# Identifying Code Smells with Collaborative Practices: A Controlled Experiment

Roberto Oliveira[1], Bernardo Estácio[2], Alessandro Garcia[1],
Sabrina Marczak[2], Rafael Prikladnicki[2], Marcos Kalinowski[3], Carlos Lucena[1],

[1]PUC-Rio, Rio de Janeiro, Brazil, [2]PUCRS, Porto Alegre, Brazil, [3]UFF, Niterói, Brazil
{rfelicio, afgarcia, lucena}@inf.puc-rio.br, {bernardo.estacio, sabrina.marczak,
rafaelp}@pucrs.br, kalinowski@ic.uff.br

***Abstract*—Code smells are often considered as key indicators of software quality degradation. If code smells are not systematically removed from a program, its continuous degradation may lead to either major maintenance effort or the complete redesign of the system. For several reasons, software developers introduce smells in their code as soon as they start to learn programming. If novice developers are ought to become either proficient programmers or skilled code reviewers, they should be early prepared to effectively identify code smells in existing programs. However, effective identification of code smells is often not a non-trivial task in particular to a novice developer working in isolation. Thus, the use of collaborative practices may have the potential to support developers in improving their effectiveness on this task at their early stages of their careers. These practices offer the opportunity for two or more developers analyzing the source code together and collaboratively reason about potential smells prevailing on it. Pair Programming (PP) and Coding Dojo Randori (CDR) are two increasingly adopted practices for improving the effectiveness of developers with limited or no knowledge in software engineering tasks, including code review tasks. However, there is no broad understanding about the impact of these collaborative practices on the effectiveness of code smell identification. This paper presents a controlled experiment involving 28 novice developers, aimed at assessing the effectiveness of collaborative practices in the identification of code smells. We compared PP and CDR with solo programming in order to better distinguish their impact on the effective identification of code smells. Our study is also the first in the literature to observe how novice developers work individually and together to identify smells. Our results suggest that collaborative practices contribute to the effectiveness on the identification of a wide range of code smells. Our findings can also be used in practice to guide educators, researchers or teams on improving detection and training on code smell identification.**

***Keywords*—Code Smells; Collaborative Practices; Controlled Experiment; Program Comprehension; Software Degradation***

## I. Introduction

As software systems evolve over time, they invariably undergo changes that can lead to software degradation. Code smells are often considered as key indicators of software quality degradation [8]. They affect different code units, such as classes and methods. Hence, if these code smells are not systematically removed, the software degradation may reach a level that requires either major maintenance effort or the complete redesign of the system. In order to avoid these problems, code smells should be identified and removed as soon as possible. In reality, novice developers introduce smells in their code as soon as they start learning programming or contributing in a software project [6][9]. If novice developers are ought to become either proficient programmers or skilled code reviewers, they should be early prepared to effectively identify code smells in existing programs.

The challenge is that the identification of code smells is not a trivial task in particular to novice developers working in isolation [9]. This task often requires subjective code analysis from the developer, which makes it an error-prone activity [5]. In spite of the extensive tool support for smell detection available nowadays (e.g. [13]), developers still need to analyze each code smell for confirmation purposes. Accordingly, this confirmation might require the knowledge of various program modules (e.g. classes or packages), which may have been developed by someone else with different programming styles. These several complexities imply that the identification of code smells is often hard to be conducted and understood by a single novice developer.

Novices become skilled in non-trivial software engineering tasks, such as smell identification, only if they are trained to effectively do so in early stages in their careers [3]. There is a need for investigating whether novice developers using collaborative practices are more effective than solo developers on the identification of code smells. Collaborative practices consist of at least two developers working together on the same software development activity (i.e. editing or reviewing the source code). They mainly differ in the way that two or more participants are allocated to the software development activity. While some of them encourage work in pairs [19], others promote work in groups [20]. Pair Programming (PP) is a typical representative of the former, while Coding Dojo (CD) is an emerging practice for group-based work. There are several variants of CD and one of them is called Coding Dojo Randori (CDR) [20]. Section II-B presents the dynamics of these practices in detail. Over the years, collaborative practices have also been used in software companies, such as Microsoft Corporation [1], Siemens [12] and ThoughtWorks [7] for improving the effectiveness of developers with limited or no knowledge in software engineering tasks. Through the use of collaborative practices, novice developers can exchange information, share their knowledge, learn specific tasks, and analyze the source code with more confidence.

Although collaborative practices seem promising to support novices on code smell identification, there is no empirical evidence about the (dis)advantages of such practices on improving

978-1-5090-5086-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SBCARS.2016.18

TABLE I. CODE SMELLS

| Class | Smell Type | Definitions |
|---|---|---|
| Intra-Class | Long Method | Method characterized by being long and excessively complex. |
| | Unused Field | A variable that is created but it is never used in the code. |
| | Duplicated Code[1] | The same code structure appears in more than one place in the same class. |
| Inter-Class | Data Class | Classes that have only fields as well as getting and setting methods, and nothing else. |
| | Feature Envy | A method is more interested in methods of another than in its own class. |
| | Lazy Class | The Lazy Class is a class that does not do enough and its members should be placed in other class. |
| | Intensive Coupling | When a method is tied to many other scattered methods in the system, whereby provider operations are dispersed only into one or a few classes. |

the effectiveness of this task. Even though analyzing how a novice deal with code smells has been a recent trend (e.g. [9]), there is no investigation on the impact of collaborative practices in this context. We performed a controlled experiment in order to investigate the impact of novice developers using collaborative practices when identifying code smells. We selected PP and CDR as representatives of pair-based and group-based practices, respectively. We compared these practices with solo programming in order to better distinguish their impact on the effective identification of code smells. As we are comparing different practices for identification of code smells, we consider a practice to be effective if the developers, using the practice, correctly identify more code smells within time constraints than when they use another practice. During the study, we also qualitatively investigated the way novices individually and collaboratively perform the identification of code smells.

