided in the controlled experiments. The problem is that the misinterpretation of metrics values may lead to **False Positives**, which in turn, directly impacts on **Precision** measures.

Table 9 – Descriptive statistic for Precision and Recall

Measure	**Code Anomaly**	**Mean (%)**		**Median (%)**		**Diff.**
		BP	***NBP***	***BP***	***NBP***	
Precision	Divergent Change	82.8	70.8	100.0	70.8	11.9
	Shotgun Surgery	48.8	54.72	33.3	50.00	6.0
	God Class	58.0	61.9	50.0	58.34	3.8
Recall	Divergent Change	37.7	30.6	27.2	27.2	7.0
	Shotgun Surgery	32.6	25.7	33.3	33.3	6.9
	God Class	85.9	65.3	100.0	100.0	20.6

After collecting measures for performed 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 architecture blueprints on the prioritization process can improve the **Precision** measures (for all code anomalies under investigation). Further discussions are provided in Section 4.4.

Our second study hypothesis (H_2) aims at investigating whether participants, 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 experiments. When analyzing the collected data (see **Table 9**), we observed that, in general, the BP group achieved better results. An increase of around 7% could be observed in prioritizing and ranking the code 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 $H_{2.0}$.

From the data collected, we could observe, in average, the **Recall** was higher when participants were provided with architecture blueprints. Thus, we can say that the effectiveness on the prioritization and ranking of critical code anomalies 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. Therefore, we conclude that the second null hypothesis $H_{2.0}$ can be rejected with a strong statistical significance. That is, the prioritization and ranking of these three code anomalies can be improved in terms of Recall when architecture blueprints are used as complementary artifacts on the prioritization process.

4.3.2. Analyzing the Time Spent on the Prioritization Process

Our third hypothesis (H_3) aims to investigate whether the use of architecture blueprints may increases the effort in terms of **Time Spent** for prioritizing and ranking critical code anomalies. Firstly, we analyzed the mean t**ime** the participants dedicated when prioritizing and ranking each code anomaly (see **Table 10**). 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. Participants in the BP group spent less time for prioritizing the anomalies *Divergent Change* and *God Class*, which was not the case for the *Shotgun Surgery* anomaly, where the NBP group spent less time.

Table 10 – Descriptive statistic for Time Spent on the prioritization process

Measure	**Code Anomaly**	**Mean (min)**		**Median (min)**		**Diff**
		BP	**NBP**	**BP**	**NBP**	
Time Spent	Divergent Change	15	19	13	15	4
	Shotgun Surgery	10	8	8	6	2
	God Class	16	30	14	25	14

To test our third hypothesis (H_3), we have also applied a two-tailed Mann-Whitney U test when analyzing the time spent by the participants on the prioritization and ranking of critical code anomalies. For each anomaly, we assigned a pair of tasks to the participants where they should identify and prioritize classes containing a specific code anomaly. Participants should inform the start and end-time for each pair of tasks, and therefore, 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. Thus, we conclude the use of architecture blueprints does not bring extra effort regarding the time for prioritizing and ranking the code anomalies under investigation.

4.4. Further Discussions

After the study hypotheses have been tested, we performed a more in-depth analysis to better understand what are the characteristics of classes identified as **False Positives** and **False Negatives**. Aiming to better organize classes identified either as **False Positives** or **False Negatives**, we used the same package structure recovered from the Mobile Media implementation. 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**. Furthermore, we investigate what information available in architecture blueprint have been mostly used in the prioritization and ranking process, and provide a discussion on how the participants' technical knowledge might impact the results observed in the controlled experiments.

4.4.1. Usefulness of Architecture Blueprints

Besides the tasks of prioritizing critical code anomalies, we asked the participants to indicate whether they judge the architecture blueprint as being useful in the prioritization process (see **Figure 5**). In this sense, around 71.4% of participants judged that the architecture blueprints provided in the experiment as being useful in the prioritization and ranking process. Only 28.6% claimed that the architecture blueprints have not been useful, so they used only the set of metrics provided in the experiment when prioritizing and ranking critical code anomalies.

