https://dx.doi.org/10.22161/ijaers.6753 *ISSN: 2349-6495(P) | 2456-1908(O)*

On the Effectiveness of Interactive Detection of Code Anomalies: An Empirical Assessment

Danyllo Albuquerque[1], Daniel Abella Mendonça de Souza[1], Wesley Gonzaga Alves[1], Ted Igor Soares Medeiros[1], Marcio Ferreira Pereira[1], Roberto Felício de Oliveira[2], Wagner Cândido da Silva[2]

[1]Department of Information System, UNIFACISA, Brazil
Email: {danyllo.albuquerque, daniel.abella, wesley.alves, ted.medeiros, marcio.pereira}@maisunifacisa.com.br
[2]Department of Information System, UEG, Brazil
Email: {prof.roberto.posse, wagner.candido}@ueg.edu.br

***Abstract*—*Background: Detection of code anomalies should be performed as early as possible in order to effectively reveal refactoring opportunities in due time. Refactoring aims at improving software maintainability, but their late application is counter-productive or even prohibitive. Detection of code anomalies is traditionally supported by non-interactive detection (NID) techniques, which encourage developers to reveal anomalies in later revisions or versions of a program. The reason is that this technique does not support progressive interaction of developers with anomalous code. In addition, it reveals anomalies in the entire source code upon an eventual developer request. More recently, the notion of interactive detection (ID) has emerged to address NID's limitations. This technique reveals anomalies when code fragments are still being edited and without an explicit developer request, thereby encouraging early anomaly detection. Problem Statement: Recent studies suggest the use of NID might lead to: (i) a low number of correctly identified anomalies, and (ii) ineffective refactoring actions. Although ID seems promising, there is no knowledge about its impact on anomaly detection and refactoring actions. Goal: Evaluate the effectiveness of an ID technique on early anomaly detection. In addition, we analyze the aid of an ID technique in performing effective refactoring actions. Method: We conducted a controlled experiment with 14 subjects that underwent tasks related to anomaly detection and judgments of required refactoring actions. Results: Our study reveals the use of ID improves anomaly detection as developers tend to early identify more anomalies when compared to the use of NID. Conclusions: Although ID contributes to detect more anomalies than NID, the former may induce to ineffective refactoring actions.*

Keywords*—*Code Anomalies, Interactive Detection, Software Refactoring.

I. INTRODUCTION

Code anomalies are structures in a program that often indicate the presence of deeper maintainability problems [1]. Code anomalies should be early detected, during the ongoing implementation of a program rather than in later maintenance tasks. Early detection of anomalies is likely to lead to effective refactoring actions [2]. Refactoring is a behavior-preserving change in the program structure intended to remove code anomalies and improve software maintainability [1]. However, the early detection of code anomalies is not a trivial task and many factors can hinder the realization of this task. Among those factors, we highlight that developers may not be able to early identify code anomalies due to their lack of experience on this task [3]. In addition, conventional techniques may offer limited support or discourage early detection of code anomalies [3].

Several techniques for (semi-) automated detection of code anomalies have been proposed in the literature (e.g. [3][5][6][7]). Most of these existing techniques are characterized as supporting non-interactive detection (NID) [3][6]. NID techniques reveal a global list of code anomalies once the source code is completed and compiled. Moreover, the use of NID demands an explicit and eventual request of the developer so that the full source code analysis is triggered. More importantly, NID techniques do not offer means for developers interact with the anomalous code elements while they are producing, editing or inspecting their program statements. All these characteristics of NID techniques encourage late detection of code anomalies.

On the other hand, the notion of interactive detection (ID) has been recently proposed [6]. An ID technique is intended to reveal code anomalies in program fragments

https://dx.doi.org/10.22161/ijaers.6753 ISSN: 2349-6495(P) | 2456-1908(O)

without an explicit developer request, thereby encouraging early detection of code anomalies. In contrast to NID, ID provides support for developers interacting with anomalous code as they edit or browse program statements. Unfortunately, there is little empirical knowledge about the effectiveness of interactive detection of code anomalies [6].

Most of the empirical studies on anomaly detection strictly focuses on the evaluation of NID [9][10][11][12]. These studies pointed out NID techniques induce to a low number of correctly identified code anomalies. Other studies also suggested NID techniques induce to the realization of ineffective refactoring actions [21][22].Therefore, the expectation is that ID techniques can better promote early identification of code anomalies and, as a consequence, effective refactoring actions. Even though organizations and developers might want to consider the adoption of ID techniques, there is no evidence in the literature about its effectiveness on anomaly detection. In other words, there is still a lack of empirical knowledge about the use of ID.

Therefore, our goal is to address the following research question: "Can the use of ID improve the effectiveness on anomaly detection and refactoring actions?". For doing so, we conducted a controlled experiment involving 14 subjects with different working experience and technical knowledge. Subjects performed tasks related to anomaly detection and judgments of refactoring with support of both ID and NID techniques. In order to evaluate the effectiveness of both techniques, we used two measures: precision and recall. We select these two measures because they have been widely adopted in other effectiveness studies involving code anomaly detection [13][14][15]. Our comparative analysis allowed us to evaluate whether some ID characteristics could bring benefits or drawbacks for effective anomaly detection.

The experimental results revealed the use of ID has achieved better effectiveness on code anomaly detection when compared to NID techniques. Developers identified a much higher number of code anomalies when using the ID. On the other hand, we have observed the use of ID might lead to a high number of false positives and, consequently, developers can be induced to perform ineffective refactoring actions.