Our comparative analysis of the collaborative practices is intended to be relevant to: (i) novice learners (and their educators) that plan to acquire (teach) skills on code smell identification, and (ii) training of project teams with limited or no experience, in which they collaborate to learn together without the availability of more experienced developers (e.g. Kazan project by ThoughtWorks [7]). Therefore, all the subjects taking part in our experiment were novice developers. They were classified according to two levels of experiences: (i) novice developers without experience in industrial software projects, and (ii) novice developers who worked in at least one industrial software project (Section IV-B). Our results suggest that collaborative practices contribute to the effectiveness on the identification of a wide range of code smells. We characterized which smells benefit from collaborative practices (w.r.t. identification effectiveness) in the aforementioned settings. Therefore, our results can be used in practice to guide educators, researchers and teams on improving detection and training on code smells.

The remaining sections of this paper are structured as follows. Section II introduces basic terminology. Section III discusses related work. Section IV shows the experimental design. Section V presents results. Section VI presents potential threats to validity. Section VII shows our conclusions.

## II. BACKGROUND

Our study aims at evaluating the effectiveness of collaborative practices in the identification of code smells (Section II-A). To this end, we choose PP and CDR as representatives of pair-based and group-based practices, respectively (Section II-B). This section discusses these concepts underlying our study.

[1] Depending of the context, Duplicated Code can be classified as intra-class or inter-class [8]. In our study, we classified it as an intra-class instances.

### A. Code Smells

Code smells are often considered as key indicators of software degradation. They can be classified in two categories according to the granularity (or scope) of their structure [8]: intra-class and inter-class smells. Intra-class smells are anomalous structures that are located within a particular class and, therefore, their definitions refer to inner elements of a particular class. An example of intra-class smell is Long Method [8]. This code smell occurs whenever a method is long and excessively complex. On the other hand, inter-class smells are those that affect the structure of two or more classes. Therefore, their definitions refer to elements scattered in two or more classes. An example of inter-class smell is Feature Envy [8], which occurs whenever a method of a class is more interested in the methods of another class excessively. Table I presents the definitions (last column) and the classifications (second column) of smells selected for this study as defined by Fowler [8]. We selected all these types of code smells as we needed various representatives of both intra-class and inter-class smells for our study. Section IV-D presents further justification for the selection of each smell.

The identification of code smells is important because without this step it is not possible to know where refactoring actions are needed. Refactoring represents basic clean-up actions required to improve software longevity [8][15]. Unfortunately, several recent studies involving industry software projects have reported how hard the process of identifying code smells is due to various factors. First, there are several types of code smells catalogued [8][10]. Therefore, a single developer may not be able to effectively identify occurrences of all types of code smell even based on state-of-practice tool support [13]. Second, in spite of the extensive tool support for smell detection available nowadays (e.g. [13]), this task still has to be manually performed by developers. For example, the metrics and thresholds used in smell detection need to be tailored for each project or component of a system. In addition, after smell candidates are identified by an existing tool, developers still need to analyze each smell for confirmation purposes. Third, reasoning about a smell often requires a non-local comprehension of various program modules (e.g. classes or packages) somehow related to that smell. Therefore, such program structures may be better understood by multiple developers.

Thus, given the need for global reasoning, information exchange among two or more developers may help to better identify certain smells. The limited knowledge of a single developer may not suffice to fully detect the occurrence of a smell. Thus, the number of developers involved in smell identification may affect the task effectiveness. In particular, the use of collaborative practices may help developers to perform specific actions that contribute to improve effectiveness on the

identification of code smells.

### B. Collaborative Practices

Collaboration plays an important role in the context of acquiring skills in software engineering tasks, through exchanging information and sharing knowledge among developers. Thus, collaborative practices have been recently used to address limitations in software engineering tasks performed by individuals. In addition, software companies are increasingly adopting collaborative practices for improving the effectiveness of developers with limited or no knowledge in software engineering tasks, including code review tasks (e.g., [2][4]). Collaborative practices mainly differ in the way that two or more participants are allocated to the same software development activity. In this context, we selected Pair Programming and Coding Dojo Randori as representatives of collaborative practices, in which the former is a pair-based practice and the latter is a group-based practice.

**Pair Programming** (PP) consists of, as the name suggests, two developers working collaboratively on the same development-related activity, such as: designing an algorithm, editing the code, reviewing code or analyzing and testing a system [1]. In a PP session, one of the developers acts as the driver of the activity, controlling the keyboard and mouse. The other developer acts as the navigator (or observer), being responsible for supporting the target activity, such as code reviews. During the session, the pairs can switch the roles in specific time boxes. On the other hand, in case of purely analytical activities (e.g. code review), pairs act simultaneously on code analysis [6] - i.e. the two individuals act as observers.

**Coding Dojo** (CD) consists in a collaborative practice in which a group of developers working on the same development-related activity. There are several types of CD, and one of the variants is called Coding Dojo Randori (CDR) [20]. In a CDR session, a group of participants works together with the following dynamics: (i) one participant acts as the driver, (ii) another one acts the navigator, and (iii) the remaining participants act as the audience, which stays most of the time in silence, paying attention to the pairs. However, the entire audience is able to participate in a coordinated way when certain events occur; for instance, if the unit tests are passed [20]. The roles of individuals are changed in rounds. When a round ends, the driver moves to the audience, the navigator turns into the driver and someone of the audience starts to act as a navigator [20]. Every participant acts at least once as the driver and once as the navigator. In the context of our study, the CDR practice is also called as Group Programming (GP).

## III. RELATED WORK

Pair Programming (PP) is one of the best-documented and most popular collaborative practices, and it has been the subject of a considerable number of empirical studies [2][4] involving developers with different knowledge levels and professional experience. In order to identify the benefits of PP, Visaggio [21] conducted an experiment with a group of undergraduate students. The study assessed (i) productivity, (ii) reduction of coding errors, and (iii) knowledge transfer between students. The results pointed out the use of PP improve developer productivity and promotes the transference of knowledge between developers.

Unlike the studies involving the use of PP, only a few studies explore empirical evidence about Coding Dojo Randori (CDR). According to Rooksby et al. [18], there is a lack of studies that evaluate the CDR effectiveness in different tasks of software development. In addition, a few studies have made comparative analysis of use of different collaborative practices. For instance, the study conducted by Estácio et al. [7] evaluated the use of CDR and PP in the teaching of mockup developments. The authors concluded the use of PP presented positive results in learning, motivation and user experience between the students, while CDR showed good results only in the learning. However, these studies did not evaluate whether CDR supports improving the accuracy of the developers in the identification of code smells.

