Towards a Taxonomy of Bad Smells Detection Approaches

Mouna Hadj-Kacem and Nadia Bouassida
Mir@cl Laboratory, Sfax University, Tunisia

Keywords: Detection Approaches, Code Smells, Anti-patterns, Design Smells, Taxonomy.

Abstract: Refactoring is a popular maintenance activity that improves the internal structure of a software system while maintaining its external behaviour. During the refactoring process, detecting bad smells plays a crucial role in establishing reliable and accurate results. So far, several approaches have been proposed in the literature to detect bad smells at different levels. In this paper, we focus on reviewing the state-of-the-art of object-oriented bad smells detection approaches. For the purpose of comparability, we propose a hierarchical taxonomy by following a development methodology. Our taxonomy encompasses three main dimensions describing the detection approach via the used method, analysis and assessment. The resulting taxonomy provides a deeper understanding of existing approaches. It highlights many key factors that concern the developers when making a choice of an existing detection approach or when proposing a new one.

1 INTRODUCTION

With the growth of software complexity, maintenance has become increasingly more arduous and time-consuming phase. According to many researchers in the field, the cost required to carry out maintenance activities accounts for about 90% of the total project budget (Erlikh, 2000). The main reason behind the high cost associated with this phase is driven by the continuous changes that occur throughout the software life cycle. These changes, like new environmental constraints and imposed customer requirements, may lead to the emergence of several design problems that negatively impact the software quality.

Presence of bad smells (Fowler et al., 1999) is one of the most known and serious design problems that frequently deteriorates the software structure and hence quality. According to (Fowler et al., 1999), a bad smell is defined as 'a surface indication that usually corresponds to a deeper problem in the system'. In the literature, many studies have investigated empirically the impact of bad smells on the software quality (Soh et al., 2016; Khomh et al., 2012) and indicated that their presence makes the system more fault-tolerant and more difficult to maintain and evolve. Mostly, this kind of problems is introduced unwittingly and unknowingly by developers either due to inappropriate design decisions or the lack of experience and practice. So, in order to avoid problems inherent to bad smells, a set of refactoring operations should be performed (Fowler et al., 1999).

In fact, refactoring is recognized as one of the efficient maintenance activities that aim at improving the software quality with economical costs (Mens and Tourwe, 2004). As stated by (Fowler et al., 1999), refactoring is a process that applies changes in the internal software structure without altering its external behaviour. The starting point in the refactoring process performs through the search and identification of bad smells locations, called detection phase (Mens and Tourwe, 2004). Then, when accurately detected, a set of refactoring operations are applied and afterwards evaluated in order to ensure the preservation of the software quality. In the refactoring process, the detection phase plays a fundamental role since its results have a decisive impact on the other phases.

In this context, a wide range of approaches have been suggested to detect bad smells at both source and model levels. Each approach has its own particular strengths and weaknesses whose understanding can provide for a fair comparison and a decisive choice between them based on the developer's needs. Towards this end, we provide in this paper a survey of the research studies that detect bad smells in object-oriented software systems. The outcome of this phase is a classification of the existing approaches into a comprehensive taxonomy that is broadly represented by three different dimensions depending upon the used methods, analysis and assessment criteria. The aim of the proposed taxonomy is to give a thorough overview of different approaches in order to help developers to understand and compare between

Hadj-Kacem, M. and Bouassida, N.
Towards a Taxonomy of Bad Smells Detection Approaches.
DOI: 10.5220/0006869201640175
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 164-175
ISBN: 978-989-758-320-9

existing detection approaches.

It is important to note that, in the surveyed studies, bad smells are interpreted under several terms, including but not limited to, code smells, design smells, anti-patterns (Brown et al., 1998), design flaws (Salehie et al., 2006), etc. In the detection approaches, some authors treated them indifferently, while others consider minor differences among them. To a certain extent, these terms share the similar basic principle of bad smells; they refer to poor solutions of recurring design and implementation problems. Nonetheless, they differ in other aspects like granularity and the abstraction level.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 presents the taxonomy and explains its development methodology. In the following three sections, the main dimensions of the taxonomy are separately described in detail. Section 7 concludes with some recommendations for future research directions.

2 RELATED WORK

Historically, the term of bad smells has been a subject of interest for many years. A number of books have been written about them and the way to mitigate their negative effects. Over time, several designations have been appeared in the literature referring to the same problem, with some differences in the granularity levels.

Because of the ambiguity inherent to the similar definitions, several taxonomies have been proposed to facilitate their analysis and understanding. (Mantyla et al., 2003) suggested a taxonomy that categorizes similar bad smells into seven classes (bloaters, object-orientation abusers, change preventers, dispensables, encapsulators, couplers, and others). Besides the taxonomy, the authors recognized the existing correlations between the smells. Later, (Moha et al., 2005; Moha et al., 2010) performed another categorization which includes both anti-patterns and code smells. These two latter defects are classified into intra-class (related to the inner workings of classes) and inter-class (related to the relationships among classes). In another work (Ganesh and Sharma, 2013), the authors tried to resolve the ambiguity between the different concepts reported in the literature. They presented their own observation at cataloguing and classifying design smells based on the object-oriented design principles that are violated.

As mentioned in the introduction, bad smells are known under a variety of terms in many studies. Even though the original definitions stated that bad smell presents a probably defect and anti-pattern is an actual defect in the system, they are however frequently treated as synonymous because of the considerable overlaps existing between them. For example, the code smell God Class and the anti-pattern Blob are interpreted as similar in the detection approaches. Accordingly, in our taxonomy, we also do not make a distinction between these terms in order to cover a large variety of interesting detection approaches.

Several studies have been carried out on the detection of bad smells, but very few researchers have performed literature reviews related to the study of the developed approaches. According to the abstraction level, the previous works can be divided into two main categories. At code level, (Ghulam and Zeeshan, 2015) have presented a systematic literature review complemented by a snowballing. Some types of techniques were excluded from the review; they are manual code smell detection techniques and duplicated code smell detection techniques. In a second part, the authors conducted an empirical comparison between three available tools on four code smells. (Palomba et al., 2014) have performed a literature review of 11 anti-patterns defined by (Brown et al., 1998; Fowler et al., 1999) together with 6 linguistic anti-patterns defined by (Arnaoudova et al., 2013). The authors have reviewed only the methods used for the detection of the 11 anti-patterns. For the linguistic anti-patterns, the methods for their detection were not provided. At model level, (Din et al., 2012) examined 11 papers that have appeared between 2009 and 2011. Mainly, they have discussed the basic functionalities of each approach. However, they did not provide their own classification for recognizing them.