The remainder of this paper is organized as follows. Section 2 introduces basic concepts required to understand the analysis performed in our study. Study settings are described in Section 3 while the results associated with interactive detection of code anomalies are discussed in Section 4. In Section 5, we present the threats to validity observed in our study. Related work is discussed in Section 6. Finally, we present our conclusions and point out directions for future work in Section 7.

II. BACKGROUND

This section presents essential concepts related to code anomalies, code refactoring and support for anomaly detection.

2.1 Code Anomalies and Refactoring

Code anomalies are symptoms on the program structure that may indicate the presence of deeper maintainability design problems [1]. They suggest where perfective maintenance is required in the source code [1]. Several code anomalies have been proposed and cataloged by several researchers, including Fowler [1], van Emden and Moonen [13], and Arevalo [16]. Typical examples of code anomalies are Feature Envy and Long Method [1].

Early detection of code anomalies is the only possibility of promoting the longevity of a software system. Early detection is the ability of identifying opportunities for refactoring [1][19][20] as soon as anomalies are introduced in the source code by programmers. Longer the code anomalies remain in the source, harder it becomes to refactor out these anomalies from a program. Refactoring [1][17] is defined as behavior-preserving change made in structure of a program with the aim of improving software maintainability. Fowler [1] has identified more than 70 different types of refactoring, which range from local changes in a specific code element (as the Extract Local Variable refactoring) to a global change (as the Extract Class refactoring).

The effectiveness of refactoring actions is largely dependent on the effectiveness of detecting the code anomalies. Preliminary studies [21][22] have exposed negative consequences on code quality whenever ineffective and late refactoring actions are performed. Thus, developers need to identify anomaly instances more effectively and opportunely so that refactoring actions can be performed. In contrast, if developers miss the occurrences of anomaly instances, developers can perform ineffective refactoring actions in the source code.

2.2 Support to Detection of Code Anomalies

Usually, developers use (semi)automated techniques to guide their effort on anomaly detection [18][23]. These techniques are basically comprised of two components [3][7]: (i) a mechanism for anomaly detection; and (ii) a user interface responsible for displaying detected anomaly instances, i.e. occurrences of code anomalies identified by

https://dx.doi.org/10.22161/ijaers.6753 *ISSN: 2349-6495(P) | 2456-1908(O)*

the detection mechanism. The detection mechanism may allow developers to choose or define algorithms for anomaly detection. Developers can choose some metrics and thresholds to compose their own detection algorithms [5]. Based on developer's interaction with the aforementioned components and the anomalous code elements, anomaly detection can be classified according to two different techniques, as shown in Figure 1.

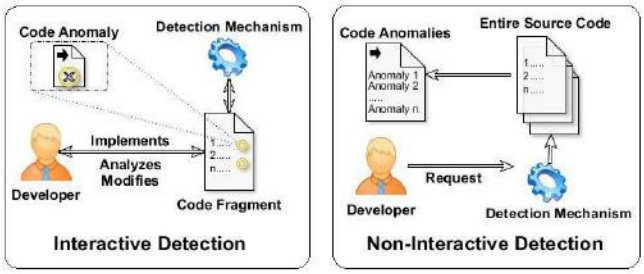

Fig.1: Comparison between techniques for anomaly detection.

Interactive detection (ID) is a technique that supports developer's interaction with anomalous code elements (Figure 1). The ID techniques reveal anomaly instances in code fragments without an explicit request from the developer. Thus, the ID techniques constantly work on detecting anomaly instances in code fragments being manipulated by the developer. Thereby, a developer using ID techniques can early identify instances of code anomalies. Once developers do not directly interact with the mechanism for anomaly detection, they are able to perform other programming activities. In summary, developers are able to analyze, modify and implement the source code while they interact with the anomalous code elements [6].

Non-interactive detection (NID) is a technique that does not support developer's interaction with anomalous code elements (Figure 1). The NID techniques reveal anomaly instances in the entire source code upon an explicit request from the developer. The mechanism for anomaly detection receives the request, and then, it detects anomaly instances in the entire source code. Thereby, developers using NID techniques identify anomaly instances only later (e.g., when code is already implemented). Once developers directly interact with the mechanism of anomaly detection, they are not able to concurrently perform other programming activities in the source code [6].

We analyze the ID technique through Stench Blossom [3]. This tool provides the programmer with three different views, which progressively offer information about the anomaly instances in the code fragment being visualized or edited. Initially, the developer interacts with the Ambient View (Figure 2A). This view relies on the metaphor of a "flower", where each "petal" represents the possible occurrence of a specific anomaly in the code fragment. Higher the radius of a "petal", the higher is the probability of occurrence of the anomaly. The mechanism for anomaly detection of Stench Blossom calculates this probability. For more information about a specific anomaly instance, the developer must click on the "petal" displayed in the Ambient View. When the developer selects an anomaly, the name of code anomaly is presented in a dialog box and then, the Active View is displayed to the developer (Figure 2B).

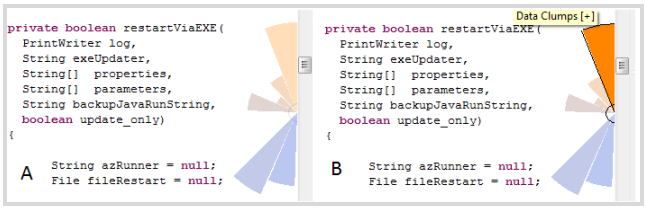

Fig.2. Ambient View (A) and Active View (B).

