Towards a systematic approach to manual annotation of code smells

Nikola Luburić, Simona Prokić, Katarina-Glorija Grujić, Jelena Slivka, Aleksandar Kovačević, Goran Sladić, and Dragan Vidaković

nikola.luburic@uns.ac.rs, simona.prokic@uns.ac.rs, katarina.glorija@uns.ac.rs, slivkaje@uns.ac.rs, kocha78@uns.ac.rs, sladicg@uns.ac.rs, vdragan@uns.ac.rs

Faculty of Technical Sciences, University of Novi Sad, Serbia

Author Note

Correspondence concerning this article should be addressed to Jelena Slivka, Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovića 6, 21 000 Novi Sad, Serbia. Email: slivkaje@uns.ac.rs, mobile phone: +381/65-85-17-269.

This research was supported by the Science Fund of the Republic of Serbia, Grant No 6521051, AI-Clean CaDET. Our funders had no involvement in the study design, collection, analysis, and interpretation of the data, writing of the report, or the decision to submit the article for publication.

Author contributions: Nikola Luburić, Jelena Slivka, and Aleksandar Kovačević constructed the initial annotation model and procedure. Nikola Luburić, Simona Prokić, and Katarina-Glorija Grujić executed and refined the procedure to create the dataset. Simona Prokić and Aleksandar Kovačević performed the statistical analysis of the annotation process. Nikola Luburić and Simona Prokić wrote the paper and Goran Sladić, Dragan Vidaković, Jelena Slivka, and Aleksandar Kovačević revised the paper and improved it with valuable suggestions.

Abstract

Code smells are structures in code that indicate the presence of maintainability issues. A significant problem with code smells is their ambiguity. They are challenging to define, and software engineers have a different understanding of what a code smell is and which code suffers from code smells.

A solution to this problem could be an AI digital assistant that understands code smells and can detect and even resolve them. However, it is challenging to develop such an assistant as there are few usable datasets of code smells on which to train and evaluate it. Furthermore, the existing datasets suffer from issues that mainly arise from an unsystematic approach used for their construction.

Through this work, we address this issue by developing a procedure for the systematic manual annotation of code smells. We use this procedure to build a dataset of code smells. During this process, we refine the procedure and identify recommendations and risks for its use. The primary contribution is the proposed annotation model and procedure and the annotators' experience report. The dataset and supporting tool are secondary contributions of our study. Notably, our dataset includes open-source projects written in the C# programming language, while almost all manually annotated datasets contain projects written in Java.

Keywords: code smell dataset, manual annotation, clean code, software quality, refactoring, machine learning

1 Introduction

Software code is written to answer specific requirements and enable use cases required of the complete software solution. These requirements state *what* the code must do (e.g., what output should it produce for the given input) but do not care for *how* it is designed or implemented. This abstraction, coupled with the software's softness, has the following consequence – a requirement can be fulfilled by a near-infinite set of different code configurations. Even when limited to a single programming language and a simple requirement, it is easy to list many code samples that fulfill the requirement using different coding styles and language features.

While many code solutions can fulfill a requirement, not all of them are acceptable. Some solutions cause subtle bugs, performance loss, or expose security vulnerabilities. Furthermore, many of the possible solutions present another severe but less obvious problem. Code that is hard to understand and modify harms the software's maintainability, evolvability, reliability, and testability (Sharma & Spinellis, 2018), introducing technical debt. Such code requires more significant mental effort to process and understand before a programmer can reliably modify it. Consequently, the programmer's morale and productivity decline as they spend more time and energy reading old code (Tom et al., 2013), increasing the overall cost of development (Sharma & Spinellis, 2018). Researchers (Sharma & Spinellis, 2018; Hozano et al., 2018) and software industry leaders (Fowler, 2018; Martin, 2009) note that such solutions suffer from code smells – properties of the code that might harm its readability and understandability, and as a consequence, the related software quality attributes. Removal of harmful code smells results in sustainable software development (Sharma & Spinellis, 2018; Hozano et al., 2018; Fowler, 2018; Martin, 2009).