Unlike the aforementioned works, the aim of the present paper is to build a broader taxonomy that covers the life cycle of the approaches performing detection of bad smells at either source or model level.

3 TAXONOMY DEVELOPMENT METHODOLOGY

In this paper, we present a current state-of-the-art covering researches on bad smells detection approaches at source as well as at model level. Our aim is to make an attempt towards a comprehensive taxonomy encompassing the existing detection approaches. On the one hand, this taxonomy reflects the evolving trajectory of research in the identification of design problems. On the other hand, it is intended to help developers to make appropriate decisions.

The process we followed for the taxonomy development is based on the basic guidelines sugge-

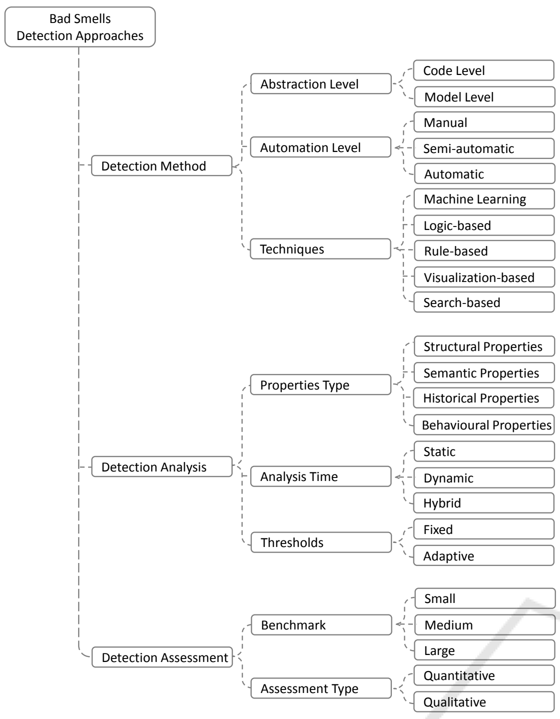

Figure 1: Hierarchy of the proposed taxonomy.

sted by (Nickerson et al., 2013). It focuses on the combination of the empirical-to-deductive and deductive-to-empirical approaches in order to identify the main dimensions and their corresponding meta-characteristics.

According to (Nickerson et al., 2013), the process of the taxonomy development starts with the definition of meta-characteristics in order to define later the corresponding dimensions. Then, the ending conditions are drawn to portray the expected structure. After that, a number of iterations are performed through two different approaches that are empirical-to-deductive and deductive-to-empirical. In the first one, a subset of the collected data is categorized according to the common meta-characteristics and dimensions. In the second one, the meta-characteristics and dimensions are extracted from the literature. When the ending conditions are met, the iterations stopped and the process of the taxonomy development ends.

Our proposed taxonomy encompasses three dimensions: detection method, detection analysis and detection assessment. For more details, our three-dimensional taxonomy is graphically depicted in Figure 1. The dimensions are examined in turn in the next sections.

4 DETECTION METHOD

The first dimension in our taxonomy focuses on presenting the approach in a general way to put it in context. It deals with the abstraction level (where), the automation level (how) and the used techniques (what). Each one of them is explained in more details in the following.

4.1 Abstraction Level

Since the focus of this paper is on object-oriented detection approaches, two major abstraction levels are considered. They are mainly code level (called code smells) and model level (called design smells). As shown in Table 1, we found that at least two-thirds of detection approaches are applied at code level.

Code Level. The majority of approaches performed at code level because of its richness with extra information useful for the detection of bad smells, e.g., number of lines, number of parameters, etc. However, despite the helpful information coming from the source code, the detection at this level is considered too late. As stated by (Akiyama et al., 2011), it is significantly practical to identify and fix bad smells as early as possible at the model level in order to gain effort in later steps of development.

Model Level. The detection at the model level is more challenging than at the code level. The challenge is due to the different modelling notations and diagrams upon which the model can be based. For example, the detection approach of (Travassos et al., 1999) is based on three UML diagrams: state diagram, class diagram and sequence diagram. As the two latter diagrams are the most popular diagrams in object modelling, detection approaches at model level are generally based on one or both of them. In (Fourati et al., 2011), the authors have used and extracted the needed information of detection through these two diagrams to identify five anti-patterns.

Recently, a study (Karasneh et al., 2016) has empirically investigated the translation of four anti-patterns occurrences from the models to the source code. The authors have shown that around 37% of the affected classes in the models persist at the code level under the same type of anti-pattern. So, as the anti-patterns appear early in the design phase, it is strongly recommended to identify them before the coding phase. However, despite the importance of detecting early the anti-patterns at the model level, this task is challengeable and less covered in the literature (see Table 1). On the one hand, the lack of semantic traceability among the various diagrams hinders the detection. On the other hand, as argued by (Ghannem et al., 2016), the main issue at this level is that there are many metrics applied during the detection at the code level cannot be mapped to the model level. Consequently, an extra effort is needed to tackle this hard task.

4.2 Automation Level

The detection approaches can be classified based on their automation level into three categories: manual, semi-automatic and automatic.

Manual. The manual approaches offer basically human processes. They are conducted by experts who spent too much time and effort to identify inconsistencies by inspecting the design parts that correspond to the definition of bad smells. Detecting manually bad smells is regarded as a costly, time-consuming and error-prone procedure. When analysing large-scale systems, the identification becomes more tedious and complicated. In addition, it is subjective because it depends on the expert's personal perception.

In his book, (Fowler et al., 1999) listed 22 code smells together with guidance rules to locate them. For each code smell, they proposed a set of refactoring operations to limit their negative impact. Also, the approach led by (Travassos et al., 1999) was one of the first manual detection approaches. The authors define a set of inspection rules referred to as reading techniques that help to detect defects in UML artifacts. Each inspection rule is a guideline assisting the designer to identify inconsistencies in class, sequence and state diagrams.

Semi-automatic. In order to avoid the aforementioned limitations of manual detection, semi-automatic approaches have emerged as a partial solution. They are a good compromise between fully automatic detection techniques that can be efficient but loose track of context, and pure human inspection that is slow and inaccurate (Langelier et al., 2005). In this type of detection, the decision making about whether an anomaly candidate is an actual defect or not, is asserted by an expert. According to (Dhambri et al., 2008), when facing a complicated situation, a human intervention is mandatory to provide its own perception.

Although they are less tedious than manual detection, the semi-automatic detection approaches are still time-consuming and subjective as they necessitate a human intervention.