In our previous effort, we empirically evaluate CDR and PP practices, aiming to analyze the impact of different type of collaboration in relation to acquiring programming skills [6]. According to the results PP had good results in learning and motivation, whereas, CDR had good results in the learning of algorithm concepts. Furthermore, CDR presented some challenges on motivation and lower rate of code quality comparing to PP. These findings provide important directions in the research of the impact of different types of collaboration on the identification of code smells. To the best of our knowledge, the study reported in this paper was the first attempt addressing the gap of evaluating different collaborative practices in the context of identifying code smells.

## IV. STUDY SETTINGS

### A. Goal

The goal of our study is to investigate the use of collaborative practices on the identification of code smells. From this goal, we derived the following two research questions (RQs).

RQ1: *Do collaborative practices improve the effectiveness of novice developers on the identification of code smells when compared to the solo programming?*

One of the most important criteria for choosing a practice for identifying code smells is its effectiveness [14]. As mentioned in Section I, we consider a practice to be effective if the developers, using the practice, correctly identify more code smells within time constraints than when they use another practice. In order to address this research question, we analyzed both qualitative and quantitative data (Section V-C). The following null and alternative hypotheses were formulated for this RQ:

- **H0** The use of collaborative practices does not affect the effectiveness on the identification of code smells.
- **HA1.1** Novice developers using pair programming identify more code smells within time constraints than novice developers using solo programming.
- **HA1.2** Novice developers using group programming identify more code smells within time constraints than novice developers using solo programming.

RQ2: *How do novice developers individually and collaboratively perform the identification of code smells?*

During the study, we also investigated qualitatively the way that developers individually and collaboratively perform the identification of code smells. We collected the data through the answers provided in a post-experiment questionnaire, the screen-shots, the audio and video records. These data were transcribed and the transcriptions were validated by 1st and 3rd authors to eliminate misunderstandings. After that, we adopted a content analysis technique [17] to coding the transcription. Then, we generate our information network and reveal a set of features of our collaborative strategy (Section V-B). In addition, from this analysis, we transcript some segments of the collected data in Section V-A and Section V-C.

### B. Participants

Software developers begin a transition from novice to expert at least twice in their careers – once in their first year in university, and second when they about to start their first industry project [3]. Experienced developers are never available in the former, and might not be available in training courses in the second transition like projects run by companies, such as ThoughtWorks/Kazan [7]. Hence, we consider these two settings in our study.

The study was conducted with 28 novice developers. They were classified according to two levels of experience: (i) novice developers without experience in industrial software projects and (ii) novice developers who worked in at least one industrial software project. They were selected either from an undergraduate course in computer science or from an Agile Software Development course, called Software Kaizen [7]. This course provides undergraduate Computer Science students an immersion of four months in industrial software projects. In order to participate in the study, all the developers signed a consent form. They also filled out a characterization form with objective questions about their knowledge in the topics related to the study: (a) programming; (b) Java; (c) PP; and (d) GP (i.e., CDR).

We collected the data from the characterization form from each participant and ranked their knowledge into: none (N), low (L), medium (M), and high (H) for each expertise topic. For instance, regarding Java programming, the knowledge of the subject was characterized as: (a) None, if she never had contact with the Java; (b) Low, if she had contact with Java only in the classes or reading some support material; (c) Medium, if she had contact with Java only in the context of an academic project; or (d) High, if she had worked with Java in at least one industrial software project.

Table II summarizes the knowledge that participants claimed to have in each topic. In addition, we highlighted with grey-tone the participants that look like having richer ("higher") knowledge on the subjects. To be considered as having better knowledge on thus subjects, a novice developer must have medium knowledge in at least two of the four expertise topic of evaluation. In general, we have observed that: (i) 78.57% of the participants have medium to high knowledge in Programming Concepts and Java Programming; (ii) 89.28% of the participants have low to medium knowledge in PP and GP; and (iii) 14.28% of the participants have no knowledge in any of the four topics.

TABLE II. EXPERTISE PER PARTICIPANT IN EACH GROUP

| Novice developers who worked in at least one industrial software projects | | | | | |
|---|---|---|---|---|---|
| **Group** | **Participant** | **Programming** | **Java** | **PP** | **GP** |
| 1 | 1 | High | High | Medium | Medium |
| | 2 | High | High | Medium | Low |
| | 3 | High | High | Low | Low |
| | 4 | High | High | Low | Low |
| | 5 | High | High | Low | Low |
| | 6 | High | High | Medium | Medium |
| 2 | 7 | High | High | Low | Low |
| | 8 | High | High | Medium | Medium |
| | 9 | High | High | Low | Low |
| | 10 | High | High | Medium | Low |
| 3 | 11 | High | High | Medium | Low |
| | 12 | High | High | Low | Low |
| | 13 | High | High | Low | Low |
| | 14 | High | High | Medium | Medium |
| Novice developers without experience in industrial software projects | | | | | |
| Group | Participant | Programming | Java | PP | GP |
| 1 | 15 | Low | Low | Medium | None |
| | 16 | Medium | Medium | Low | Low |
| | 17 | Medium | Medium | Medium | Low |
| | 18 | Low | Low | Medium | None |
| | 19 | Medium | Low | Low | Low |
| | 20 | Low | Low | Medium | Low |
| 2 | 21 | Medium | Medium | Low | None |
| | 22 | Low | Low | Low | Low |
| | 23 | Medium | Medium | Medium | Low |
| | 24 | Medium | Medium | Low | Low |
| 3 | 25 | Medium | Medium | Low | Low |
| | 26 | Low | Low | Medium | None |
| | 27 | Medium | Medium | Medium | Low |
| | 28 | Medium | Medium | Low | Low |

After characterizing the participants' knowledge, we applied the principles of balancing, blocking and random assignment [22] to mitigate threats to validity concerning the distribution of participants between the groups. For balancing, we tried equally-sized groups. However, we had to create one of the groups with 6 participants because we aimed to evaluate the interaction of developers within a large group programming session. Concerning blocking, we avoided that one team had more participants with more knowledge than the other. We performed it in order to avoid biased results of a team performing better in the assigned tasks. Finally, participants of similar experience were randomly assigned to the groups.