Unfortunately, removing code smells is not easy, as many code smell definitions are vague and lack a concrete heuristic that can unambiguously determine the smell's presence. For example, the Long Method code smell (Fowler, 2018) is present in functions that try to do too many things (Martin, 2009), requiring the programmer to analyze regions of the function to understand their intent before understanding the overall function. Such methods require much cognitive power to understand (Hozano et al., 2018; Fowler, 2018). Notably, this smell is not strictly tied to the method's length in terms of code lines. Functions with 30 lines of code might not suffer from the Long Method smell if they contain repeated and easy-to-understand instructions. In contrast, functions with ten sophisticated lines might require refactoring. As Fowler (2018) noted, even a single code line might be suitable for extraction into a separate function if its intent is unclear.

Because of their impact on the software's quality and the ambiguity concerning their identification and resolution, it would be helpful if code smells could be automatically detected and even resolved. However, without a clear definition and set of heuristics, it is impossible to rely on simple rules based on metrics and thresholds (e.g., the number of code lines is higher than ten) to automatically identify code smells. Such solutions result in many false positives when the threshold is too low or poor recall when it is high (Sharma & Spinellis, 2018). Furthermore, such solutions do not consider the application domain, context, or architectural style and development practices needed to avoid false-positive instances (Fontana et al., 2016b). More sophisticated artificial intelligence (machine learning) models are needed to understand the code's semantics or provide more advanced smell detection rules.

Azeem et al. (2019) conducted a systematic literature review that analyzes machine learning (ML) approaches used for code smell detection. They concluded that ML models generally outperformed

heuristics-based approaches. However, the authors note that the reviewed studies are affected by several threats to validity. Notably, most studies used small or poorly constructed datasets to train and test their models, limiting their generalizability.

Many larger code smell datasets are automatically labeled using heuristic-based tools (Azeem et al., 2019). Such datasets exclude instances that do not satisfy some threshold, eliminating positive instances that an expert would otherwise identify. While some studies manually filter the generated dataset to remove false positives, there is no way of knowing the number of false negatives (Walter et al., 2018; Palomba et al., 2015). This issue is especially relevant for code smells such as Long Method and Large Class (Fowler, 2018) that significantly depend on the code's semantics and not on, for example, the number of code lines.

Manually annotating code smells is time-consuming (Azeem et al., 2019) and challenging (Hozano et al. 2018), where a high disagreement exists between software engineers on which code snippets suffer from some code smell (Hozano et al., 2018). Taibi et al. (2017) presented code snippets that suffer from some code smell to engineers and found that only 29% of them could name the smell, while 41% could describe the problem imposed by the smell. Because of these issues, there are no large-scale manually constructed datasets.

While the lack of large manually annotated datasets is a problem in and of itself, a more severe issue is that most of the available datasets lack a systematic approach to their construction. As mentioned, some annotation procedures heavily rely on automated tools and ignore false negatives (Azeem et al., 2019; Walter et al., 2018; Fontana et al., 2016a; Rasool & Arshad, 2017). Others purposefully avoid training the annotators, aiming to get pure results (Madeyski & Lewowski, 2020; Palomba et al., 2015). Notably, most studies start with vague definitions of what they are annotating (Hozano et al., 2018) and produce datasets that are not published in a form that can be used for reliable reproduction, as pointed out in (Madeyski & Lewowski, 2020).

In this study, we work towards developing a systematic approach to creating a code smell dataset useful for training machine learning smell detectors. The primary contributions of this paper are:

- the proposed methodology for developing a code smell annotation model,
- the proposed dataset annotation procedure, and
- the annotators' experience report.