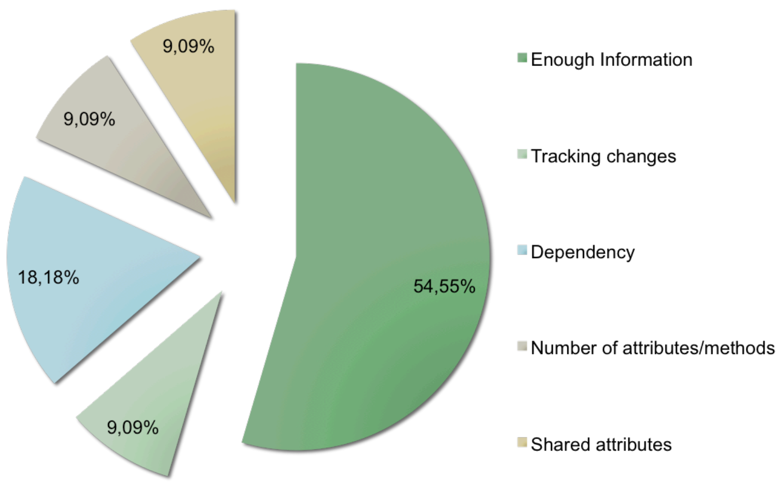

Figure 5 – Information Suggested by the Participants

For the participants that indicated the architecture blueprints have been useful in the prioritization and ranking process, we also asked the participants to point out what kind of information could be added to the architecture blueprint. Most part of the participants said the architecture blueprints are complete and no additional information is required. Thus, for those participants the information presented on architecture blueprints was sufficient to assist them in the prioritization and ranking process. Based on this information, the participants could pre-select classes that might have a specific anomaly and their suspicion would be confirmed or rejected through the evaluation of software metrics. Another suggestion was that the information about classes having a high number of dependencies could be useful if included in the architecture blueprint. In this case, participants were able to prioritize and rank classes infected with the code anomalies *Divergent Changes* and *Shotgun Surgery*. Once the classes are pre-selected, the metrics can be analyzed to confirm whether a given class has an anomaly or not. In summary, architecture blueprints were considered to be useful on the prioritization and ranking process, since participants could initially identify classes, which are potential candidate to be infected with code anomalies.

4.4.2.
Participants' Technical Knowledge

Aiming to provide discussions about the results found through the controlled experiments, we decided to investigate how experienced the participants are. The analysis of participants' technical knowledge is important to understand whether they were able to interpret the information presented in the architecture blueprints. Even having blueprints representing information regarding the descriptive architecture, participants should be capable reason about all the artifacts to prioritize and rank the critical code anomalies. Thus, the technical knowledge was assessed considering three main topics required for the execution of the controlled experiments: 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 participants. When analyzing the knowledge of participants in the BP group, we observed that 14,4% have no knowledge related with code anomalies prioritization and ranking before the training session. In addition, around 76,2% and 9,5% of them has a moderate and advanced knowledge in code anomalies, respectively. Therefore, more than 80% of the total of participants have a moderate or advanced knowledge on the prioritization and ranking of critical code anomalies. In addition, we observed around 85% of participants in the **BP** group have moderate or advanced knowledge on software architecture. On the other hand, when analyzing the results of the **NBP** group, we observed 93% of participants have moderate knowledge on code anomalies. Moreover, all the participants in this group have experience with software architecture. Once there are groups of participants with different technical knowledge and working experience, it was possible to balance the groups defined in the controlled experiments.

4.4.3. Mapping Architecture Blueprints to Source-Code

After the prioritization tasks, participants were asked to inform how (and whether) architecture blueprints are useful on the prioritization and ranking process. All the blueprints represented information about the descriptive architecture, such as the main interfaces/connectors defining the communication between components as well as the concerns each architectural component is responsible for realizing. In this sense, we decided to analyze properties of architecture *blueprint* provided as artifact to perform the experimental tasks. We have specifically analyzed the architecture blueprints in terms of consistency, completeness and level of abstraction.

Table 11 summarizes the analysis of consistency and completeness when mapping the descriptive architecture (represented by the blueprints) and source code elements in the Mobile Media system. For the consistency measures, we identified occurrences of different types of consistency problems. For example, no instances of architectural components without interface and architectural components with the same name were found. Given the number of inconsistencies observed in the architecture blueprints and the number of source code elements participating in the mapping, the architecture blueprint achieved a consistency of 70%. The consistent elements are the core elements of Mobile Media architecture. However, the 30% of inconsistencies might somehow impact on the effectiveness of the architecture blueprints as artifact for guiding participants on the prioritization and ranking of critical code anomalies. On the other hand, the required interfaces are the only architectural elements have not reached at least 60% of completeness. 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 system 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% considered all the elements participating in the mapping process.

Table 11 - Analyzing consistency and Completeness of architecture blueprint

Consistency	Value	Completeness	Value
Dependency Not Mapped	16	Architectural Component Mapped	100%
Inverted Dependency	2	Required Interface Mapped	100%
Provided Interface (Same Name)	6	Provided Interface Mapped	55%
Required Interface (Same Name	2	Concerns Mapped	100%

It is important to mention both consistency and completeness 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 analyzed the level of abstraction for the component diagram. We observed 34 out of 50 classes implemented were mapped. It is interesting to observe that 8 classes are specifically implementing the *Exception Handling* concern and other 3 are utilitarian classes. The remaining classes are related with controller, datamodel or screen functionalities. Thus, we could identify the mapping between 18 architectural components and 34 code elements (i.e. classes, interfaces) in the system actual 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. Therefore, the level of abstraction for the architecture blueprint used in the controlled experiments is around 65%. This measures of level of abstraction means 2 or 3 classes on the system implementation. The only exception was the architectural component *SmsController*, which is implemented by 5 classes.

