Universidade Federal de Alagoas
Instituto de Computação
Programa de Pós-Graduação em Informática

Dissertação de Mestrado

On the agreement among developers when detecting code smells aided by decision tree

Christiano Rossini Martins Costa
christianorossini@ic.ufal.br

Orientador:

Baldoino Fonseca dos Santos Neto

Maceió, abril de 2020

Christiano Rossini Martins Costa

On the agreement among developers when detecting code smells aided by decision tree

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Curso de Mestrado em Informática do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Baldoino Fonseca dos Santos Neto

Maceió, abril de 2020

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

C837o Costa, Christiano Rossini Martins.
On the agreement among developers when detecting code smells aided by decision tree / Christiano Rossini Martins Costa. – 2020.
59 f. : il.

Orientador: Baldoino Fonseca dos Santos Neto.
Dissertação (mestrado em Informática) - Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2020.

Texto em inglês.
Bibliografia: f. 52-54.
Apêndices: f. 56-59.

1. *Code smells*. 2. Concordâncias. 3. Sistemas de suporte de decisão. I. Título.

CDU: 004.424.25

Folha de Aprovação

Christiano Rossini Martins Costa

"On the agreement among developers when detecting code smells aided by decision tree"

Dissertação submetida ao corpo docente do Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas e aprovada em 06 de abril de 2020.

Banca Examinadora:

[illegible]

Prof. Dr. Baldoino Fonseca dos Santos Neto
UFAL – Instituto de Computação
Orientador

[illegible]

Prof. Dr. Márcio de Medeiros Ribeiro
UFAL – Instituto de Computação
Examinador Interno

Rafael M. de Mello

Prof. Dr. Rafael Maiani de Mello
CEFET – RJ
Examinador Externo

To my parents, who gave me all support to overcome life's challenges, despite all hardnesses.
Now here I am. Thank you, for everything.

Acknowledgements

I would like to express my gratitude to my family for all the support and encouragement throughout this journey.

To the members of the examining board, Prof. Márcio de Medeiros Ribeiro and Prof. Rafael Maiani de Mello, for kindly accepted the invitation to participate and collaborate with this dissertation.

To the research group EASY and my classmates for the uncountable kindnesses and the companionship that guided my entire trajectory in this difficult challenge.

Especially, I'd like to thank my advisor Prof. Baldoino Fonseca dos Santos Neto for all his patience, support and constructive criticism during this period. This work, as in every major challenge in life, was made up of good and bad times. But it was the bad times that made me understand how important a good orientation work is in the life of a student.

Resumo

O *code smell* é um sintoma de um mau design de código em desenvolvimento de software. Geralmente ocorre quando os desenvolvedores conceberam mal o design de um componente de código ou porque não projetaram a solução adequadamente devido a prazos rígidos. Na literatura, o *code smell* é descrito informalmente, isto é, não é possível identificá-lo objetivamente. Essa informalidade pode levar dois ou mais desenvolvedores a raciocinar sobre cada ocorrência de *smell* a sua maneira. Como conseqüência de diferentes pontos de vista, percepções conflitantes nas mesmas bases de código podem ser notáveis, afetando a consistência entre as revisões de código. Trabalhos anteriores que abordam a subjetividade do *code smell* carecem de elementos informativos que possam descrever por que certos códigos-fonte foram anteriormente classificados como *code smell*, a fim de ajudar os desenvolvedores a raciocinar sobre a ocorrência de um *code smell* com mais eficácia. Em nossa pesquisa, propomos mostrar ao desenvolvedor uma visualização de um classificador de árvore de decisão, composto por regras baseadas em métricas de software, com o objetivo de informar as razões pelas quais algum código-fonte foi classificado anteriormente como um *host* de um *code smell*. O fornecimento de novos *insights* pode levar o desenvolvedor a raciocinar de forma mais ampla sobre a ocorrência de um *code smell*, não se restringindo a fatores relacionados apenas a experiências e vida profissional. Nosso objetivo é, após várias avaliações de *code smells*, investigar como a concordância entre desenvolvedores pode ser influenciada pela visualização fornecida por um modelo de classificação compreensível, o classificador de árvore de decisão. Realizamos um experimento on-line onde reunimos colaborações de 30 desenvolvedores da indústria e da academia. Os resultados indicam que: (i) a detecção de *code smells* auxiliada por uma árvore de decisão leva a uma melhora relativa da concordância em relação às detecções com base apenas na análise de código; (ii) a detecção de *code smell* auxiliada pela árvore de decisão não diminui o esforço para detectar *smells* (iii) nosso experimento sugere que as árvores de decisão usadas para dar suporte à detecção de *code smells* são úteis para o desenvolvedor em termos de *insights* para tomada de decisão.

Palavras-chave: Odores de código. Concordâncias. Revisão de código. Árvore de Decisão. Estudo empírico.

Abstract

Code smell is a symptom of poor code design on software development. It often occurs when developers poorly conceived the design of the code component or because they did not properly designed the solution due to strict deadlines. In literature, code smell is described informally, i. e., it isn't possible to identify a smelly code objectively. Such informality may lead two or more developers to reason about each smell occurrence in their own way. As a consequence of different viewpoints, conflicting perceptions on the same code bases may be notable, impacting the consistency across code reviews. Previous works that addresses code smell subjectivity lacks of some informative element that may describe why certain suspicions source code was previously classified as code smell in order to aid developers to reasoning about the occurrence of a code smell with more effectivity. In our research, we propose to show to the developer a visualization of a decision tree classifier, composed by some metric-based rules, aimed to inform the developer the reasons why some source code was previously classified as a host of a certain code smell. Providing new insights may tease the developer to reasoning the occurrence of a code smell widely, not restricting to factors concerned only to past experiences and backgrounds. Our objective is, after several code smell evaluations, investigate how the agreements among developers may be influenced by the visualization provided by a comprehensible classification model, the decision tree classifier. We performed an on line experiment where we gather collaborations from 30 developers from industry and academy. The results indicate: (i) code smells detection aided by a decision tree leads to a relative improvement of agreement in relation to detections based solely on code analysis; (ii) the detection of code smell aided by decision tree do not decrease the effort to detect smells (iii) our experiment suggests that the decision trees used to support code smell detection are useful to the developer in terms of insights for decision making.

Keywords: Code Smell. Agreements. Code Review. Decision Tree Classifier. Detection. Identification. Empirical study.

List of Figures

List of Tables

Contents

1 Introduction

1.1 Context and Problem

Code smell is a symptom of poor code design on software development (FOWLER et al., 1999). It often occurs when developers poorly conceive the design of the code component or because they did not properly design the solution due to strict deadlines (PALOMBA et al., 2014). Empirical studies indicate that code smells leads to a lack of code comprehensibility (ABBES et al., 2011) and increases change and fault-proneness, affecting negatively the software maintainability (YAMASHITA; MOONEN, 2013b). Refactoring (FOWLER et al., 1999) is a useful technique that relies on fixing software code that hosts a smell through applying the best Oriented Object practices as long as keeping the core business logic untouched.

Fowler and Beck (FOWLER et al., 1999) describes each type of code smell informally, i. e., they do not provide formal rules for detecting code smells. Such informal definition of code smell may lead two or more developers to reason about each smell occurrence in their own way, subjectively. For example, from Long Method emerges some questions such as (i) When a method is actually long? (ii) How many responsibilities characterize a long method? (iii) What is the threshold of source code lines that characterize a long method? These questions may result in divergent opinions, depending on each developer's experiences and backgrounds (HOZANO et al., 2018). The subjectivity issue turns more relevant when we assume that a considerably large proportion of software professionals (32%) states that they did not know about code smells (YAMASHITA; MOONEN, 2013a). As a consequence of different viewpoints, conflicting perceptions on the same code bases may be notable, impacting the consistency across code reviews (HOZANO et al., 2018).

Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for reducing software defects and improving the quality of software projects (BACCHELLI; BIRD, 2013). Although the availability of tools (FONTANA; BRAIONE; ZANONI, 2012) and machine learning techniques (AZEEM et al., 2019) for smell detection, code reviewers still need to analyze code design problems individually to confirm its occurrence on the system. For instance, in pull-based development (GOUSIOS et al., 2015), a distributed software development that allows developers to work on a software project even being geographically dispersed, there is a reviewer staff of core developers that is responsible to accept or reject pushed changes to the repository. In the reviewing process, most of the rejected changes refer to design problems, including architecture violations, use of bad programming practices, violations of good object-oriented design principles, lack of abstraction and poor implementation (SILVA; VALENTE; TERRA, 2016). Thus, a code review process requires them to detect smells in unfamiliar code.

The subjectiveness concerned with individual code smells assessment was addressed by previous studies (HOZANO et al., 2018; PALOMBA et al., 2014; TAIBI; JANES; LENARDUZZI, 2017). These were empirical studies aimed to expose developers to software code snippets and gather their viewpoints about the occurrence of certain types of code smells, obtaining qualitative findings regards to how similar they detect code smells (HOZANO et al., 2018) and how they perceive and associate the problem to the symptoms of smell definition (PALOMBA et al., 2014). In the case of Hozano et al. (HOZANO et al., 2018), they explored personal factors, such as experiences and heuristics formulated by developers to discover agreements among them when evaluating code smells in code. In the experimental phase, the authors present to the developer a suspicious source code that hosts a code smell as well as a description of the informal definition of the smell to be evaluated, asking the developer to confirm or not the occurrence of the described code smell type in source code. However, it lacks some informative element that may describe why certain suspicions source code was previously classified as code smell in order to aid developers in reasoning about the occurrence of a code smell with more effectivity.

In our research, rather than merely show a suspicions source code and the respective informal definition of the code smell, we propose to show to the developer a visualization of a decision tree classifier, composed by some metric-based rules, aimed to inform the developer the reasons why some source code was previously classified as a host of a code smell. Thereafter, the developer can confirm or not the occurrence of certain code smell aided by such rules that indicates the reason why the source code is "smelly". Inspecting a decision tree model representation, through its nodes and rules, can give new insights to developers in order to identify which attributes (metrics) are the strongest predictors of the class variable (smelly or not) (FREITAS, 2014). These new insights may tease the developer to reasoning the occurrence of a code smell widely, not restricting to factors concerned only to past experiences and backgrounds.

1.2 Objective

As an evolution of previous study (HOZANO et al., 2018), we aim at investigating how the code smell agreement among developers may be influenced by the visualization provided by a comprehensible classification model, the decision tree classifier. (FREITAS, 2014). This classifier was largely employed (AZEEM et al., 2019) in recent studies focused on automatic smell detection and offered good effectivity measures (AMORIM et al., 2015). Moreover, decision trees have an output that is pretty easy to interpret, giving the opportunity to properly understand the mechanisms that lead to the detection of a certain code smell instance (AZEEM et al., 2019).

In this study we address the following hypotheses:

- **(H1)** developers exposed to the rules of a classification model, each rule with transparent software metric, tends to agree on the ocurrence of a code smell, minimizing the subjectivity inherited from code smell informality.

- **(H2)** the evaluation process takes less effort to be concluded when the developer is exposed to elements which characterize the smelly code.
- **(H3)** from the perspective of decision making, the decision tree visualization is useful to the evaluation process.

1.3 Execution and main results

To perform the objectives aforementioned, we gather collaborations from 30 developers from industry and academy. They evaluated 2 groups with 4 tasks each with source codes potentially affected by some type of smell. Each group of task expose the developer to a source code in distinct perspectives, so one shows the classification model that predicted the code as "smelly" whereas the another group doesn't (blind evaluation). The experiment was carry out through a web-based app created exclusively to collect the evaluations from participants, designed to comply with the requirements of study design.

The main results obtained are:

- We got evidence that code smells identification aided by a decision tree leads to a relative improvement of agreement in relation to detections based solely on code analysis. After detaching different kinds of participants from the whole set of participants, we discovered groups that behave distinctly. The agreement on the detections aided by decision tree performed better with experienced code smell detection participants, which brings the evidence that such profile got the best out of our approach.
- As for the effort, the identification of code smell aided by decision tree did not decrease in time compared to detection based only on code inspection. For both groups of tasks, there isn't any evidence that indicates the benefits related to the effort reduction when detecting code smells with decision tree, i. e., the time spent to detect smells with decision tree tend to be equivalent or worst than the time spent to detect code smells based solely in code inspection.
- Finally, based on the answers provided by the majority of participants, our experiment suggests that the decision trees used to support code smell detection are useful to the developer for decision making. The classification models (decision trees) that represent the detection rules of a God Class and a Long method were the best-evaluated models in terms of usefulness, i. e., models that offered good contributions to decision making.

1.4 Contributions

Our approach offered a novel way to detect code smells using metric-based rules to improve reasoning about the smelliness of the source code under analysis. From this study emerges the following contributions:

- favors the decision making during the code review process;
- minimization of conflicting code smell evaluations, favoring the consistency across code reviews;
- identification of the most concerned issues about the metric-based rules provided by the decision tree models.

1.5 Research structure

This research is structured as follows.

- Section 2 describes some background useful for the understanding of this work, including a brief description of code smell and decision tree classifier.
- Section 3 presents the research study and its goals, the design of our online experiment, and how the data will be analyzed.
- Section 4 and 5 present and discuss the results obtained.
- Section 6 discuss past research of related to code smells, agreements among developers and detection methods.
- Section 7 details the limitations and threats to validity of this study.
- Section 8 presents our conclusion and ideas which could lead to future contributions in the field of Software Engineering.

2 Study Background

In this chapter we contextualize our work, giving an overview of code smells and machine learning, the base concepts used in this study.

2.1 Code Smell

Code anomaly, so-called "code smell", is an implementation choice that indicates a design problem which violates the well-known principles of cohesion, abstraction and separation of concerns (OIZUMI et al., 2016) on software development. There are more than 20 (FOWLER et al., 1999) different types of code smells that contributes to increase comprehension effort (ABBES et al., 2011) and leads to design degradation (OIZUMI et al., 2016). The adoption of refactoring, a clean-up process that improves the code's internal structure (FOWLER et al., 1999), is the way out to prevent the presence of code smells in code. Fowler et. al. (FOWLER et al., 1999) about 20 sets of symptoms of code smells. These include large classes, feature envy, long parameter lists, and lazy classes. Each code-smell type is accompanied by refactoring suggestions to remove it.

We chose four different types of code smells to carry out experiments, which are detailed in Section 3.1.

2.2 Machine Learning

(VANDERPLAS, 2016) states that Machine learning involves building mathematical models to help understand data. Once these models have been fit to previously seen data, they can be used to predict and understand aspects of newly observed data. The learning process might be categorized into two main types: supervised learning and unsupervised learning.

2.3 Supervised and Unsupervised Learning

Supervised learning is a method that attempts to discover the relationship between input attributes (independent variables) and a target attribute (referred to as a dependent variable). The relationship that is discovered is represented in a structure referred to as a Model. Models describe and explain phenomena which are hidden in the dataset and which can be used for predicting the value of the target attribute whenever the values of the input attributes are known. The supervised methods can be implemented in a variety of domains such as marketing, finance and manufacturing (ROKACH; MAIMON, 2008).

Unsupervised learning involves modeling the features of a dataset without reference to any label. These models include tasks such as clustering, in which data is automatically assigned to some number of discrete groups. As a result, the model can infer labels on unlabeled data (VANDERPLAS, 2016).