### C. Experiment Design

We planned our experiment design to include one factor with three treatments: (i) solo programming, (ii) pair programming (PP), and (iii) group programming (GP). We also included three tasks: identifying code smells in three different programs with several code smells. We adopted a crossed design in order to enable all the treatments to be similarly applied to all the tasks. This helped to mitigate threats to validity of our experiment concerning: (i) the influence of the learning curve over the results, and (ii) the fact that one task could favor a specific treatment. It is noteworthy that the principles applied to distribute the participants between the groups in the crossed design still enable comparing the results for each individual exercise. The crossed design is shown in Table III.

Concerning the configuration of the crossed design, in the

TABLE III. CONFIGURATION OF THE CROSSED DESIGN

| | Program A -step one- | Program B -step two- | Program C -step three- |
|---|---|---|---|
| Solo Programming | Participants 1 to 6 | Participants 7 to 10 | Participants 11 to 14 |
| | 15 to 20 | 21 to 24 | 25 to 28 |
| Pair Programming | Participants 11 and 12 13 and 14 | Participants 1 and 2 3 and 4 5 and 6 | Participants 7 and 8 9 and 10 |
| | 25 and 26 27 and 28 | 15 and 16 17 and 18 19 and 20 | 21 and 22 23 and 24 |
| Group Programming | Participants 7 to 10 | Participants 11 to 14 | Participants 1 to 6 |
| | 21 to 24 | 25 a 28 | 15 to 20 |

first step of the experiment, we can verify, for instance, that participants from 1 to 6 analyzed program A with the solo programming treatment. Then in the second step, these same participants analyzed the program B with the PP treatment. Finally, in the third step they analyzed the program C with the group programming treatment. Figure 1 shows how the phases and activities of the experiment were organized.

**Activity 1: To apply the questionnaire for participants' characterization**: As previously mentioned in Section IV-B, a questionnaire was designed in order to gather participant's information on several aspects. For example, it involved questions about programming languages already used by them, their affinity with programming and their notion about the concepts of pair programming and coding dojo. The responses obtained through this questionnaire allowed us to identify some key characteristics of each participant.

**Activity 2: To train the participants**: This step aimed to train participants through a presentation that explained the various concepts, along with examples. Specially, the presentation addressed concepts associated with the following topics: (i) Pair Programming, (ii) Coding Dojo, and (iii) Code Smells. As far as code smells are concerned, we also made their particular definitions explicit, introduced examples and discussed the relevance of identifying the smells selected for our experiment. The first author performed the same training for both samples, i.e. novice developers with no experience in industry and novice developers with experience in industry. The training was organized in 2 parts: the first aimed the explanation of the concepts and took 25 minutes and the second aimed to provide room for discussion about the concepts and lasted 10 minutes.

**Activity 3: To execute the identification tasks**: Participants identified the occurrences of seven (7) different types of smells in three different programs (they will be explained in Section IV-D) according to the configuration of the presented crossed design (Table III). The data related to the identification of code smells were transcribed into a list. The formation of the pairs in PP sessions within each group was similar to how it was performed by Braught et al. [4], pairing an experienced participant with a less experienced one. In the group programming session, the sequence of the pairs was determined by the convenience of the participants. The study was conducted online, i.e. simultaneously and observed by researchers. The duration of each step was one hour for each group. They worked on their tasks simultaneously in different

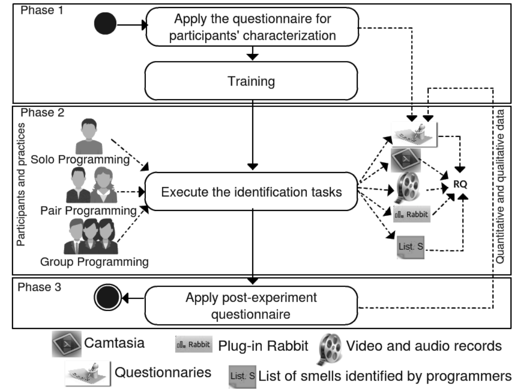


Fig. 1. The experimental design

rooms. We also performed a qualitative analysis through the logs generated by the plug-in Rabbit[2]. The screenshots came through Camtasia[3], and the audio and video records form our set up environment. This setting helped us to explore how the participants make decisions on confirmation of the smells when they work individually and collaboratively.

**Activity 4: To apply post-experiment questionnaire**: The participants were asked about their general perception on using the individual and collaborative practices for the smell detection tasks. The questionnaire also asked participants to particularly assess the usefulness of each practice to identify smells. These questions were made in order to enable us to reveal a set of features for an idealized collaborative strategy. This idealized collaborative strategy is intended to improve effectiveness on the collaborative identification of smell (Section V-B). Also, the questions were made in order to extract complementary qualitative data concerning our research question.

### D. Smell Reference List

We used the same procedure and materials in each group (Section IV-B). The experiment involved the identification of code smells using solo programming and collaborative practices. We applied these techniques on different programs that contained seven types of code smells, classified in two categories: inter-class and intra-class smells (Section II-A). We selected these smells because they cover both categories, and previous studies have reported them as the most frequent types of code smells in software projects [8][23]. For instance, Yamashita and Moonen [23] reported that duplicate code is the smell most frequently mentioned by developers in software projects. Other smells also often mentioned by developers is Long Method [23]. An additional reason for the selection of these smells is that they affect different program structures, such as instructions, attributes, methods and classes [8]. Consequently, they may impose different levels of difficulty to the identification process. Finally, that these smells require the analysis of different characteristics of a program, such as size, coupling, cohesion, complexity and many others [8].

[2]In: https://marketplace.eclipse.org/content/rabbit
[3]In: https://www.techsmith.com/camtasia.html

TABLE IV. AVERAGE AMOUNT OF CODE SMELLS IDENTIFIED PER HOUR

| | Program A | Program B | Program C |
|---|---|---|---|
| Solo Programming | 3.33 | 1.38 | 6.88 |
| Pair Programming | 7.25 | 5 | 14 |
| Group Programming | 6 | 3 | 5 |

Prior to the beginning of the experiment, we defined an oracle consisting of a reference list of code smells. This reference list can be found on our website [16]. We selected three programs existing in our repository of software projects. The choice of these programs was based on the fact that they had similar types of smells in them. In addition, the oracle's smells present in those programs were detected, inspected and evaluated by applying two complementary strategies for the identification of code smells: (i) manual inspection, and (ii) inspection with the aid of a tool for semi-automatic identification of smells. The result of the intersection of these two strategies determined the similarity in the list of existing smells in those three programs, setting the oracle required for each program.