By surveying the literature, we defined an initial version of the annotation model and procedure to resolve some threats to the validity of the existing datasets. We then refined our procedure while building a medium-sized corpus[1] for the Long Method and Large Class code smells (Fowler, 2018), which we selected due to their prevalence and impact. Though we derived the annotation model on the example of two code smells, researchers can extend it to other code smells by following the proposed methodology. We report on our annotation experience while following the proposed procedure, highlighting any identified recommendations and risks.

We developed an acquisition tool to simplify the process of collecting data for annotation. The tool is open-sourced as part of the *Clean CaDET* platform (Prokić et al., 2021) and supports data acquisition

[1] The dataset is available at https://github.com/Clean-CaDET/clean-cadet-dataset. Note: For now, we only provide a sample of the dataset. We will publish the full dataset upon publication of this article.

from C# source code repositories. We have also published manuals to help other researchers benefit from our tool[2]. The dataset and supporting tool are secondary contributions of our study. Notably, our dataset includes open-source projects written in the C# programming language, while almost all manually annotated datasets contain projects written in Java. C# language dominates the developers' discussions on code smells' (Tahir et al., 2020) but is poorly supported in code analysis tools (Tahir et al., 2020; AbuHassan et al., 2021). Finally, C# is similar to the Java programming language, making our contributions relevant to the large research community.

The rest of the paper is structured as follows: Section 2 presents the related work. We examine procedures used to create existing datasets and comment on the threats to validity we look to address. In Section 3, we present our annotation model, which includes a generic conceptual model applicable to all code smells, and its concretization for the Long Method and Large Class smells. Section 5 describes our annotation procedure and its composing steps. In Section 0, we present our findings, including the annotators' experience reports, the characteristics of the constructed dataset, and our study's limitations. Finally, Section 7 concludes our work and lists opportunities for future work.

2 Related Dataset Annotation Procedures

Experimental studies on source code usually rely on data from three sources, including commercial projects, academic projects, and open-source projects (Malhotra, 2019). Researchers favor open-source projects, as the study results are relatively easy to reproduce, validate, and compare with other studies (Walter et al., 2018). We reviewed the studies that produced code smell datasets from open-source code snippets and analyzed their annotation procedure.

Walter et al. (2018) developed a dataset from 92 Java open-source projects, which are part of the *qualitas corpus* (Tempero et al., 2010) curated Java code collection for empirical studies. They used 11 automated tools for smell detection, which could collectively identify 14 code smells. They ran each code snippet through a set of tools that could identify a particular smell and defined a label for the percentile agreement (grouped to 25%, 50%, 75%, and 100%) of the tools for the smell. The authors note that the descriptions of code smells are usually vague, and detectors interpret them differently. A dataset generated by heuristics-based tools can be used as a training set. However, manual annotation is necessary to produce better results (Fontana et al., 2016b). Aniche et al. (2016) warn that metrics cannot be examined out of their context. For example, classes that fulfill a specific architectural role (e.g., view models, data transfer objects) present metric value distributions significantly different from other classes. Consequently, code assessment tools that use a single threshold per code metric, regardless of the architectural role of the class in the system, may perform doubtful assessments.

Fontana et al. (2016a) also worked on the *qualitas corpus* (Tempero et al., 2010). They used five automated detectors to identify four code smells, where at least two detectors could identify each smell. This automated annotation identified a set of code smell candidates. Following a semi-random sampling procedure, three MSc students selected code snippets and manually validated 1986 instances, determining that over half (1160) were incorrectly classified. They finally produced a dataset of 420 instances for each smell, where one-third included positive instances, while the rest were negative instances. The authors purposefully selected this distribution for their final dataset to enable machine

[2] The Dataset Explorer tools, along with the documentation, is available at https://github.com/Clean-CaDET/platform/wiki/Module-Dataset-Explorer

learning models to work with a more balanced dataset. However, this unrealistic distribution might affect the generalizability of machine learning models trained on this dataset, as pointed out in (Di Nucci et al., 2018). Concretely, the dataset instance distribution responsible for the model's high performance significantly differs from a realistic software project (Palomba et al., 2018a).