4.4.4. Critical Code Anomalies and False Positives

We now discuss the proportion of **False Positives** observed for each code anomaly when analyzing the results achieved in the controlled experiments (see **Table 12**). Firstly, when prioritizing and ranking instances of the *God Class* anomaly, the **BP** group identified **False Positives** in classes implemented in 5 different packages, while the **NBP** group identified classes implemented in 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**. The results for the **BP** group showed classes in the *Controller* package are responsible for 30.61% of **False Positives**, while classes in the *Datamodel* package are responsible for 42.86%. Based on the feedback provided by the participants, the *God Class* anomaly was identified with the highest number of **False Positives**.

On the other hand, analyzing the **False Positives** for the **NBP** when prioritizing and ranking critical code anomalies, we observed classes implemented 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), considering 5 different classes. Due the fact 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) and 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 instances), AbstractController (3 instances) and SelectMediaController (4 instances).

When analyzing the *Shotgun Surgery* anomaly, we identified classes implemented 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** in the **BP** group. Classes responsible for more instances of **False Positives** are MediaData, AlbumData and AbstractController (each one with 7 instances). On the other hand, analyzing the data in the **NBP** group we identified classes in the same packages were responsible for a high number of **False Positives**: *Datamodel* (62.16%), *Controller* (29.73%) and *Screens* (8.11%). In addition, the classes MediaData and AlbumData presented the same number of **False Positives** in the **BP** group, where each class had (at least) 9 instances. Considering instances of all code anomalies, the *Divergent Change* was the one identified 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** when prioritizing and ranking the *Divergent Change* anomaly.

Table 12 - Characteristics of False Positives

Group	Package	# of Instances	God Class	Shotgun Surgery	Divergent Change
Blueprint	Controller	41	30.61%	54.35%	12.29%
	Datamodel	43	42.86%	34.35%	85.71%
	Screens	10	10.20%	10.87%	--
Non-blueprint	Controller	33	40.00%	63.16%	25.00%
	Datamodel	23	40.00%	29.73%	50.00%
	Screens	8	15.00%	8.11%	25.00%

As aforementioned, classes implemented in the *Datamodel* and *Controller* packages are responsible for most part of **False Positives**. Furthermore, we briefly discuss the main characteristics of those classes in terms of metrics used in the prioritization and ranking process. For the sake of simplicity, we selected three most recurrent classes identified as **False Positives** for each code anomaly under investigation. Since those classes implement *Controller* or *Datamodel* functionalities, they are likely to have similar characteristics. Thus, when prioritizing and ranking the *God Class* and *Shotgun Surgery* anomalies, the classes recurrently identified as **False Positives** are MediaData, AbstractController and SelectMediaController, respectively. On the other hand, the three main classes identified as **False Positives** when prioritizing and ranking the *Divergent Changes* anomaly are, respectively, MediaData, AbstractController and AlbumData.

Table 13 summarizes the metrics for those 4 classes. We only show the metrics the participants used when prioritizing the code anomalies, and for some reasons lead to the identification of **False Positives** (e.g. misinterpretation of metric values). In addition, other classes implemented in the *Datamodel* or *Controller* packages were also responsible for the **False Positives**, but each class presented less than 2 instances. In total we identified 16 classes as **False Positives**, of which 11 and 6 classes implement, respectively, *Controller* and *Datamodel* functionality. As we can observe, classes implementing the *Controller* functionality have high *coupling* (**CBC**) and low *cohesion* (**LCOM**). The *number of attributes* (**NOA**) and *number of methods* (**NOM**) of those classes cannot be considered high. However, when analyzing the data classes, we can observe the MediaData have high *coupling* (**CBC**) and very low *cohesion* (**LCOM**).

Table 13 - Software Metrics and 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

Furthermore, the *number of attributes* (**NOA**) and *number of methods* (**NOM**) are not considered high. Based on the feedback provided by the participants, we observed usually the classes they selected as candidates for a given anomaly have similar characteristics, but they are not specifically the same. When participants have not correctly used the information provided on the architecture blueprint, the prioritization and ranking process was based on metrics. However, some participants have not considered, for example, the number of dependencies or the number of concerns implemented according to the architecture specification represented in the blueprints. Therefore, the solely use of metrics is not sufficient to indicate a class is a potential candidate to present code anomalies harmful to the architecture design.