2.4 Decision Tree Classifier

A decision tree is a type of supervised learning that uses a tree structure to represent a number of possible decision paths and an outcome for each path. They're very easy to understand and interpret, and the process by which they reach a prediction is completely transparent (GRUS, 2019). Due to its transparency, this technique is frequently used in applied fields such as finance, marketing, engineering and medicine (ROKACH; MAIMON, 2008). For having this self-explanatory feature, there is no need to be a data mining expert in order to follow a certain decision tree. (ROKACH; MAIMON, 2008)

Decision trees can easily handle a mix of numeric and categorical attributes and can even classify data for which attributes are missing (GRUS, 2019). When a decision tree is used for classification tasks (which produce categorical outputs), it is most commonly referred to as a classification tree. When it is used for regression tasks, it is called a regression tree (which produce numeric outputs) (ROKACH; MAIMON, 2008). The Figure 1 shows a typical decision tree classifier, an example that predicts an animal, based on a set of characteristics. Instances are classified by navigating them from the root of the tree down to a leaf according to the outcome of the tests along the path.

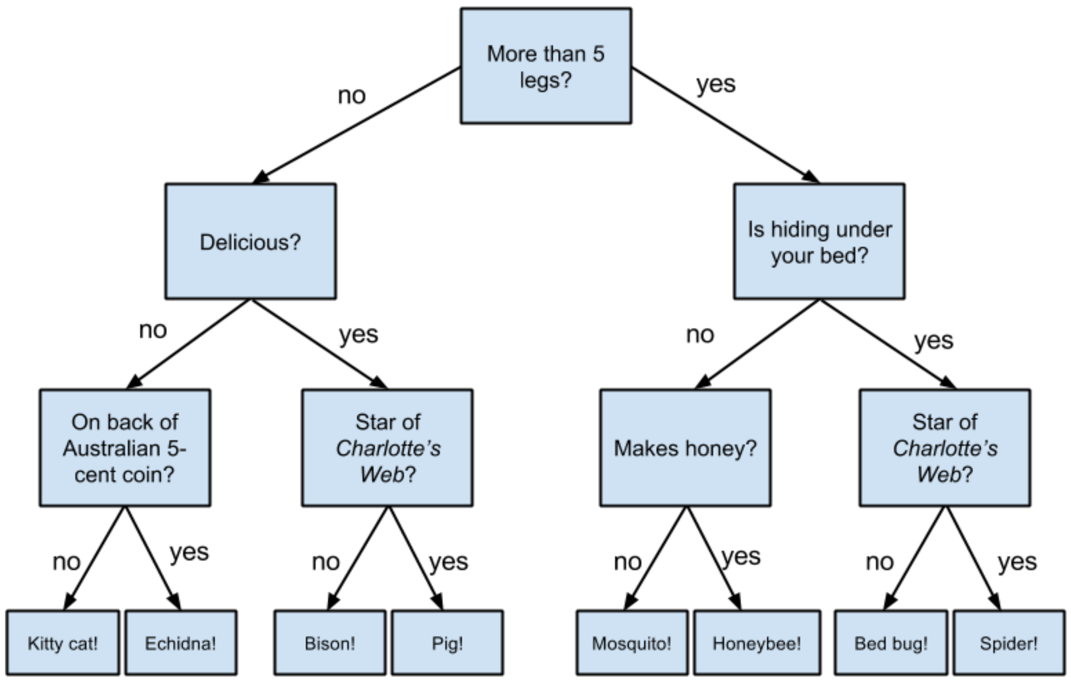


Figure 1 – Example: A decision tree to guess an animal. (GRUS, 2019)

As seen in Figure 1, the decision tree consists of a node called a “root” that has no incoming edges. All other nodes have exactly one incoming edge. A node with outgoing edges is referred to as an “internal” node or a “test” node. All other nodes are called “leaves”. In a decision tree, each internal node splits the instance space into two or more sub-spaces according to a certain discrete function of the input attributes values. In the simplest and most frequent case, each test considers a single attribute, such that the instance space is partitioned according to the attributes value. In the case of numeric attributes, the condition refers to a range. Each leaf is assigned to one class representing the most appropriate target value (ROKACH; MAIMON, 2008).

To comprehend any predictions, we start with a root of a tree; we consider the characteristic that corresponds to the root and we define to which branch the observed value of the given characteristic corresponds. Then, we consider the node in which the given branch appears. We repeat the same operations for this node until we reach a leaf (ROKACH; MAIMON, 2008).

Algorithms for constructing decision trees work top-down, by choosing a feature at each step that best splits the set of instances (ROKACH; MAIMON, 2008). Different algorithms use different metrics for measuring the "best", generally measuring the homogeneity of the target feature within the subsets. So these metrics are applied to each candidate subset, and the resulting values are combined to provide a measure of the quality (or impurity) of the split. There are two well known splitting criteria which can be cited:

Information Gain. is an impurity-based criteria that uses the entropy measure as the impurity measure (ROKACH; MAIMON, 2008). Entropy is defined as:

$$Gini(T) = -\sum_{i=1}^{J} p_i \log_2 p_i \tag{2.1}$$

where $p_1, p_2, ...$ are fractions of samples and represent the percentage of each class present in the child node that results from a split in the tree. Thus, the Information gain is the degree of impurity among parent and children:

$$\overbrace{IG(T,a)}^{Information\ Gain} = \overbrace{H(T)}^{Entropy\ (parent)} - \overbrace{H(T|a)}^{Weighted\ sun\ of\ Entropy\ (Children)} \tag{2.2}$$

Gini Index. Is an impurity-based criteria that measures the divergences between the probability distributions of the target attributes (ROKACH; MAIMON, 2008) values and is defined as:

$$Gini(p) = 1 - \sum_{i=1}^{J} {p_i}^2 \tag{2.3}$$

where $i \in \{1, 2, ..., J\}$ and p_i the fraction of items labeled with class i in the set.

3 Study Design

The main challenge in the identification of code smells is that their definitions are based on inaccurate concepts that make developers to have divergent reasoning whether a piece of code is a smell or not. The divergences impact negatively the agreement on code smell evaluations (HOZANO et al., 2018).

After a learning process, with software metrics as independent variables, we intend to discover statistically to what extent the insights provided by the predictor's rules (decision tree) may influence the agreement among developers after several code smell evaluations. In summary, this study intends to contribute to answering the following questions:

RQ1: *Do an aided approach of code smell detection - using a tree classifier representation – influence on agreement among developers*?

The motivation for this question is to investigate how developers agree after evaluating candidate source codes from real projects that are suspicious to host a certain type of code smell. During the experiment, two distinct perspectives are considered: evaluations aided by decision tree visualization and evaluations without any visualization (only "raw" source code). We analyze the degree of agreement from both perspectives separately and compare. These analysis and comparison are made considering the (i) totality of developers that contributes to our work and (ii) by grouping according to common characteristics that they share. Thus, we aimed at performing a deeper investigation in order to increase the knowledge about to what extent the rules provided by a comprehensible classification model can influence the agreement among developers when the developers detect smells in unfamiliar source code. Our results may shed light on the construction of comprehensible machine learning models that are able to aid developers during the process of code smell detection.

RQ2: *How much time do developers spend during code smell detection with a tree classifier support?*

This RQ aims at measuring the time spent to accomplish each task evaluation of code smell detection, separating evaluations aided by decision tree visualization and evaluations without any visualization. We built an experiment that has a stopwatch integrated into each task. After each task accomplishment, the experiment counts the time-lapse from the start of the task until the task submission. Thus, we measure how long time the developer takes to detect smells when he is aided by a decision tree classifier and when he is not.

RQ3: *How useful is the decision tree visualization for decision making?*

The motivation of this question aims at analyzing if transparent metric-based rules from a decision tree classifier are relevant to aid the decision making during smell detection. By collecting opinions through an open question, the empirical experiment gave us several insights about to what extent a visual representation of a classification model contributes to detecting a code smell. The result of this RQ is important to find out the strong and weak points of our approach, mainly concerning the structure of the available classification trees.

In the next sections, we describe in detail the steps performed in our study.

3.1 Code smell types selection

In our study, we considered the four code smell types briefly described in Table 1. We chose these types based on previous studies about the developer's perceptions (PALOMBA et al., 2014) and agreements (HOZANO et al., 2018). The list varies from highly perceived and identified types (god class, long method) to moderate and less perceived ones (long parameter list, refused bequest) (PALOMBA et al., 2014), similarly we chose types that vary from fair agreement level (god class, long method) among developers to slight agreement (long parameter list, refused bequest) (HOZANO et al., 2018). Therefore, these code smell types weren't chosen randomly but driven by how developers recognize each type of chosen smell.

Furthermore, these smell types affect different scopes of a program. While God Class and Refused Bequest affect mainly classes, the Long Method and Long Parameter List are related to methods.

Table 1 – Types of code smells selected (FOWLER et al., 1999)

name	**scope**	**description**
God Class (GC)	class	A God Class implements several different responsibilities and centralize most of the system processing.
Refuse Bequest (RB)	class	A class redefining most of the inherited methods, thus signaling a wrong hierarchy.
Long Method (LM)	method	A method that is unduly long in terms of lines of code.
Long Parameter List (LPL)	method	A method having a long list of parameters, some of which avoidable.

3.2 The oracle dataset

We call our source to train the machine learning technique as an oracle dataset. This dataset contains instances that represent java classes or methods mined from several projects and columns composed by independent and dependent variables, which are software metrics and a boolean-type classification attribute, that marks the occurrence of a code smell, respectively.

We imported the work made by Palomba et al. (PALOMBA et al., 2018) to gather occurrences of java classes and methods that are affected by the code smell types listed in Table 1. They

Table 3 – Open source projects involved in the study

Project	**Version**	**Description**
Ant	1.8.3	Build System
Cassandra	1.1	Database Management System
Eclipse Core	3.6.1	Integrated Development Environment
Elastic Search	0.19	RESTful Search and Analytics Engine
Hadoop	0.9	Tool for Distributed Computing
Hbase	0.94	Distributed Database System
Hive	0.9	Data Warehouse Software Facilitates
Lucene	3.6	Search Manager
Nutch	1.4	Web-search Software
Karaf	2.3	Standalone Container
Pig	0.8	Large Dataset Analyzer
Qpid	0.18	Messaging Tool
Ivy	2.1.0	Dependency Manager
Wicket	1.4.20	Java Application Framework
Xerces	2.3.0	XML Parser

developed a tool to perform smell detection in 30 open source systems. Afterward, the authors manually validated the candidate code smells suggested by the tool, classifying as true or false positives all candidate code smells. Due to the difficulty faced to find some projects and version on Github, we selected only 15 of 30 available projects with validated code smells. These projects are listed in Table 3.

Largely used as an input for predictive code smell detection in machine learning-based approaches (FONTANA et al., 2016; NUCCI et al., 2018; HOZANO et al., 2017; FONTANA et al., 2013; AZEEM et al., 2019), software metrics are utilized to measure quality as well as estimate the cost and effort of software projects (FENTON; BIEMAN, 2014). We used the educational license of software Understand [1] to calculate and extract source code metrics by analyzing the open-source projects listed in Table 3. After metric extraction, we got 32 class-level metrics and 14 method-level metrics.

The complete list of the collected metrics is available in the Table 22 and Table 23, given in Appendix A. These metrics cover different aspects of source code such as complexity, size, cohesion and inheritance. Besides, many of them were mentioned in various publications that involve software engineering (NUÑEZ-VARELA et al., 2017), mainly machine learning applications (AZEEM et al., 2019).

[1] https://scitools.com/

3.3 Training the algorithm and generating the classifiers

Decision tree refers to a hierarchical model of decisions and their consequences, used to classify an object or an instance into a predefined set of classes based on their attributes values (ROKACH; MAIMON, 2008). In artificial intelligence, this algorithm is known as an induction algorithm because it obtains a training set and forms a model that generalizes the relationship between the input attributes and the target attribute (ROKACH; MAIMON, 2008). Researches recognize decision trees as a comprehensible (or interpretable) classifier for humans (GUIDOTTI et al., 2018) due to its intuitive graphical structure and its capability to help users to focus their analysis on the most relevant attributes, rejecting non-relevant ones (FREITAS, 2014).

An interpretable classification model is a relevant property that allows users to trust the model's predictions and follow the recommendations associated with those predictions. Besides, comprehensible classification models can give new insights to users about important predictive relationships in the data, i.e., identifying which attributes are the strongest predictors of the class variable (FREITAS, 2014). To our context, the insights provided by rules, drawn in a tree format, offer transparent pieces of information that may aid the developers to interpret each type of code smell. Thus, new agreement patterns may be discovered, giving a novel approach regarding previous studies (HOZANO et al., 2018) that similarly addressed the agreement subject. The algorithm was trained using the values of software metrics from oracle dataset instances, i. e., from classes or methods of the pre-annotated training dataset. In other words, we have used the oracle as a training dataset containing various metrics calculated for each java code, an entire class or a code snippet (method), in addition to the information on whether that java code contains each of the studied code smells - the target attribute. We adopted Python's Pandas [2] and Sklearn [3] API to carry out the training process, which includes data preparation, model fitting and effectivity evaluation. Sklearn API implements C.A.R.T. (Classification and Regression Trees), one of the variants of decision tree algorithm that include C5.0, C4.5 and ID3. During algorithm training, some hyperparameters were tested in order to obtain the best tradeoff between the number of leaves, tree depth and effectivity, i. e., we went towards generating trees with low complexity considering the highest possible effectivity level. We were looking for such tradeoff because, regarding user perspective, the number of leaves and tree depth have a straight correlation with the interpretation difficulty (LUŠTREK et al., 2016) of the model, that's why we attempted to generate short trees to facilitate participant's visualization during the experiment, taking into account the best effectivity measurements.

To evaluate the effectivity of the trained models, we use the widely-adopted Information Retrieval (IR) metrics, namely recall and precision (BAEZA-YATES; RIBEIRO-NETO et al., 1999; ROKACH; MAIMON, 2008):

$$Precision = \frac{TP}{FP + TP}$$

[2] https://pandas.pydata.org/

[3] https://scikit-learn.org/

Code smell	Leaves	Depth	TN	TP	FN	FP	Precision	Recall	F-measure
lpl	6	5	106	29	9	6	0,841	0,856	0,849
lm	6	3	462	137	4	2	0,973	0,976	0,972
gc	3	2	47	14	2	2	0,843	0,889	0,878
rb	3	2	84	27	9	5	0,750	0,724	0,721

Table 4 – Effectivity result

$$Recall = \frac{TP}{FN + TP}$$

where,

- **TP**: True positives is the number of positives instances correctly classified as positive.
- **TN**: True negatives is the number of negatives instances correctly classified as negative.
- **FP**: False positives is the number of negative instances incorrectly classified as positive.
- **FN**: False negatives is the number of positives instances incorrectly classified as negative.

As an aggregate indicator of precision and recall, F-measure, i.e., the harmonic mean of precision and recall, is represented by:

$$F\text{-}measure = 2 \times \frac{precision \times recall}{precision + recall}$$

Furthermore, we use a 10-fold cross-validation approach to assessing the performance of a predictive model and preventing the over-fitting phenomenon. Thus, we calculated the average effectivity measures over all folds. Table 4 shows the overall results obtained by the decision tree classifier for each code smell type. After hyperparameter testing, we obtained decision tree models that range from 3 until 6 leaves and from 2 until 5 of tree depth.