Automatic. Several approaches have appeared aiming at the automation of the whole detection process to overcome the above problems of manual and semi-automatic ones. They reflect the extent to which the tool could examine the system by itself without requiring a human intervention. Many methods have been exploited to reduce the time of the detection in particularly large-scale systems. However, despite the significant progresses with the automatic approaches, there is still a need for an additional effort to standardize and calibrate bad smells definitions in a formal way to optimize the detection results (De Mello et al., 2017).

4.3 Techniques

Surveyed detection techniques consist of five broad categories: machine learning, logic-based, rule-based, visualization-based and search-based.

Machine Learning. Machine learning techniques are an effective way to detect the existence of bad smells. They are based on a training set of information collected and evaluated by experts. A classifier learns using the training set of information, and it is then used for testing systems to detect bad smells.

Different learning algorithms have been used in the detection approaches, including Support Vector Machines (Maiga et al., 2012a; Maiga et al., 2012b), Bayesian Belief Networks (Khomh et al., 2009; Khomh et al., 2011), Association Rule Mining (Palomba et al., 2015; Fu and Shen, 2015), etc.

Recently, (Arcelli Fontana et al., 2016) have experimented with 16 supervised machine learning algorithms on four code smells that are Data Class, Large Class, Feature Envy and Long Method. The selected algorithms are J48 (with pruned, unpruned and reduced error pruning), JRip, Random Forest, Naïve Bayes, SMO (with Radial Basis Function and Polynomial kernels) and LibSVM (with the two algorithms C-SVC and ν-SVC in combination with Linear, Polynomial, RBF and Sigmoid kernels).

Logic-based. Logical reasoning represents another interesting alternative for detecting bad smells. Generally, it relies on a mathematical basis and it is able to provide faster results. (Tourwe and Mens, 2003) have demonstrated a logic-based approach that uses a logic meta-programming for detecting bad smells and for providing the appropriate set of refactoring operations. The logic programming language used for the implementation is called SOUL, known as a variant of Prolog with some minor differences. Similarly, (Stoianov and Şora, 2010), proposed a logic-based detection approach using Prolog predicates. According to the authors, the approach is characterized by its simplicity since it is able to detect 5 design-patterns and 6 anti-patterns by means of their structural and behavioural aspects. The achieved results show that no false positives were found among the automatic detected anti-patterns. These results are proven by a manual inspection of the code.

Rule-based. Most of existing bad smells detection approaches are based on rules that are established at one of the mentioned automation levels. Generally, the rules tend to closely describe the characteristics of bad smells via a particular combination of known and/or newly defined metrics. For each bad smell, its related rules are implemented through combining a metric with its appropriate threshold. However, defining the right threshold value of a given metric for a given bad smell is not obvious and necessitates a significant analysis effort because there is no consensus for their calibration. Therefore, several interpretations exist to translate the detection of the same bad smell definition into a rule. This non-compromise may conduct to the decrease of accuracy results reported by this type of approaches.

A detection strategy suggested by (Marinescu, 2004) formulates metrics-based rules that capture deviations from good design principles and heuristics. The strategy is based on two mechanisms of filtering and composition. The filtering aims to detect design fragments with specific properties captured by a metric. However, the composition is based on a set of operators which are used to associate the metrics in an articulated rule. Likewise, (Munro, 2005) proposed precise definitions of bad smells based on the informal descriptions provided originally by (Fowler et al., 1999) in order to prevent the diversity of interpretations varying between developers. The proposed specifications are in form of template that consists of three parts: the bad smell name, main descriptions of its characteristics and design heuristics for its identification. The rules are constructed using if-then-else conditional statements combining the metric with its threshold.

(Moha et al., 2010) proposed a method called DETEX (DETection EXpert). The DETEX is an instance of DECOR (DEtection & CORrection) that represents all the steps necessary for the specification and detection of bad smells. Using a domain-specific language, DETEX automatically generates the detection algorithms. These algorithms are obtained from the models of rule cards which describe the properties of a class having been considered as a smell. Besides the detection, this approach enables the correction of the defects. (Hozano et al., 2015) exploited the developers' feedback of specific projects in order to produce personalized rules. Four code smells are recognized as relevant according to the collected feedback.

Visualization-based. (van Emden and Moonen, 2002) presented a code smell browser called jCOSMO which detects and visualizes code smells in java code. After having parsed the source code, the tool shows a graphical overview containing the parts affected by bad smells besides the relations between them. In another work, (Langelier et al., 2005) proposed a visualization framework that exploits perception capabilities of the human visual system in order to support quality analysis in software systems. Based on the main features of the latter approach, (Dhambri et al., 2008) propose a detection approach combining automatic pre-processing with visual representation and analysis of data. The authors develop a tool called VERSO. For a given system, the tool generates 3D representations of anti-pattern locations that necessitate the intervention of a human analyst to provide his judgement.

Unlike the other visualization tools which are limited to represent modular structures based on a single view, a multiple-view approach enriched with concern properties is proposed by (Carneiro et al., 2010). The authors develop SourceMiner, an Eclipse plug-in that integrates different sets of view and is able to detect three code smells: Feature Envy, God Class and Divergent Change. Another Eclipse plug-in called Stench Blossom for detecting code smells is suggested by (Murphy-Hill and Black, 2010). The plug-in provides the programmer with three different views: ambient, active and explanation views. These views are designed to give developers progressively more informations on the bad smells in the code being visualized.

Search-based. Search-based techniques have been more recently considered in bad smells detection approaches. They are used in SBSE (Search-Based Software Engineering) to solve optimization problems in many domains. When a software engineering issue is structured as a search problem, different meta-heuristic search techniques like genetic programming, simulated annealing and chemical reaction optimization can solve it. For example, the genetic programming is used by (Ouni et al., 2013) in their approach to automatically generate detection rules. For the approach proposed in (Ghannem et al., 2016), the authors used a meta-heuristic search based on genetic algorithms. By exploiting an existing corpus of known design defects, the approach detects three anti-patterns in UML class diagrams. Also, (Kessentini et al., 2011) have used and compared between three search algorithms that are Harmony Search, Particle Swarm Optimization, and Simulated Annealing. The proposed approach is based on a dataset of examples containing only the instances of the detected defects. The dataset of examples consists of detected design defects that are collected from previously inspected projects.

(Sahin et al., 2014) were the first to propose a method for generating code smells detection rules as a Bi-Level Optimization Problems (BLOPs). The approach is composed of two levels of optimization tasks: lower-level and upper-level. In their adaptation, the upper-level optimization generates a set of detection rules in order to cover a base of code smell examples and populate the lower-level with artificial code smells. The lower-level optimization will generate a maximum number of artificial code smells that cannot be identified by the rules produced by the upper-level.

5 DETECTION ANALYSIS

The second taxonomy's dimension describes the main features of the approach during the analysis.

