Smells are sensitive to developers! On the efficiency of (un)guided customized detection

Mario Hozano[*], Alessandro Garcia[†], Nuno Antunes[‡], Baldoino Fonseca[§] and Evandro Costa[§]
[*]Department of Computing Systems, UFCG, Paraíba - Brazil,
Núcleo de Ciências Exatas, UFAL, Brazil - Email: hozano@arapiraca.ufal.br
[†]Opus Research Group – LES, Informatics Dept., PUC-Rio, Brazil. Email: afgarcia@inf.puc-rio.br
[‡]CISUC, Department of Informatics Engineering, University of Coimbra, Portugal. Email: nmsa@dei.uc.pt
[§]Computing Institute, UFAL, Brazil. Emails: {baldoino,evandro}@ic.ufal.br

***Abstract*—Code smells indicate poor implementation choices that may hinder program comprehension and maintenance. Their informal definition allows developers to follow different heuristics to detect smells in their projects. Machine learning has been used to customize smell detection according to the developer's perception. However, such customization is not guided (i.e. constrained) to consider alternative heuristics used by developers when detecting smells. As a result, their customization might not be efficient, requiring a considerable effort to reach high effectiveness. In fact, there is no empirical knowledge yet about the efficiency of such *unguided* approaches for supporting developer-sensitive smell detection. This paper presents *Histrategy*, a guided customization technique to improve the efficiency on smell detection. *Histrategy* considers a limited set of detection strategies, produced from different detection heuristics, as input of a customization process. The output of the customization process consists of a detection strategy tailored to each developer. The technique was evaluated in an experimental study with 48 developers and four types of code smells. The results showed that *Histrategy* is able to outperform six widely adopted machine learning algorithms – used in unguided approaches – both in effectiveness and efficiency. It was also confirmed that most developers benefit from using alternative heuristics to: (i) build their tailored detection strategies, and (ii) achieve efficient smell detection.**

I. Introduction

Code smells indicate poor implementation choices that usually worsen software comprehensibility [1] and maintainability [2]. The growing incidence of code smells is also an indicator of design degradation [3] and fault proneness [4]. Thus, code smell detection is an elementary technique to improve software longevity. Several studies show the smelliness of the source code is often subjective and sensitive to each developer perception [5], [6], [7], [8]. It is the developer who often recognizes and removes anomalous structures from existing programs and, therefore, it is important that smells are identified in accordance with the developer's perception.

A smell *detection strategy* is composed of metrics, thresholds and logical operators. Smell detection strategies seem to achieve acceptable accuracy in experimental scenarios [9], [10], [11], [12], [13], [14], [15]. However, these studies **rely on universal, non-customized strategies**. Unfortunately, the use of non-customized strategies is known to be inefficient when applied in real project settings [16]. Various studies report that, in general, different developers present low agreements on the characteristics of each code smell [6], [7], [8]. They diverge in terms of the employed detection strategies, such as the particular metrics and thresholds associated with each strategy. Thus, developers often need to customize strategies in order to detect smell instances to their particular context.

The customization process consists of two main steps: (1) the formulation of the underlying *heuristic* to detect each smell type – e.g. a *God Class* is a class with high coupling and many non-cohesive methods, and (2) the refinement of this high-level heuristic in terms of a concrete detection strategy, including its specific metrics, thresholds and logical operators [17], [9], [10], [11]. These two steps are sensitive to the experience of each developer [5], [6], [7], [8]. Thus, a key challenge on smell detection is how to achieve high efficiency in the customization process, i.e. achieve high effectiveness on developer-sensitive smell detection while reducing the customization burden on developers. An industrial study [18] revealed developers might spent days of work in order to customize smell detection strategies with acceptable accuracy.

Existing tool support is limited to the adaptation of existing strategies by modifying thresholds [19], [20], [21], [22], [15] or the manual definition of their own smell detection strategies, requiring extra knowledge and effort [13]. The lack of support for customizing smell detection may discourage developers to identify and refactor smelly code in their programs [16], [18]. The success of smell detection is directly dependent on the efficiency of an automated customization process [16].

Machine Learning algorithms [23] are used to somehow automate part of the customization process, without asking the developer to indicate how the strategy must be defined. These algorithms use examples of code smells previously reported by the developer in order to learn how he detects such smells. However, there is still no knowledge how these approaches are customizable and sensitive to each developer perception. In fact, previous studies [24], [23] do not investigate the efficiency of these approaches from the perspective of each developer. Furthermore, the required high number of examples might introduce an unfeasible additional effort.

This paper presents ***Histrategy*, a technique for *guided customization* of smell detection strategies**. *Histrategy* is guided by a set of previously defined detection heuristics. First, *Histrategy* uses the heuristics to produce a restricted set of detection strategies. Then, it customizes such strategies based

978-1-5386-0535-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPC.2017.32

on the individual perception of each developer. *Histrategy* keeps all the produced strategies, and after converging to the developer's perception, it evaluates and selects the best one to be used. **Our hypothesis is that the *guided customization* can reduce the number of examples required to produce an effective detection strategy sensitive to each developer's perception.**

To evaluate the technique, we defined an experiment focused on four types of code smells. The study **compares the efficiency of *Histrategy* with the *unguided customization* for detecting developer-sensitive smells**. The *unguided customization* mimics the techniques evaluated by Fontana *et al.* [16]. These techniques do not provide any guidance in terms of using heuristics to produce a restricted set of detection strategies. Our experiment also addresses another limitation found in the study of Fontana *et al.*. They have not assessed the efficiency and ability of *unguided customization* techniques to be sensitive to various developers, who often have different perceptions about code smells.

During the experiments, *Histrategy* was able to reach a high effectiveness with a low training effort in the vast majority of the analyzed cases. For instance, *Histrategy* reached a f-measure that vary from 66.67% up to 100%, reaching more than 70% in 46 of the 48 analyzed cases. In addition, *Histrategy* reached these results by performing less than 10 requests in the majority of the cases. Finally, *Histrategy* reached a efficiency higher than the unguided techniques in almost all analyzed cases.

The remainder of this paper is structured as follows. Section II presents the background on code smells and discusses previous work on customization in code smell detection. Section III describes *Histrategy* in detail. Section IV presents the experimental study aiming at evaluating *Histrategy* while the results are presented and discussed in Section V. Section VI presents the threats to validity and, finally, Section VII concludes this work.

II. Background and Related Work

Previous researches [13], [25], [15] investigated approaches to detect code smells. Some of the recent approaches are based on the use of machine learning techniques [26], [12], [24], [27], [23]. Such approaches are able to customize code smell detection from a set of examples used for training. Although these studies have evaluated the efficiency of techniques based on machine learning algorithms to detect code smells, they present two limitations concerning how they can aid developers on detecting smells efficiently.

First, the studies did not evaluate the efficiency of the techniques to detect smells according to the view of each developer. Some of these studies involved different developers on experimenting the proposed approaches. However, the accuracy of such approaches is not evaluated and presented individually for each developer. The accuracy is computed according to a restricted set of examples built from developers that were forced to share the same perception about some code smells. By considering the developers often have different perceptions on the presence of smells in a program [5], [6], [7], [8], we do not know if the proposed approaches are efficient to detect code smells individually for developers that detect smells differently.