Finally, the C.A.R.T algorithm is able to generate distinct Decision Trees (as the one in Figure 2), each one trained to determine whether a certain class (or method) contains a certain Code Smell as long as its metrics are known. The nodes of the tree represent software metrics, whereas the leaves represent the prediction that points to the existence of the code smell analyzed. The decision tree algorithm tends to allocate the most significant metrics for the detection of the Code Smell in the top of the tree, while the less significant metrics appear as the tree deepens and gets more specialized.

3.4 Study Procedure

The core procedure of our study is to test two distinctive treatments for different subjects (participants): evaluation of code smell with a decision tree visualization and evaluation of code

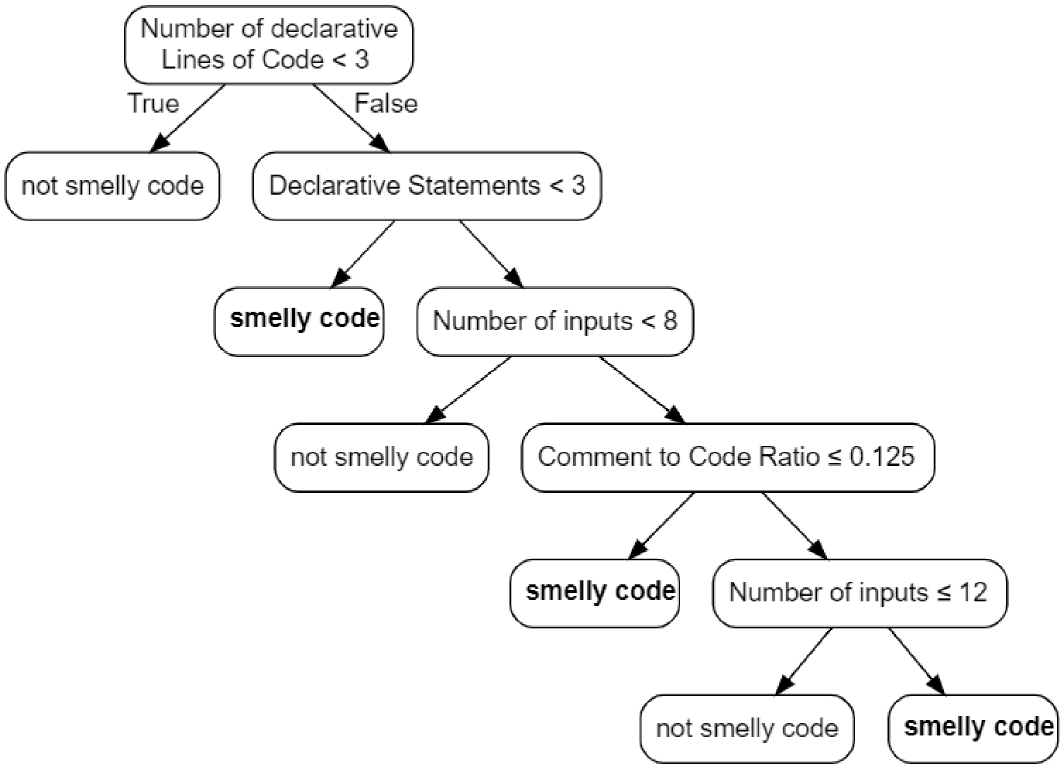


Figure 2 – Decision tree model which indicates a Long Parameter List.

smell without decision tree visualization. At the same time, we must assure the equivalence of evaluations from participants splitting several code snippets into two groups: in each group, each participant evaluates a task (a code snippet that hosts a code smell) along with a distinctive treatment.

From the previous paragraph, we may infer two independent variables with two levels each: two groups of tasks with two treatments each. Hence a 2 by 2 matrix-like design, or Latin Square design (BOX; HUNTER; HUNTER, 2005), is suitable to accomplish our research needs. Figure 3 represents the experiment design applied to the present research and it is disposed as follows:

- **Rows**: participants are shuffled in rows, namely participant 1 and participant 2;
- **Columns**: in the columns, there are 2 fixed groups of tasks, each task containing a source code (class or method) affected by a specific code smell type;
- **Cells**: treatment applied to some task, addressed to some participant. "DT" stands for "Decision Tree" - the decision tree model is available to be visualized - whereas "No DT" stands for "No Decision Tree" - the decision tree model visualization is omitted.

This experiment aims to collect the opinion of many participants about to what extent they (dis)agree about the existence of certain code smell type in selected code snippets. A user-friendly internet-based application was designed and suited to gather such opinions, having a business

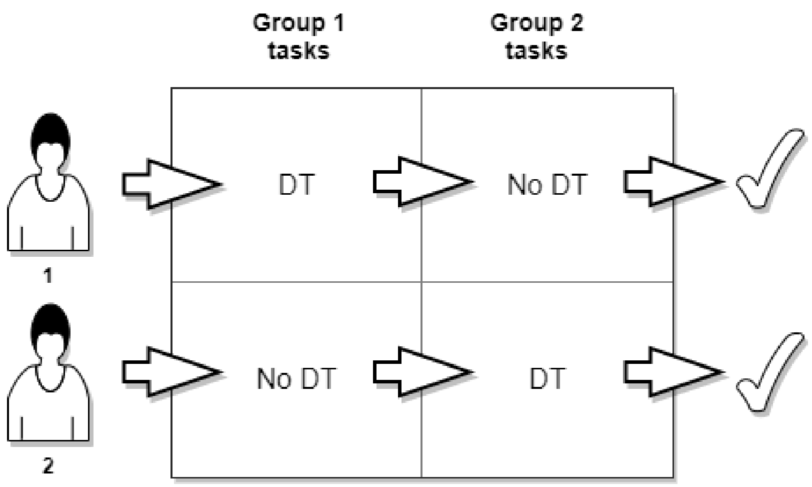


Figure 3 – Experiment design. A 2x2 latin square.

logic that complies with the experiment design showed in Figure 3. The application is fully available at Github [4].

The application's URL was published on the Internet, mainly in social networks focused on IT professionals involved with Software Engineering, to invite potential participants. After the invitation acceptance, the candidate finds the following flow, illustrated in Figure 4:

1. *Welcome page*: In the beginning, we present a welcome page to the participant with a brief description about the context of the study.

2. *Participant's background*: the participant rates its own experience about software development, object-oriented development, Java programming language, code revision and code smell detection. Each developer had to assign a rating which varies from "I do not have any experience" until "Very High". Besides, the participant is asked about his highest academic degree so far, what's his origin (Industry, academy or both) and what is his experience in software development in years. The complete questionnaire can be found in the Appendix B.

3. *Startup instructions*: at this stage, the experiment tool explains what is expected from the participant throughout the experiment. It includes how many tasks must be accomplished, what is the purpose of the tasks, what is the possible scenarios of code evaluation and a brief introduction about what is a decision tree. An example of a decision tree, as similar to those presented during experiment execution, is shown to the participant in order to get familiarity with the metric-based rules. After clicking the button "Start experiment", the participant is forwarded to the tasks accomplishment step, where he starts to evaluate Java codes.

4. *Tasks accomplishment*: at this point, the participants should accomplish a sequence of 12 tasks. This step is detailed in Section 3.5.

[4] https://github.com/christianorossini/masterSurvey

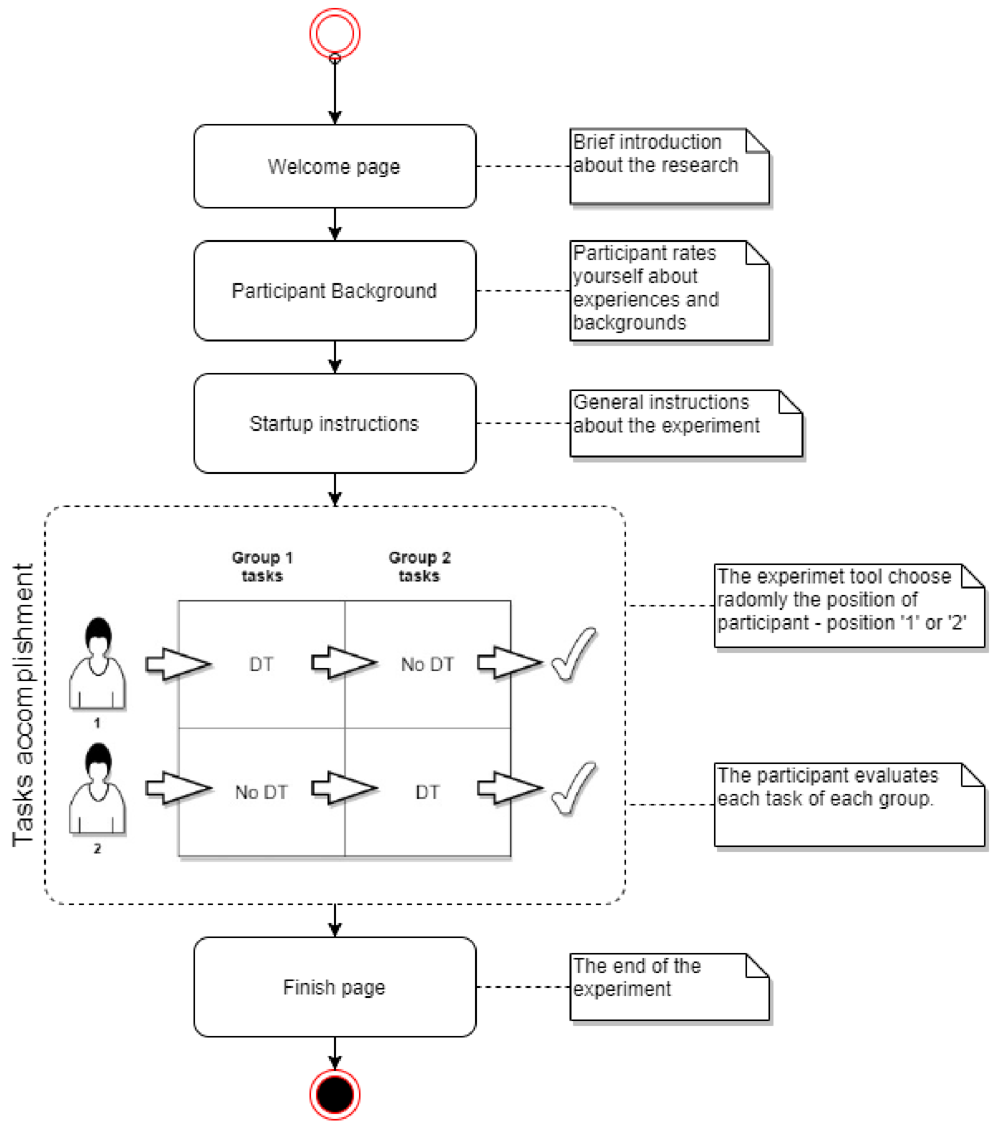

Welcome page
Brief introduction about the research
Participant Background
Participant rates yourself about experiences and backgrounds
Startup instructions
General instructions about the experiment
Tasks accomplishment
Group 1 tasks
Group 2 tasks
1
DT
No DT
2
No DT
DT
The experimet tool choose radomly the position of participant - position '1' or '2'
The participant evaluates each task of each group.
Finish page
The end of the experiment

Figure 4 – Steps of the experiment

5. *Finish page*: once all tasks are accomplished, the experiment tool shows a page confirming that the experiment is ended.

3.5 Tasks accomplishment

To our context, a task is an evaluation unit which has a correspondent Java code which is suspicions to host one of the code smells of Section 3.1. As we can observe in the experiment design illustrated in Figure 3, the tasks are divided into two groups, each group with six tasks. Four of these tasks, i. e., $1/3$ of the total, are merely control tasks or, as we call, "dummy" tasks. The dummy tasks aim to limit the biased (PALOMBA et al., 2014) and carelessly participations, so we might filter participants that indicate the same answers without any criterion. Therefore, we have twelve tasks, but only eight are considered valid tasks (see Table 5 and Table 6).

In order to suits the experiment design (Figure 3), the experiment tool logically groups

Task ID	Class or method	Code smell Type
#11	org.apache.nutch.tools.Benchmark.benchmark	lpl
#7	org.apache.qpid.ra.inflow.QpidActivationSpec org.apache.qpid.ra.ConnectionFactoryProperties	rb
#4	org.apache.ivy.plugins.resolver.BasicResolver.getDependency	lm
#3	org.apache.tools.ant.DirectoryScanner	gc

Table 5 – Classes or methods that belongs to tasks of group 1

Task ID	Class or method	Code smell Type
#12	org.apache.hadoop.fs.ProxyFileSystem.create	lpl
#5	org.apache.qpid.ra.QpidRAStreamMessage org.apache.qpid.ra.QpidRAMessage	rb
#6	org.apache.pig.impl.logicalLayer.optimizer.PushUpFilter.check	lm
#8	org.eclipse.jdt.internal.core.util.PublicScanner	gc

Table 6 – Classes or methods that belongs to tasks of group 2

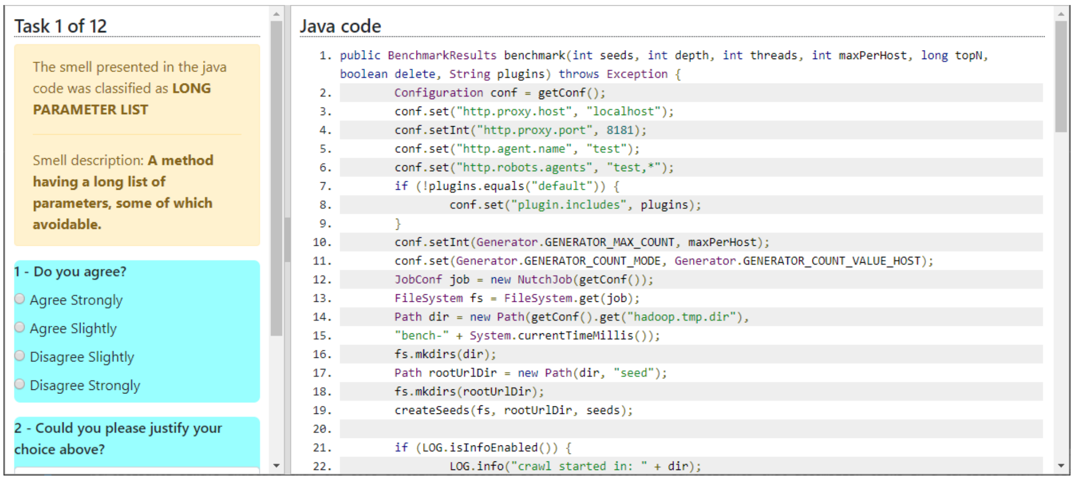
the participants in pairs, so each participant is assigned randomly as Participant 1 (row 1) or Participant 2 (row 2). Therefore, the experiment tool continuously checks the number of participants labeled as "1" or "2" in order to maintain the groups equalized.

Depending on the row that was chosen for the participant, first or second row of the Latin Square, labeled Participant "1" or "2" respectively, it determines how they're going to accomplish each group of tasks, i. e., what is the sequence of treatment that will be applied for each group of tasks. The treatments come inside each cell of the experiment design, so each treatment appears only once in every row and every column (see Figure 3). As a result, each participant performs both groups of tasks and has the opportunity to visualize the correspondent decision tree model only in one of the groups of tasks. Such tasks divided into groups favor the minimization of the learning effect because the treatments are completely isolated from each other.