Finally, if the developer requires detailed information about a specific instance of a code anomaly, the Explanation View (Figure 3) can be displayed from a new click on the name of the anomaly under analysis. The developer can use the color gradation to verify which code fragments are related to a specific instance of code anomaly. Therefore, the interaction with anomalous code elements provided by Stench Blossom, allows developers better understanding the origins of different instances of a given code anomaly.

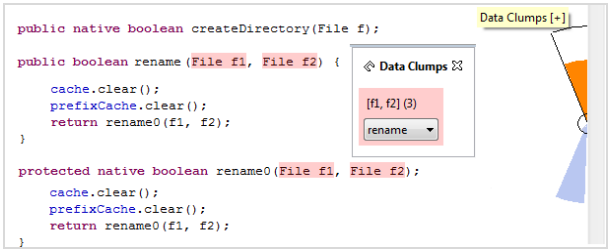

Fig.3. Explanation View.

III. STUDY SETTINGS

This section presents the main concepts related to execution of this research. The details related to the experiment, the choice of subjects and procedures for data analysis are described below.

3.1 Effectiveness evaluation

Effectiveness on detection of code anomalies is one of most important criteria for choosing a technique to perform this activity [8][9]. When a technique for detection of code anomalies is considered effective, it means the technique is able to detect a high number of anomaly instances in a program. In addition, effective techniques should ideally detect only anomaly instances

https://dx.doi.org/10.22161/ijaers.6753 ISSN: 2349-6495(P) | 2456-1908(O)

are indeed a maintainability problem. If developers use effective techniques, they can identify anomaly instances and consequently refactoring opportunities in order to improve the software maintainability [8][9].

We used precision and recall to evaluate the effectiveness of anomaly detection. In the following, we define the concepts required to understand these two measures. Existing code anomalies (ECA) are actual anomaly instances identified by the technique for anomaly detection, where these instances are indeed confirmed by the experts as a maintainability problem. Experts are developers with deep knowledge about the system and its maintainability problems. Detected code anomalies (DCA) are anomaly instances identified through the use of an anomaly detection technique. Not all the detected code anomalies are confirmed as existing (actual) code anomalies by the experts. True positives (TP) are those anomaly instances present in both DCA and ECA sets – i.e. anomaly instances identified by experts that actually represent a maintainability problem. False positives (FP) are anomaly instances identified by the programmers using a detection technique, but they are not in the ECA set. Finally, False negatives (FN) are anomaly instances not identified by the developers, which are in ECA.

The precision and recall measures defined in above equations (Eq) were adapted from Rijsbergen [26] and have been widely used in other studies [13][14][15]. These previous studies were also focused on comparing techniques for anomaly detection. Precision quantifies the rate of true positives by the number of detected code anomalies. Recall quantifies the rate of true positives by the number of existing code anomalies.

3.2 Research Questions

In order to address our general research question (Section 1), we defined two specific goals: (i) assess whether developers using the interactive detection (ID) technique identify code anomalies more effectively compared to the non-interactive detection (NID) technique; and (ii) assess whether using ID technique lead developers to perform ineffective refactoring compared to NID technique. Thus, we defined three research questions (Table 1) to achieve the aforementioned goals.

Table 1. Specific Research Questions

RQ	Description
RQ1	Does the ID technique improve the *recall* in detection of code anomalies?
RQ2	Does the ID technique improve the *precision* in the detection of code anomalies?
RQ3	Does the ID technique contribute to perform ineffective refactoring actions?

The first research Question (RQ1) compares both techniques using precision measure. This analysis is important because it shows the effectiveness of the ID technique regarding the number of true positives and false positives. Similarly, in our second research question (RQ2), we compared the recall measure of ID and NID techniques. The recall is as important as the precision. For example, it allows us to find which technique induced developers to miss more anomaly instances.

Finally, our third research question (RQ3) is focused on evaluate how the techniques interfere in the refactoring actions. As we discussed, code anomalies are considered indicators for refactoring actions. Thus, our work consider as effective refactoring actions, those modifications applied over anomalous code elements in order to improve the system maintainability. Although, techniques for anomaly detection might indicate false positives, and hence, developers will apply refactoring actions over code elements that do not represent a true threaten the system maintainability (i.e. ineffective refactoring actions).

For each research question, we defined hypotheses (H) which are summarized in Table 2. Thereby, we defined H1 and H2 due to empirical evidence found in the work of Murphy-Hill and Black [3]. This work pointed out the use of interactive detection (ID) technique can increase the number of anomaly instances found in the source code. Therefore, our expectation is that the use of ID technique may improve the effectiveness on detection of code anomalies in terms of precision and recall measures. We defined H3 as consequence of H1 and H2. Since ID technique constantly provides (i.e. regardless developers' request) information about anomaly instances, this amount and availability of information may improve the developers' reliability on anomaly detection. Consequently, our expectation is that developers may reduce the amount of false positives and hence, a smaller amount of ineffective refactoring actions would be performed.

Table 2. Hypotheses

H	Description
H1	The ID technique has a better recall than the NID technique.
H2	The ID technique has a better precision than the NID technique.
H3	The ID technique leads to performing less ineffective refactoring actions than the NID technique.