Second, the proposed approaches required a high number of examples to customize the detection. In this way, the construction of a training set containing a high number of examples may require so much time and effort from developers, thereby questioning the applicability of the detection approach in realistic settings. Next, we describe and discuss previous work that investigated the use of machine learning algorithms to customize code smell detection.

Khomh *et al.* [26] proposed the use of a *Bayesian Belief Network* (BBN) algorithm to detect instances of *God Class* smell in two open source projects. The authors involved 4 graduate students to validate manually a set of classes, reporting if each class contains a *God Class* instance or not. From a set containing 15 smell instances, the authors performed a 3-fold cross-validation procedure in order to calibrate the BBN and assess its performance on detecting *God Class*. The BBN was able to detect all smells, resulting in a *recall* of 100%. However, the algorithm misclassified non-smelly classes and reached a *precision* of 68%.

In a follow-up study [12], the same authors extended their previous work [26] by applying BBNs to detect three types of code smells. In this work the authors followed the same procedure to produce the training set that was used to calibrate and assess the BBN. Again, the BBNs were able to reach a *recall* of 100%. On the other hand, they presented a worse *precision* on detecting the analyzed smells. They reached a precision of almost 33%.

The work described in [24] evaluated the efficiency of an approach based on *Support Vector Machine* (SVM) to detect smells. The proposed approach was evaluated by using a training set containing 250 examples manually validated by different developers. According to the results, the SVM-based approach was able to reach, in average, a *recall* and *precision* equal to 70% and 74%, respectively.

In [27] the authors proposed the use of a *Decision Tree* algorithm to detect smells. Similar to the previous work, the authors used a training set containing a huge number of examples and validated by different developers. The paper reported the algorithm was able to reach, in average, a recall of 71% and a precision up to 78%.

In this context, Fontana *et al.* [23] presents the larger study that compares and experiments different configurations of six machine learning algorithms for code smell detection. For training, the authors considered a set composed of more than 1900 examples of code smells manually validated by different developers. According to the authors, all evaluated techniques present a high accuracy, the highest one was obtained by two algorithms based on *Decision Trees* (*J48* and *Random Forest* [28]). In addition, the authors also reported the techniques needed a hundred training examples to reach an accuracy of at least 95%.

III. Histrategy: improving smell detection with guided customization

Developers have different backgrounds, experience and skills. They are also aware and influenced by their particular project characteristics. These and other factors naturally lead them to have different perceptions about the occurrence of code smells [5], [6], [7], [8]. It is necessary to detect smells in accordance with the individual perception of each developer, as it is the developer is who actually often recognizes and removes smells from existing programs. Thus, we present *Histrategy*, a guided customization technique able to detect code smells that are sensitive to each developer's perception.

Fig. 1 presents an overview of the approach. Inputs are depicted in the top, while each key step is marked with the corresponding circled number. In the next sections, we describe in detail the *process inputs*, the *customization process* and an example of how the *Histrategy* works.

A. Process Inputs

Histrategy receives as input: (i) the type of smell to be detected; (ii) a set of detection strategies produced from heuristics; and (iii) the source code where instances of this smell type must be detected.

Each heuristic represents the way in which the developer detects a specific smell type. For example, a developer can consider *the number of lines of code and the complexity of a method* to judge if the method is a *Long Method* instance. On the other hand, other developer can use a different heuristic. He can judge the smelliness of the method only analyzing *the number of lines of code*. Such heuristics are intended to guide the definition of detection strategies, which are able to detect occurrences of the smell type under analysis. A detection strategy is defined as logical expression composed of metrics, relational operators and thresholds. For instance, the strategies ($LOC^1 > 90$ and $CC^2 > 15$) and ($LOC > 100$) could be used to operationalize the heuristics adopted by the aforementioned developers on detecting a *Long Method* instance. The process of gathering the heuristics and mapping them to detection strategies must be performed before starting the customization process performed by *Histrategy*.

Finally, the source code refers to the software in which the developer intends to detect the smell type under analysis. The software under analysis can be either an entire system or a subset of it, composed by some classes and methods. This will be used in **step 2** of the approach.

B. Customization Process

Based on the received inputs, *Histrategy* starts the developer-driven guided customization process by executing the following steps.

1) Select current best strategy. From the set of strategies available to detect the analyzed smell type, only one must be defined as the *best* strategy for him. For this, in each iteration, *Histrategy* assesses the strategies given as input as well as those produced during the customization; it looks for the strategy that presents the highest effectiveness on classifying code snippets already classified by the developer. In cases that two or more stored strategies present the same best performance, or when there are still no snippet classified by the developer, the strategy is randomly selected.

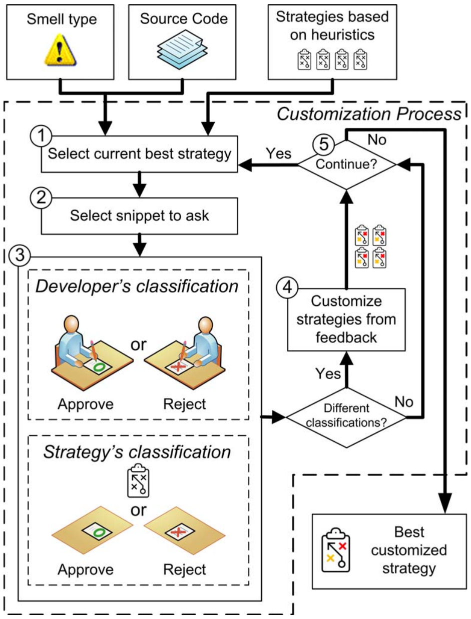

Smell type
Source Code
Strategies based on heuristics
Customization Process
1
Select current best strategy
Yes
5
No
Continue?
2
Select snippet to ask
3
Developer's classification
or
Approve
Reject
4
Customize strategies from feedback
Yes
Different classifications?
No
Strategy's classification
or
Approve
Reject
Best customized strategy

Fig. 1. Histrategy's flow to customize strategies

2) Select snippet. Based on the selected strategy, *Histrategy* selects a code snippet from the source code given as input. To select this snippet, *Histrategy* identifies the snippets whose metrics values are closest to the thresholds of the metrics that compose the *best* strategy. For example, if we have a strategy as $LOC > 80$, then *Histrategy* can select a method that contains 81 lines of code. In case there are more than one snippet, *Histrategy* randomly selects one of them.

3) Classify the snippet. The selected snippet is presented to the developer that must classify this snippet indicating if it contains (or does not contain) the smell type under analysis. After collecting the developer's feedback, the *Histrategy* also verifies how the *best* strategy classifies the code snippet. This way, *Histrategy* can analyze if the classification performed by the *best* strategy matches the developer's classification, concerning the presence of the smell in the analyzed snippet.