Lenarduzzi et al. (2019) created the technical debt dataset, where they collected 33 Java projects and ran them through a series of automated tools for quality evaluation and code smell detection. The automated tools analyzed historical changes for the selected projects by processing their state for multiple commits and uncovered 38 thousand code smells. All code smells were automatically detected and the authors did not perform manual validation of the results.

Recently, Sharma and Kessentini (2021) published a large dataset of code smells and quality metrics. They automatically analyzed over 55 thousand Java and 31 thousand C# code repositories to determine their quality metrics and code smells. While the size of the dataset is impressive (counting over a million code smells), it was automatically generated using their existing smell detection tool with no manual validation of the results.

Palomba et al. (2015) built a dataset by annotating 20 open-source Java projects for five different code smell instances, defining 243 positive instances. One author examined the projects to identify the initial set of code smell candidates. A second author validated the set of candidates and discarded any false positives. While such a procedure increases the likelihood that the remaining positive instances are correctly classified, it does not account for the false negatives the first author might have made. This issue is particularly relevant for annotating code smells. Other studies have shown a high subjectivity and disagreement among engineers for determining the presence of a code smell in code (Hozano et al., 2018; Mäntylä et al. 2004).

Another study conducted by the same group (Palomba et al., 2018a) presents a dataset made from 30 open-source Java projects, where the authors manually validated 17350 positive instances of 13 different code smells. The authors used an automatic detection tool to gather a list of code smell candidates. Two annotators have manually validated the candidate code smells. The detection tool uses simple rules with low thresholds that overestimate the presence of code smells to ensure a high recall. Significantly, a Long Method smell candidate is selected if a function's number of lines of code (LOC) exceeds the average of the project, while a God Class is detected if a class has LOC above 500 and its cohesion is lower than the average of the project. These rules can vary greatly depending on what the average LOC for the project is. For projects with inexperienced engineers, the average can quickly go above the conventional recommendations from the industry and other research, where functions with LOC above 30 (Hozano et al., 2018; Fowler, 2018; Martin, 2009) and classes with LOC above 100 (Bafandeh Mayvan et al., 2020) might be affected by a smell. Furthermore, even functions with LOC above ten can impose a quality issue if the code is sufficiently complex (Fowler, 2018; Martin, 2009).

Madeyski and Lewowski (2020) developed a dataset from 792 open-source Java projects. A total of 26 software engineers looked for four different code smells and annotated 4770 code snippets, randomly selected from the project pool. As multiple engineers labeled each code snippet, the dataset includes 14739 independent annotations. Notably, 16 annotators individually labeled less than 300 instances (positive and negative) of the four smells collectively. This means that each of the 16 engineers might have labeled less than 50 instances for a specific smell. In our experience, the annotators had to examine many instances for each smell (e.g., over a hundred) before they could annotate it reliably and

consistently, regardless of their previous experience. Therefore, the labels made by these annotators might present a threat to the dataset's validity. To the study's credit, the five most active annotators had an average of 10 years of professional programming experience. Another limitation is that some instances were labeled by a single annotator, which is problematic as the labeling process is error-prone.

Rasool and Arshad (2017) present one of the few studies that include an open-source dataset of code smells found in C# projects. However, this dataset has several limitations, as it includes four C# projects where the annotation was done automatically using simple heuristics. A single MSc student subsequently validated the candidates to produce the final set of code smell instances.

The chief threat to the validity of many annotated datasets is the starting premise – the definition of a code smell. Most studies roughly define a code smell, usually with a few high-level sentences, and in the best case, with illustrative examples. For manually annotated datasets, the studies either purposefully avoid training the annotators or list a short workshop. This preparation might be problematic, as manual code smell detection is subjective (Mäntylä et al. 2004), produces high disagreement among experienced software engineers (Hozano et al., 2018; Taibi et al., 2017), and significantly differs between the scientific literature (Bafandeh Mayvan et al., 2020) and industry best practices (Fowler, 2018; Martin, 2009). Another common limitation is the lack of a usable and easily accessible dataset. Many datasets are not published in a form that can be used for reliable reproduction, missing vital information such as source code revision and URLs to code snippets, as pointed out in (Madeyski & Lewowski, 2020). While not a shortcoming of any one study, almost all available datasets contain projects written in the Java programming language.