## V. RESULTS

### A. Collaborative Effectiveness

The average code smell identification effectiveness (number of code smells found) per hour by each treatment per program is shown in Table IV. To test the hypotheses defined in Section IV-A statistical analyses were carried out with support of the R tool. Given our small samples for each exercise, we avoided assumptions on the data (e.g., normal distribution and homocedasticity) and decided to apply non-parametric statistical tests. Since the experiment has three treatments, a first step was applying Kruskal Wallis (K-W) to identify if there were significant differences between the samples. Thereafter we used Mann-Whitney as a post-hoc test comparing the groups pairwise according to the hypotheses to be tested. To handle the fact that we had three possible pairwise comparisons, the very conservative Bonferroni correction was applied when using the obtained p-values aiming at confirming our hypotheses.

Concerning the overall number of code smells identified per treatments, K-W clearly showed differences among the groups for all three programs (p-values 0.0006, 0.0064 and 0.0009 for programs A, B and C, respectively), which encouraged us to conduct post-hoc tests to discover if the differences were related to the assumptions described in our alternative hypotheses.

Regarding **Hypothesis HA1.1**, novice developers using PP found a significantly higher amount of code smells in all three programs (p-values 0.0001, 0.0015, 0.0059). Given these p-values, even with the Bonferroni correction (dividing the alpha value by 3 - the number of possible pairwise comparisons - before comparing it to the p-values), this hypothesis can be confirmed with significance level of 5% for all three programs.

With respect to **Hypothesis HA1.2**, novice developers using GP found a significantly higher amount of code smells for two of the three programs (p-values 0.0004, 0.0029, 0.8520). Given these p-values, this hypothesis can be confirmed with (Bonferroni corrected) significance level of 5% only for programs A and B. Actually, for program C the average number of code smells identified per hour by solo programmers was higher. It is noteworthy that in program C the GP treatment had more participants than for programs A and B. Hence, it seems that a large amount of participants in a group does not contribute to code smell identification. In fact, our qualitative analysis indicated that the process for converging opinions can be time consuming. However, this could also be due to particularities of the program C, which reinforces the value of our experiment design with independent trials on different programs.

Drilling the analysis down to the types of code smells it was possible to observe that the average number of identified code smells was higher when the participants were working collaboratively than when working individually for nearly all types. Figure 2 presents the average number of smells identified within time constraints by the participants in three different programs. The light gray bars represent the results of solo programming, the black bars represent the results of pairs, and dark gray bars represent the results of groups. The values were calculated based on the number of identified code smells divided by average hours spent during the identification task. This figure represents only the results in which the average of identified code smells was equal or higher than 1.0 for at least one of the practices.

Additionally, we also performed the same analysis in each one of the samples individually, i.e., novice developers with and without experience in industrial software projects. Interestingly, we verified that the developers working collaboratively were better than when they were working individually in both samples. This result is more significant in the sample related to the more experienced novices, i.e. novices who have experience in an industrial software project.

We observed that developers working collaboratively clearly achieved more success in the identification of inter-class smells. As we explained in section II-A, inter-class smells are those that affect the structure of at least one class, and their definitions refer to elements scattered in two or more classes, such as Feature Envy, and Lazy Class. For several inter-class smells, the average of identified code smells by pairs or groups were over 40% higher than the corresponding average by individuals. We could confirm through our qualitative analysis that this superiority is due to the inherent complexity of the smells that involve several classes or methods, which require more knowledge to grasp the anomalous structure. We observed that each developer has a peculiar way to identify inter-class smells. During our experiment, the opinion of only one developer was not enough to identify inter-class smells precisely. When developers worked together, they were more precise in the identification of inter-class smells by considering each other's opinions. This observation is reinforced by the opinion of the developers: 89.28% of developers reported that they were more confident on the identification of inter-class smells when they were working collaboratively than when they were working alone.

We also observed that, for some of these smells (e.g. Feature Envy and Lazy Class), when the developers were working in pairs, they identified a higher number than when they were working in groups. We can attribute this finding to

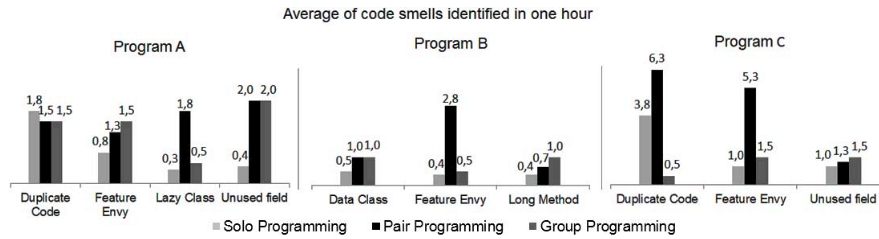


Fig. 2. Comparison between practices

at least two reasons. First, the use of groups in the detection of code smells may be ineffective for more complex code smells due to the existence of differing opinions on the smell validation activity (Section V-B will further elaborate on this matter). Second, converging opinions can be costly in terms of time. The several developers of a group have to share their time. The larger the group, the less average time per person is available and the fewer opportunities each member will likely have to contribute to the discussion. This observation is reinforced by the opinion of the participants in our post-experiment questionnaire. For example, participant P4 said: *"When we had many contradicting views regarding a code smell, the work in group may lead to considerable difficulties in the decision-making process"*.

Moreover, we observed that whenever developers work in collaboration, they avoid making mistakes (i.e., false positives in the reported list of code smells). Furthermore, we found that they follow a less optimistic approach on guessing where the smells are located in the program. They also tend to define together more elaborate detection strategies than when they work individually. For example, a participant (P1) marked a piece of a method structure as hosting a Feature Envy smell. However, the other participant (P2) in the pair did not agree with the decision. According to the second participant, *"...that part of the source code does not have a Feature Envy because the suspect method (marked as containing the code smell) is not calling other methods several times"*. In order to show that method does not contain a Feature Envy, he asked to the first participant to run through the classes thereby explaining to him why the other classes are not contributing to the code smell. After that, both participants agreed that there was no Feature Envy in that method after all. This collaborative behavior avoided developers to report false positives. Therefore, the collaborative analysis of the source code allows developers to discuss and share ideas about the severity of an anomalous structure, thus complementing the knowledge of each other. Consequently, this behavior of pair developers tends to increase the rate of success in identifying smells.