4) Customize Strategy. If the classifications of the *current best strategy* and developer diverge, the strategy needs to be customized to be in accordance with the developer. The customization consists in tuning the thresholds of the strategy so that *Histrategy* will be hopefully able to classify snippets according to the developer's classification. For example, consider a strategy *LOC > 100* to detect *Long Methods*. If a developer considers that a method with 95 lines of code is a *Long Method* instance, then the thresholds that compose this strategy must be customized to *LOC* $\geq$ *95*. Moreover, if the

[1] Lines of Code

[2] Cyclomatic Complexity

best strategies derived from other heuristics diverge from the developer's classification, new strategies are also created from them. This allows that all the strategies can evolve together, independently if they are selected in a given iteration.

5) Analyze the stopping criteria. Whenever a strategy is customized, or when it classifies the snippet equal to the developer's classification, *Histrategy* needs to decide if the strategy is appropriate to detect smells to the developer or the customization process must be executed again from **step 1** by considering a new iteration. This decision depends on the stopping criteria be satisfied. The stopping criteria must considers two factors. First, how similar are the developer's perceptions and the classifications performed by the strategy. In this case, this similarity can be considered as the *effectiveness* of the strategy. Second, the *effort* required from the developer to tune the strategy. During the customization process, the *Histrategy* frequently asks the developer to indicate if the code snippet contains (or does not contain) a smell. If the technique performs a high number of requests, the customization can become a impractical or time-consuming task [29]. Therefore, it is important to balance the *effectiveness* and the *training effort* to stop the customization process.

Given the aforementioned steps, we conjecture that early definition of detection strategies from heuristics may reduce the number of examples required by *Histrategy* in order to customize the detection for the developer. After all, we expect that the developer will perceive the analyzed smell from one of the heuristics considered by *Histrategy*. On the other hand, if *Histrategy* does not have proper guidance, i.e. at least a proper heuristic for the developer, the customization will not be able to produce an effective detection strategy and it will require a lot of examples to produce it. Section IV evaluates the efficiency (in terms of *accuracy* and *effort*) of the *Histrategy* on producing customized strategies driven by the perspective of each developer.

C. Execution Example

To better understand the customization process performed by *Histrategy*, Fig. 2 exemplifies how the *Histrategy* customizes strategies to detect *Long Method* instances for a specific developer. Initially, the *Histrategy* receives two detection strategies: **S1** ($LOC > 100$) and **S2** ($LOC > 90\ and\ CC > 15$). In addition, our technique also receives five methods, named from *M1-M5*, which are used to gather the developer's feedback. For each method, we describe the LOC and CC values. The customization process is based in a sequence of iterations with the developer, as follows:

(**Iter. 1**) *Histrategy* selects the best strategy among the initial strategies (**S1** and **S2**). As the developer has not provided feedback at the beginning of the process, the selection of the *current best strategy* is performed randomly (**S1** is selected). Next, *Histrategy* selects a method with metric values closest to the selected strategy thresholds. As **S1** is defined as $LOC > 100$, the method *M4* (with $LOC = 99$) is selected to be presented to the developer to receive his classification. The developer classifies *M4* as *Long Method*, but **S1** does not. Due to this divergence, the **S1** strategy is customized in order to fit it to the developer's perception, originating the strategy **S3** ($LOC \geq 99$). After this, *Histrategy* verifies if the strategies derived from other heuristics (**S2**) also classify the *M4* method different than the developer. No strategy is derived from **S2** because it matches the developer's feedback.

(**Iter. 2**) *Histrategy* selects the *current best strategy* from the three strategies available. Only **S2** and **S3** are able to classify the method presented in the previous iteration as the developer classified in his feedback (**S2** is randomly selected). From the **S2** strategy, our technique retrieves the *M3* method, which presents metric values closest to the **S2** thresholds. Both the **S2** strategy and the developer agree that the method **M3** is not a *Long Method* instance. Once the strategy derived from other heuristic (**S3**) also classifies the method as the developer did, no new strategy is created in this iteration.

(**Iter. 3**) Again, *Histrategy* needs to select the *best current strategy* available. Only **S2** and **S3** are able to classify the methods asked in previous iterations as the developer did (**S2** is randomly selected). The *M2* method is selected to gather the developer's feedback, and the developer and the **S2** strategy classify **M2** differently. Thus, **S4** ($LOC \geq 80\ and\ CC > 15$) is created by adapting the **S2** thresholds in order to classify the method as developer's did. Also **S3** diverges from the developer's feedback, thus *Histrategy* creates a new strategy (**S5**: $LOC \geq 80$) by adapting the thresholds of **S3** to match the developer's feedback.

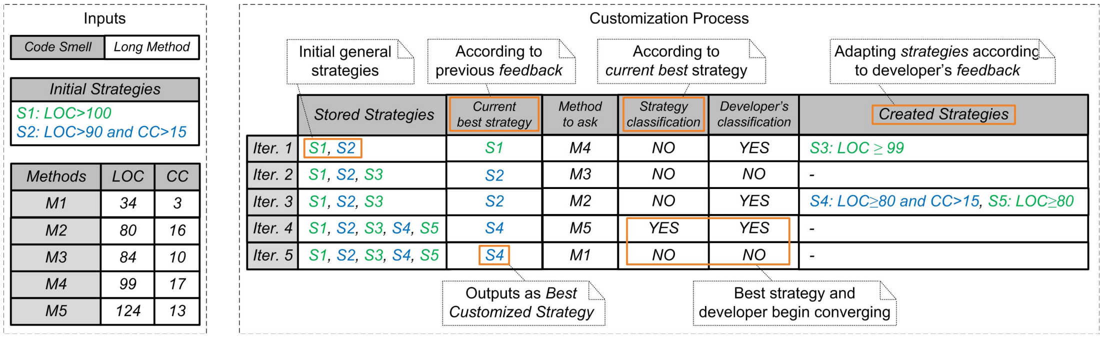
In **Iter. 4** and **Iter. 5**, the **S4** was able to classify methods *M5* and *M1* in accordance with the developer. Then, it was not necessary to customize new strategies. As a consequence, the strategy **S4** can be used to detect *Long Method* instances to the developer. After all, **S4** is able to classify all methods presented in the example as the developers did.

IV. Study Design

This section describes a study to evaluate *Histrategy*, which uses a *guided customization* to produce strategies able to detect smells sensitive to each developer's perception. The study focuses on two particular properties: **effectiveness**, i.e. how effective are the generated customized strategies in correctly identifying or not a code smell, and **efficiency**, i.e. the relation between the effectiveness of the customized strategy and the *training effort* required from each developer to produce a proper customized strategy.