From this point to further work the treatments mentioned above we refer as scenarios: in one scenario the participant will have the opportunity to evaluate a Java code with the support of the rules provided by a visual representation of the predictor (decision tree) while in another the participant evaluates the code without any support.

The Figures 5 and 6 exhibit the environments designed to task accomplishment and they represents the scenarios explained previously. In both cases, the experiment tool presents the name and description of the current code smell that is being evaluated. This description is based on an informal definition presented by Fowler (FOWLER et al., 1999) as well as derived descriptions (HOZANO et al., 2018; PALOMBA et al., 2014), so it's a starting point to the task evaluation.

For the scenario without decision tree visualization (Figure 5), the task environment has the following elements:



Task 1 of 12

The smell presented in the java code was classified as **LONG PARAMETER LIST**

Smell description: **A method having a long list of parameters, some of which avoidable.**

1 - Do you agree?
- Agree Strongly
- Agree Slightly
- Disagree Slightly
- Disagree Strongly

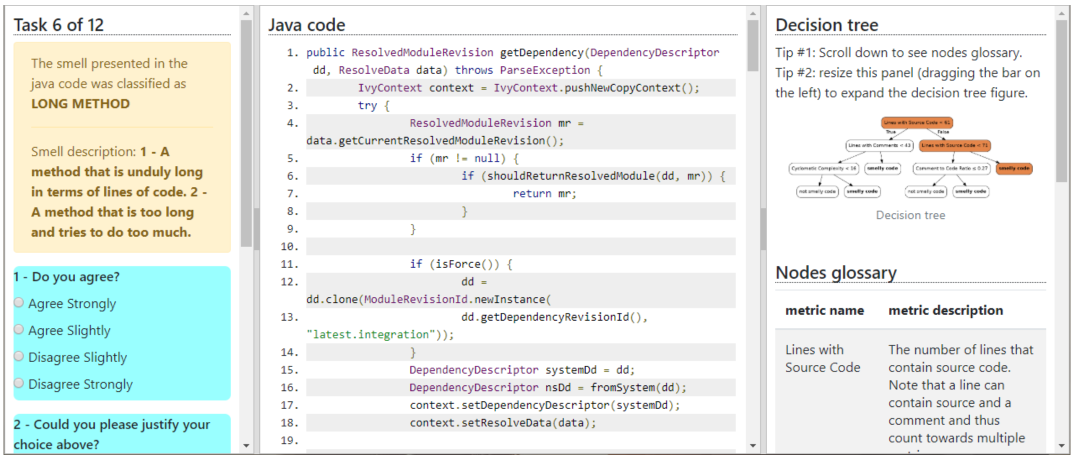
2 - Could you please justify your choice above?

Java code

```java
public BenchmarkResults benchmark(int seeds, int depth, int threads, int maxPerHost, long topN,
boolean delete, String plugins) throws Exception {
        Configuration conf = getConf();
        conf.set("http.proxy.host", "localhost");
        conf.setInt("http.proxy.port", 8181);
        conf.set("http.agent.name", "test");
        conf.set("http.robots.agents", "test,*");
        if (!plugins.equals("default")) {
                conf.set("plugin.includes", plugins);
        }
        conf.setInt(Generator.GENERATOR_MAX_COUNT, maxPerHost);
        conf.set(Generator.GENERATOR_COUNT_MODE, Generator.GENERATOR_COUNT_VALUE_HOST);
        JobConf job = new NutchJob(getConf());
        FileSystem fs = FileSystem.get(job);
        Path dir = new Path(getConf().get("hadoop.tmp.dir"),
        "bench-" + System.currentTimeMillis());
        fs.mkdirs(dir);
        Path rootUrlDir = new Path(dir, "seed");
        fs.mkdirs(rootUrlDir);
        createSeeds(fs, rootUrlDir, seeds);

        if (LOG.isInfoEnabled()) {
                LOG.info("crawl started in: " + dir);
```

Figure 5 – Task environment without decision tree



Task 6 of 12

The smell presented in the java code was classified as **LONG METHOD**

Smell description: **1 - A method that is unduly long in terms of lines of code. 2 - A method that is too long and tries to do too much.**

1 - Do you agree?
- Agree Strongly
- Agree Slightly
- Disagree Slightly
- Disagree Strongly

2 - Could you please justify your choice above?

Java code

```java
public ResolvedModuleRevision getDependency(DependencyDescriptor
 dd, ResolveData data) throws ParseException {
        IvyContext context = IvyContext.pushNewCopyContext();
        try {
                ResolvedModuleRevision mr =
data.getCurrentResolvedModuleRevision();
                if (mr != null) {
                        if (shouldReturnResolvedModule(dd, mr)) {
                                return mr;
                        }
                }

                if (isForce()) {
                        dd =
dd.clone(ModuleRevisionId.newInstance(
                        dd.getDependencyRevisionId(),
"latest.integration"));
                }
                DependencyDescriptor systemDd = dd;
                DependencyDescriptor nsDd = fromSystem(dd);
                context.setDependencyDescriptor(systemDd);
                context.setResolveData(data);
```

Decision tree

Tip #1: Scroll down to see nodes glossary.
Tip #2: resize this panel (dragging the bar on the left) to expand the decision tree figure.

Lines with Source Code < 61
True
False
Lines with Comments < 43
Lines with Source Code < 71
Cyclomatic Complexity < 16
smelly code
Comment to Code Ratio ≤ 0.27
smelly code
not smelly code
smelly code
not smelly code
smelly code

Decision tree

Nodes glossary

metric name	metric description
Lines with Source Code	The number of lines that contain source code. Note that a line can contain source and a comment and thus count towards multiple

Figure 6 – Task environment with decision tree

1. **Task submission (left panel)**. For each task, the participant should submit a form composed of the following questions:
   1 - Do you agree? - based on the code smell type described by the task, in this question the participants rates their degree of agreement regarding the smelliness of the Java code. Inspired by the Likert scale (OPPENHEIM, 2000), four options are available:

   - Agree Strongly;
   - Agree Slightly;
   - Disagree Slightly;
   - Disagree Strongly.

   2 - Could you please justify your choice above? - the participant should justify why he agrees or disagrees.

2. **Java code (middle panel)**. The java code has to be evaluated by the participant in order to locate the suspicious code smell type described by the task.

The scenario with decision tree visualization (Figure 6) has almost the same elements as the previous one, except by:

1. **Task submission (left panel)**. Compared to the previous scenario, it has an extra question :

   3 - What insights (contributions) did the decision tree give you in order to support your decision? - to this question the participant should explain whether the decision tree model visualization contributes to reasoning about his evaluation.

2. **Decision tree (right panel)**. The decision tree contains metric-based rules that lead to the code smell type that affects the java code presented by the task. It serves as a support for decision making. The highlighted path from the root node until leaf stands for the satisfied conditions within nodes that lead the predictor (machine learning classifier) to classify the current java code as smelly code (Figure 7). Additionally, the panel provides a sub-section called "Nodes Glossary" which contains a list of metrics and its descriptions aimed to facilitates the comprehension of the rules.

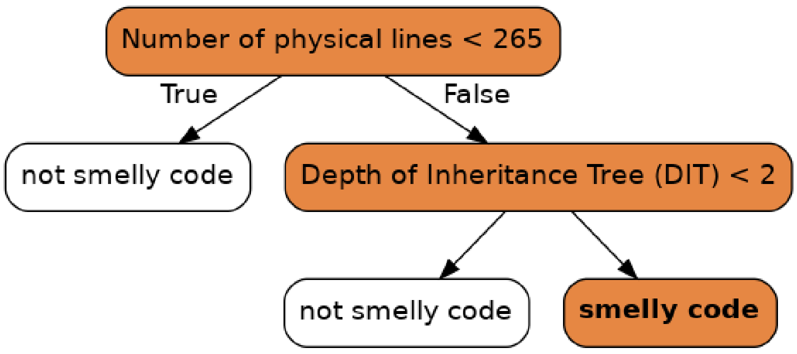

Number of physical lines < 265
True
False
not smelly code
Depth of Inheritance Tree (DIT) < 2
not smelly code
smelly code

Figure 7 – During a task evaluation, a highlighted path of a decision tree model is shown to the participant.

3.6 Experiment execution and data inspection

The experiment lasts about 4 weeks and had 30 participants who completed all the required tasks successfully. Afterward, we collected all the data from a relational database and we performed a data inspection to guarantee the quality of the data observed in the study. Thus, we analyzed the evaluations and open questions in order to identify problems that could interfere with the data analysis negatively.

As we mentioned in section 3.5, we created "Dummy tasks" which are randomly selected java codes not affected by any of the code smells considered in our study. This was done to limit careless participations (PALOMBA et al., 2014) in the study, i.e., avoid that participants who always indicate that the code contains the suggested problem without reasoning about it. So, we first search for participants that accomplish tasks in a careless and/or biased way, analyzing how

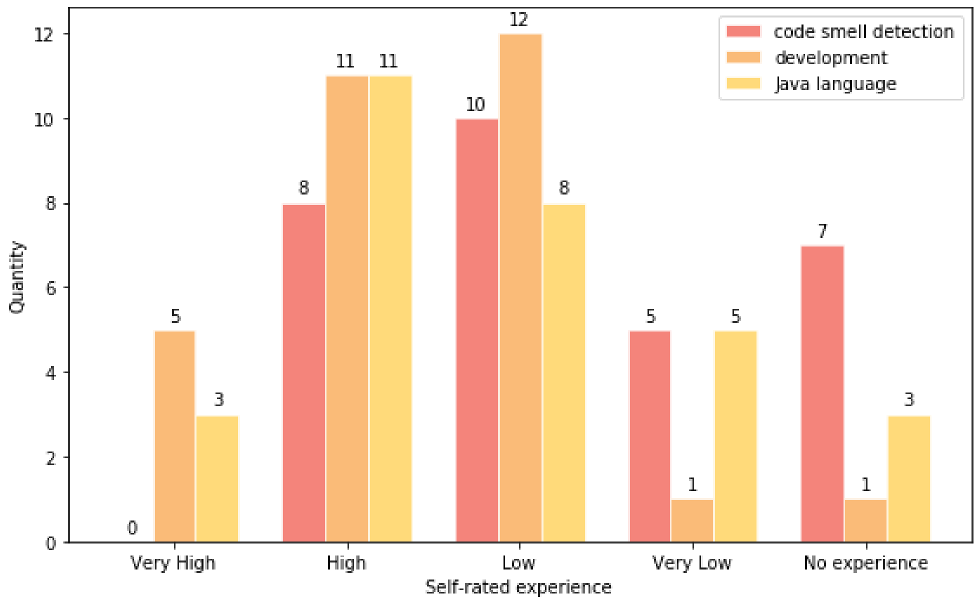


Figure 8 – Self-rated experience from participants

they performed tasks when faced with dummy tasks. For instance, we intentionally created a dummy task that defines and describes a god class. However, the associated java code has just a few lines of codes. Thus, we expected that the participant would mark "I strongly disagree" or, at minimum, "I Slightly disagree" in most of the dummy tasks presented to him. After data inspection, we noted that all of the participants not behave biased or carelessly.

As mentioned previously, we obtained answers from 30 participants with different experience levels in software development, Java programming language and code smell detection. The participants rated their personal experience that varies from "No experience" up to "Very High", distributed as illustrated in Figure 8. We note that the participants reported different experience levels (x-axis) and the great majority indicated experience higher or equal to "Low". For instance, the graph shows us that most of the participants have excellent skills in development, some of them having very high experience on it. It allows us to explore possibilities considering developers with different experience levels and with at least certain knowledge about the investigated context.

3.7 Data analysis

After experiment execution and data collection we summarized the data to answer the research questions. We use methods to (i) analyze the inter-rater agreement among developers as well as developer's experiences and backgrounds that may influence such agreement, (ii) compute the confidence of evaluations considering the scenario with decision tree visualization and the scenario without decision tree and (iii) analyse the time spent by participants to accomplish each task considering both scenarios.

task	Participants													
	01	02	03	04	05	06	07	08	09	10	11	12	13	14
#3	■	■	■	■	■	■	■	■	■	□	■	■	■	■
#4	□	■	■	■	■	■	■	■	■	■	■	■	■	■
#11	■	■	■	■	■	■	■	■	■	■	■	■	■	■
#7	■	■	□	□	■	□	□	■	□	■	■	■	■	■

Table 7 – Participants' evaluations for tasks group 1, in DT scenario. The grey cell represents the 'Agree', the white is 'Disagree'.

3.7.1 Inter-rater agreement

To answer the **RQ1**, we use a statistical method that calculates the inter-rater agreement among the participants' evaluations. Fleiss Kappa (FLEISS, 1971) measure the degree of agreement of the nominal or ordinal assessments made by multiple participants when assessing the same samples. Thus, this measure evaluates the concordance or agreement among multiple raters (FLEISS, 1971). This measure reports a number lower or equal to 1. If the Kappa value is equal to 1, then the raters are in complete agreement, unlike if the agreement is lower than 0 (zero), it means that there is no agreement among raters or it is weaker than expected by chance. Thus, we perform the following actions:

- We collect the evaluations made for each group of tasks and split in two: evaluations aided by decision tree visualization and evaluations done without decision tree. Such evaluations produce a $4 \times N$ matrix, where 4 is the number of tasks from a group and N is the number of raters (participants), as exemplified in Table 7. The "Agrees" is obtained by evaluations marked as "Strong Agree" or "Slight Agree" as long as the evaluations assigned as "Strong Disagree" or "Slight Disagree" was computed as "Disagree". That convergence of two levels of "agree" or "disagree" in one was mandatory to calculate the Fleiss Kappa agreement properly.

- To each matrix of evaluations we compute the Fleiss' Kappa measure in order to obtain the degree of agreement. As a result, we obtain 4 numeric values, each one representing the intersections (column $\times$ row) of the Latin Square design.

- We calculate the average of Kappa measures separated by scenario and compare.

3.7.2 Time spent to accomplish tasks

To answer the **R2**, we measure the time that the participant spend to accomplish tasks with or without decision tree visualization, as a measurement of effort. During the experiment execution, all tasks were time-sensitive. It means that the experiment tool was designed with a time trigger that starts a time count whenever the participant explicitly starts a task and it stops when the participant submits such task. The time interval is stored in seconds and after converted in minutes to improve the readability.

3.7.3 The usefulness of decision trees for decision making

To answer the **RQ3**, we gathered all answers from the open question which asks the participant about what the insights/contributions were given by decision tree in order to support the decision making, considering the context of the task that is under evaluation. All the answers were analyzed by the authors and after we use the coding technique (SEAMAN, 1999) to categorize the detected contributions from qualitative data. From the analysis of the answers, we classified as good contributions to decision making the opinions which stated that the decision tree was useful for the presented evaluation context.

4 Results

In the following sections, we present the study results for all the outcomes brought by data analysis, which supports the research questions defined in our study. In the first subsection (Section 4.1) we present the agreement considering all members who participate in our study, we summarize the evaluations and calculate the agreement using Fleiss Kappa measure. In further sections, we will refine the results from Section 4.1 presenting the agreement among participants by considering different factors related to them, such as their origin (Section 4.2) and experience (Section 4.3). These results help us to answer **RQ1**. In our study, we do our best to describe and discuss only statistically significant Kappa values by considering a *p-value* < 0.05. Despite it was not possible in some cases.