### B. Individual vs. Collaborative Strategies

In order to address our second research question (Section IV-A), we qualitatively analyzed how developers worked individually and collaboratively during the identification of code smells. This analysis also allowed us to infer which activities performed by developers contributed (or not) to the effective identification of code smells.

Figure 3 summarizes the results of such qualitative analyses based on a feature model notation [12]. The figure shows all the activities performed by developers, either individually or collaboratively. Each feature (rectangle) in Figure 3 represents a specific activity and a different number is attached to it. These numbers are mentioned in parentheses throughout this section in order to refer to specific activities represented in Figure 3. The style of the rectangle border indicates whether the activity was performed individually or collaboratively, either in pairs or in groups (see the legend). Certain activities were performed by all subjects, i.e. individuals, pairs or groups. These omnipresent activities are represented by rectangles with thick borders. Figure 3 only represents an activity whenever it was performed by the majority of the subjects who used the corresponding practice(s) expressed in that feature. We only represent such activities as we want to capture those activities that consistently contributed to improve or decrease the effectiveness on code smell identification. By 'majority' we mean two-thirds (66.67%) of the subjects. For instance, more than 80% of the subjects working individually used a checklist of examples (activity 4.2 in Figure 3) in order to detect code smells. The specific percentage of subjects performing each activity is not shown in in Figure 3, but some percentages are mentioned throughout this section.

Figure 3 shows that all subjects, independently from the employed practice, have performed six mandatory activities for smell identification. These general activities are named phases and they are presented by the top-level features in Figure 3. These six phases are: (1) Smell selection, (2) Metrics selection, (3) Navigation through the program classes, (4) Detection of smell candidate, (5) Smell validation, and (6) Decision making. The developers pass over sequentially in each phase. In the context of each phase, alternative activities may exist. They are represented in Figure 3 through OR and XOR relations based on the feature model notation. For instance, the subjects perform the smell selection (phase 1) using one of the two approaches: randomly (1.1) or a knowledge-based (1.2) approach. Two or more alternatives (sub-activities) may be used together in order to realize a more complex activity. For example, pairs or groups in phase 5 may carry out two or three sub-activities (5.1, 5,2 or 5.3) in order to realize the validation of smell candidates. For each sub-activity, Figure 3 indicates whether it contributes to improve (+) or decrease (-) the effectiveness of the smell identification task. We now describe and discuss each phase individually in the following paragraphs.

The **first phase** corresponds to the selection of types of code smells to be identified in the source code. During the experiment, the developers performed the selection of code smells using random selection (1.1) or a knowledge-based (1.2) approach. A random selection occurs when the developers

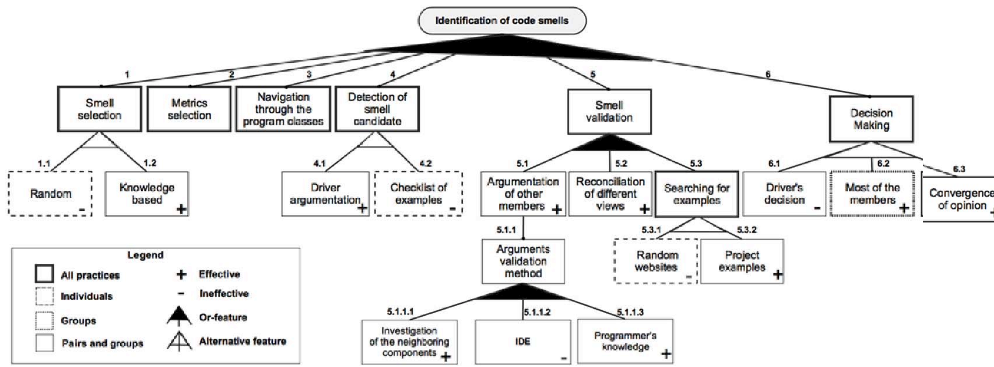


Fig. 3. Collaborative Strategy

simply follow the list of smells presented during the training activity (Section IV-C), without determining a less obvious but important criterion. The knowledge-based selection occurs when the developers select types of code smells in a particular order. In this selection approach, they were encouraged to use their previous experience and knowledge to select the smell types. We observed that the random selection was used by 71.4% of developers when they were working individually. This demonstrates that developers working individually often do not prioritize wisely the types of code smells to be identified. Furthermore, by observing the questionnaire answers and the videos, we concluded the random selection is less effective than the knowledge-based selection. There are three key reasons that explain this phenomenon: (i) the developers spend too much time searching for smells which they do not have much knowledge, consequently, (ii) they feel discouraged to perform the smell identification, and (iii) they start to work on identifying relevant smells too late. We also found that 78.5% of pairs and 66.67% of the groups implemented the knowledge-based selection. Therefore, pair programming encouraged more the use of knowledge-based selection than group programming.

In the **second** and **third phases**, we did not observe recurring alternative approaches used by pairs or groups that contributed to improve their effectiveness. Still, we make some observations on how they used collaborative practices to minimize (or not) the burden of the corresponding activities in those phases. The second phase concerns the selection of metrics or indicators. These metrics or indicators are used as criteria to support the identification of code smells of a particular type. For instance, some of the subjects used the metric lines of code (LOC) for supporting the detection of occurrences of certain smell types (e.g. Long Method and Lazy Class). We observed that developers worked collaboratively to minimize the burden of certain aspects of this task. For instance, they benefited of the knowledge of their peers to choose more wisely the appropriate metrics for each smell type. Pairs and groups also systematically defined how to interpret the results of each metric when identifying a particular smell type. They also identified and discussed potential false positives generated by the use of particular metrics. Finally, the teams covered a higher number of metrics than individuals.