Previous studies (Section II) have not assessed the role of *guided customization*, neither compared it against *unguided* techniques [30] on producing detection strategies tailored to a single developer. Thus, we compare the efficiency of the guided customization, as supported by *Histrategy* (Section III), with *unguided customization*, as supported by existing approaches. In summary, the study tries to answer the following research questions:

- **RQ1**: *How effective are the strategies customized by the Histrategy in detecting developer-sensitive smells?*
- **RQ2**: *How much training effort is required by Histrategy to detect developer-sensitive smells?*


Inputs
Customization Process
Initial general strategies
According to previous feedback
According to current best strategy
Adapting strategies according to developer's feedback
Outputs as Best Customized Strategy
Best strategy and developer begin converging

Code Smell	Long Method

Initial Strategies
S1: LOC>100 S2: LOC>90 and CC>15

Methods	LOC	CC
M1	34	3
M2	80	16
M3	84	10
M4	99	17
M5	124	13

	Stored Strategies	Current best strategy	Method to ask	Strategy classification	Developer's classification	Created Strategies
Iter. 1	S1, S2	S1	M4	NO	YES	S3: LOC ≥ 99
Iter. 2	S1, S2, S3	S2	M3	NO	NO	-
Iter. 3	S1, S2, S3	S2	M2	NO	YES	S4: LOC≥80 and CC>15, S5: LOC≥80
Iter. 4	S1, S2, S3, S4, S5	S4	M5	YES	YES	-
Iter. 5	S1, S2, S3, S4, S5	S4	M1	NO	NO	-

Fig. 2. Example on customizing the detection for *Long Method* smell

- **RQ3**: *Does Histrategy detect smells more effectively and efficiently than unguided customization approaches?*

A. Subjects and Heuristics

Our study recruited 48 developers with different backgrounds and experience, according to the following criteria. We wanted to gather a group of developers from different companies and who previously worked in various projects from different domains. We sent an invitation message to several contacts in different companies. At the end, we were able to recruit participants with: (i) different levels of experience on software development (e.g. Java and other programming technologies), (ii) previous experience on code smell detection, and (iii) at least 3 years of experience on software projects, which had emphasis on structural software quality.

In addition, we analyzed 4 different types of code smells, presented in Table I. We chose these smell types because they affect different scopes of a program, i.e. classes, methods or parameters. These smell types were also investigated in previous work on code smell detection [23], [12], [13], [31].

TABLE I
TYPES OF CODE SMELLS INVESTIGATED IN THIS STUDY

Name	Description
Long Method	A method that is too long and tries to do too much.
God Class	The God Class code smell refers to classes that tend to centralize the intelligence of the system.
Data Class	These are classes that have fields, getting and setting methods for the fields, and nothing else.
Primitive Obsession	Primitive types are overused in software. Small classes should be used in the place of primitive types in some situations.

The code snippets analyzed in our experiments were extracted from 2 open source Java projects: GanttProject (2.0.10) and Apache Xerces (2.11.0). We selected such projects because they had been investigated by existing smell detection studies [13], [12], [23] and the source code of these projects contains a variety of suspicious code smells that enable the execution of our experiments.

As mentioned in Section III, heuristics are used to produce a restricted set of detection strategies, which will be used as input in the customization process performed by *Histrategy*. In our experiment, we reused heuristics reported on previous studies that investigated subjective issues concerning the developer's perception on detecting smells [5], [6], [7], [8]. Moreover, we used a limited set of strategies existing in the literature [10], [32], [13], [23] to operationalize these heuristics. This process represents an alternative way for organizations that do not have a pre-defined set of heuristics and well accepted strategies amongst its developers. The strategies considered in our study are available at [33].

B. Data Collection

To support our study, we collected a set of potentially-smelly code snippets manually validated by the 48 developers that participated of our experiment. Each developer validated the snippet by classifying either as a *smelly* or *smell-free* code snippet. This data supports the evaluation of the detection approaches investigated in our study. Initially, we grouped the 48 participants into 4 groups. Each group, composed of 12 developers, was responsible to classify the same 15 code snippets concerning the occurrence (or not) of a smell type analyzed in our study. At the end, we collected more than 700 classifications.

Each classification concerns the analysis of the code snippet by looking for a specific code smell. The code snippets comprehend the scope of the smell type under analysis. For example, the code snippets for *Long Method* candidates contained at least a specific method (the smell candidate); the developer could inspect the class when needed. The participant must conclude each classification by answering one from two possible choices: **`YES`**, when he agrees that the given code snippet contains the specified smell type; and **`NO`**, otherwise.

C. Data Analysis

To answer the **RQ1**, we used the classification data to produce a detection strategy customized for each developer that participated in our study. In particular, the customization was guided by the 15 classifications performed by the developer and a set of heuristics to detect a given smell type.

We evaluated the effectiveness of each customized strategy as follows. For each developer, we apply the customized strategy to classify the same 15 code snippets analyzed by him. After, we compare the developer's evaluations with the classification performed by the customized strategy. As a result of this comparison, we assessed the effectiveness of each customized strategy from three widely-adopted Information Retrieval (IR) metrics, namely *recall*, *precision* and *f-measure* [34]. Such metrics are frequently used in other studies involving code smell detection [12], [24], [23].

To answer **RQ2**, we considered as *training effort* the number of examples required by *Histrategy* in order to produce the customized strategies. After all, as detailed in Section III, the customized strategies are created during an iterative process that considers the developer's feedback. In this way, we evaluate and discuss about the number of examples required to produce each one of the strategies produced to answer **RQ1**.

Finally, to answer **RQ3**, we compare the effectiveness and efficiency of *Histrategy* against unguided customization based on 6 machine learning (ML) techniques, namely J48, JRip, Random Forest, SMO, Naive Bayes and SVM. Such techniques were investigated in a recent study that evaluated the efficiency of each ML approach on detecting smells [23]. In this case, we evaluate and compare the effectiveness and efficiency of these techniques on detecting smells for each developer that participated in our study. The effectiveness was evaluated by analysing the performance of each customized strategy on detecting smells over the 15 classifications produced by each developer. For that, we evaluate the *f-measure* by applying a 5-fold cross validation procedure aiming at limiting overfitting problems.

To analyze the efficiency, we evaluated the *effectiveness* of each technique by varying the number of examples considered in the learning. The effectiveness was evaluated by considering the 15 classifications performed by each developer. However, we ranged the number of learning examples from 2 to 13, in order to guarantee that both, the learning and test datasets, were composed of snippets classified by smell and non-smell by the developer. In some cases, the number of examples to perform the learning of the techniques was very low. As a consequence, we could not compute the effectiveness in terms of *f-measure* on evaluating the efficiency. Hence, we used the *accuracy* [34] measure to calculate the effectiveness on analysing the efficiency of each technique. Similarly, the *accuracy* measure was adopted in several works that investigated the use of machine learning algorithms for code smell detection [26], [35], [23]. The developer's evaluations that were used to compute the effectiveness and efficiency, as well as all other materials and results are available in [33].

V. Results and Discussion

In this section, we describe and discuss the main results of the study. We structure the data presentation and discussion in terms of our three research questions.

A. *RQ1: How effective are the strategies customized by the Histrategy in detecting developer-sensitive smells?*

Table II presents the main results that support the discussions about **RQ1**. We discuss the effectiveness of *Histrategy* for each smell type, as presented in the first column. The second column identifies the developers that participated of the study using their *ID*s. The third column reports the heuristic that defined the best customized strategy for each developer. The following three columns report the *recall*, *precision* and *f-measure* of the customized strategy on classifying the code smells for each developer. Finally, the last column indicates the number of examples required by *Histrategy* to produce each strategy. The latter will support the discussion of **RQ2**.