3.3 Method and Subjects

We use the recommendations outlined in the work of Kitchenham et al. [24] as a guide for establishing and implementing a controlled experiment. The subjects

https://dx.doi.org/10.22161/ijaers.6753 ISSN: 2349-6495(P) | 2456-1908(O)

accomplished tasks related to detection of code anomalies and identification of refactoring opportunities (Section 3.4). They performed these tasks with support of ID and NID techniques. We choose the ID technique provided by Stench Blossom [3] for two main reasons: (i) it provides support to all ID features [6], as previously discussed (Section 2.2); and (ii) to the extent of our knowledge, it is the only robust solution that provides automated support for ID.

We choose the NID technique provided by manual inspection due to it has been widely used in other comparative studies [10][3][4] of techniques for anomaly detection. In addition, this technique does not require automated detection, thereby providing no specific influence of a particular detection mechanism. Similarly, we have also not found any other automated detection technique that supports the same set of anomalies addressed by Stench Blossom. For instance, the automated detection proposed by van Emden and Moonen [13] provides support to only two code anomalies (Instanceof and Typecast). Conducting a comparative experiment against just these two code anomalies would produce quite limited results. Finally, it also provides us with a reference to analyze the impact of an automated ID technique.

The comparison between ID and NID techniques allowed the analysis of whether particular characteristics of ID (e.g. early detection) bring apparent (dis)advantages. It is not the intent of this experiment to compare various ID techniques, such as the one realized by Stench Blossom. This choice is because, to the extent of our knowledge, there is no other robust automated solution that offers an interactive technique for supporting anomaly detection. Finally, many would consider ID and NID complementary rather than competitive techniques as they are naturally targeted at different development stages (Section 2.2). Although ID and NID can be used in a complementary way, they can also be used with the same purpose during a programming activity (e.g. analysis of code fragments). In the context of our experiment, the techniques for anomaly detection were evaluated with the same purpose: detection of code anomalies while browsing code elements.

Regarding to the subjects of this study, we recruited two main groups: (i) postgraduate students and (ii) professionals developers. These subjects were selected based on the criteria of interest in participating of the experiment. We expected from subjects, at least, intermediate knowledge in Java and refactoring. However, we did not expect from subjects knowledge about code anomalies or the interactive detection technique used in the experiment. Due to space constraints, detailed description of subjects profile may be found online in our paper supplementary material [25].

3.4 Experiment Description

The subjects performed tasks related to identification of code anomalies and refactoring opportunities. In these tasks, the subjects manipulated Java code files extracted from Java Core Library [25]. We have chosen this project because is an open source industrial system, making it easier to replicate this study by independent researchers. Four code files were selected according to the similar size and amount of the code anomalies. The experimental phase required two code files – one file for the ID task (e.g. file A) and the other to NID task (e.g. file B). This criterion was adopted because both files could be used in the tasks, regardless of the order, reducing their influence on the results of the experiment. Each experimental task was individually conducted with the first author as an observer of the experiment. It is also important to mention we already provided the environment with all the files and tooling support required to execute the experimental tasks. The maximum time each subject had available for executing the experimental tasks was 60 minutes. A detailed description of experimental tasks may be found online in our paper supplementary material [25]. Finally, we organized the experiment into three different phases, namely:

Phase 1 – Pre-Experiment: Initially, the subjects answered a questionnaire to collect the necessary data for definition of subjects' profile (Section 4). Then, the subjects received a material with the definition of eight (8) code anomalies supported by Stench Blossom, as well as an example of the occurrence of each one. A detailed description of code anomalies used may be found online in our paper supplementary material [25]. A deadline of 15 minutes (maximum) was given for the subject to understand these definitions. This step was intended at leveling the knowledge of the subjects. Finally, the subjects underwent a training session about Stench Blossom and the Eclipse IDE version used in the experiment.

Phase 2 – Identification of Code Anomalies: Subjects identified the occurrences of eight (8) different types of code anomalies supported by Stench Blossom. The data related to identification of code anomalies were transcribed into a form. During the use of ID technique, the subjects could agree or disagree with the detections proposed by this technique. Thus, false positives arising from the ID technique could be omitted when subjects used their knowledge in making decisions about the existing anomaly instances. Two tasks were performed in

https://dx.doi.org/10.22161/ijaers.6753 ISSN: 2349-6495(P) | 2456-1908(O)

this experiment phase: one with the ID and another one with the NID. We computed for each task: the total (T) number of detected code anomalies (DCA), the number of true positives (TP) and the number of false positives (FP). The data obtained from these tasks will be used to evaluate the first and second hypothesis (H1 and H2) and the Section 4.1 provides its detailed description.

Phase 3 – Judgments of Refactoring: Subjects performed judgments of refactoring using ID and NID techniques. This phase consisted in identifying of Feature Envy anomaly. We decided to focus on Feature Envy for this experimental phase, as this is the only code anomaly currently supported by the implementation of the Explanation View (Section 2.2). After the identification of Feature Envy, the subject should infer about the usefulness of applying a refactoring action. In positive case, the subject should answer the following questions: (i) how scattered is the anomaly in the analyzed code, (ii) how likely removing this anomaly and (iii) which refactoring actions are required. The aforementioned questions are directly related to judgments of refactoring [1][2]. The following concepts are required to understand this task: Ineffective Refactoring (IR) occurs when the developer positively infers about refactoring from an instance of Feature Envy anomaly, which has been considered a false positive. Effective Refactoring (ER) occurs when the developer positively infers about refactoring necessity from an instance of Feature Envy anomaly, which has been considered a true positive. The data obtained from these tasks will be used to evaluate the third hypothesis (H3) and its description can be seen in Section 4.2.