In Section 4.4 we address the effort spent by participants to accomplish the tasks considering both evaluation modes - with and without decision tree visualization, whose result answers the **RQ2**. Finally, the results related to the usefulness of decision tree for decision making, which helps to answer the **RQ3**, is described in Section 4.5. The results and materials used in our empirical study are available at https://git.io/Jv6cM.

4.1 Overall participants' agreement

We summarize all evaluations provided by participants separated by scenarios, i. e., the scenario with decision tree visualization (*DT*) and without decision tree visualization (*No DT*), as we may see in Table 9. Each row corresponds to the task accomplished by the participants, the column *Code Smell* identifies the type of code smell that was suggested by the task. Each cell, under the columns named *Disagreed* and *Agreed*, represents the sum of evaluations made by participants.

The inter-rater agreement is calculated using Fleiss Kappa statistical method and the results are showed in Table 10, with associated *p-Value* ranged from 0.0388 to 0.2710. The agreement values were calculated considering all tasks from the same group, divided by evaluations made considering the scenario with decision tree (*DT*) and without decision tree (*No DT*). In other words, the agreements were calculated based on each cell of the experiment design illustrated previously in Figure 3.

To classify the agreement strength properly, we interpret what is the degree of agreement of a measured sample based on a scale of categories. These categories were proposed by Landis e Koch (LANDIS; KOCH, 1977) and have been adopted by previous works (SCHUMACHER et al., 2010; ZHANG; HALL; BADDOO, 2011) in order to verify the strength of the Kappa value. The scale of categories is shown in Table 8. Therefore, we may see that all measures of Table 10 have a slight agreement strength, regardless of task group or scenario. It confirms the thesis (HOZANO et al., 2018) that the agreement among developers is commonly low.

Kappa	Agreement Strength
< 0	Less than chance agreement
0.01–0.20	Slight agreement
0.21– 0.40	Fair agreement
0.41–0.60	Moderate agreement
0.61–0.80	Substantial agreement
0.81–0.99	Almost perfect agreement

Table 8 – Agreement strength categories

			DT scenario		No DT scenario	
Group	Task ID	Code Smell	Disagreed	Agreed	Disagreed	Agreed
1	3	god class	1	13	2	14
	4	long method	1	13	0	16
	11	long parameter list	0	14	5	11
	7	refused bequest	5	9	4	12
2	8	god class	2	14	1	13
	6	long method	2	14	3	11
	12	long parameter list	7	9	7	7
	5	refused bequest	8	8	5	9

Table 9 – Overall evaluations

	Task Group 1	Task Group 2	Avg
DT	**0.108**	**0.087**	**0.098**
No DT	0.041	0.058	0.049

Table 10 – Overall Fleiss Kappa agreement measures

The showed Kappa measures tell us that the values from the *DT* scenario (highlighted in bold) overcome the *No DT* scenario in both assessed groups, as well as the average considering the two groups. Hence, the *DT* scenario presented a better agreement among participants compared to the *No DT* scenario. Therefore, it means that the scenario in which the participant obtained insights provided by a visualization from a decision tree model (*DT*) has a relative advantage in terms of agreement.

4.2 Developer's agreement considering their origin

During the experiment, we collected information about what is the participant's origin, if he is from Academy, Industry or both. In the analysis described in the previous section - the overall agreement among participants - we evaluated the inter-rater agreement among all participants responsible for evaluating tasks. At this point, in order to investigate the influence of the origin

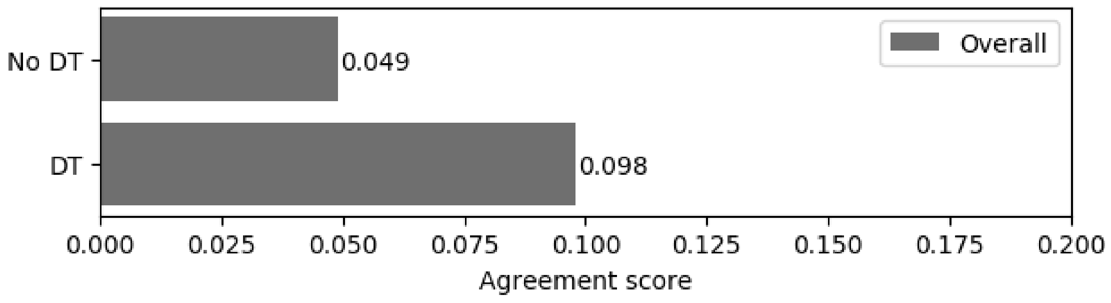


Figure 9 – Average agreement for all participants.

on the agreement among their evaluations, we split the whole set of participants into subgroups. Due to limitations regarding the number of participants, the only subgroup that deserves a deeper investigation in this subsection is the participants from the academy.

Afterward, we calculate the agreement among the evaluations considering scenarios with and without decision tree visualization. Finally, we compare such subgroup agreement with the overall agreement calculated in section 4.1. Such comparison allows us to understand whether the origin of the participant plays some role to increase or decrease the agreement among developers when compared to the whole set of participants.

4.2.1 Participants from academy

In this section, we investigate the agreements reached by participants from the academy, a similar approach as the previous section (Section 4.1). The Table 12 shows the Kappa values and the Table 11 clarifies how was distributed the evaluations through tasks and scenarios. The obtained Kappa measures show us that the agreement strength across task groups and scenarios is classified starting from Poor ($Kappa < 0$) up to Slight ($0.01 < Kappa < 0.2$). The Kappa measures highlighted in bold indicate that the agreement in *No DT* scenario overcomes the *DT*scenario in *Task Group 1* whereas the agreement in *DT* scenario overcomes the *No DT* scenario in *Task Group 2*. Despite the average Kappa agreement indicates the *DT* scenario has a better agreement in general, the agreement reached by the *DT* scenario was not better in both tasks group.

Figure 10 illustrates the average agreement observed in the evaluations done by the participants from the academy and the average agreement from the whole set of participants. In this figure, we use a dark gray bar to represent the overall agreement levels replicated from the average Kappa value of section 4.1. This plot shows that the average agreement provided by evaluations made by participants from the academy is lower than the overall agreement, both in the *DT* scenario and in the *No DT* scenario. Therefore, the average agreement from participants from academy is fewer compared to the overall.

			DT scenario		No DT scenario	
Group	Task ID	Code Smell	Disagreed	Agreed	Disagreed	Agreed
1	3	god class	1	7	1	12
	4	long method	1	7	0	13
	11	long parameter list	0	8	5	8
	7	refused bequest	1	7	3	10
2	8	god class	1	12	1	7
	6	long method	2	11	3	5
	12	long parameter list	6	7	3	5
	5	refused bequest	7	6	2	6

Table 11 – Evaluation table of participants from academy

	Task Group 1	Task Group 2	Avg
DT	-0.103	**0.112**	**0.004**
No DT	**0.082**	-0.082	0

Table 12 – Kappa agreement measures of participants from academy

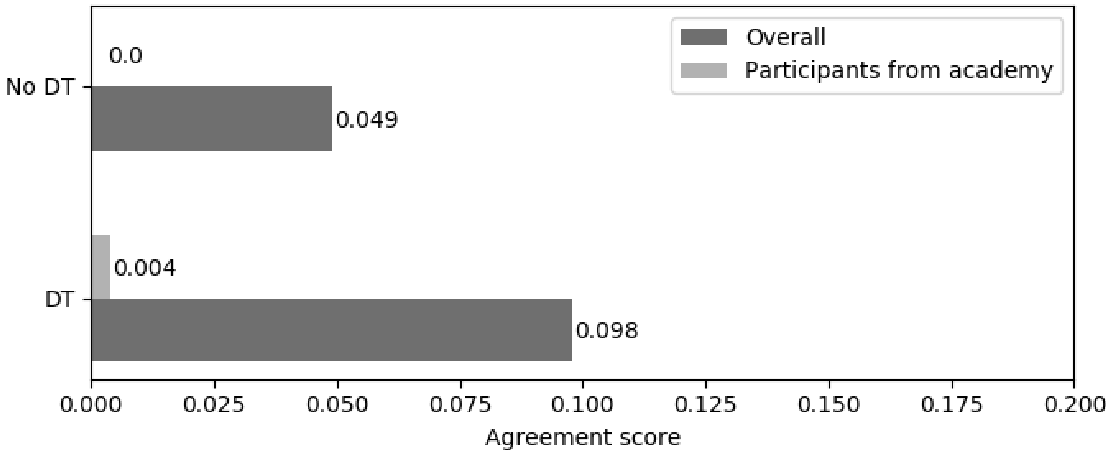


Figure 10 – Average agreement comparison: Overall × participants from academy.

4.3 Agreements by experience

Previously we investigate the inter-rater agreement based on participants' origin. In this turn, we investigate the agreements taking into account the participants' experience. In this way, we analyze the agreement among the more experienced participants in three different profiles: code smell detection, Java programming and development.

During the experiment execution, the developer rated his experience by choosing one of five options: "I do not have any experience", "Very low", "Low", "High" and "Very high". So the experience of each developer is defined by the self-assigned ratings reported by him, so we consider an experienced developer the one who have experience rating above "High". Due to the constraints regarding the number of participants, we also include participants self-rated "Low", it contributed to becoming the results more statistically significant.

4.3.1 Experienced participants on code smell detection

In this section, we extract a subset of participants who are experienced in code smell detection, putting side by side the agreements obtained by the *DT* scenario and the *No DT* scenario.

			DT scenario		No DT scenario	
Group	Task ID	Code Smell	Disagreed	Agreed	Disagreed	Agreed
1	3	god class	0	9	2	7
	4	long method	0	9	0	9
	11	long parameter list	0	9	2	7
	7	refused bequest	4	5	2	7
2	8	god class	1	8	0	9
	6	long method	1	8	2	7
	12	long parameter list	2	7	6	3
	5	refused bequest	3	6	3	6

Table 13 – Evaluation table of experienced participants on code smell detection

	Task Group 1	Task Group 2	Avg
DT	**0.299**	-0.064	**0.116**
No DT	-0.050	**0.183**	0.066

Table 14 – Kappa measure of experienced participants on code smell detection

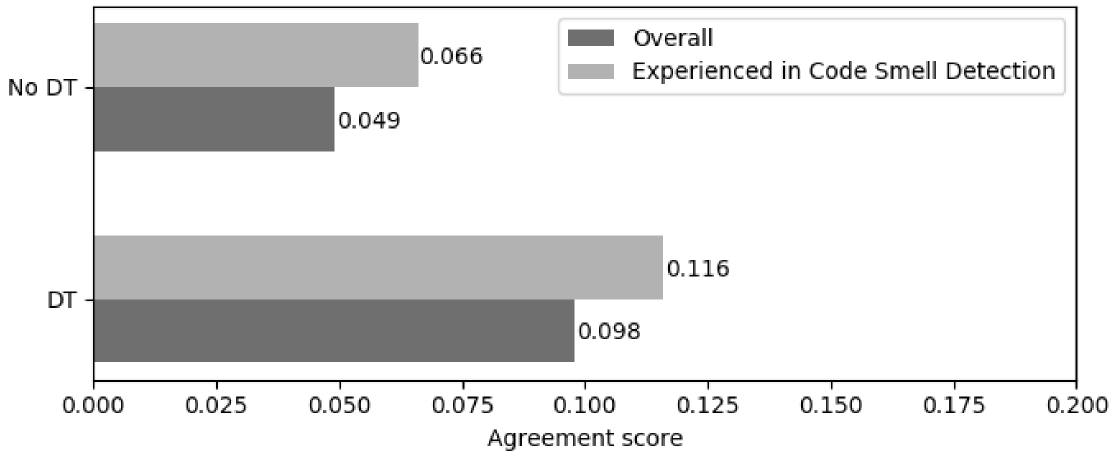


Figure 11 – Average agreement comparison: Overall × Experienced participants in code smell detection

The Table 13 shows the total of assignments divided by scenario and group of tasks, the Table 14 shows the comparison of inter-rater kappa measure among *DT* scenario and *No DT* scenario and the Figure 11 compares the average agreement that was performed by experienced developers on code smell detection in contrast to the average agreement performed by overall participants.

In the Table 14, we can see that the task evaluations carried out by a decision tree visualization (*DT*) have a better agreement on average than the evaluations done without decision tree rules (*No DT*), even though the agreement in *No DT* overcomes the *DT* in Task Group 2. Thus, despite the average of Kappa agreement indicates the *DT* scenario have a better agreement in general, it is not better considering both task group individually.

Comparing the agreement computed through the average of Fleiss Kappa measure with the overall agreement, as shown in Figure 11, the average of agreement from experienced developers on code smell detection performed better than the overall participants. Such result indicates that the expertise of such type of participant in detecting code smells in several software projects brings some uniformity of evaluations, as both scenarios of agreement from experienced developers overcome the overall agreement. Moreover, we may infer that the rules from the decision tree were an important factor for increasing the agreement among developers, thus it may help for accurate reasoning about the smelliness or absence of smell during its detection.

4.3.2 Experienced participants on Java language

In this section we analyze how the experienced participants on Java language agree in code smell detection, computing the Kappa agreements obtained by *DT* scenario and *No DT* scenario.

Table 16 shows the Kappa agreement values and Table 15 shows a detailed view of the distribution of evaluations across tasks and scenarios. Differently of previous participant profiles, the obtained Kappa measures indicate that the agreement ranges from 0.018 up to 0.233, i. e., the table indicates that the agreement strength ranges from *slight agreement* up to *fair*. The Kappa measures highlighted in bold indicate that the agreement in *No DT* scenario overcomes the *DT* scenario in Task Group 2 whereas the agreement in *DT* scenario overcomes the *No DT* scenario in Task Group 1. The average agreement considering both tasks group indicates that *No DT* scenario has, in general, a better agreement than the *DT* scenario.

Comparing the average agreement with the overall agreement, as shown in Figure 12, the agreement average of experienced Java language developers performed better than the overall participants in both scenarios. But, among experienced participants in Java language, unlike the experienced participants on code smell detection, the *DT* scenario had a lower agreement than the *No DT* scenario. So we may infer that the decision tree didn't have such importance on the agreement, taking into account that, when it was omitted, they reached a better agreement.

			DT scenario		No DT scenario	
Group	Task ID	Code Smell	Disagreed	Agreed	Disagreed	Agreed
1	3	god class	0	10	2	10
	4	long method	1	9	0	12
	11	long parameter list	0	10	3	9
	7	refused bequest	4	6	4	8
2	8	god class	1	11	0	10
	6	long method	1	11	2	8
	12	long parameter list	4	8	7	3
	5	refused bequest	5	7	3	7

Table 15 – Evaluation table of experienced developers on Java language

	Task Group 1	Task Group 2	**Avg**
DT	**0.162**	0.046	0.104
No DT	0.018	**0.233**	**0.125**

Table 16 – Kappa measure of experienced Java language participant

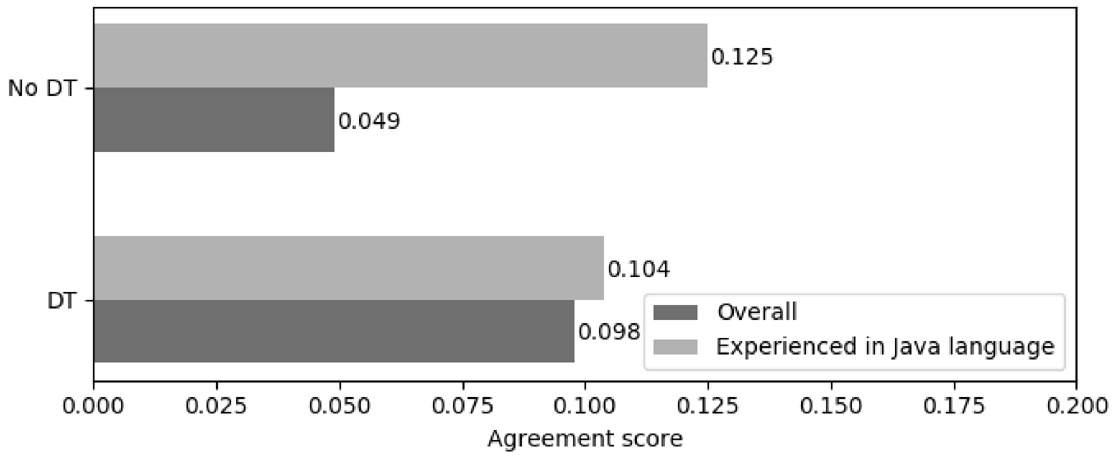


Figure 12 – Average agreement comparison: Overall × Experienced in Java Language

4.3.3 Experienced participants on development

Now we investigate how the experienced developers agree in code smell detection, computing the Kappa measures obtained by *DT* scenario and *No DT* scenario.