TABLE II
Detecting Smells with Histrategy

Code Smell	Dev	Heuristic	Histrategy			
			R (%)	P (%)	FM (%)	# examples
God Class	1	H1	100.00	57.14	72.73	8
	2	H1	100.00	87.50	93.33	11
	3	H1	100.00	60.00	75.00	12
	4	H1	100.00	100.00	100.00	6
	5	H2	100.00	100.00	100.00	2
	6	H2	85.71	100.00	92.31	4
	7	H2	66.67	100.00	80.00	5
	8	H2	100.00	100.00	100.00	2
	9	H3	100.00	55.56	71.43	3
	10	H3	100.00	62.50	76.92	15
	11	H3	80.00	66.67	72.73	13
	12	H4	80.00	80.00	80.00	10
	Average (%)		92.70	80.78	84.54	7.58
Long Method	13	H5	100.00	100.00	100.00	5
	14	H5	100.00	64.29	78.26	6
	15	H5	90.91	90.91	90.91	2
	16	H5	91.67	100.00	95.65	2
	17	H5	100.00	88.89	94.12	0
	18	H5	75.00	85.71	80.00	4
	19	H5	90.00	90.00	90.00	8
	20	H5	100.00	100.00	100.00	3
	21	H6	88.89	88.89	88.89	7
	22	H6	100.00	84.62	91.67	10
	23	H6	100.00	90.00	94.74	8
	24	H6	100.00	76.92	86.96	10
	Average (%)		94.71	88.35	90.93	5.42
Data Class	25	H7	77.78	87.50	82.35	11
	26	H7	100.00	87.50	93.33	7
	27	H7	88.89	100.00	94.12	8
	28	H7	87.50	87.50	87.50	15
	29	H7	88.89	100.00	94.12	8
	30	H7	100.00	62.50	76.92	15
	31	H7	88.89	100.00	94.12	8
	32	H7	100.00	62.50	76.92	7
	33	H8	88.89	80.00	84.21	5
	34	H8	100.00	84.62	91.67	1
	35	H8	100.00	64.29	78.26	2
	36	H8	100.00	80.00	88.89	3
	Average (%)		93.40	83.03	86.87	7.50
Primitive Obsession	37	H9	100.00	66.67	80.00	6
	38	H9	71.43	83.33	76.92	3
	39	H9	71.43	71.43	71.43	2
	40	H9	100.00	80.00	88.89	10
	41	H9	100.00	100.00	100.00	15
	42	H9	83.33	55.56	66.67	14
	43	H9	100.00	100.00	100.00	7
	44	H9	100.00	85.71	92.31	10
	45	H9	100.00	66.67	80.00	15
	46	H10	83.33	55.56	66.67	11
	47	H10	80.00	80.00	80.00	15
	48	H11	100.00	83.33	90.91	5
	Average (%)		90.79	74.90	82.82	9.42

God Class. *Histrategy* was guided by four heuristics to customize strategies able to detect *God Class* instances sensitive to the individual perception of each developer. While the heuristic `H1` was more appropriate to produce strategies for the developers 1–4, the heuristics `H2` and `H3` were sensible for the developers 5–8 and 9–11, respectively. The `H4` was sensible only for the developer 12. Such results suggest that developers perceive *God Class* instances differently. After all, heuristics indicate the way in which developers detect a specific smell type. If we analyze the strategies used to operationalize these heuristics, the divergence among the developers tends to be even more significant. Indeed, even for the developers that followed a same heuristic, the strategies customized for them presented different threshold values after performing the *Histrategy's* customization process, as one can see at [33]. This finding reinforces the fact that smell detection needs to be sensitive to each developer. Previous studies [26], [12], [24], [27], [23] could not confirm this expectation as they do not explicitly assess developer-sensitive customization.

As a result of the guided customization, we observe that *Histrategy* obtained a high *recall* in the vast majority of the analyzed cases. In particular, *Histrategy* reached a *recall* equal to 100% in 8 of the 12 analyzed cases. In the remaining 4 cases, *Histrategy* reached a *recall* equal or greater than 80%, except in the case of the developer 7 for which the *recall* was equal to 66.67%. Such results indicate that the vast majority of the smells detected by *Histrategy* were relevant to the developers. We also note that *Histrategy* reached a high *precision* in the majority of the analyzed cases. For instance, it reached a *precision* equal or greater than 80% in 7 of the 12 analyzed cases, reaching 100% in 5 cases. Considering these values of *recall* and *precision* reached by *Histrategy*, we obtained *f-measure* values that vary from 71.43% up to 100%.

Long Method. Only two heuristics were used by *Histrategy* in order to customize strategies able to detect *Long Method* instances for each developer. In this case, while the developers 13–20 followed the heuristic `H5`, the best strategies defined for the developers 21–24 were derived from the heuristic `H6`.

Similarly to *God Class*, *Histrategy* also reached a high *recall* in the majority of the cases related to *Long Method*. While guided customization reached a *recall* equal to 100% in 8 of the 12 cases related to *God Class*, it reached the same *recall* in 7 cases related to *Long Method*. In addition, all the *recall* values reached by *Histrategy* in *God Class* and *Long Method* are equal or greater than 75%. Regarding the *precision*, *Histrategy* was able to reach 100% in 3 cases. In the remaining 9 cases, the values of *precision* vary from 64.29% up to 90.91%. As a consequence, the *f-measure* of *Histrategy* was greater than 78% in all the analyzed cases.

Data Class. Just as *Long Method*, *Histrategy* used only two heuristics to customize strategies to detect *Data Class*. The heuristic `H7` was more appropriate for the developers 25–32 and the `H8` was more sensible for the developers 33–36. Regarding the *recall* and *precision* values, the *Histrategy* was also able to reach high values. For instance, while the *recall* vary from 77% up to 100%, reaching 100% in half the 12 analyzed cases, the *precision* vary from 62% to 100%. These results enable the *Histrategy* reaches only *f-measure* values greater than 76%.

Primitive Obsession. The *Histrategy* used only 3 heuristics to customize strategies related to *Primitive Obsession* detection. The heuristic `H9` was followed by most of the developers (37–45) that evaluated such smell type. The other developers followed the heuristics `H10` (46–47) and *H11* (48). Similarly to *Long Method*, *Histrategy* reached only *recall* values greater than 71% and equal to 100% in in 7 of the 12 analyzed cases. Regarding *Precision*, *Histrategy* reached *precision* values greater than 70% in half of the 12 cases, reaching 100% in 2 of these cases.