5.1 Properties Type

There are four categories of properties used in the analysis: structural, semantic, historical and behavioural properties.

Structural Properties. As depicted in Table 1, the majority of approaches exploit structural properties in the detection. This type of properties relates to the structure of the system constituents ranging from fine-grained elements to coarse-grained elements (e.g. class, interface, method, field, parameter, etc.) (Moha et al., 2010). For instance, LOC (Lines of Code), CYCLO (Cyclomatic Complexity) and ATFD (Access To Foreign Data) are some examples of metrics that respectively characterize the size, complexity and coupling that are related to the software quality (Lanza and Marinescu, 2007). Based on the structural information, the metric values needed for the detection are calculated.

In addition to the standard metrics in the literature, it is possible to develop complementary metrics to deal with specific requirements. In (Fourati et al., 2011), four new metrics are created at the model level that describe both the complexity and coupling of the system. In the same way, (Nongpong, 2015) proposed a new metric called FEF (Feature Envy Factor) to detect the smell of Feature Envy.

Semantic Properties. The semantic properties rely on linguistic information provided by the main characteristics of bad smells classes. According to (Dhambri et al., 2008), semantic properties are not represented in the source code in an explicit manner as they refer to the application domain knowledge. In their approach, these properties are required by the analyst to ensure that the inspected fragment plays the targeted roles. In (Fourati et al., 2011), the determination of the semantic properties of anti-patterns is handled through lexical dictionaries like the WordNet dictionary. The proposed approach detects five anti-patterns in UML diagrams. It relies on a set of existing and other design metrics newly defined by the authors. In this approach, three types of model properties were considered: structural and semantic informations extracted from the class diagram, as well as behavioural informations extracted from the sequence diagram.

Historical Properties. (Ratiu et al., 2004) extended the original concept of detection strategy (Marinescu, 2004) by extracting historical information from the suspected defect structure. The authors define history measurements that describe the evolution of the bad smells and then combine the results with the original detection strategies. The proposed approach is evaluated on God Class and Data Class, and provides more accurate detection results. (Palomba et al., 2013; Palomba et al., 2015) proposed an approach named HIST (Historical Information for Smell deTection) to detect five types of bad smells. In this work, only historical information extracted from version control systems was used. Likewise, (Fu and Shen, 2015) proposed a detection approach by mining the evolutionary history of projects extracted from revision control system. Three code smells are chosen to be detected from 5 projects, whose the duration of the evolutionary history vary from 5 to 13 years.

Behavioural Properties. In (Fourati et al., 2011), the extraction of behavioural properties is based on the transformation of the sequence diagram into an XML document and metric calculation such as method calling, the sender, the receiver, etc. Also, (Stoianov and Şora, 2010) used the Prolog predicates to describe behavioural properties of both design patterns and anti-patterns.

5.2 Analysis Time

Analysis time identifies the moment when bad smells are detected. Three types of analysis time are defined: static, dynamic and hybrid.

Static. As summarized in Table 1, static analysis is the most commonly used way to analyse a system code. It takes into account the examination of either the source code form or the meta-model form that is the abstract representation of the code. Static analysis is still the most adopted choice because of its simplicity and rapidity comparing to the dynamic analysis which is costly as it necessitates more time and other resources for the running of the program. However, in such a scenario, this type of analysis can indefinitely make hypothesis about the running behaviour of a system. Consequently, many detected bad smells could be rarely or never executed, which leads to the problem of false positives. For this reason, few studies recently opted for combining static with dynamic analysis (Ligu et al., 2013).

Dynamic. Dynamic analysis is performed during the execution of the system under evaluation. In other words, it is the inspection of a running system, such as a unit test. (Kumar and Chhabra, 2014) proposed an approach based on dynamic analysis to detect Feature Envy smell. Their approach consists of two levels. At the first level, the methods that may suffer from the bad smell Feature Envy are obtained dynamically. At the second level, the non-suspect methods are eliminated and thus the detection becomes more efficient as the overhead is reduced. According to the authors, the dynamic analysis is more accurate in object-oriented environment.

Hybrid. Hybrid analysis is the combination of static and dynamic analysis. In (Ligu et al., 2013), the authors proposed an approach that is implemented as an extension of the detecting JDeodorant Eclipse plug-in (Fokaefs et al., 2007). Based on an hybrid analysis, the approach is able to detect the code smell Refused Bequest more accurately than using only static analysis. The static analysis of the source code is employed to identify suspicious hierarchies. However, the dynamic analysis serves to determine the subclass that actually exhibits the smell.

5.3 Thresholds

The detection results are heavily affected by the diversity and the estimation methods of threshold values. Because of the lack of a standard method for measuring suitable thresholds, the detection approaches calculate these values in different ways (Fontana et al., 2016). While some rely on defining fixed thresholds based on expert knowledge, others propose adaptive threshold values. In the literature, several alternatives exist to overcome this problem (Alves et al., 2010; Oliveira et al., 2014; Fontana et al., 2015; Liu et al., 2016), but there is not yet an optimal solution that fulfils all the requirements to equally perform an accurate detection.

Fixed. A fixed threshold is a pre-defined value that pertains to the expert knowledge and opinion. This type of threshold may be interpreted incorrectly as it is inflexible. In fact, the estimation of appropriate thresholds depends on many factors, such as the size of the system, its application domain, the organization best practices, the perception of the developer who defines these values (Fontana et al., 2012). However, a fixed threshold value cannot deal with the variation and the acceleration information of the system during its evolution in the way that the adaptive threshold does.

Adaptive. Unlike the fixed threshold, the adaptive threshold is more flexible as it is based on one or a set of the system features upon which its value is formulated in different ways. In many research studies, the calibration of threshold values is inferred by a tuning machine (Mihancea and Marinescu, 2005). By creating a dataset of bad smell instances, the tuning machine selects the thresholds values. The larger the dataset is, the more accurate the threshold will be.

6 DETECTION ASSESSMENT

This dimension refers to the assessment of the performance and efficiency of the approach in terms of the used benchmark and the assessment type. For the benchmark, we follow the classification suggested in (Radjenovi et al., 2013) that categorizes the systems into small, medium and large according to the size (number of lines of code and/or number of classes). Then, when finally detected, a quantitative and/or a qualitative assessment is carried out to provide the accuracy of the approach.