3.5 Analysis Method

We applied statistical analysis on the data obtained from experimental tasks. Such statistical analyzes were carried out with support of the R tool [27]. This tool provides means for calculating statistical tests considered in this study: (i) Wilcoxon signed-rank test [28], and (ii) paired T-Test [28]. The first one is applied to the values associated with the correctly identified anomaly instances. This test was selected since the data were not following a normalized distribution. The second one is applied to the values of recall and precision since the obtained measures were following a normalized distribution. The execution of the experimental tasks derived data for two samples: the sample with the aid of ID and the sample with the aid of NID technique. The aforementioned statistical tests can be applied since each observation in the first sample can be paired with one observation of the second sample.

IV. RESULTS AND DISCUSSION

In this section, we present the results of the experimental tasks described in Section 3.4. Each subject spent on average 45 minutes to execute the experiment. Therefore, the upper limit of one hour was enough for the subjects conclude the tasks. Whenever it is appropriate, statistical analyzes are presented. The first phase (Section 3.4) of the experiment involved the application of a questionnaire aiming to determine the subjects' profile. Table 3 summarizes the main characteristics of the subjects' profile. Their profile meets our study assumptions since all subjects have at least intermediate knowledge about Java, detection of code anomalies and program refactoring. The following subsections present the key results and findings revealed by our study.

Table 3. Results of the pre-experiment questionnaire

Question	Results
Professional practice	7 Subjects were postgraduate students and 7 subjects were professional developers
Experience time	Half of the sample had between 5 and 8 years of experience in software development
Using IDE	All subjects have used some IDE
Java proficiency	On a scale from 0 to 4 **(*)**, 36% of subjects answered 2 and 57% of subjects answered 3
Anomaly detection proficiency	On a scale from 0 to 4 **(*)**, approx. 80% of the subjects answered 1 or 2.
Refactoring proficiency	On a scale from 0 to 4 **(*)**, approx. 60% of the subjects answered 3 or 4.
(*) 0 means "not proficient" and 4 "very proficient"	

4.1 Identification of Code Anomalies

The second phase involved the execution of the tasks related to identification of code anomalies using non-interactive detection (NID) and interactive detection (ID) techniques. The tasks focused on analyzing the effectiveness of using ID on the detection of code anomalies. Table 4 describes the results per subject or full sample (FS) with respect to the detected code anomalies (DCA), true positives (TP), and false positives (FP).

Table 4. Results of identification of code anomalies

Subject	NID			Subject	ID		
	DCA	TP	FP		DCA	TP	FP
Developer 1	4	4	0	Developer 1	6	5	1
Developer 2	7	6	1	Developer 2	15	13	2
Developer 3	9	8	1	Developer 3	16	14	2
Developer 4	6	5	1	Developer 4	9	7	2

https://dx.doi.org/10.22161/ijaers.6753 *ISSN: 2349-6495(P) | 2456-1908(O)*

Developer 5	10	8	2	Developer 5	11	9	2
Developer 6	12	9	3	Developer 6	14	11	3
Developer 7	6	5	1	Developer 7	5	5	0
Total	**54**	**45**	**9**	**Total**	**76**	**64**	**12**
Student 1	4	3	1	Student 1	10	8	2
Student 2	4	4	0	Student 2	5	4	1
Student 3	8	6	2	Student 3	12	9	3
Student 4	5	4	1	Student 4	3	3	0
Student 5	7	5	2	Student 5	10	7	3
Student 6	5	4	1	Student 6	6	5	1
Student 7	2	2	0	Student 7	8	6	2
Total	**35**	**28**	**7**	**Total**	**54**	**42**	**12**
FS Total	**89**	**73**	**16**	**FS Total**	**130**	**106**	**22**
FS Average	**6,1**	**5,2**	**1,1**	**FS Average**	**9,3**	**7,6**	**1,6**

ID technique increases both true and false positives: We observed the subjects identified 22 false positives when using the ID technique. That is, the number of false positives is approximately 38% higher than the number of false positives (16) produced when subjects used the NID technique. Similarly, the subjects identified 106 true positives (i.e. anomalies correctly identified) based on the use of ID technique, while subjects identified 73 true positive based on the use of NID technique. Therefore, the use of ID increased in 45% the total of true positives by the subjects when identifying code anomalies. Finally, the data related to true positives generated with ID and NID techniques were statistically significant ($p = 0.002$, $df = 12$, $z = 3.05$, using a Wilcoxon signed-ranks test [28]).

Aiming to provide an additional perspective on the effectiveness of the interactive detection of code anomalies, we also analyzed precision and recall measures. Therefore, we applied those collected measures in the equations defined in Section 3.1. The Table 5 illustrates the results of these metrics for both ID and NID techniques. The precision and recall measures were calculated in order to address the research questions RQ1 and RQ2. In addition, these results were used in order to test the hypotheses H1 and H2, respectively.

ID increases recall: When analyzing recall measures, we observed that, in average, the subjects using the ID technique achieved a score of 0.30, while the use of the NID achieved 0.21. Thus, the results represent a difference of approximately 30% in favor of the ID technique. Similar results could be observed when analyzing different samples (e.g. students or developers). For instance, the developers' sample improves recall values in 40%, while the students' sample improves recall values in 50%. Likewise, the data related to recall in this task through ID and NID was statistically significant ($p = 0.0013$, $df = 13$, $t = 4.06$, using a Paired T-Test [28]) in the task of identification of code anomalies.