The Table 18 shows the Kappa agreement values and the Table 17 shows a detailed view of the distribution of evaluations from tasks and scenarios. From the obtained results of the Kappa measure, we observe that the measures for both scenarios keep the low agreement as usual, with an agreement strength classified as slight. The Kappa measures highlighted in bold indicate that the agreement in *DT* scenario overcomes the *No DT* scenario in Task Group 1 whereas the agreement in *No DT* scenario overcomes the *DT* scenario in Task Group 2.

			DT scenario		No DT scenario	
Group	Task ID	Code Smell	Disagreed	Agreed	Disagreed	Agreed
1	3	god class	1	13	2	12
	4	long method	1	13	0	14
	11	long parameter list	0	14	4	10
	7	refused bequest	5	9	3	11
2	8	god class	2	12	1	13
	6	long method	2	12	3	11
	12	long parameter list	5	9	7	7
	5	refused bequest	7	7	5	9

Table 17 – Evaluation table of experienced participants in development

	Task Group 1	Task Group 2	Avg
DT	**0.108**	0.044	**0.076**
No DT	0.012	**0.058**	0.035

Table 18 – Kappa measure of experienced participants on development

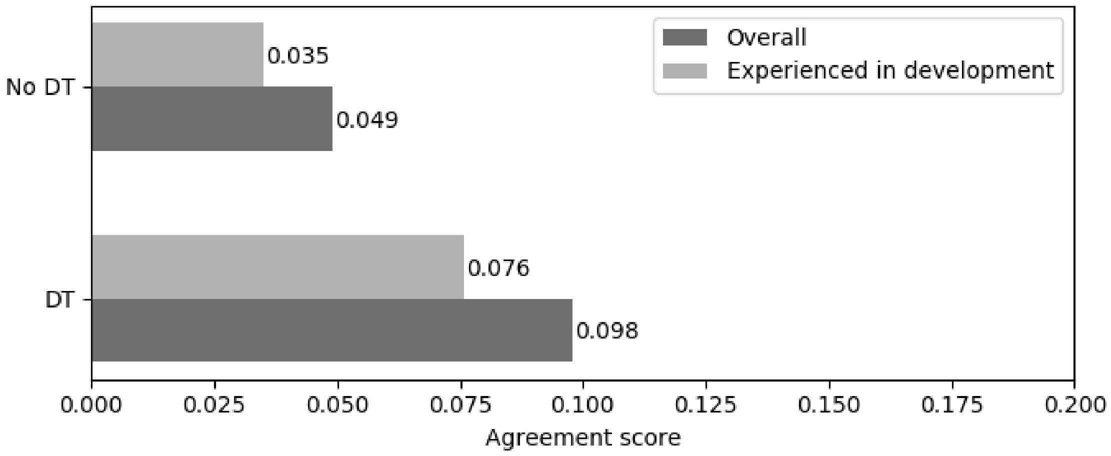


Figure 13 – Average agreement comparison: Overall × Experienced participants in development

Comparing the average agreement of experienced developers with the overall agreement, as shown in Figure 13, the agreement of experienced developers performed lower than the overall participants, considering both scenarios. Considering the agreement among experienced participant in development, the *DT* scenario had a better agreement than the agreement from *No DT* scenario. That result suggest that, for such participant profile, the decision tree visualization have a relative importance on agreement when compared to code inspection without decision tree.

4.4 The effort spent to answer the tasks

For each task of the experiment, we measured the time that the participant spent to accomplish tasks in order to measure effort. Figure 14 illustrates boxplots containing the time distribution evolving the two groups of tasks. We dismissed the outliers to adjust the range of analysis from 0 to 16 minutes (y-axis), increasing the comprehension of the graph.

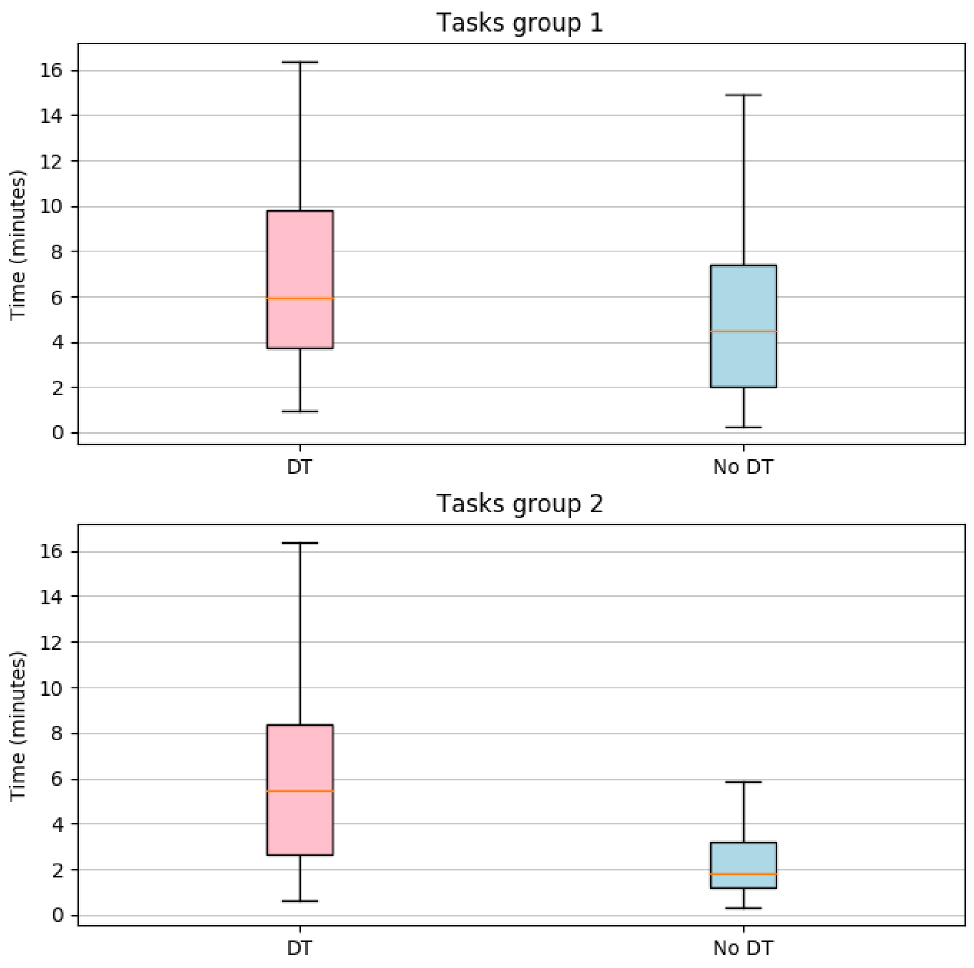


Figure 14 – Time spent to answer the tasks

In the *Tasks group 1*, the time spent to detect code smells aided by decision tree (*DT*) is slightly higher than detecting without it (*No DT*). The slowest time when not aided by the decision tree is faster than the slowest time when detecting code smell aided by a decision tree. The same happens with the quantiles and the minimum.

The *Tasks group 2* presents a higher difference between scenarios *DT* and *No DT* than group 1 regarding the time spent to detect code smells (tasks accomplishment). For this group, the time spent to detect code smells aided by decision tree (*DT*) is considerably higher than detecting without it. The slowest time for *No DT* reaches almost the median of the sample from the opposite scenario *DT*. According to *T-Test*, applied to tasks of Group 2, there is statistically significant evidence (*p-value* = 5.331e-05) that the decision tree visualization doesn't reduce the

Table 19 – Example of statements from participants and the detected insight/contribution

ID	Statement	Detected contribution
491	*"The lack of cohesion in methods is something I would not be able to compute visually. Thus, the decision tree was handy."*	Information provided by metrics
260	*"Made the decision more objective (saves time)."*	Facilitates decision
222	*"Another psychological effect on decision making. A little more confidence in what I had in mind."*	Improved confidence

time to detect code smells, so the time is increased though.

Therefore, for both tasks group, there isn't any evidence that indicates the benefits related to the effort reduction when detecting code smells with a decision tree, i. e., the time spent to detect code smells visualizing the rules provided by a decision tree tends to be equivalent or to overcome the time spent to detect code smells based solely on code inspection.

4.5 The usefulness of decision tree visualization for decision making

Throughout the participation in the experiment, when faced with the scenario with a decision tree, the participant answered an open question that asked him to inform what insight or contribution the decision tree gave him in order to accomplish the task. We analyzed manually each answer to discover patterns that may indicate the importance of the decision tree for decision making. From these answers, we apply a coding technique (SEAMAN, 1999) to recognize these patterns.

During the analysis of the open answers, we reject the vague ones. It means that those answers which look like "Helped me so much" or "It's very helpful" weren't considered. Rather we considered as valid answers those which mention some relevant characteristics like "Number of declarative Lines of Code and Declarative Statements", in this case referring to the metrics within nodes.

Table 19 shows some examples of the statements and the recognized patterns assigned to each, presented in the third column (Detected contribution). Each statement belongs to an answer instance which refers to a task. The ID identifies uniquely the answer given by a participant. The complete list of statements is available in the artifact's repository which accompanies this research[1]. In total, we categorized 3 contributions that the participants wrote in open question.

Figure 15 presents an overview of number of times these detected contributions was observed in total. From the domain of categorized contributions, the most seen was "The information

[1] https://github.com/christianorossini/masterProject

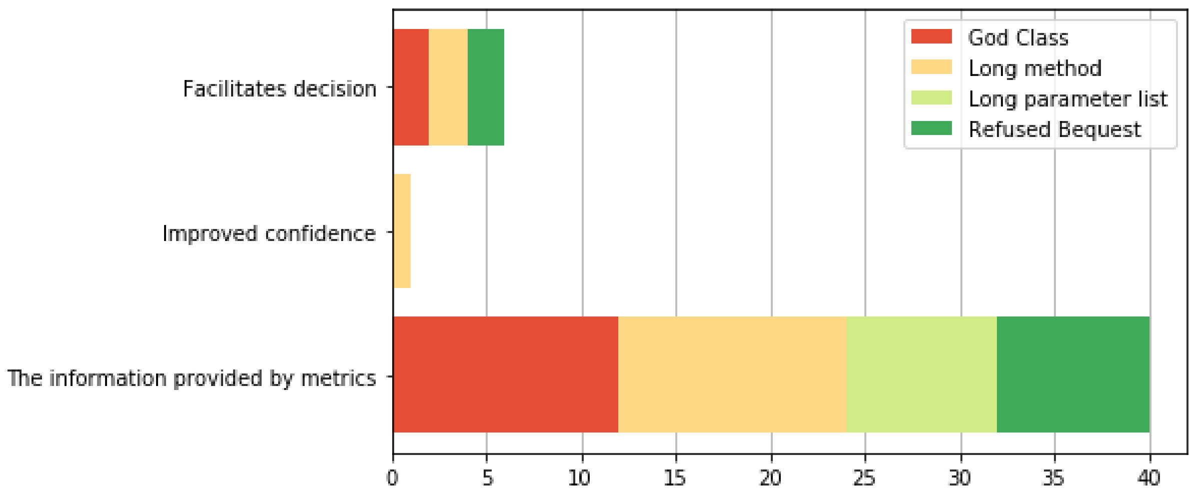


Figure 15 – The frequency of categorized contributions stated by participants

provided by metrics". So amongst the answers provided by participants, around 85% stated that the metrics was a important insight to guide their decisions, followed by the fact that the decision tree facilitates the decision of detecting smell (13%) and the fact that it improves confidence (2%). From the last two detected contributions, we can infer that the metric-based rules were a key factor to facilitate smell detection and improve confidence, respectively. Hence, all the categorized contributions involves the rationale of gaining information by metric-based rules.

When analyzing all the detected contributions in Figure 15 focusing on a type of code smell (that is, a single color), we obtain quantitative information on how many code smells (each one represented by a single decision tree) are present in each detected contribution. For example, the classification models that represent God Class and Long Method are the most evidenced models by those who consider the information provided by metrics as a relevant contribution. Following theses models, on a smaller scale, are the classification models that represent the Long Parameter List and Refused Bequest. Among the available representations of code smell, the Long Parameter List is the only one that is included in all detected contributions.

On the other hand, we also captured the opinions stated by participants who disagree about the usefulness of the decision tree visualization on smell detection, i. e., the participants that stated that certain decision tree didn't contribute to decision making. As previously, we apply a coding technique (SEAMAN, 1999) to recognize patterns obtained from the open question and the Table 21 shows some examples of the statements and the recognized patterns assigned to each one manually. Again, the complete list of statements is available in the artifact's repository[2]. In total, we categorized 7 different patterns mined from the open answers.

In the Figure 16, we present the number of occurrences of the categorized patterns from the type of statements exemplified in Table 21. A large amount of participants states that "The tree rules mismatch the smell", corresponding to 54% of occurrences, followed by "Useless

[2] https://github.com/christianorossini/masterProject

ID	Statement example	Recognized pattern
242	*"The choose by shown decision tree is based only number of lines, this is bad."*	Lack of confidence
192	*"I was a kind of confuse if number of physical lines are related with only the lines of the parent, used in the child class."*	Confusing rules
224	*"None. In fact, the tree does not explicitly present "parameters" as a decision criterion."*	Lack of essencial informations
361	*"couldn't quite understand the "Percent Lack of Cohesion in Methods" metric"*	The metrics understanding
152	*"Looking at the code was better for me than looking at the tree."*	Code analysis was more effective
121	*"Not much. The low number of declarative statements says nothing to me about a long parameter list. From my point of view, a method with no declarative statements and few parameters is perfectly possible."*	Useless informations from DT
488	*"This decision tree gave me the impression of ignoring complexity when deciding on the method's smelliness. Particularly, I disagree with that, because some long methods are not smelly at all. They may be long and, still, easy to read and understand."*	The tree rules mismatch the smell

Table 21 – Examples of statements from participants who disagrees about the usefulness of decision tree visualization for detecting smells.

informations from DT" (14%), "Code analysis was more effective" (10.71%), "Lack of essential informations" (7.1%), "The metrics understanding" (7.1%), "Lack of confidence" (3.5%) and "Confusing rules" (3.5%). Unlike the answers that characterized the classification models as useful for detecting code smells (Figure 15), in this case the classification model that represents the God Class has a minor number of occurrences in relation to the other smells. For instance, in the answer categorized as "The tree rules mismatch the smell", the classification model which represents the god class has the lowest number of occurrences. This indicates that the rules contained in the decision tree that represents a god class, basically composed of metrics of size and complexity, proved to be very useful for decision making. The classification model that represents the Long Parameter List has the majority of opinions that point to the mismatch of the tree rules and the smell that is under evaluation. Besides, it is the only smell representation that is characterized as it lacks essential information for decision making.