6.1 Benchmark

A benchmark is a standard for evaluating the performance of approaches. It is a set of known systems upon which the experiments are conducted. Another purpose of using a benchmark is to enable comparative analysis between studies. Actually, the benchmarks vary in their features, such as programming language (e.g. java, C++), availability (open source, commercial project, constructed project, student project), size (small, medium, large), etc. In our taxonomy, the benchmark categorization depends on the size. Our focus on the benchmark size is justified by its important aspect for determining the external validity of the approach (Radjenovi et al., 2013). In fact, when using only small benchmark, it is highly possible that the approach's accuracy may be affected, and accordingly, the reported results will be subjectively interpreted. Therefore, in order to maximize objectivity, evaluations on different sizes are strongly recommended and particularly on the larger ones. Indeed, this is crucial to ensure the generalization and the validation of the approach's findings.

In order to distinguish between small, medium and large benchmarks, we follow the guidelines presented in (Radjenovi et al., 2013). Based on the statistics information provided by the surveyed studies, the system is classified into one of the three groups. Overall, the statistics information are performed through the lines of code (LOC) and/or the number of classes. For a small benchmark, the lines of code or the number of classes are, respectively, less than 50 KLOC (thousands of LOC) or 200 classes. In a medium benchmark, the number of lines is restricted between 50 KLOC and 250 KLOC, or the number of classes is restricted between 200 classes and 1000 classes. Finally, a benchmark is classified as large if the lines of code or the number of classes are, respectively, more than 250 KLOC or 1000 classes. When a study provides the values of both lines of code and number of classes, the system is classified in the higher category (Radjenovi et al., 2013). For example, if a system contains 100 KLOC and 150 classes, the system is classified as medium according to the lines of code, and as small according to the number of classes. Thus, the system is affected to the higher category which is medium. However, when the needed information is not stated, the assigned category is by default small. The more the benchmark is large and varied, the more the approach proves its efficiency.

When performing a comparison between tools, it is recommended to conduct experiments on known and commonly used benchmark. Thus, the results of each approach can be measured and accordingly they can be compared equally against others. To this end, we extract a list of the benchmark exploited in the surveyed studies. This list includes only open source systems because they are the most frequently used benchmark in the evaluations. Because of lack of space, we will not show the tabulated list but rather illustrate the final result in Figure 2. To improve comparability, this histogram illustrates the systems that are used at least in two studies. Xerces was found the most frequently used system in the evaluation of approaches, followed by GanttProject and Log4J.

6.2 Assessment Type

Once the bad smells are detected, we conclude with the accuracy that is an important factor for determining the whole performance of the approach. In most studies, the accuracy is assessed quantitatively in order to ensure the objectivity of the approach. Nonetheless, there are other studies that are based only on

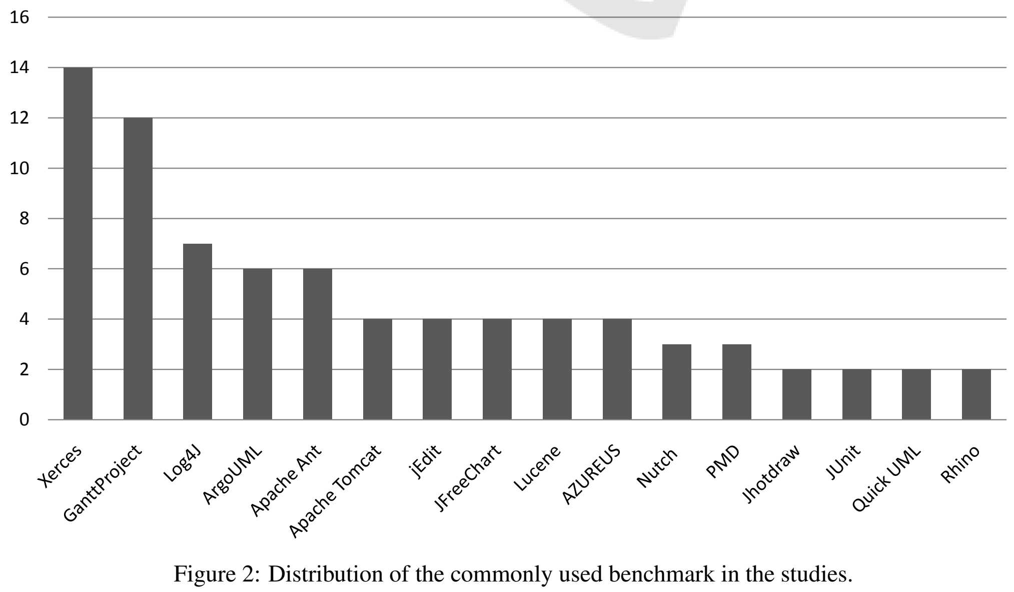

Figure 2: Distribution of the commonly used benchmark in the studies.

qualitative assessment which may decrease their objectivity.

Quantitative. In order to provide an objective assessment, it is necessary to follow a standard method to calculate the accuracy of an approach. Broadly, two metrics are used to assess the accuracy, namely precision and recall. They are well-known metrics in information retrieval (Baeza-Yates et al., 1999). The precision metric indicates the correctness of the approach, it gives the rate of correctly identified bad smells by the number of detected bad smell candidates. However, the recall metric measures the completeness of the approach, it is the rate of correctly identified bad smells by the totally number of actual bad smells. These two metrics are calculated as follows:

$$\begin{aligned} Precision &= \frac{|\{Existing\ Bad\ Smells\} \cap \{Detected\ Bad\ Smells\}|}{|\{Detected\ Bad\ Smells\}|} \\ &= \frac{True\ Positive}{True\ Positive\ +\ FalsePositive} \end{aligned} \tag{1}$$

$$\begin{aligned} Recall &= \frac{|\{Existing\ Bad\ Smells\} \cap \{Detected\ Bad\ Smells\}|}{|\{Existing\ Bad\ Smells\}|} \\ &= \frac{True\ Positive}{True\ Positive\ +\ False\ Negative} \end{aligned} \tag{2}$$

A third metric, called F-measure (Baeza-Yates et al., 1999), is the harmonic mean of precision and recall. Overall, it reflects the whole detection accuracy in one value. It is defined as follows:

$$F-Measure = 2 \times \frac{Precsion \times Recall}{Precsion + Recall} \tag{3}$$

Qualitative. The qualitative assessment is purely based on the judgement of either the authors of the approach or external analysts. This type of evaluation depends on the knowledge and the work-experience of the evaluators. Their own perceptions to the domain may be different and may guide to conflicting results. Therefore, the qualitative assessment is mainly known by its subjectivity because the evaluators may exaggerate or underestimate the actual performance of the approach. For this reason, in the studies using only qualitative assessment, there is a greater possibility that their findings are limited and cannot be generalized. This fact makes impossible to establish a fair comparison with other approaches.

Notwithstanding the aforementioned limitations, it would be interesting to complement this type of assessment with the determination of the precision and recall metrics. Consequently, a high validity is established.