We also found that recall suffers direct influence regarding the subjects' working experience. The results allowed us to conclude the use of ID can directly affect the recall values. The use of ID implies the interaction of subjects with the anomalous code elements as they progressively analyze code fragments. Therefore, developers are able to achieve more coverage with ID regarding the correctly identified instances of code anomalies. Finally, we can confirm the first hypothesis (H1), since the use of ID led to better recall values compared to the use of NID.

Table.5. Precision and recall

Subject	**ID**		**Subject**	**NID**	
	P	**R**		**P**	**R**
Developer 1	0,83	0,20	Developer 1	1,00	0,16
Developer 2	0,87	0,52	Developer 2	0,86	0,24
Developer 3	0,88	0,56	Developer 3	0,89	0,32
Developer 4	0,78	0,28	Developer 4	0,83	0,20
Developer 5	0,82	0,36	Developer 5	0,80	0,32
Developer 6	0,79	0,44	Developer 6	0,75	0,36
Developer 7	1,00	0,20	Developer 7	0,83	0,20
Average	**0,85**	**0,37**	**Average**	**0,86**	**0,26**
Student 1	0,80	0,32	Student 1	0,75	0,12
Student 2	0,80	0,16	Student 2	1,00	0,16
Student 3	0,75	0,36	Student 3	0,75	0,24
Student 4	1,00	0,12	Student 4	1,00	0,08
Student 5	0,70	0,28	Student 5	0,71	0,20
Student 6	0,83	0,20	Student 6	0,80	0,16
Student 7	0,75	0,24	Student 7	0,80	0,16
Average	**0,80**	**0,24**	**Average**	**0,83**	**0,16**
Total Average	**0,82**	**0,30**	**Total Average**	**0,84**	**0,21**

ID and NID techniques have similar precision: We observed the average of precision measures with ID was 0.82, while the use of NID achieved 0.84. As opposed to recall values, the difference of precision measures with NID and ID was not significant. This finding is revealed when analyzing percentage values. We also realized the subjects' working experience directly affected the results. The professionals' sample achieved better precision values compared to the students' sample. As previously discussed, although the use of the ID technique increases the number of false positive, it also tends to increase the number of true positive - which directly affect precision values. According to results illustrated in Table 5, there is no evidence to support that the subjects using ID have worse (or better) precision than subjects using NID technique. Therefore, we cannot confirm or refute the second hypothesis (H2).

https://dx.doi.org/10.22161/ijaers.6753 *ISSN: 2349-6495(P) | 2456-1908(O)*

The results indicated there is no negative impact when the interactive detection of code anomalies is performed progressively - i.e. while the developer is browsing or editing the code. Software developers are likely to benefit from detecting anomalies earlier, when they constantly receive feedback provided by ID. Moreover, the constant availability and higher amount of information through ID led developers to accept a higher number of anomaly instances. However, if the subject holds a higher level of working experience, he can be more confident to infer (i.e. accept or reject) about the suggestions of anomaly instances from ID. The data described in Table 4 allow us confirm this assumption. More experienced developers using ID obtained a lower number of false positives compared to the students (fewer working experience) using the same technique. In a similar way, developers identified a higher number of true positives compared to students. Finally, these results are similar to those presented in the work of Murphy-Hill and Black [3], as developers identify more true positives using ID compared to developers using NID technique.

4.2 Judgments of Refactoring

In the third phase (Section 3.4), subjects performed judgments of refactoring using non-interactive detection (NID) and interactive detection (ID) techniques. These tasks were performed in order to address the research question RQ3, which is validated by testing the hypothesis H3. In summary, we analyzed whether the subjects performed ineffective refactoring (IR) or effective refactoring (ER) related to occurrence of Feature Envy anomaly. Section 3.4 shown a detailed description of the judgments of refactoring. Finally, the Table 6 illustrates results from the accomplishment of aforementioned tasks.

Table 6. Results on judgments of refactoring

Subject	NID		Subject	ID	
	IR	ER		IR	ER
Developer 1	-	X	Developer 1	X	-
Developer 2	-	X	Developer 2	-	X
Developer 3	-	X	Developer 3	-	X
Developer 4	-	X	Developer 4	-	X
Developer 5	-	X	Developer 5	-	X
Developer 6	-	X	Developer 6	-	X
Developer 7	X	-	Developer 7	X	-
Total	**1**	**6**	**Total**	**2**	**5**
Student 1	X	-	Student 1	X	-
Student 2	-	X	Student 2	-	X
Student 3	-	X	Student 3	X	-
Student 4	-	X	Student 4	-	X
Student 5	-	X	Student 5	X	-
Student 6	-	X	Student 6	-	X
Student 7	X	-	Student 7	X	-
Total	**2**	**5**	**Total**	**4**	**3**
FS Total	**3**	**11**	**FS Total**	**6**	**8**

ID technique may increase IR: We verified the subjects performed 3 ineffective refactoring when using the NID, while the subjects using the ID performed 6. That is, the use of ID occasioned a growth of 50% in the ineffective refactoring performed by subjects. Moreover, when analyzing the results achieved by the developers' sample, subjects performed only 1 ineffective refactoring when using NID, while 2 ineffective refactoring were performed when ID was employed. The same proportion of growth (i.e. 50%) occurs in the results obtained from students' sample. We noticed 2 ineffective refactoring were performed when the NID was used, while subjects using the ID performed 4.