In order to summarize the number of opinions categorized as useful or useless, we plot the chart presented in Figure 17. Through the figure we can see, by the number of ocurrences, that the decision tree (or classification model) which represents the God Class was the most qualified

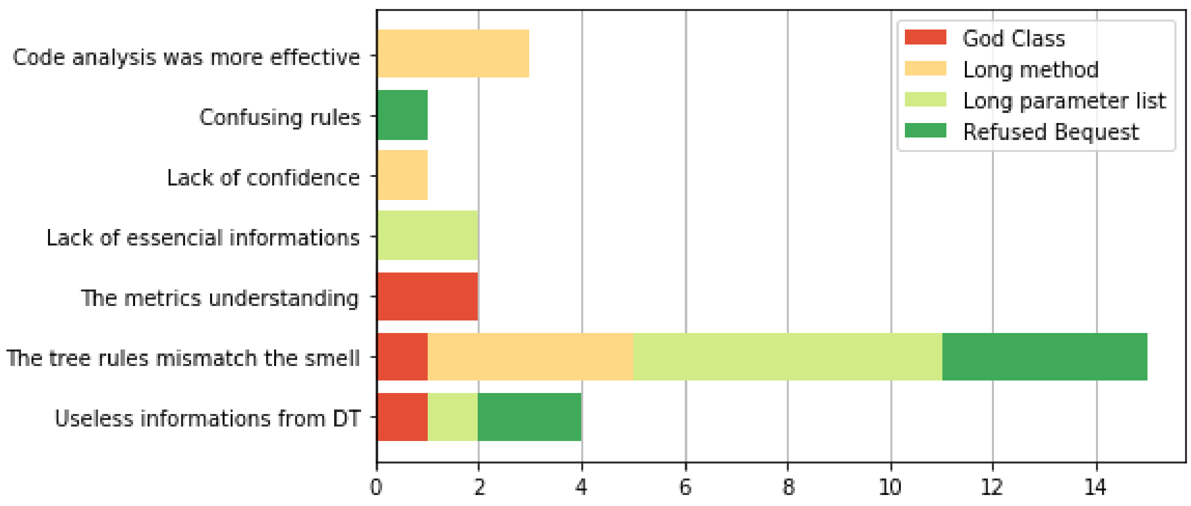


Figure 16 – The frequency of categorized patterns from statements which considers the decision tree not useful to the context

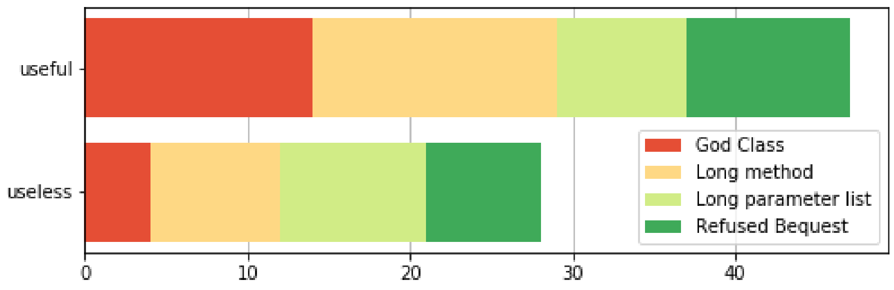


Figure 17 – The sum of all opinions about the usefulness (contributions) of the classification models (decision tree)

model in terms of the contribution to decision making among the models used, followed by the long method. They have more qualifications defined as 'useful' and less qualifications defined as 'useless'. Therefore, in general, we can conclude that, from the perspective of the participants, there are more answers which indicate that the decision trees gives important insights to the code smell detection process ('Useful' - 63%) than the opposite ('Useless' - 37%).

5 Discussions

In this section we discuss the results aiming at answering the research questions.

5.1 RQ1: Do an aided approach of code smell detection - using a tree classifier representation – influence on agreement among developers?

Based on the results obtained in the experiments conducted to respond to RQ1, considering the total number of participants involved, there is evidence that code smells detection aided by a decision tree leads to a relative improvement of agreement in relation to code smell detection based solely on pure and simple code analysis, although in both scenarios the resulting Kappa measure was considered slight regarding the agreement strength scale shown in Table 8. However, this low agreement (less than 0.1) agrees with past publications (HOZANO et al., 2018) when it was stated that by default the agreement between developers is predominantly low, regardless of the analyzed smell.

Soon after, we segregated the total number of participants to investigate the agreement considering different experiences and backgrounds. In this case, 4 different types of user profiles were analyzed, including participants originating from the academy, experienced participants in code smell detection, experienced in java language and experienced in development. Considering the average agreement obtained in each profile, the inspection of code aided by decision tree favored the agreement for all user profiles except for the profile of experienced participants in the Java language. However, considering each group of tasks individually, even for the profiles in which the decision tree favored the agreement, the agreement of the *DT* scenario did not take advantage over the *No DT* in all comparisons.

The profiles of participants that stood out were the participants experienced in code smell detection and those experienced in Java language. The resulting agreement obtained by both exceeded the agreement considering all participants (overall agreement), both for the scenario with decision tree (*DT*) and for the scenario without decision tree (*No DT*). Nevertheless, the *DT* scenario was more relevant on the agreement for the most experienced participants in code smell detection, i. e., the analysis of the metric-based rules in the decision tree favored the agreement for such type of participant.

5.2 RQ2: How much effort do developer spend during code smell detection with a tree classifier support?

In relation to RQ2, the detection of code smell aided by decision tree did not decrease in time compared to detection based only on code inspection. For both groups of tasks, there isn't any evidence that indicates the benefits related to the effort reduction when detecting code smells with decision tree, i. e., the time spent to detect smells with decision tree tend to be equivalent or to overcome the time spent to detect code smells based only on code analysis. This result can be considered a non-fortuitous case, given that the participant had to observe other screen elements during task evaluation to obtain more information on the metrics presented, as is the case of the metrics glossary presented in the environment represented in Figure 6. In addition, as discussed in section 4.5, for a good portion of the open answers given concerned to the decision tree contribution for the task accomplishment, many generated models were endowed with inappropriate information, the rules contained in the tree did not exactly match the smell presented, essential information regarding the smell was omitted and the bad understanding of the metrics presented by certain models. These are evidence that made the experience of viewing the decision tree costly from an effort perspective.

5.3 RQ3: How useful is the decision tree visualization for decision making?

Regarding RQ3, our experiment suggests that the decision trees used to support code smell detection are useful to the developer. About 63% of the participants, who completed the experiment and had their responses analyzed, considered the decision trees useful for supporting smell detection. From this percentage, about 85% opined that the information provided by the presented software metrics was important for decision making. On the other hand, 37% of the participants considered that the decision trees were not useful for decision making and, from this percentage, the majority considered that the presented decision tree mismatch the code smell under evaluation. The classification models that represent the detection rules of a God Class and a Long method were the best-evaluated models in terms of usefulness, i. e., models that offered good contributions to decision making. On the other hand, the models that represented the Long Parameter List and the Refused Bequest performed worse in terms of usefulness.

From these findings, for future works, we are going to consider reviewing the independent variables used for training the classification algorithms in order to generate models more refined to the chosen types of code smells, since the main complaints of the participants is related to useless information that is presented by the decision tree, the difficulty of understanding the metrics used and the lack of essential information related to the smell under evaluation.

6 Related Work

To the best of our knowledge, this is the first work that utilizes a comprehensible machine learning classifier to aid developers to detect code smell and how this approach deals with the agreement among them. However, there are some studies that investigate the agreement among developers when detecting code smells and the factors that influence their evaluations. The decision tree algorithm, which is an important part of this study, was also included in the literature addressing the effectiveness of the automatic code smell detection.

In (MANTYLA, 2005), the author presented a study that investigated the agreement among developers about 3 types of code smells: Long Method, Long Parameter List and Feature Envy. Their research results indicated a low agreement regarding Feature Envy, but a good agreement on Long Method and Long Parameter List evaluations. According to the authors, these smells are easier to understand and detect, becoming evaluations more similar. The results showed also that the number of lines of code (MLOC) and the number of parameters in a method (NPARAM) could be used, respectively, as predictors of the evaluations related to Long Method and Long Parameter List. Even though the results of the study reported interesting findings concerning how similar the developers detect smells in code, the study analyzed the agreement among developers on evaluating only 3 smell types.

Mika Mäntylä et al. (MÄNTYLÄ; LASSENIUS, 2006) expanded the previous research by involving 12 developers analyzing code snippets of 23 types of code smells. The author presented a definition and an example of each type of code smell to the developers as well as the study asked them to evaluate some code snippets related to the analyzed code smell. The participants of the empirical study evaluated the presence or absence of each smell across several code snippets. The authors noticed a perfect agreement of 5 developers in only 1 of 46 analyzed code snippets presented in their publication, regarding the analysis of a Long Method smell. Otherwise, the authors reported a high disagreement in the evaluations of the Switch Statement, Inappropriate Intimacy and Message Chains smells. Although the interesting findings regarding the agreement among developers, the authors stated that the data used in the experiment was not enough to make the results more reliable. Furthermore, the reduced number of analyzed code snippets and developers involved in the experiment made it harder to check the statistical significance of the agreement among the developers.

Hozano et al. (HOZANO et al., 2018) presented a study aimed to investigate how similar the developers detect smells in code and identify possible factors that influence in the agreement of their evaluations. They made an empirical study involving 75 developers evaluating 15 types of code smells into code snippets of 5 open source projects. In total, more than 2700 evaluations of code snippets from real software projects were performed. From the collected evaluations, they identified a great disagreement among the developers' evaluations considering the code smells investigated in the study. The results evidenced, in general, that developers detect code smells in different ways. In contrast, developers that followed the same heuristic to evaluate a given code smell presented a consistent agreement when analyzed separately, so the authors stated that the heuristics plays an important role to determine the agreement among the developers.

The effectiveness of the decision tree on code smell detection was addressed by Amorim et al. (AMORIM et al., 2015). In their work, they presented a practical experience report that evaluates the use of Decision Tree classification algorithm to recognize code smells in different software projects. They aimed at contributing to the state-of-the-art on code smells detection by analyzing the use of the Decision Tree algorithm to detect code smells automatically. They evaluated the performance of the Decision Tree to detect 12 types of different smells across 4 open source projects. They compared the precision, recall and F-measure results with other machine learning techniques, such as SVM and Bayesian Beliefs Networks, as well as the results generated by general rules-based approaches. For both cases, the decision tree reaches better performance.

7 Threats to Validity

In this section, we discuss the threats to validity, including internal, external, construct, and conclusion validity.

Conclusion validity. Threats in this category impact the relationship between treatment and outcome. The biggest threat to the conclusion validity was the statistical significance obtained in the calculation of agreement using the technique proposed by Joseph L. Fleiss (FLEISS, 1971). Given the time constraints to conduct the research, it was not possible to prospect a sufficient number of subjects (participants) in order to create a satisfactory database, although every effort was made to publicize and expand the number of participants to the experiment. Thus, despite the mitigation efforts, for a significant part of the agreement scores established in **RQ1**, the expected statistical significance was not achieved.

Internal validity. Threats refer to the factors that may have influenced our study. First, our concern when designing the experiment was to make it easy to use and that its duration occurs in the shortest possible time, to prevent the experiment from being boring. Although there was a concern to mitigate this issue, including a brief pilot experimentation, the duration of each participation was approximately 60 minutes on average. Given the amount of time that or experiment lasts, we can assume that, after a certain point, the participants felt tired or unmotivated, directly affecting the quality of the information provided.

Second, we are aware that social interactions can be another threatening factor to the validity of the results: due to the fact that the experiment execution was not controlled, the participants who performed the experiment had time to interact with the applicants. Therefore, we assume that this social interaction can cause changes in the behavior of the candidate participants during the execution of the experiment. However, we detected that this situation is a necessary trade-off: we needed to create a web experiment that could reach the largest possible number of software engineering professionals in exchange for exposure to the threats resulting from the interaction between people.

External Validity. The threats here concern the degree to which the findings can be generalized to the wider classes of subjects. Our study is based on an experiment published on the internet to reach as many volunteers as possible. Due to the number of participants that committed our research instrument, less than expected, we are aware that our research may be subject to the problem of representativeness, which may affect the generalization of the results. Possible mitigation to this problem would be to substantially increase the number of participants and make our experiment reach different profiles of professionals, which we aim for further works.

As for the number of types of code smells, we chose 4 out of 20 possible types cataloged by Fowler (FOWLER et al., 1999). This small sample was necessary to make the experiment executable within a reasonable time. To mitigate possible threats to the generalization validity, we focus our choice on how developers recognize each type of code smell in terms of perceptiveness and level of agreement pointed out by previous studies (PALOMBA et al., 2014; HOZANO et al., 2018), going from the most perceived smell to the least as a criterion of choice.

Construct Validity. This is an uncontrolled experiment, which raises some concerns. The most important is that which affects the dedicated effort to solve tasks, as discussed in RQ2. Without the effective control of the participants' activities, it is not possible to accurately measure the time actually spent on completing the activities. As a result, eventual dispersions by the participants may overestimate the completion time of the activities, impacting the result of RQ2.

8 Conclusion and Future Works

In this research, an empirical study was presented to investigate the agreement among developers when they are aided by a comprehensible classification model to support decision on smell detection. Following the same way, the study investigates the required effort to perform evaluations by visualizing the classifier model and the usefulness of our approach by participant perspective.

To answer the raised questions, we made an empirical study involving 30 participants who evaluate 8 tasks, divided into 2 groups, with source codes potentially affected by 1 of 4 selected types of smell. Each group of tasks exposed the developer to a source code with different perspectives, so one shows the classification model that predicted the code as "smelly" whereas another group doesn't. The experiment was carried out through a web-based app developed exclusively to collect the evaluations performed by participants. In total, more than 230 evaluations of code snippets from real software projects were performed.

Throughout our work, we got evidence that code smells detection aided by a decision tree leads to a relative improvement of agreement in relation to detections based solely on pure and simple code analysis, although in both scenarios the resulting Kappa measure was considered slight regarding the agreement strength. After detaching different profiles of participants from the whole set of participants, we discovered groups that behave distinctly. The agreement on the detections aided by decision tree performed better with experienced code smell detection participants, which brings the evidence that such profile got the best out of our approach. As for the effort, the detection of code smell aided by decision tree did not decrease in time compared to detection based only on code inspection. For both groups of tasks, there isn't any evidence that indicates the benefits related to the effort reduction when detecting code smells with decision tree, i. e., the time spent to detect smells aided by decision tree tend to be equivalent or to be highest than the time spent to detect code smells based solely in code inspection. Finally, based on the answers provided by the majority of participants, our experiment suggests that the decision trees used to support code smell detection are useful to the developer in terms of insights to decision making.