7 CONCLUSION

Throughout the refactoring process, the more accurate are the detection results, the more effective and correct are the later phases. Bad smells detection phase has a decisive influence on the whole refactoring process, and accordingly on the results of the maintenance. Over the last decade and half, this subject has attracted substantial interest from both academia and industry. Numerous approaches have been, and still are, appearing to deal with this challenging problem.

In this paper, we provide a comprehensive taxonomy to classify and characterize detection approaches. This taxonomy is derived from a survey of the research studies that detect bad smells in object-oriented software systems. Three dimensions are proposed to describe the whole detection approach starting from the choice of the used techniques, the analysis characteristics to the assessment criteria. In addition to its knowledgeable value, this taxonomy is addressed to researchers as a basis for classifying and evaluating different works. Also, it can be used by developers who may suggest promising opportunities for creating new approaches or combining between existing techniques.

Owing to its important role in the maintenance, the field of bad smells detection gets more and more attention. Despite the variety of existing detection approaches and the remarkable advances reached up to now, a number of research issues remain open and require further investigation in future works. Among the most common problems encountered during detection process is the ambiguity of bad smells definitions that should be carefully resolved by providing a standard upon which the next detection approaches are based on. Also, there is the issue of threshold that could be alleviated if the first problem is resolved. In fact, a wrong threshold calculation may lead to false or missing detections. Another issue is the impact of the chosen benchmark on the integrity of the obtained results. This may lead to biased comparisons between approaches.

To summarize, it would be fruitful to develop a framework encompassing all the necessary ingredients for the detection process, i.e. formal definitions of bad smells, unified benchmark for the evaluation. Also, it is strongly recommended to broaden the applicability of the detection approaches to commercial systems.

Table 1: Detection approaches according to the proposed taxonomy.

Approaches	Detection Method			Detection Analysis			Detection Assessment	
	Abstraction Level	**Automation Level**	**Techniques**	**Properties Type**	**Analysis Time**	**Thresholds**	**Assessment Type**	**Benchmark**
(Arcelli Fontana et al., 2016)	Code	Automatic	Machine learning	Structural	Static	Adaptive	Quantitative	74 systems S: 19 - M: 38 - L: 17
(Fu and Shen, 2015)	Code	Automatic	Machine learning	Historical Structural	Static	Adaptive	Quantitative	5 systems S: 5 - M: 0 - L: 0
(Kumar and Chhabra, 2014)	Code	Automatic	Search-based	Structural Behavioural	Dynamic	Adaptive	Quantitative	2 systems (selected parts) S: 2 - M: 0 - L: 0
(Munro, 2005)	Code	Semi-automatic	Rule-based	Structural	Static	Fixed	Qualitative	2 case studies S: 2 - M: 0 - L: 0
(Ghannem et al., 2016)	Model	Automatic	Search-based	Structural	Static	Adaptive	Quantitative	4 systems S: 0 - M: 3 - L: 1
(Hozano et al., 2015)	Code	Automatic	Rule-based	Structural	Static	NM	Quantitative	2 systems S: 2 - M: 0 - L: 0
(Nongpong, 2015)	Code	Automatic	Rule-based	Structural	Static	NM	Qualitative	Student Projects S: 2 - M: 0 - L: 0
(Carneiro et al., 2010)	Code	Semi-automatic	Visualization	Structural	Static	NM	Quantitative	An academic project (5 versions) S: 1 - M: 0 - L: 0
(Hassaine et al., 2010)	Code	Automatic	Machine learning	Structural	Static	Adaptive	Quantitative	2 systems S: 1 - M: 1 - L: 0
(Moha et al., 2010)	Code	Automatic	Rule-based	Structural	Static Semantic	Adaptive	Quantitative	10 systems S: 5 - M: 3 - L: 2
(Murphy-Hill and Black, 2010)	Code	Semi-automatic	Visualization	Structural	Static	Fixed	Qualitative	2 systems (selected parts) S: 2 - M: 0 - L: 0
(Stoianov and Şora, 2010)	Code	Automatic	Logic-based	Structural Behavioural	Static	NM	Qualitative	6 systems S: 6 - M: 0 - L: 0
(Sahin et al., 2014)	Code	Automatic	Search-based	Structural	Static	Adaptive	Quantitative	9 systems S: 2 - M: 5 - L: 2
(Dexun et al., 2013)	Code	Automatic	Rule-based	Structural	Static	Fixed	Quantitative	6 systems S: 4 - M: 1 - L: 1
(Ligu et al., 2013)	Code	Automatic	Search-based	Structural	Hybrid	NM	Qualitative	1 system S: 0 - M: 1 - L: 0
(Khomh et al., 2009)	Code	Semi-automatic	Machine learning	Structural	Static	Adaptive	Quantitative	2 systems S: 1 - M: 1 - L: 0
(Dhambri et al., 2008)	Code	Semi-automatic	Visualization	Structural Semantic	Static	NM	Quantitative	2 systems S: 0 - M: 2 - L: 0
(Ouni et al., 2013)	Code	Automatic	Search-based	Structural	Static	Adaptive	Quantitative	6 systems S: 2 - M: 2 - L: 2
(Palomba et al., 2013)	Code	Automatic	Machine learning	Historical	Static	Adaptive	Quantitative	8 systems S: 0 - M: 6 - L: 2
(Saranya et al., 2017)	Model	Automatic	Search-based	Structural	Static	Adaptive	Quantitative	3 systems S: 0 - M: 3 - L: 0
(Salehie et al., 2006)	Code	Automatic	Rule-based	Structural	Static	Fixed	Quantitative	1 system S: 0 - M: 0 - L: 1
(Kreimer, 2005)	Code	Automatic	Machine learning	Structural	Static	NM	Quantitative	2 systems S: 2 - M: 0 - L: 0
(Langelier et al., 2005)	Code	Semi-automatic	Visualization	Structural	Static	NM	Qualitative	1 system S: 0 - M: 0 - L: 1
(Palomba et al., 2015)	Code	Automatic	Machine learning	Historical	Static	Adaptive	Quantitative	20 systems S: 6 - M: 9 - L: 5
(Walter et al., 2015)	Code	Automatic	Rule-based	Structural	Static	Fixed	Quantitative	1 system (selected parts) S: 0 - M: 1 - L: 0
(Kessentini et al., 2014)	Code	Automatic	Search-based	Structural	Static	Adaptive	Quantitative	9 systems S: 2 - M: 5 - L: 2
(Travassos et al., 1999)	Model	Manual	Rule-based	Structural Semantic	Static	NM	Qualitative	An Academic project S: 1 - M: 0 - L: 0
(Maiga et al., 2012a)	Code	Automatic	Machine learning	Structural	Static	Adaptive	Quantitative	3 systems S: 0 - M: 1 - L: 2
(Marinescu, 2004)	Code	Automatic	Rule-based	Structural	Static	Fixed	Quantitative	2 industrial case studies S: 0 - M: 2 - L: 0
(Ratiu et al., 2004)	Code	Automatic	Rule-based	Historical	Static	Fixed	Quantitative	3 case studies S: 2 - M: 1 - L: 0
(Tourwe and Mens, 2003)	Code	Automatic	Logic-based	Structural	Static	NM	Qualitative	1 system S: 1 - M: 0 - L: 0
(Fourati et al., 2011)	Model	Automatic	Rule-based	Structural Behavioural Semantic	Static	Fixed	Qualitative	Several designs collected from literature S: NM - M: 0 - L: 0
(Mansoor et al., 2017)	Code	Automatic	Search-based	Structural	Static	Adaptive	Quantitative	7 systems S: 0 - M: 2 - L: 5
(van Emden and Moonen, 2002)	Code	Semi-automatic	Visualization	Structural	Static	NM	Qualitative	1 academic project S: 1 - M: 0 - L: 0
(Akiyama et al., 2011)	Model	Automatic	Logic-based	Structural	Static	NM	Qualitative	Industrial system S: 1 - M: 0 - L: 0
(Kessentini et al., 2011)	Code	Automatic	Search-based	Structural	Static	Adaptive	Quantitative	2 systems S: 0 - M: 2 - L: 0
(Khomh et al., 2011)	Code	Semi-automatic	Machine learning	Structural	Static	Adaptive	Quantitative	2 systems S: 1 - M: 1 - L: 0
(Oliveto et al., 2010)	Code	Automatic	Machine learning	Structural	Static	Adaptive	Qualitative	2 systems S: 2 - M: 0 - L: 0