In this paper, we develop an annotation model and procedure to address the listed limitations by:

- Establishing a common understanding of the smells, related heuristics, and code characteristics used for their identification through the annotation model and training the annotators appropriately.
- Creating an annotation procedure with opportunities for cross-validation and discussion to align understanding as new coding styles are uncovered.
- Defining a schema for the dataset that supports reliable reproduction and contains the necessary information.
- Building a dataset of code smells in C# software projects.

3 Methodology

The most important consideration for developing a code smell dataset is the definition of the code smell. Without a good understanding of what is being annotated, it is impossible to produce a useful dataset. Providing the annotators solely with code smell definitions is problematic as most definitions are vague textual descriptions (Moha et al., 2010). Consequently, without precise guidelines, annotators highly disagree on what constitutes a code smell (Hozano et al., 2018).

Moha et al. (2010) strived to solve this issue through constructing more precise smell definitions than vague text-based specifications in their framework DECOR. They first analyze the text-based smell definitions to extract the smell vocabulary, i.e., a list of keywords relevant for smell detection. They use the resulting vocabulary to classify smells and build their taxonomy. Based on these results, they build specifications detailing the smell detection rules.

Moha et al.s' (2010) framework DECOR is among the most cited detection tools (Lacerda et al. 2020). However, some experiments (Lacerda et al. 2020) indicate its performance is not as high as reported in the initial experiments conducted by Moha et al. (2010). Azeem et al. (2019) report that the ML-based smell detection techniques outperform heuristic-based approaches. We also emphasize that strict code smell definitions do not perfectly relate to code smell understanding in practice. In our earlier study (Kovačević at al., 2021), we applied multiple metric-based heuristics to the Madeyski and Lewowski (2020) dataset that aimed to extract professional developers' contemporary understanding of code smells. We showed that the ML-based learn-by-example approach outperforms several heuristic-based approaches on this dataset.

Thus, in contrast to DECOR, we do not aim to construct strict definitions of code smells that can straightforwardly be translated into rules to automatically annotate code snippets. Instead, we adopt the ML learn-by-example approach and strive to develop guidelines for creating a necessary dataset. Like Moha et al. (2010), we examine the literature to extract these guidelines. In addition to analyzing smell definitions and rules as Moha et al. (2010), we include empirical studies on how contemporary developers perceive code smells. We also note that we do not consider high-level design smells considered in the DECOR framework. Instead, we focus on the implementation-level code smells.

Figure 1 summarizes the steps of our methodology. We present the first four steps in which we analyzed the existing literature in Sections 3.1 - 3.4. Section 4 presents the derived conceptual code smell model and two concrete examples of this abstraction for Long Method and Large Class annotation. Section 5 presents the dataset annotation procedure used to refine the Annotation Model derived from the literature and annotate a medium-sized Long Method and Large Class dataset. In Section 6, we validate the created dataset.