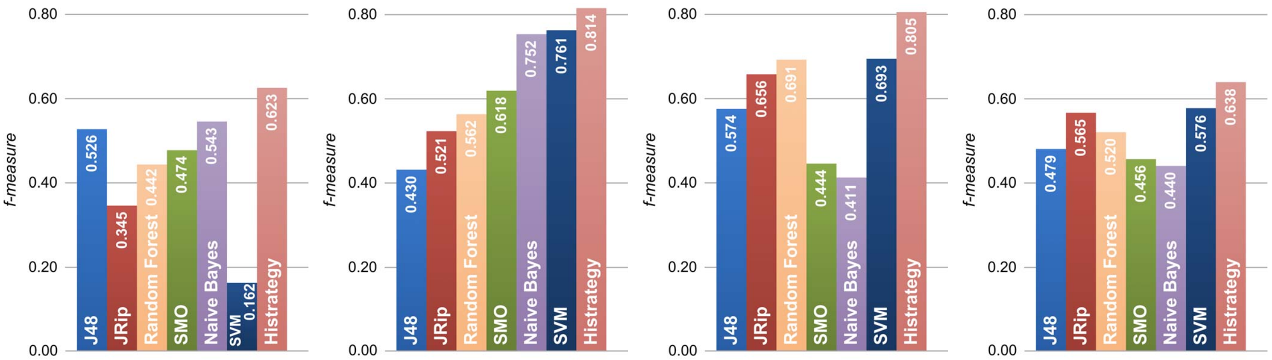
The results indicate that *Histrategy* reached recall equal to 100% in 28 of the 48 analyzed cases. In addition, *Histrategy* reached a *recall* greater than 80% in 43 cases.Regarding the precision, *Histrategy* reached values equal to 100% in 25% of the analyzed cases, obtaining a *mean* of precision that varied from 74.90% to 88.35%. As a consequence, *Histrategy* obtained *f-measure* values that vary from 66.67% to 100%, reaching *f-measure* values greater than 70% in 46 of the 48 cases. Such results suggest that ***Histrategy* was able to detect code smells with high effectiveness for the most of the developers**. Moreover, for each analyzed smell type, it was necessary to use different heuristics in order to produce strategies tailored to each developer's perception. Another interesting point to be observed in these results is the fact that differently from previous studies [24], [23], where hundreds of examples had to be manually validated by developers, in our study we achieved high effectiveness with *Histrategy* by using only a few examples. The use of guided customization clearly reduces the effort involved in the training session.

*B. **RQ2**: How much training effort is required by Histrategy to detect developer-sensitive smells?*

In the previous section, *Histrategy* was able to detect smells sensitive to each developer's perception. However, we still do not know the effort to perform such detection. For this purpose, we considered the number of examples required by the *Histrategy* to produce a detection strategy for each developer, as described in the last column of Table II.

Firstly, we observe that the customization of strategies to detect *Long Method* instances required a *mean* of 5.42 requests, which represents the lowest *mean* of requests among all the types of smells analyzed. Still regarding the *Long Method*, we note that it was not necessary to perform any request to the developer 17. In this case, the customization was not able to increase the effectiveness reached by the strategy that *Histrategy* initially defined. Even though has been needed to perform up to 10 requests in two cases (developers 22 and 24), in half of the cases, *Histrategy* requested at most 5 requests to each developer. Therefore, in the vast majority of the *Long Method* cases, the *Histrategy* was able to detect developer-sensitive smells by performing only few requests.

In case of the *Data Class*, the *Histrategy* reached a *mean* of requests of 7.50. However, we observe that *Histrategy* was able to detect *Data Classes* instances for each developer by performing less than 10 requests to 9 of the 12 developers. The increase in the *mean* of requests was influenced mainly by the developers 28 and 30, where *Histrategy* needed to perform 15 requests for each one in order to produce the customized strategies. In such cases, where a high number of examples are needed, it is necessary to consider new heuristics that better represents the way in which the developers 28



(a) God Class (b) Long Method (c) Data Class (d) Primitive Obsession

Fig. 3. Mean of *f-measure* obtained by *Histrategy* and other ML-approaches

and 30 detect smells. After all, a proper set of heuristic may support *Histrategy* on customizing the detection according to the perception of these developers.

Regarding the *God Class*, *Histrategy* needed to perform a *mean* of requests slightly greater than the *Data Class*. This increase was influenced by the number of cases where *Histrategy* performed more than 10 requests. While *Histrategy* performed more than 10 requests in only 3 of the 12 *Data Class* cases, this number of requests occurred in 5 cases related to the *God Class*. The highest *mean* of requests, equal to 9.42, occurred in the cases related to the *Primitive Obsession*. However, the *mean* is still lower than 10. We observe that *Histrategy* performed less than 10 requests only in 5 of the 12 analyzed cases. As a consequence of this high number of requests to detect *Primitive Obsession*, *Histrategy* obtained the lowest mean values of *recall*, *precision* and *f-measure* among all the analyzed smell types, as described in Table II.

As a result of this discussion, we observe that ***Histrategy* was able to detect developer-sensitive smells by performing less than 10 requests in the vast majority of the analyzed cases**. Therefore, when compared with existing smell detection approaches [23] that used more than 1900 examples validated manually by developers, *Histrategy* is able to detect smells with low *training effort*.

C. ***RQ3****: Does Histrategy detect smells more effectively and efficiently than unguided customization approaches?*

According to the results for **RQ1** and **RQ2**, *Histrategy* was able to reach high values of effectiveness by performing few requests for each developer. However, we do not know how efficient is the *Histrategy* when compared with *unguided customization* techniques. Hence, **RQ3** intends to compare the effectiveness (in terms of *f-measure*) and efficiency (in terms of *accuracy* and *training effort*) of the *guided* and *unguided* customization approaches.

Effectiveness. For each smell type analyzed in our study, we evaluated the effectiveness of the *Histrategy* and existing techniques, based on *unguided* customization, on detecting smells for each developer responsible at evaluating the analyzed type. Next, we calculated the *mean* of the effectiveness values obtained by each technique (guided and unguided). The Figures 3a – 3d present the results related to the smell types *God Class*, *Long Method*, *Data Class* and *Primitive Obsession*, respectively. In each figure, the *y-axis* describes the *mean* values reached by each technique on detecting the analyzed smell type. In addition, we attach the exact *mean* value to the bar associated with each technique.