S: Small, M: Medium, L: Large

REFERENCES

Akiyama, M., Hayashi, S., Kobayashi, T., and Saeki, M. (2011). Supporting design model refactoring for improving class responsibility assignment. In *Model Driven Engineering Languages and Systems*.

Alves, T. L., Ypma, C., and Visser, J. (2010). Deriving metric thresholds from benchmark data. In *IEEE Int. Conf. on Software Maintenance*, pages 1–10.

Arcelli Fontana, F., Mäntylä, M. V., Zanoni, M., and Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191.

Arnaoudova, V., Penta, M. D., Antoniol, G., and Guhneuc, Y. G. (2013). A New Family of Software Anti-patterns: Linguistic Anti-patterns. In *17th Eur. Conf. on Software Maintenance and Reengineering*.

Baeza-Yates, R., Ribeiro-Neto, B., et al. (1999). *Modern information retrieval*. ACM press New York.

Brown, W. H., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*.

Carneiro, G. d. F., Silva, M., Mara, L., Figueiredo, E., Sant'Anna, C., Garcia, A., and Mendonca, M. (2010). Identifying Code Smells with Multiple Concern Views. In *Br. Symp. on Software Engineering*.

De Mello, R. M., Oliveira, R. F., and Garcia, A. F. (2017). On the Influence of Human Factors for Identifying Code Smells: A Multi-Trial Empirical Study. In *ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement*, pages 68–77.

Dexun, J., Peijun, M., Xiaohong, S., and Tiantian, W. (2013). Detection and refactoring of bad smell caused by large scale. *Int. J. of Soft. Eng. & Applications*.

Dhambri, K., Sahraoui, H., and Poulin, P. (2008). Visual Detection of Design Anomalies. In *12th Eur. Conf. on Software Maintenance and Reengineering*, pages 279–283.

Din, J., AL-Badareen, A. B., and Jusoh, Y. Y. (2012). Anti-patterns detection approaches in Object-Oriented Design: A literature review. In *7th Int. Conf. on Computing and Convergence Technology*, pages 926–931.

Erlikh, L. (2000). Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3):17–23.

Fokaefs, M., Tsantalis, N., and Chatzigeorgiou, A. (2007). JDeodorant: identification and removal of feature envy bad smells. In *Int. Conf. on Soft. Maintenance*.

Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*.

Fontana, F. A., Dietrich, J., Walter, B., Yamashita, A., and Zanoni, M. (2016). Antipattern and code smell false positives: preliminary conceptualization and classification. In *23rd IEEE Int. Conf. on Software Analysis, Evolution, and Reengineering*, pages 609–613.

Fontana, F. A., Ferme, V., Zanoni, M., and Yamashita, A. (2015). Automatic metric thresholds derivation for code smell detection. In *6th IEEE/ACM Int. Workshop on Emerging Trends in Software Metrics*, pages 44–53.

Fourati, R., Bouassida, N., and Abdallah, H. B. (2011). A metric-based approach for anti-pattern detection in uml designs. In *Computer and Information Science*.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: improving the design of existing code*.

Fu, S. and Shen, B. (2015). Code bad smell detection through evolutionary data mining. In *Int. Symp. on Empirical Soft. Engineering and Measurement*, pages 1–9.

Ganesh, S. and Sharma, T. (2013). Towards a Principle-based Classification of Structural Design Smells. *Journal of Object Technology*.

Ghannem, A., El Boussaidi, G., and Kessentini, M. (2016). On the use of design defect examples to detect model refactoring opportunities. *Software Quality J.*

Ghulam, R. and Zeeshan, A. (2015). A review of code smell mining techniques. *J. of Soft.: Evolution and Process*.

Hassaine, S., Khomh, F., Gueheneuc, Y. G., and Hamel, S. (2010). IDS: an immune-inspired approach for the detection of software design smells. In *7th Int. Conf. on the Quality of Information and Communications Tech.*

Hozano, M., Ferreira, H., Silva, I., Fonseca, B., and Costa, E. (2015). Using Developers' Feedback to Improve Code Smell Detection. In *30th Annual ACM Symp. on Applied Computing*, pages 1661–1663.

Karasneh, B., Chaudron, M. R. V., Khomh, F., and Gueheneuc, Y. G. (2016). Studying the Relation between Anti-Patterns in Design Models and in Source Code. In *23rd IEEE Int. Conf. on Software Analysis, Evolution, and Reengineering*, volume 1, pages 36–45.

Kessentini, M., Sahraoui, H., Boukadoum, M., and Wimmer, M. (2011). Search-based design defects detection by example. In *Fundamental App. to Soft. Eng.*

Kessentini, W., Kessentini, M., Sahraoui, H., Bechikh, S., and Ouni, A. (2014). A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection. *IEEE Trans. Softw. Eng.*

Khomh, F., Penta, M. D., Guéhéneuc, Y.-G., and Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275.