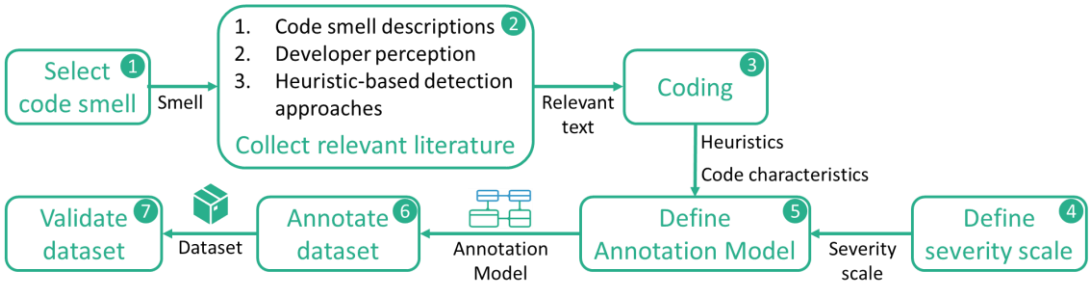

Figure 1 Flowchart summarizing the steps of our methodology

3.1 Select code smells

The first step of our methodology is to select code smells for annotation**.** In this paper, we derived the dataset annotation methodology using a concrete example of two code smells[3], Long Method and Large Class[4]. After we explain the derived annotation model, Section 4.4 will analyze the generalizability of the proposed annotation model to other code smells. We chose Long Method and Large class for the following reasons:

[3] In Section 4.4, we discuss how this model generalizes to other smells.

[4] We consider severe instances of the Large Class code smell to be akin to the God Class (Santos & de Mendonça, 2015).

- **Their harmfulness**. The evidence of code smells harmfulness is conflicting (Fontana et al. 2016a). Azeem et al. (2019) urge the researchers to focus on code smells harmful to the codes' quality. A tertiary systematic review by Lacerda et al. (2020) accentuated Long Method and Large Class code smells as having the highest negative impact on quality attributes and bug-proneness. Palomba et al. (2014a) conducted an empirical study to analyze developers' perception of code smells as actual design or implementation problems. They categorized the smells as (1) not perceived as design problems, (2) may or may not represent a problem based on the smells' intensity, and (3) perceived as significant threats. The smells in the third category are those related to complex/long source code that encompasses the Large Class and Long Method smells.
- **Their prevalence in code**. Palomba et al. (2018a) found that the Long Method and Large Class smells are among most prevalent smells. More alarmingly, they frequently co-occur with other smells (Palomba et al., 2018b), which is shown as particularly problematic (Palomba et al., 2018a; Yamashita & Moonen, 2013).
- **The availability of empirical studies concerning these smells**. Our annotation procedure relies on empirical studies on how developers perceive code smells in practice. Large Class and Long Method are amongst the most investigated smells (Lacerda et al. 2020). Unfortunately, there is a limited number of empirical studies concerning developers' annotations of other code smells (Figure 2).
- **The subjectivity of the annotation task**. Some studies reported high developer agreement for Long Method detection (Mäntylä, 2005; Mäntylä & Lassenius, 2006). However, a recent larger-scale evaluation by Hozano et al. (2018) reported high disagreement for the Long Method and Large Class smells. The dataset produced by Madeyski and Lewowski (2020) also displays non-negligible disagreement between the annotators. Aniche et al. (2016) and Fontana et al. (2016b) warn that classes with specific architectural roles may display characteristics that indicate bad practice but are a natural consequence of classes' specific responsibilities.
- **Their interaction**. Large Classes and Long Methods in isolation may not have a significant effect; however, when combined, they significantly impact codes' quality (Abbes et al., 2011). Therefore, a dataset enabling the detection of both these smells might be helpful to detect such critical cases.