In summary, we observed the subjects using ID are likely to perform more ineffective refactoring compared to subjects using NID technique. Moreover, we could observe that working' experience also influences the results of this task, since the developers performed 50% fewer ineffective refactoring than the students. During the second experimental phase, we observed most of the false positives were related to occurrences of the Feature Envy anomaly. Furthermore, the students using ID pointed out the majority of false positives. This fact led us to conclude occurrences of false positives might be directly associated with developers' working experience. Moreover, the use of the ID technique for the anomaly detection also directly affects the refactoring actions.

Concluding, we can refute the third hypothesis (H3) by analyzing the collected data associated with judgments of refactoring (Table 6). The use of ID might induce developers to perform ineffective refactoring actions because the existence of the anomaly instance that could indicate the refactoring action may be untruthful. In short, if the developer performs refactoring on a false positive related to some anomaly, the effort to accomplishment this task might not contribute to improving the system maintainability.

V. THREATS TO VALIDITY

Sample size and diversity: Fourteen subjects performed the controlled experiment. The results may have direct influence from size of the sample and the subjects' working experience on anomaly detection and refactoring. To mitigate this threat, we choose a sample comprising by students and developers. Furthermore, we conducted training sessions in order to leveling the knowledge of subjects with respect to these topics.

Experiment Complexity: Other threats to validity are related to: (i) the difficulty in understanding code files

https://dx.doi.org/10.22161/ijaers.6753 ISSN: 2349-6495(P) | 2456-1908(O)

chosen for the experiment, and (ii) the nature of the selected experimental tasks. With the aim of minimizing the first threat, the code files were selected according to the size and time available for each task. Furthermore, we performed a pilot experiment in order to adjust the time required to perform these tasks. Aiming at mitigate the second threat, one of the paper's authors monitored all the experimental tasks. In addition, subjects received instructions for completing the questionnaires and demonstrations prior to the completion of the experiment tasks.

Integrated Development Environment: We used the version 3.3 (Europe) of the eclipse IDE due to compatibility problems of Stench Blossom. Thus, the subjects' experience in using this IDE version may have been harmed because it was a version older than the one being used by developers nowadays. However, observations of the subjects did not lead us this phenomenon had any influence on their performance. In addition, aiming to minimize this threat, we provided specific training on the use of the Eclipse IDE 3.3.

Sample of Code Anomalies: Finally, we restricted subjects to discussing only eight types of code anomalies. In contrast, Fowler has cataloged a list with more than twenty code anomalies [1]. Therefore, the eight code anomalies supported by Stench Blossom may not necessarily be a representative sample of anomalies found in certain programs. Likewise, we only focused on one code anomaly in the Explanation View - Feature Envy anomaly. Therefore, subjects' judgments of refactoring may be different for other kinds of code anomalies.

VI. RELATED WORK

This study represents a first independent assessment of interactive detection (ID) of code anomalies. We have chosen the ID technique supported by the Stench Blossom (Section 2.2). This technique was proposed and implemented by Murphy-Hill and Black [3]. We have chosen this particular technique for three main reasons: (i) it provides support to all main features of an ID technique [6]; (ii) it offers automated robust support for ID; and (iii) to the extent of our knowledge, it is the only robust solution that provides automated support for ID.

In our evaluation, we compared the effectiveness on detection of code anomalies with ID and NID techniques. Mäntylä et al. [10] also conducted an empirical evaluation comparing two different techniques. However, they did not evaluate the ID technique for anomaly detection. Instead, they compared manual inspection in contrast to semi-automatic technique (i.e. both techniques were non-interactive). Since ID technique provided by Stench Blossom presents a visualization environment, we seek in the literature related works that also present use of these environments in detection of code anomalies. Parnin et al. [11] evaluated the impact of visualization techniques in the anomaly detection. However, they did not evaluate the use of ID-sensitive visualization of code anomalies.

None of the aforementioned studies presented information about false positives found from the use of different detection techniques. Moreover, none of them analyzed the impact of ID on the identification of refactoring opportunities and their consequences. Although Macia et al. [12] evaluated the number of false positives on anomaly detection, the technique used by them does not support ID. Finally, only Murphy-Hill and Black [3] evaluated the use of their ID technique. However, they prioritized aspects related to usability guidelines such as availability, lucidity and context sensitiveness. Consequently, they did not observe if the ID technique improved the effectiveness on anomaly detection and, consequently, on judgments of refactoring.

VII. CONCLUSION AND FUTURE WORKS

The use of interactive detection (ID) technique can lead developers to early identify opportunities for refactoring actions and hence, bring significant benefits to system maintainability. Using ID: (i) developers can perform other programming activities in source code concomitantly to anomaly detection; (ii) developers are constantly aware about the anomaly instances when analyzing different code fragments; and (iii) developers tend to find early a higher number of anomaly instances due to the amount and availability of information related to code anomalies.

Although developers using ID may identify more anomaly instances found in their code, its use may also increase the number of false positives in early anomaly detection activities. Findings of our study point out that these differences of ID and NID occur for different reasons. First, the amount and availability of information may confuse the developer in the task of interactive identification of code anomalies. Second, the lack of developers' working experience directly contributes to a higher acceptance of suggestions of anomaly instances yielded by an ID technique.