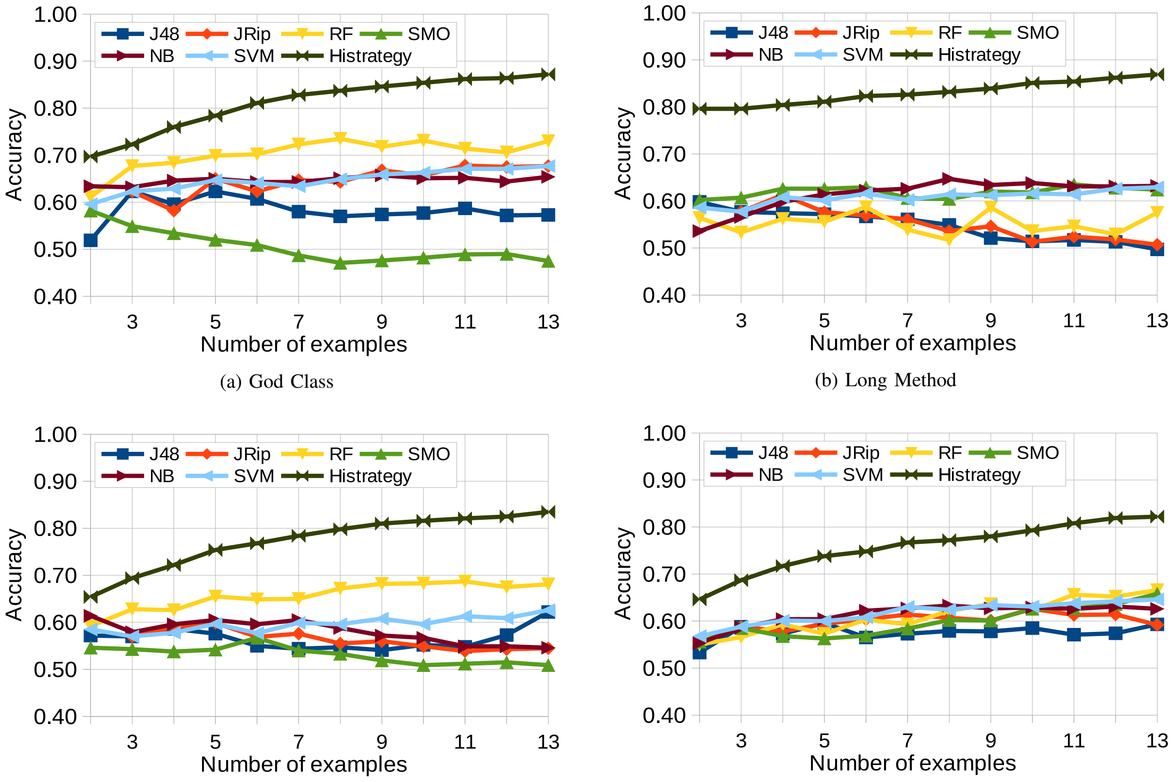
The results depicted in Fig. 3a indicate that *Histrategy* reached a *mean* equal to 0.623 on detecting *God Class* instances. This *mean* is greater than the ones reached by the other evaluated techniques. Among *unguided* techniques, *Naive Bayes* reached the highest *mean* equal to 0.543. The *J48* reached a *mean* equal to 0.526, which is slightly lower than the *Naive Bayes*. On the other hand, *SVM* reached the worst *mean*, equal to 0.162, among all the analyzed cases.

In the case of *Long Method*, *Histrategy* reached a *mean* equal to 0.814, as depicted in Fig. 3b. This *mean* is the highest one reached by a technique, among all the analyzed smell types. Regarding the unguided techniques, *SVM* reached a *mean* greater than the other techniques. Note that the *mean* reached by *Naive Bayes* is only slightly lower than *SVM*. While *SVM* reached a *mean* equal to 0.761, the *mean* obtained by *Naive Bayes* was equal to 0.752. Still concerning *Long Method* detection, *J48* reached the worst *mean* equal to 0.430.

Similarly to the *God Class* and *Long Method*, *Histrategy* reached a *mean* greater than the ones obtained by unguided techniques on detecting *Data Class* instances. As depicted in Fig. 3c, *Histrategy* was able to reach a *mean* equal to 0.805. *SVM* obtained the second highest *mean*, equal to 0.693. *Naive Bayes* reached the lowest *mean*, equal to 0.411.

Regarding the *Primitive Obsession*, the results described in Fig. 3d indicate that *Histrategy* obtained the highest *mean* when compared with the unguided techniques. Differently of the smell types previously discussed, the unguided techniques were not able to reach a *mean* greater than 0.600. Similarly to the *Data Class*, the *Naive Bayes* also reached the lowest *mean*, equal to 0.440, on detecting *Primitive Obsession* instances

According to the results illustrated in Fig. 3, we note



(a) God Class

(b) Long Method

(c) Data Class

(d) Primitive Obsession

Fig. 4. Learning curves of the detection approaches

that *Histrategy* reached *mean* values greater than *unguided* techniques in all analyzed cases. We also observe that even though there is a consistency on the *mean* values obtained by *Histrategy*, the *unguided* techniques did not present the same consistency on detecting the analyzed smell types. While the *SVM* reached the *highest mean* when compared with the other unguided techniques on detecting *Long Method*, *Data Class* and *Primitive Obsession*, it also reached the worst *mean*, among all the analyzed cases, on detecting *God Class* instances. On the other hand, the *Naive Bayes* reached a *mean* greater than the other *unguided* techniques on detecting *God Class* instances, but it also reached the lowest *mean* values on detecting *Data Class* and *Primitive Obsession*. Therefore, **differently from unguided techniques, we observe a consistent influence of guided customization on improving the effectiveness in the detection of developer-sensitive smells**.

Efficiency. After analyzing the effectiveness of the techniques, we compared their efficiency in terms of its: (i) accuracy, and (ii) the training effort required by each one to reach the reported accuracy. Figures 4a, 4b, 4c and 4d describe the *Learning curves* that represent the efficiency reached by each technique on detecting a specific smell type. In particular, for each analyzed smell type, a curve describes the *mean* of the accuracies obtained by each technique as we increase the number of examples considered to customize a detection strategy. The *mean* of the accuracies was obtained as described in Section IV-C. The *x-axis* and *y-axis* represent the *mean* and the number of examples, respectively. The legend on top of the figures describes the colors used to identify each technique. Regarding the *God Class*, *Histrategy* was able to reach an efficiency higher than the unguided techniques by considering from 2 to 13 examples, as described in Fig. 4a. Indeed, while *Histrategy* was able to reach an accuracy equal to 0.70 by using only two examples, the other techniques were not able to reach an accuracy greater than 0.65. We also observe that, differently from unguided techniques, *Histrategy* obtained an increase in its efficiency as we increase the number of examples, reaching an efficiency greater than 0.85 on using 13 examples. All the unguided techniques reached an efficiency below 0.75. At the end, we observe that *SMO* presented the lowest efficiency after considering at least 3 examples.

Similarly to the *God Class*, *Histrategy* reached an efficiency higher than unguided techniques in all analyses related to *Long Method*, as described in Fig. 4b. We observe that *Histrategy* reached an accuracy equal to 0.8 by using only two examples. This accuracy was the highest one obtained by a technique, considering only two examples. This high accuracy obtained by *Histrategy* tended to increase as the number of examples was increased, reaching an accuracy greater than 0.8 by using only 5 examples. The unguided techniques were not able to reach an accuracy greater than 0.65 even when using all the available examples.

In the case of *Data Class*, *Histrategy* presented an efficiency greater than unguided techniques in all analyzed scenarios, as described in Fig. 4c. For this smell, Histrategy could reach an accuracy of 0.84 by using 13 examples. The unguided techniques only reached an accuracy lower than 0.70.

Histrategy obtained an efficiency similar to the previous analysis on detecting *Primitive Obsession*, reaching an efficiency higher than unguided techniques in all the analyses, as described in Fig. 4d. We also observe that, similar to the previous analysis, *Histrategy* tended to increase as we increase the number of examples. Note that *Histrategy* was able to increase its accuracy from 0.65 to 0.75 by using from 2 to 5 examples, reaching an accuracy close to 0.83 when all the available examples were explored. The unguided techniques reached only accuracy below 0.67.

The results indicate that *Histrategy* reached an efficiency greater than unguided techniques in the vast majority of all analyzed cases, for the four smell types. We also observed that while the guided customization tended to improve its accuracy as we increase the number of examples, some unguided techniques decreased its accuracy, as observed for *SMO* in the *God Class* and *J48* for *Long Method* detection. Such results suggest that the ***Histrategy* is able to detect developer-sensitive smells more effectively and with lower number of examples than *unguided* techniques.** Such findings suggest that the guided customization may be an important alternative to accommodate the different perceptions of the developers.