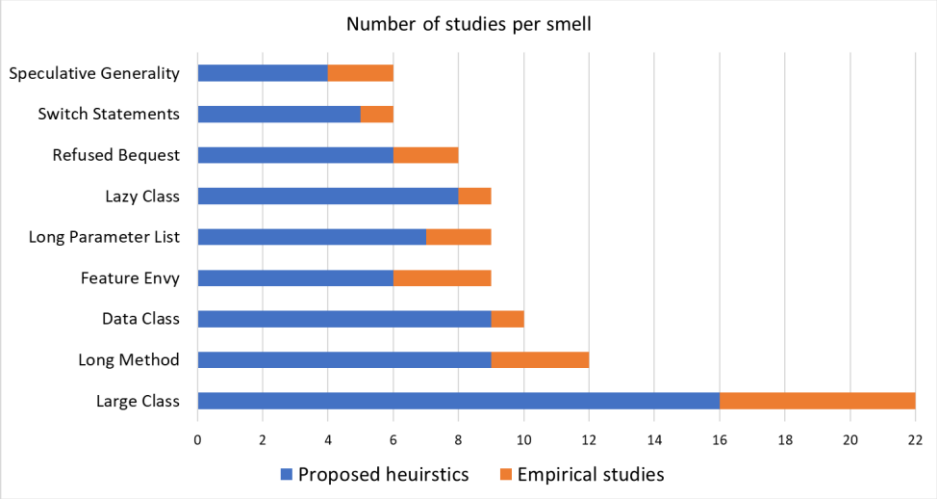

Figure 2 Number of studies per smell for the top 10 smells (we consider preliminary studies and the study that integrates them a single study). "Proposed heuristics" denote the number of heuristics collected in the Systematic Literature Review by Bafandeh Mayvan et al. (2020). "Empirical studies" denotes the number of studies that examine engineer perception of code smells (Section 3.2 describes our process of finding these studies)

3.2 Collect relevant literature

The second step of our methodology collecting the literature describing the code smell**.** We focused on three sources of literature:

1. **Code smell descriptions.** This category contains books, whitepapers, and blogs authored by notable subject matter experts (Fowler, 2018; Martin, 2009; Martin et al., 2003) and tool vendors that specialize in code quality analysis (Campbell, 2018). From this literature, we extracted what engineers look for when doing code reviews and which metrics the quality analysis tools calculate for their smell detection engines.
2. **Developer perceptions of code smells**. We started our search for the literature from a set of "seed" papers. For each seed paper, we performed forward and backward snowballing (Wohlin, 2014) to uncover additional relevant papers. We repeated the forward and backward snowballing process for each relevant paper we identified. Fontana et al. (2016b) conducted a similar search procedure in their meta-synthesis strategy. Here, we explored two categories of scientific literature for related research, including:
    - **Dataset annotation studies**. We examined the literature described in the previous section to extract any annotation guidelines useful to our context. We used the recent systematic literature review (SLR) by Azeem et al. (2019) and the recently published MLCQ dataset by Madeyski and Lewowski, T. (2020) as the seed papers.
    - **Studies that examine engineer perception of code smells.** We used the empirical study by Hozano et al. (2018) as our seed paper for this search. We examined these studies (Hozano et al., 2018; Padilha et al., 2014; Palomba et al., 2017; Santos & de Mendonça, 2015; Palomba et al., 2014a; Mäntylä & Lassenius, 2006; Schumacher et al., 2010) to extract the guidelines stemming from human intuition regarding code smell annotation and smell severity.
3. **Studies of automated smell detection**. We examined studies gathered in the recent systematic literature review on heuristic-based code smell detection (Bafandeh Mayvan et al., 2020) to extract the domain knowledge researchers strived to encode in their algorithms.

Authors NL, JS, and AK read the collected literature and extracted the parts of text specifying code smells, developer reasoning when detecting code smells, and proposed code smell detection heuristics.

3.3 Coding

Annotating a high-quality dataset requires agreeing on annotation guidelines and training the annotators (Hovy, 2010; Ide & Pustejovsky, 2017). Thus, in the coding step, we analyze the extracted relevant text to derive guidelines that help annotators assess the higher-level concept of the code smell. We aim to harmonize the annotator understanding of the code smell annotation task through the extracted guidelines. Authors JS, SP, and NL performed coding independently. Then they resolved disagreements through discussion.

Code smell annotation guidelines improve the annotator agreement. For example, an empirical study on code smell perception by Hozano et al. (2018), purposely omitted precise annotation guidelines or examples to derive developers' unbiased code smell perception. They observed a generally high disagreement between software engineers on the code smell detection task. However, they also perceived that those developers that follow the same heuristics to assess the existence of a given code smell rendered a moderate to a substantial agreement.