Khomh, F., Vaucher, S., Guhneuc, Y. G., and Sahraoui, H. (2009). A Bayesian Approach for the Detection of Code and Design Smells. In *9th Int. Conf. on Quality Software*, pages 305–314.

Khomh, F., Vaucher, S., Guhneuc, Y.-G., and Sahraoui, H. (2011). BDTEX: A gqm-based Bayesian approach for the detection of antipatterns. *J. of Sys. and Software*.

Kreimer, J. (2005). Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*.

Kumar, S. and Chhabra, J. K. (2014). Two level dynamic approach for Feature Envy detection. In *Int. Conf. on Computer and Communication Technology*.

Langelier, G., Sahraoui, H., and Poulin, P. (2005). Visualization-based Analysis of Quality for Large-scale Software Systems. In *20th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 214–223.

Lanza, M. and Marinescu, R. (2007). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*.

Ligu, E., Chatzigeorgiou, A., Chaikalis, T., and Ygeionomakis, N. (2013). Identification of refused bequest code smells. In *Int. Conf. on Software Maintenance*, pages 392–395.

Liu, H., Liu, Q., Niu, Z., and Liu, Y. (2016). Dynamic and automatic feedback-based threshold adaptation for code smell detection. *IEEE Trans. Softw. Eng.*

Maiga, A., Ali, N., Bhattacharya, N., Saban, A., Guhneuc, Y. G., and Aimeur, E. (2012a). SMURF: A SVM-based Incremental Anti-pattern Detection Approach. In *19th Working Conf. on Reverse Engineering*.

Maiga, A., Ali, N., Bhattacharya, N., Saban, A., Guhneuc, Y. G., Antoniol, G., and Ameur, E. (2012b). Support vector machines for anti-pattern detection. In *27th Int. Conf. on Automated Software Engineering*.

Mansoor, U., Kessentini, M., Maxim, B. R., and Deb, K. (2017). Multi-objective code-smells detection using good and bad design examples. *Software Quality J.*

Mantyla, M., Vanhanen, J., and Lassenius, C. (2003). A taxonomy and an initial empirical study of bad smells in code. In *Int. Conf. on Software Maintenance*.

Marinescu, R. (2004). Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE Int. Conf. on Software Maintenance*, pages 350–359.

Mens, T. and Tourwe, T. (2004). A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139.

Mihancea, P. F. and Marinescu, R. (2005). Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems. In *9th Eur. Conf. on Software Maintenance and Reengineering*.

Moha, N., Gueheneuc, Y. G., Duchien, L., and Meur, A. F. L. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Softw. Eng.*, 36(1):20–36.

Moha, N., Huynh, D.-l., Guéhéneuc, Y.-G., and Team, P. (2005). A Taxonomy and a First Study of Design Pattern Defects. *STEP 2005*, page 225.

Munro, M. J. (2005). Product metrics for automatic identification of bad smell design problems in Java source-code. In *11th IEEE Int. Software Metrics Symp.*

Murphy-Hill, E. and Black, A. P. (2010). An Interactive Ambient Visualization for Code Smells. In *5th Int. Symp. on Software Visualization*, pages 5–14. ACM.

Nickerson, R. C., Varshney, U., and Muntermann, J. (2013). A method for taxonomy development and its application in information systems. *Eur. J. of Information Systems*.

Nongpong, K. (2015). Feature envy factor: A metric for automatic feature envy detection. In *7th Int. Conf. on Knowledge and Smart Technology*, pages 7–12.

Oliveira, P., Valente, M. T., and Lima, F. P. (2014). Extracting relative thresholds for source code metrics. In *IEEE Conf. on Software Maintenance, Reengineering, and Reverse Engineering*, pages 254–263.

Oliveto, R., Khomh, F., Antoniol, G., and Gueheneuc, Y. G. (2010). Numerical Signatures of Antipatterns: An Approach Based on B-Splines. In *14th Eur. Conf. on Software Maintenance and Reengineering*.

Ouni, A., Kessentini, M., Sahraoui, H., and Boukadoum, M. (2013). Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1):47–79.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., and Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *28th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 268–278.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Poshyvanyk, D., and Lucia, A. D. (2015). Mining Version Histories for Detecting Code Smells. *IEEE Trans. Softw. Eng.*, 41(5):462–489.

Palomba, F., Lucia, A. D., Bavota, G., and Oliveto, R. (2014). Anti-pattern detection: Methods, challenges, and open issues. 95:201 – 238.

Radjenovi, D., Heriko, M., Torkar, R., and ivkovi, A. (2013). Software fault prediction metrics: A systematic literature review. *Inf. and Software Technology*, 55(8):1397 – 1418.

Ratiu, D., Ducasse, S., Gîrba, T., and Marinescu, R. (2004). Using History Information to Improve Design Flaws Detection. In *8th Eur. Conf. on Software Maintenance and Reengineering*, pages 223–232.

Sahin, D., Kessentini, M., Bechikh, S., and Deb, K. (2014). Code-Smell Detection As a Bilevel Problem. *ACM Trans. on Software Engineering and Methodology*.

Salehie, M., Li, S., and Tahvildari, L. (2006). A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. In *14th IEEE Int. Conf. on Program Comprehension*, pages 159–168.

Saranya, G., Nehemiah, H. K., Kannan, A., and Nithya, V. (2017). Model level code smell detection using EGAPSO based on similarity measures. *Alexandria Engineering Journal*.

Soh, Z., Yamashita, A., Khomh, F., and Guhneuc, Y. G. (2016). Do Code Smells Impact the Effort of Different Maintenance Programming Activities? In *23rd IEEE Int. Conf. on Software Analysis, Evolution, and Reengineering*, volume 1, pages 393–402.

Stoianov, A. and Şora, I. (2010). Detecting patterns and antipatterns in software using Prolog rules. In *Int. Joint Conf. on Computational Cybernetics and Technical Informatics*, pages 253–258.

Tourwe, T. and Mens, T. (2003). Identifying refactoring opportunities using logic meta programming. In *7th Eur. Conf. on Software Maintenance and Reengineering*.

Travassos, G., Shull, F., Fredericks, M., and Basili, V. R. (1999). Detecting defects in object-oriented designs: using reading techniques to increase software quality.

van Emden, E. and Moonen, L. (2002). Java quality assurance by detecting code smells. In *9th Working Conf. on Reverse Engineering*, pages 97–106.

Walter, B., Matuszyk, B., and Fontana, F. A. (2015). Including Structural Factors into the Metrics-based Code Smells Detection. In *Scientific Workshop Proc. of XP*.