VI. Threats to Validity

In this section, we present the threats to validity by following the Wohlin *et al.* validity criteria [36].

Construct Validity. In our study, we collected a set of potentially-smelly code snippets manually validated by developers. This data supports the evaluations performed in our experiment. We enabled the developers to evaluate each code snippet by reporting the option **`YES`** or **`NO`**. Providing only these two options may be a threat, since they did not enable the developer to inform the degree of confidence in his answers. However, we adopted such procedure to ensure that the developers were able to decide about the existence of a code smell and to enable we evaluate the efficiency of the classification techniques analysed in our study.

Internal Validity. We followed the configurations defined by Fontana *et al.* [23] to execute the unguided techniques evaluated in our analysis. Although the use of other configurations could have influenced some of these techniques to increase their efficiency, we adopted the configurations defined by Fontana *et al.* because the authors performed a variety of experiments in order to find the best adjust for each technique. In addition, the results described in Fontana *et al.* indicate that the techniques were able to reach high values of accuracy.

External Validity. We used code snippets from only 2 Java projects. Although these projects contain different sizes and domains, and they have been widely used in existing works related to code smells [26], [12], [13], [24], our results might not hold to other projects, since some of them have very distinctive characteristics. Moreover, even though we have performed our experiments by using 48 developers with different background and experience, our results might not also hold for other developers since they have different perceptions about the presence of a same smell [5], [6], [7], [8]. Finally, we selected a set of heuristics to perform the experiments. Although we have collected such heuristics from different sources, the selection process may be discarded heuristics that could have been important for the developers, or for the *Histrategy* to increase its efficiency.

VII. Conclusion

We proposed *Histrategy*, a technique that uses a guided customization to detect code smells that are sensitive to the developer's perception. Moreover, we presented a study aiming at investigating the effectiveness and efficiency of *guided* and *unguided* techniques to identify developer-sensitive smells. This investigation is important because developers can have different perceptions about the presence of a same smell [6], [37].

To evaluate *Histrategy*, we performed an experiment involving 48 developers to evaluate snippets extracted from 2 open source projects looking for 4 types of code smells. More than 700 evaluations were performed resulting in a huge portion of quantitative and qualitative data. From such data, we verified that *Histrategy* was able to reach a high efficiency on detecting code smells for each developer. In average, *Histrategy* required at most 7 examples to produce strategies that reached a recall up to 92% and a precision up to 81%.

We compared the effectiveness and efficiency of the *Histrategy* against unguided customization via 6 machine learning techniques investigated in a recent study [23]; they are: *J48*, *JRip*, *Random Forest*, *SMO*, *Naive Bayes* and *SVM*. The results indicate that *Histrategy* was able to reach an effectiveness and efficiency higher than the *unguided* techniques in the vast majority of the analyzed cases. Therefore, the results indicate that the guided customization may improve the efficiency on code smell detection.

As future work, we pretend extend this investigation by evaluating the efficiency of *Histrategy* to detect other smell types. In addition, we also pretend to repeat this study in a controlled scenario, considering developers and projects of a same organization. Such investigation will enable to analyze if developers, who work together, detect smells similarly (or not) and require a proper customization technique.

Acknowledgment

This work has been partially supported by CAPES/Procad (grant `175956`), CNPq (grants `483425/2013-3`, `309884/2012-8`, `312149/2016-6`), FAPERJ (grants `E26-102.166/2013`, `E26-225.207/2016`), FAPEAL and by the project **DEVASSES** (www.devasses.eu), funded by the European Union's FP7 under grant agreement no `PIRSES-GA-2013-612569`.

REFERENCES

[1] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 181–190.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[3] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 440–451. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884868

[4] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, Aug. 2011.

[5] M. V. Mäntylä, "An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement," in *International Symposium on Empirical Software Engineering*, 2005.

[6] M. V. Mäntylä and C. Lassenius, *Subjective evaluation of software evolvability using code smells: An empirical study*. Springer, May 2006, vol. 11.

[7] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10*, p. 1, 2010.

[8] J. A. M. Santos, M. G. de Mendonça, and C. V. A. Silva, "An exploratory study to investigate the impact of conceptualization in god class detection," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '13. New York, NY, USA: ACM, 2013, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/2460999.2461007

[9] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ser. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 350–359. [Online]. Available: http://dl.acm.org/citation.cfm?id=1018431.1021443

[10] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[11] M. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," *11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 15–15, 2005.

[12] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, Apr. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2010.11.921

[13] N. Moha, Y.-G. Gueheneuc, L. Duchien, and a. F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[14] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An Experience Report on Using Code Smells Detection Tools," *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 450–457, Mar. 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5954446

[15] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining Version Histories for Detecting Code Smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, 2015.

[16] F. A. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 15–22, Jun. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6225993

[17] R. Marinescu, "Detecting Design Flaws via Metrics in Object-Oriented Systems A Metrics-Based Approach for Problem Detection," *International Conference and Technology of Object-Oriented Languages and Systems (TOOLS)*, pp. 173–182, 2001.

[18] M. Ferreira, E. Barbosa, I. Macia, R. Arcoverde, and A. Garcia, "Detecting architecturally-relevant code anomalies: a case study of effectiveness and effort," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1158–1163.

[19] "PMD," http://pmd.sourceforge.net/, March, 2017.

[20] "Checkstyle," http://checkstyle.sourceforge.net/, March, 2017.

[21] "inFusion," https://www.intooitus.com/products/infusion, March, 2017.

[22] "JDeodorant," http://www.jdeodorant.com/, March, 2017.

[23] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, Jun. 2015.

[24] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, "SMURF: A SVM-based Incremental Anti-pattern Detection Approach," *2012 19th Working Conference on Reverse Engineering*, pp. 466–475, Oct. 2012.

[25] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, 2012.

[26] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 305–314.

[27] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, "Experience report: Evaluating the effectiveness of decision trees for detecting code smells," in *IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, ser. ISSRE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 261–269.

[28] T. M. Mitchell, *Machine learning*, ser. McGraw-Hill series in computer science. Boston (Mass.), Burr Ridge (Ill.), Dubuque (Iowa): McGraw-Hill, 1997. [Online]. Available: http://opac.inria.fr/record=b1093076

[29] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, "Facilitating software refactoring with appropriate resolution order of bad smells," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 265–268. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595738

[30] M. Hozano, N. Antunes, B. Fonseca, and E. Costa, "Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers," in *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems, Porto, Portugal, April 26-29*, 2017 (in press).

[31] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, pp. n/a–n/a, sep 2015. [Online]. Available: http://doi.wiley.com/10.1002/smr.1737

[32] N. Roperia, *JSmell: A Bad Smell detection tool for Java systems*. California State University, 2009.

[33] M. Hozano, "ICPC 2017 - Replication Package," http://goo.gl/tuvxUy, March, 2017.

[34] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[35] N. Maneerat and P. Muenchaisri, "Bad-smell prediction from software design model using machine learning techniques," *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 331–336, May 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5930143

[36] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[37] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells," *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 101–110, Sep. 2014.