The **third phase** corresponds to the way that developers run through the modules (e.g. classes and methods) during the code review process. We found that most developers, regardless of working individually or collaboratively, navigate through the modules sequentially: one by one. We observed that either individual or teams failed to determine a prioritization approach on how to navigate through program modules effectively. In the feedback form, the subjects mentioned that this activity could be more effective if all group members could navigate through classes independently. Group programming requires that only two members effectively work on the code analysis.

The **fourth phase** comprises the approach used by developers to perform the detection of code smell candidates. These candidates represent code smells that have not been confirmed (validated) by the individual or team yet. In this phase, we noticed two main approaches used by developers to support the activities: driver argumentation (4.1) and checklist of examples (4.2). Individuals often use the latter, while pairs or groups often employ the former. When developers worked collaboratively, they selected one smell candidate and, then, the driver argued about it thoughtfully with the other members of the team. They also exposed their certainties and uncertainties about each smell candidate. When the team members were mostly uncertainly about it, they ended up not confirming a particular candidate as an actual smell. According to the subjects' feedback in the questionnaire, the fact that collaborative practices (either PP or Group programming) impose the selection of a driver contributed to further improve the activity effectiveness. According to them, the presence of a driver helped to streamline the discussions. On the other hand, when developers were working individually, they tended to just check the existence of code smells comparing with a checklist of examples. These examples are randomly selected by each individual based on his personal experience and clearly influenced the occurrence of several false positives. The reason is that the examples often had no semantic equivalence to the smell candidates in the experiment.

The **fifth phase** is associated with the smell validation activity. This activity consists of confirming or discarding code smell candidates identified in the fourth phase. When this activity is carried out collaboratively, the other members often require arguments in favor or against to a viewpoint expressed by one of the team members (5.1). This process contributes to the confirmation or refutation of a code smell candidate.

Each developer uses one or more methods to elaborate their arguments: (i) the driver runs through neighboring modules of the project (5.1.1.1) in order to better understand the context of a particular smell candidate, (ii) using various IDE features (5.1.1.2), and (iii) developer's knowledge (5.1.1.3) gathered in previous projects. Amongst the three argumentation methods, we found that only the use of IDE features did not contribute to improve the subjects' effectiveness, except for the Unused Field smell. In the post-experiment questionnaire, one of the participants (P23) made the following comment *"... the IDE helped me find only smell Unused Field"*.

Contradictory opinions might occur once all the arguments are brought to the discussion with respect to the confirmation of a smell candidate. Then, there was a need for an agreement among developers (5.2) and drivers needed to reconcile those different views. Another form to validate the existence of the code smell is through queries of examples in the project or in the Web (5.3). This form was used by most developers when working individually. We found that such random queries are often conducted (5.3.1) in general-purpose search engines (e.g., Google). On the other hand, the developers when working collaboratively often performed their queries of examples in the project under analysis (5.3.2). Most of the solo developers who adopted random selection did not perform the correct identification of code smells. This fact was observed from the comparison of the screenshots and the list of smells identified by programmers. The random selection on websites did not contribute to the success rate (i.e., correctly identify code smells) mainly because the examples were often different from the context in which the classes and methods were represented in the experiment design. Therefore, we noticed that the returned examples were not helpful to support solo developers in correctly validating their smell candidates.

The **sixth phase** involves the decision-making process to make a final verdict about each code smell. When developers work collaboratively, they can make decisions based on the Driver's decision (6.1). However, this activity is usually not effective as one of the developers can see it as a sign of authoritarianism. On the other hand, activities based on the opinion of the majority of the members (6.2) and according to the agreement amongst developers (6.3) were usually more effective. These activities were more effective because developers, using them, solved teams' uncertainties in a smoother way.

Given the aforementioned analyses of certain collaborative activities, the strategy (summarized in Figure 3) can be used in practice to guide: (i) either teams or educators on improving training on code smells, and (ii) researchers that need to investigate how to improve support for (collaborative) identification of code smells.

### *C. Overall Discussion*

After answering our two research questions, we realized that identification of inter-class code smells (Section V-A) tends to be an inherently social activity. Developers naturally need to make several decisions (Section V-B) in order to effectively detect these more complex code smells. Moreover, the process of defining the detection strategies, such as, selecting metrics and thresholds, is inherently a social task, which emerges from the opinion of every team member. Teamwork was important to select proper metrics and thresholds. For example, in order to validate a Long Method, the participants used different thresholds for the number of lines of code (LOC) when employing each practice. For instance, when the participant P18 was working individually, he considered LOC $> 80$ for Program B as threshold. This choice was decisive to result in several false positives. However, when participants P19 and P20 were working in pairs they considered as threshold a LOC $> 150$ for Program B, which resulted in no false positive. We also observed that the smell candidates suggested by a single individual tend to be rejected by another developers, even if the detection strategy was somehow defined and evaluated in their previous experiences [11].

Another fact that illustrates the importance of collaboration concerns the validation of smell candidates. This activity was clearly the one that benefited the most from collaborative practices. A scenario involving Data Class instances help to illustrate this point. For instance, when the participant P6 has worked alone, he only searched the Internet to decide about actual occurrences of Data Class. On the other hand, before deciding about the Data Class, the participant P25, working in a group, used other classes in the project to convince the other participants of the group about the existence of the Data Class. To convince the group, P25 said, *"...this class is a Data Class because it contains only getters and setters (methods)"*. Moreover, the participant P28 added, *"... this class contains data that is only used by other classes"*. In this case, P28 had to run among the other classes before making the comment. Using both comments, the group marked the class as a Data Class correctly. Therefore, the developers obtained more easily the knowledge to reason about code smells when they worked collaboratively than when they worked individually. Consequently, they spent less effort to perform the smell identification with higher accuracy.

## VI. THREATS TO VALIDITY

**Construct validity**: Regarding the experimental planning, we considered the following threats: (i) the set of analyzed code smells, (ii) the generation of the smell reference list, and (iii) the absence of a static analysis tools. Regarding the first threat, we tried to mitigate it by using systems that suffered from the same set of code smells. Thus, we restricted our research to discussing only seven types of code smells. To mitigate the second threat, a smell reference list had to be created in order to support the experimental task and to provide means to calculate the number of identified code smells, while doing so, several precautions were taken (explained in Section IV-D). Regarding the absence of a static analysis tools, we highlight that our goal was not to investigate the impact of a specific tool on smell identification tasks; instead, our goal was to investigate how the developers collaborative identify code smells. Nevertheless, if we had used a tool during the experiment, the results could be completely dependent on the intricacies of this particular tool. In other words, the use of a tool would introduce significant bias to the experiment.