4.4.5.
Architecture Blueprints and False Negatives

In this section, we discuss the characteristics of **False Negatives**, which represent classes should be identified as potential candidates to be infected with critical code anomalies, but were not. **Table 14** summarizes the list of **False Positives** considering all the instances found for each code anomaly. The percentage represents the total number of **False Negatives** each class were responsible considering all code anomalies under investigation. Surprisingly, in both groups the four most recurrent instances of **False Negatives** are the classes AlbumController, MediaListController, MainUIMidlet and SmsMessaging, respectively. In general, it is very difficult to prioritize and rank critical instances of the three code anomalies in those classes – even when participants are provided with architecture blueprints. Other class in the code anomalies reference list was responsible for 4 – 8% of **False Positives**.

Table 14 - False Negatives by Participants' Group

Class Name	#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%

At a first moment, it seems the use of architecture blueprints has not improved the process of prioritizing and raking of critical code anomalies through the analysis of **False Negatives -** which has direct impact on **Recall**. Although those four classes had similar instances of **False Negatives**, we need to take into consideration the **BP** group had 27 replications of the controlled experiments, while the **NBP** group had only 16 replications. Thus, considering the ratio between the total number of **False Negatives** and the number of replications, we have a density of **False Negatives** equals to 5 and 8, respectively, for the **BP** and **NBP** groups. In this sense, 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.

4.5. Summary

In this study, we performed controlled experiments to investigate how the process of prioritizing and ranking critical anomalies could be improved, when guided by architecture blueprints. Our initial findings have shown that the architecture blueprints has improved both **Precision** and **Recall** measures when prioritizing and ranking critical code anomalies. For instance, we observed the **Precision** measures were higher for when prioritizing and ranking instances of the code anomalies

Divergent Change and *Shotgun Surgery*. However, in the case of the *God Class* code anomaly the results were not expressive. The architecture blueprints have not effectively improved the prioritization and ranking of all instances of this code anomaly. In general, for the cases where the use of architecture blueprints improved **Precision** measures, the results were not higher than 20%. Moreover, according to the feedback provided by the participants in the controlled experiments, we observed that some of them have not followed correctly the inspection sequence defined in the controlled experiments because they considered the anomaly *God Class* as being more intuitive to detect. This may have led to the wrong prioritization and ranking of instances of this anomaly, and consequently, the number of **False Positives** becomes high, thereby, affecting the **Precision** measures.

On the other hand, when we observe the **Recall** measures the results were significantly different. The **Recall** measures have increased for all the three anomalies. Thus, as the **Recall** measures indicate the sensitivity of the participants when prioritizing and ranking real positive cases that are correctly predicted as positive, we could observe a lower number of **False Negatives**. A lower number of **False Negatives** implies in a higher **Recall**. Independently whether the participants have properly used the information about the descriptive architecture represented in the blueprints, we observed to some extent improvements on occurrence of **False Positives** and **False Negatives**. In summary, we observed that:

(i) even though we observed that the architecture blueprints could somehow improve **Precision** measures on the prioritization and ranking process, the statistical tests indicate the results with marginally statistical significance. It means that further investigations are still required to evaluate whether the Precision can be improved when developers are provided with architecture blueprints when prioritizing and ranking critical code anomalies.

(ii) the **Recall** measures were positively affected by the used of architecture blueprints on the process of prioritizing and ranking critical code anomalies. It means that participants have correctly prioritized and ranked instances of critical code anomalies when compared to the *ground truth* provided by the systems experts.

(iii) besides improving the **Precision** and **Recall** when prioritizing and ranking critical instances of code anomalies, the use of architecture blueprints did not bring any additional effort in terms of time.

Finally, after testing the study hypotheses we discussed the main characteristics of **False Positives** and **False Negatives** observed for each group of participants. Nevertheless, we also discussed what information the participants judged relevant when performing the experimental tasks for the prioritization and ranking critical code anomalies.

Based on the analysis of our second study, we could address the auxiliary research questions defined in this chapter regarding: (i) how the use of blueprints can help the prioritization and ranking of critical code anomalies; and (ii) how useful the architecture blueprints are when prioritizing and ranking critical code anomalies. Thus, based on the aforementioned results, we observed that the process of prioritizing and ranking critical instances of code anomalies should be automated. The fact is that participants tended to invest a significant time when performing the prioritization and ranking of critical code anomalies, since not all of them were able to optimize those activities when architectural information are provided.

In addition, we can also conclude that: (i) although the use of architectural information have improved the results when prioritizing and ranking critical code anomalies, it is not possible to know which specific architectural information should be used when performing the tasks manually; and (ii) the results are not much superior regarding not using blueprints on the prioritization and ranking of critical code anomalies. In this sense, we proposed heuristics (see Chapter 5) to automate the prioritization and ranking tasks according to different architectural drift problems observed in the early stages of the system development. The fact is that each different architectural drift problems required different architectural information when compared to the other architectural problems.