As future work, we intend to use the feedback from participants to improve the generated models and made it more useful to detecting smells. Moreover, in the next works we intend to cover a wider range of code smells and study new agreements covering these smells.

Bibliography

ABBES, M. et al. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: IEEE. *Software maintenance and reengineering (CSMR), 2011 15th European conference on*. [S.l.], 2011. p. 181–190. Citado 2 vezes nas páginas 12 and 16.

AMORIM, L. et al. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In: IEEE. *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.], 2015. p. 261–269. Citado 2 vezes nas páginas 13 and 49.

AZEEM, M. I. et al. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, Elsevier, 2019. Citado 3 vezes nas páginas 12, 13, and 21.

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: IEEE. *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.], 2013. p. 712–721. Citado na página 12.

BAEZA-YATES, R.; RIBEIRO-NETO, B. et al. *Modern information retrieval*. [S.l.]: ACM press New York, 1999. v. 463. Citado na página 22.

BOX, G. E.; HUNTER, J. S.; HUNTER, W. G. Statistics for experimenters. In: *Wiley Series in Probability and Statistics*. [S.l.]: Wiley Hoboken, NJ, USA, 2005. Citado na página 24.

FENTON, N.; BIEMAN, J. *Software metrics: a rigorous and practical approach*. [S.l.]: CRC press, 2014. Citado na página 21.

FLEISS, J. L. Measuring nominal scale agreement among many raters. *Psychological bulletin*, American Psychological Association, v. 76, n. 5, p. 378, 1971. Citado 2 vezes nas páginas 31 and 50.

FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, v. 11, n. 2, p. 5–1, 2012. Citado na página 12.

FONTANA, F. A. et al. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, Springer, v. 21, n. 3, p. 1143–1191, 2016. Citado na página 21.

FONTANA, F. A. et al. Code smell detection: Towards a machine learning-based approach. In: IEEE. *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. [S.l.], 2013. p. 396–399. Citado na página 21.

FOWLER, M. et al. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 1999. Citado 6 vezes nas páginas 9, 12, 16, 20, 27, and 50.

FREITAS, A. A. Comprehensible classification models: a position paper. *ACM SIGKDD explorations newsletter*, ACM, v. 15, n. 1, p. 1–10, 2014. Citado 2 vezes nas páginas 13 and 22.

GOUSIOS, G. et al. Work practices and challenges in pull-based development: the integrator's perspective. In: IEEE PRESS. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. [S.l.], 2015. p. 358–368. Citado na página 12.

GRUS, J. *Data science from scratch: first principles with python*. [S.l.]: O'Reilly Media, 2019. Citado 2 vezes nas páginas 8 and 17.

GUIDOTTI, R. et al. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)*, ACM, v. 51, n. 5, p. 93, 2018. Citado na página 22.

HOZANO, M. et al. Evaluating the Accuracy of Machine Learning Algorithms on Detecting Code Smells for Different Developers. In: *Proceedings of the 19th International Conference on Enterprise Information Systems*. SCITEPRESS - Science and Technology Publications, 2017. p. 474–482. ISBN 978-989-758-247-9. Disponível em: <http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006338804740482>. Citado na página 21.

HOZANO, M. et al. Are you smelling it? investigating how similar developers detect code smells. *Information and Software Technology*, Elsevier, v. 93, p. 130–146, 2018. Citado 10 vezes nas páginas 12, 13, 19, 20, 22, 27, 33, 46, 48, and 51.

LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. *biometrics*, JSTOR, p. 159–174, 1977. Citado na página 33.

LUŠTREK, M. et al. What makes classification trees comprehensible? *Expert Systems with Applications*, Elsevier, v. 62, p. 333–346, 2016. Citado na página 22.

MANTYLA, M. V. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In: IEEE. *2005 International Symposium on Empirical Software Engineering, 2005.* [S.l.], 2005. p. 10–pp. Citado na página 48.

MÄNTYLÄ, M. V.; LASSENIUS, C. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, Springer, v. 11, n. 3, p. 395–431, 2006. Citado na página 48.

NUCCI, D. D. et al. Detecting code smells using machine learning techniques: are we there yet? In: IEEE. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2018. p. 612–621. Citado na página 21.

NUÑEZ-VARELA, A. S. et al. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, Elsevier, v. 128, p. 164–197, 2017. Citado na página 21.

OIZUMI, W. et al. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: IEEE. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. [S.l.], 2016. p. 440–451. Citado na página 16.

OPPENHEIM, A. N. *Questionnaire design, interviewing and attitude measurement*. [S.l.]: Bloomsbury Publishing, 2000. Citado na página 28.

PALOMBA, F. et al. Do they really smell bad? a study on developers' perception of bad code smells. In: IEEE. *2014 IEEE International Conference on Software Maintenance and Evolution*. [S.l.], 2014. p. 101–110. Citado 7 vezes nas páginas 12, 13, 20, 26, 27, 29, and 51.

PALOMBA, F. et al. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, Springer, v. 23, n. 3, p. 1188–1221, 2018. Citado na página 20.

ROKACH, L.; MAIMON, O. Z. *Data mining with decision trees: theory and applications*. [S.l.]: World scientific, 2008. v. 69. Citado 4 vezes nas páginas 16, 17, 18, and 22.

SCHUMACHER, J. et al. Building empirical support for automated code smell detection. In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2010. p. 1–10. Citado na página 33.

SEAMAN, C. B. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, IEEE, v. 25, n. 4, p. 557–572, 1999. Citado 3 vezes nas páginas 32, 42, and 43.

SILVA, M. C. O.; VALENTE, M. T.; TERRA, R. Does technical debt lead to the rejection of pull requests? *arXiv preprint arXiv:1604.01450*, 2016. Citado na página 12.

TAIBI, D.; JANES, A.; LENARDUZZI, V. How developers perceive smells in source code: A replicated study. *Information and Software Technology*, Elsevier, v. 92, p. 223–235, 2017. Citado na página 13.

VANDERPLAS, J. *Python data science handbook: Essential tools for working with data*. [S.l.]: " O'Reilly Media, Inc.", 2016. Citado 2 vezes nas páginas 16 and 17.

YAMASHITA, A.; MOONEN, L. Do developers care about code smells? an exploratory survey. In: IEEE. *2013 20th Working Conference on Reverse Engineering (WCRE)*. [S.l.], 2013. p. 242–251. Citado na página 12.

YAMASHITA, A.; MOONEN, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: IEEE. *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.], 2013. p. 682–691. Citado na página 12.

ZHANG, M.; HALL, T.; BADDOO, N. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice*, Wiley Online Library, v. 23, n. 3, p. 179–202, 2011. Citado na página 33.

Appendix

APPENDIX A – Software metrics

A.1 List of class-level software metrics

Table 22 show the entire list of class-level metrics[1] used to collect independent variables.

metric name	description
Average Cyclomatic Complexity	Average cyclomatic complexity for all nested functions or methods.
Avg Number of Blank Lines	Average number of blank lines for all nested functions or methods.
Avg Number of Lines	Average number of lines for all nested functions or methods.
Avg Number of Lines of Code	Average number of lines containing source code for all nested functions or methods.
Avg Number of Lines with Comments	Average number of lines containing comment for all nested functions or methods.
Base Classes	Number of immediate base classes
Class Methods	Number of class methods
Class Variables	Number of class variables
Comment to Code Ratio	Ratio of number of comment lines to number of code lines.
Coupling Between Objects	The Coupling Between Object Classes (CBO) measure for a class is a count of the number of other classes to which it is coupled. Class A is coupled to class B if class A uses a type, data, or member from class B. Any number of couplings to a given class counts as 1 towards the metric total.
Declarative Statements	Number of declarative statements
Depth of Inheritance Tree (DIT)	The depth of a class within the inheritance hierarchy is the maximum number of nodes from the class node to the root of the inheritance tree. The root node has a DIT of 0. The deeper within the hierarchy, the more methods the class can inherit, increasing its complexity.
Executable Lines of Code	Number of lines containing executable source code.
Instance methods	Number of instance methods
Instance Variables	Number of instance variables - variables defined in a class that are only accessable through an object of that class
Lines with Comments	Number of lines containing comment.
Lines with Source Code	The number of lines that contain source code.
Local Default Visibility Methods	Number of local default visibility methods
Max Cyclomatic Complexity	Maximum cyclomatic complexity of all nested functions or methods.
Maximum nesting level	Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function.
Number of blank lines	Number of blank lines
Number of Children	Number of immediate subclasses. (i.e. the number of classes one level down the inheritance tree from this class).
Number of declarative Lines of Code	Number of lines containing declarative source code. Note that a line can be declarative and executable. Example: int i = 0;
Number of Local Methods (WMC)	Number of local (not inherited) methods.

[1] https://scitools.com/feature/metrics/

metric name	**description**
Number of Methods	Number of methods, including inherited ones.
Physical lines	Number of physical lines.
Percent Lack of Cohesion in Methods	100% minus average cohesion for class data members. Calculates what percentage of class methods use a given class instance variable. A lower percentage means higher cohesion between class data and methods.
Private Methods	Number of local (not inherited) private methods.
Protected Methods	Number of local protected methods.
Public Methods	Number of public methods. Only counts local (not inherited) methods.
Statements	Number of declarative plus executable statements.
Sum of Cyclomatic Complexity	Sum of cyclomatic complexity of all nested functions or methods.

Table 22 – List of class-level metrics

A.2 List of method-level software metrics

Table 23 show the entire list of method-level metrics[2] used to collect independent variables.

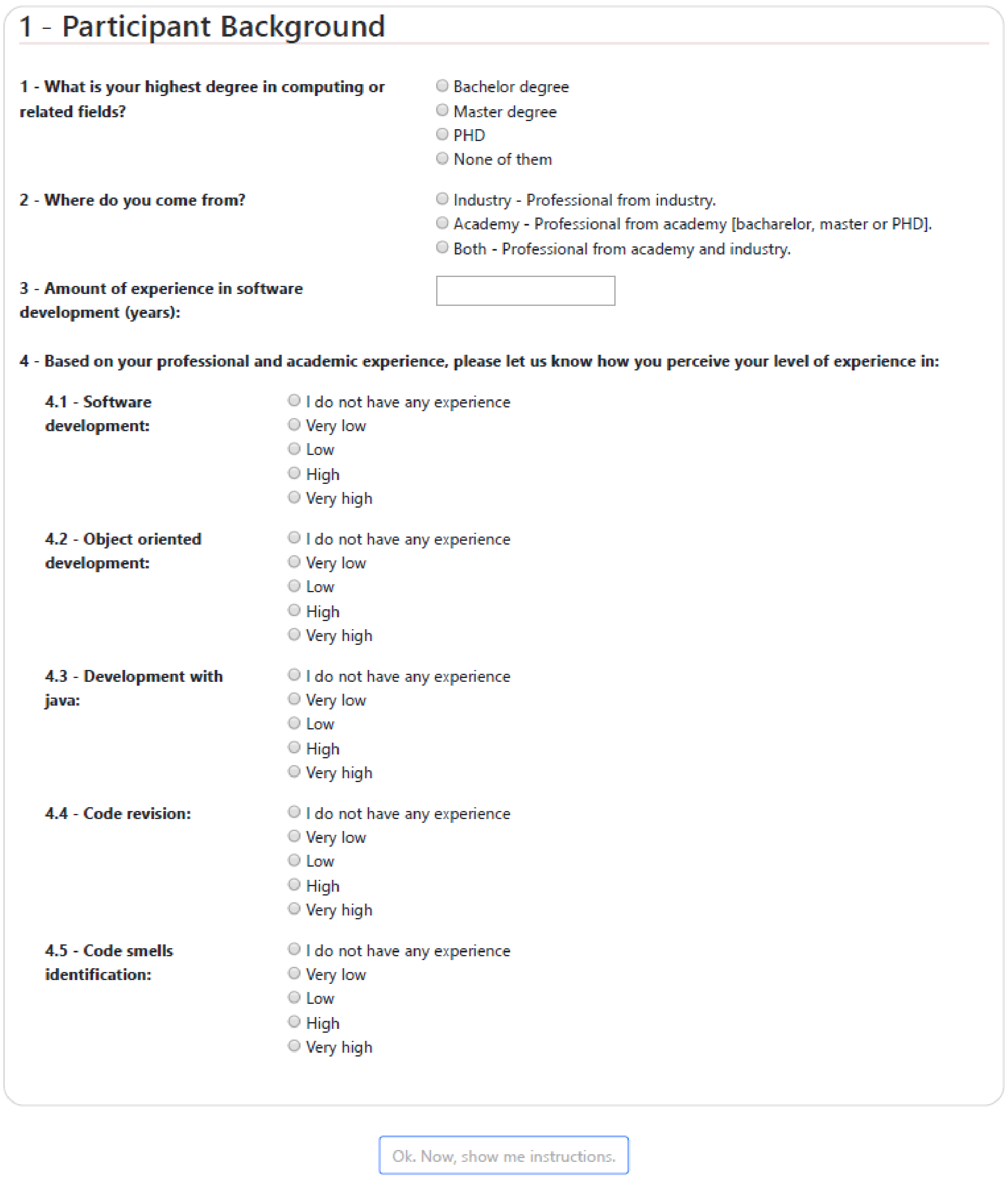
metric name	**description**
Number of inputs	The number of inputs a function uses plus the number of unique subprograms calling the function. Inputs include parameters and global variables that are used in the function.
Number of blank lines	Number of blank lines.
Source Lines of Code	The number of lines that contain source code. A line can contain source and a comment and thus count towards multiple metrics.
Number of declarative lines of Code	Number of lines containing declarative source code. A line can be declarative and executable. Example: int i =0;
Executable Lines of Code	Number of lines containing executable source code.
Lines with Comments	Number of lines containing comment.
Number of outputs (FANOUT)	The number of outputs that are SET. This can be parameters or global variables.
Paths	Number of unique paths though a body of code, not counting abnormal exits or gotos.
Statements	Number of declarative plus executable statements.
Declarative Statements	Number of declarative statements.
Executable Statements	Number of executable statements.
Cyclomatic Complexity	The cyclomatic complexity of any structured program with only one entrance point and one exit point is equal to the number of decision points contained in that program plus one. It counts the keywords for decision points (FOR, WHILE, etc) and then adds 1.
Maximum nesting level	Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function.

[2] https://scitools.com/feature/metrics/

metric name	**description**
Comment to Code Ratio	Ratio of number of comment lines to number of code lines. Some lines are both code and comment, so this could easily yield percentages higher than 100.

Table 23 – List of method-level metrics

APPENDIX B – Participant background questionnaire



1 - Participant Background

1 - What is your highest degree in computing or related fields?

- Bachelor degree
- Master degree
- PHD
- None of them

2 - Where do you come from?

- Industry - Professional from industry.
- Academy - Professional from academy [bacharelor, master or PHD].
- Both - Professional from academy and industry.

3 - Amount of experience in software development (years):

4 - Based on your professional and academic experience, please let us know how you perceive your level of experience in:

4.1 - Software development:

- I do not have any experience
- Very low
- Low
- High
- Very high

4.2 - Object oriented development:

- I do not have any experience
- Very low
- Low
- High
- Very high

4.3 - Development with java:

- I do not have any experience
- Very low
- Low
- High
- Very high

4.4 - Code revision:

- I do not have any experience
- Very low
- Low
- High
- Very high

4.5 - Code smells identification:

- I do not have any experience
- Very low
- Low
- High
- Very high