The effectiveness measurements also revealed that ID do not considerably improve the precision of early anomaly identification. However, we realized the subjects' working experience could directly affect the results associated with this measure. The higher the subjects' working experience, the higher is the values observed for precision. Analogously, the subjects'

https://dx.doi.org/10.22161/ijaers.6753 *ISSN: 2349-6495(P) | 2456-1908(O)*

working experience also directly affected the recall measures. Actually, the subjects' working experience improved recall values in a greater proportion compared to values associated with precision.

Finally, new experiments about ID effectiveness can be performed using a different set of code anomalies with different levels of granularity (i.e. anomalies that affect different code elements, such as packages, classes and methods). This recommendation is even more relevant for the second phase of the experiment (Section 3.4), which focused on the occurrence of the Feature Envy anomaly.

REFERENCES

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

[2] Tourwé, T. and Mens, T. Identifying Refactoring Opportunities Using Logic Meta-programming. In Proc. of 7th European Conference on Software Maintenance and Reengineering, pp. 91-100, 2003.

[3] Murphy-Hill, E., and Black, A.P. An Interactive Ambient Visualization for Code Smells. In Proc. of the 5th Int'l Symposium on Software Visualization, pp. 5-14, 2010.

[4] Mäntylä, M.V., Vanhanen, J., and Lassenius, C. Bad Smells – Humans as Code Critics. In Pro. of 20th Int'l Conference on Software Maintenance, pp. 399-408, 2004.

[5] Marinescu, R. Measurement and Quality in Object-Oriented Design. In Proc. of 21th International Conference on Software Maintenance, pp. 701-704, 2005.

[6] Albuquerque, D.W. et al. Detecção Interativa de Anomalias de Código – Um Estudo Experimental. In Proc. of 11th Brazilian Workshop on Software Modularity, 2014.

[7] Murphy-Hill, E. and Black, A.P. Seven Habits of a Highly Effective Smell Detector. In Proc. of Int'l Workshop on Recommendation Systems for Software Engineering, pp. 36-40, 2008.

[8] Murphy-Hill, E. and Black, A.P. Refactoring Tools: Fitness for Purpose. IEEE Software, Vol. 25, Issue 5, pp. 38-44, August 2008.

[9] Simon, F., Steinbrückner, F., and Lewerentz, C. Metrics Based Refactoring. In Proc. of 5th European Conference on Software Maintenance and Reengineering, pp. 30-38, 2001.

[10] Mäntylä, M.V. An Experiment on Subjective Evolvability Evaluation of Object-Oriented Software: Explaining Factors and Inter Rater Agreement. In Proc. of Int'l Symposium on Empirical Software Engineering, 2005.

[11] Parnin, C., Görg, C., and Nnadi, O. A Catalogue of Lightweight Visualizations to Support Code Smell Inspection. In Proc. of 3th Int'l Symposium on Software visualization, pp. 77-86, 2008.

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

[13] Van Emden, E., and Moonen, L. Java Quality Assurance by Detecting Code Smells. In Proc. of 9th Working Conference on Reverse Engineering, pp. 97-106, 2002.

[14] Moha, N. et al. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Soft. Engineering, Vol. 36, Issue 1, pp. 20-36, January 2010.

[15] Dhambri, K., Sahraoui, H., and Poulin, P. Visual Detection of Design Anomalies. In Proc. of 12th European Conf. on Soft. Maintenance and Reengineering, pp-279-283, 2008.

[16] Arevalo, G. et al. Discovering Unanticipated Dependency Schemas in Class Hierarchies. In Proc. of 9th European Conf. on Software Maintenance and Reengineering, pp. 62-71, 2005.

[17] Opdyke. W.F. Refactoring Object-Oriented Frameworks, University of Illinois at Urbana-Champaign, 1992.

[18] Murphy-Hill, E., and Black, A.P. Why Don't People Use Refactoring Tools? In Proc. of Workshop on Refactoring Tools, 2007.

[19] Kerievsky, J. Refactoring to Patterns. Pearson Higher Education, 2004.

[20] Murphy-Hill, E., Parnin, C., and Black, A.P. How We Refactor, and How We Know It. In Proc. of 31st Int'l Conference on Software Engineering, pp. 287-297, 2009.

[21] Pizka, M. Straightening Spaghetti Code with Refactoring. In Proc. of Int'l Conference on Software Engineering Research and Practice, pp. 846-852, 2004.

[22] Bourquin, F., and Keller, R. High-Impact Refactoring Based on Architecture Violations. In Proc. of 11th European Conference on Software Maintenance and Reengineering, pp. 149-158, 2007.

[23] Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In Proc. of 11th European Conf. on Software Maintenance and Reengineering, pp. 329-331, 2007.

[24] Kitchenham, B. et al. Preliminary Guidelines for Empirical Research in Software Engineering. IEEE Transactions on Software Eng., Vol. 28, Issue 8, pp. 721-734, August 2002.

[25] Repository of the controlled experiment. Available in: < http://www.inf.puc-rio.br/~inf2107/experimento.html>

[26] Rijsbergen, C. J. Information Retrieval. Butterworth 1979.

[27] R Tool. Available in: <http://www.r-project.org/>. Access in August 2014.

[28] Siegel, S., and Castellan, J. Nonparametric Statistics for the Behavioral Sciences, 2nd Edition. Mc-Grawl-Hill International Editions, 1988.