**Internal validity**: We considered the following threats to the internal validity: (i) communication among the participants, (ii) different knowledge levels among the participants, and (iii) differences among experimental tasks. To mitigate the first

threat, we made an effort to minimize communication among the participants. Moreover, we explained the experimental tasks to avoid misguidance of the participants. Concerning the second threat, we applied the design principles of balancing, blocking and random assignment, as suggested by Wohlin et al. [22]. Finally, regarding the third threat, we applied the control action of using a crossed design, in which independent groups applied all treatments to all tasks, leading us to three independent trials. This design also helped to isolate the learning effect, given that each group applied the practices in a different sequence.

**External validity**: Our study interviewed a small sample of developers and was conducted on a specific set of programs, which represent threats to the external validity. Therefore, we cannot generalize our conclusions. The sample was comprised by 28 novice developers who worked or not in real projects in the industry. The data extracted from this study presents important results related to identification of code smells. But, as an initial study on this subject, we do not raise any external validity claims and ask for replications to allow further generalizing the results.

**Conclusion validity**: This threats concerns the confidence in the relation between the treatment and the outcome. Regarding the quantitative analyses, we tested our hypotheses using non-parametric tests. This decision was taken considering our limited amount of data points and to avoid violating assumptions on their distribution. Concerning the qualitative analyses, we tried improve the conclusion validity conducting our analyses based on different sources of qualitative data obtained from Camtasia, videos, and questionnaires.

## VII. Conclusion

In this paper, we compared PP and CDR with solo programming in order to better distinguish their impact on the effective identification of code smells. In fact, our study is the first in the literature to observe how novice developers work individually and together to identify smells. Our results suggest that collaborative practices increase the effectiveness on the identification of inter-class code smells. Teams and educators can now focus on exploring the use of collaborative practices only for these smells. In fact, for nearly all types of inter-class smells, the average of smells identified by novice pairs or groups outperformed at least in 40% of the corresponding average of smells identified by individuals. Novices often do not question the possibility of being wrong when working in isolation. As a consequence, they leave various (inter-class) code smells go unnoticed. The use of collaborative practices seems to reduce this kind of undesirable behavior. Thus, collaborative practices tend to increase the rate of success in identifying more complex smells.

We also identified a list of activities performed by pairs or groups that contributed to improve the effectiveness on code smell identification. In general, our results suggest that novice developers are not yet properly equipped with practices to conduct effective smell identification activities. Hence, the collaborative strategy, which was identified in our study, can be used in practice to guide educators, researchers or teams on improving detection and training on code smell. Future steps in this work involve the planning and execution of new empirical studies in order to refine elements of the collaborative strategy while supporting novices in the smell identification process.

## Acknowledgment

This work is funded by CAPES/PGCI (No. 060/15), CAPES/Procad (grant # 175956), CNPq (grants # 483425/2013-3, 309884/2012-8 and 312127/2015-4), FAPERJ (E26-102.166/2013) and the research agreement between PUCRS and ThoughtWorks.

## References

[1] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004.

[2] A. Begel and N. Nagappan. Pair programming: what's in it for me? In *ESEM '08*, pages 120–128, 2008.

[3] A. Begel and B. Simon. Novice software developers, all over again. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 3–14, New York, NY, USA, 2008.

[4] G. Braught, J. MacCormick, and T. Wahls. The benefits of pairing by ability. In *In. SIGCSE '10*, pages 249–253, 2010.

[5] S. Bryton, F. Brito, Abreu, and M. P. Monteiro. Reducing subjectivity in code smells detection: Experimenting with the long method. In *QUATIC'10*, pages 337–342, 2010.

[6] B. Estacio, R. Oliveira, S. Marczak, M. Kalinowski, A. Garcia, R. Prikladnicki, and C. Lucena. Evaluating collaborative practices in acquiring programming skills: Findings of a controlled experiment. In *SBES'15*.

[7] B. Estácio, R. Prikladnicki, M. da Costa Móra, G. Notari, P. Caroli, and A. Olchik. Software kaizen: Using agile to form high-perfomance software development teams. In *AGILE'08*, pages 1–10, 2014.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[9] F. Hermans and E. Aivaloglou. Do code smells hamper novice programming: A controlled experiment on scratch programs? In *ICPC '16*, Austin, Texas, USA, 2016.

[10] M. V. Mantyla, J. Vanhanen, and C. Lassenius. Bad smells - humans as code critics. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 399–408, 2004.

[11] R. Marinescu. Measurement and quality in objectoriented design. Technical report, 2005.

[12] K. Michael, J. Prashant, C. Angelo, and L. David. Distributed extreme programming. In *International conference on eXtreme Programming and Agile Processes in Software Engineering*, pages 66–71, 2001.

[13] E. Murphy-Hill and A. P. Black. Seven habits of a highly effective smell detector. In *RSSE'08*, pages 36–40, New York, NY, USA, 2008.

[14] E. R. Murphy-Hill and A. P. Black. Why don't people use refactoring tools? In *WRT*, pages 60–61, 2007.

[15] W. F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[16] Open-Archives. Website. In *http://migre.me/ufioB*, 2016.

[17] M. Philipp. Qualitative content analysis. *FQS'00*, 2000.

[18] J. Rooksby, J. Hunt, and X. Wang. *Agile Processes in Software Engineering and Extreme Programming*, chapter The Theory and Practice of Randori Coding Dojos, pages 251–259. Springer International, 2014.

[19] N. Salleh, E. Mendes, and J. C. Grundy. Empirical studies of pair programming for cs/se teaching in higher education: A systematic literature review. *IEEE Trans. Software Eng.*, 37:509–525, 2011.

[20] D. Sato, H. Corbucci, and M. Bravo. Coding dojo: An environment for learning and sharing agile practices. In *Agile, 2008. AGILE '08. Conference*, pages 459–464, 2008.

[21] C. Visaggio. Empirical validation of pair programming. page 654, 2005.

[22] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2012.

[23] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251, 2013.