Guidelines can also simplify the task for the annotators. Schumacher et al. (2010) investigated the way professional software developers detect the Large Class code smell. They provided the annotators with a list of questions helping them identify Large Classes. One subject emphasized that these questions "help you to organize your mind."

We apply the coding procedure (Seaman, 1999) to derive the related set of heuristics for the code smell. Researchers commonly use coding to analyze qualitative data. For example, Hozano et al. (2018) used coding to analyze developer reasonings collected through an open question on the survey. Schumacher et al. (2010) used coding to analyze comments from the think-aloud phase from the subjects. Moha et al. (2010) used a procedure similar to coding to create the smell vocabulary. They marked key concepts in code smell definitions extracted from the literature to develop the smell vocabulary.

By applying this step, we derived heuristics and code characteristics related to the selected code smell. We use this input to derive our Annotation Model.

3.4 Define severity scale

Many studies determine smell severity automatically by measuring the chosen code metrics or using other heuristics such as historical information of component modification. However, the large variety of such approaches suggests that researchers disagree on the factors that should be used to filter and prioritize code smells (Sae-Lim et al., 2018).

Sae-Lim et al. (2018) suggest this issue should be solved by conducting empirical studies to uncover factors used by practitioners. Pecorelli et al. (2020) also emphasize how available solutions suffer from not being empirically assessed or not providing developers with recommendations aligning with their perception of design issues, making them still ineffective in practice. Therefore, we include the subjective annotation of the smells' severity in our annotation model.

We examined the studies that asked the developers' for their subjective evaluation of the code smell severity to extract more concrete annotation guidelines. Unfortunately, not many empirical studies consider this issue. Pecorelli et al. (2020) and Taibi et al. (2017) asked the developers to rank the criticality of the code smell on a scale of 1 to 5. Madeyski and Lewowski (2020) asked the annotators to manually assess the code smells on a four-level severity scale (*none* to *critical*). However, neither of these studies provided guidelines on what factors should be considered for severity assessment. A more descriptive four-level severity scale was designed in the studies (Fontana et al., 2016a; Fontana and Zanoni, 2017). Thus, we opted to use this scale in our model.

4 Annotation Model

Figure 3 describes our code smell annotation conceptual model, where we denote the entities and their relationships.

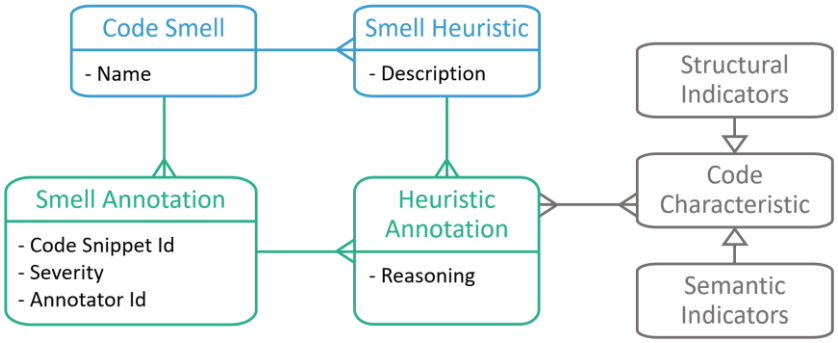

Figure 3 Code smell annotation conceptual model

4.1.1 Code Smell and Smell Heuristics

Our *Code Smell* entity is a high-level concept that denotes a category of issues that harm the code's maintainability by making the code difficult to understand or change to fulfill new requirements. We derive these entities from catalogs of code smells, such as (Fowler, 2018).

Smell Heuristics decompose code smells into less vague properties of the code and are closely related to the software engineer's cognitive load, thinking process, or experienced issues when working with the code (Hozano et al., 2018). For example, we define the "method is too complex" heuristic for the Long Method code smell. This heuristic applies to a code snippet when the engineer spends much time processing a line of code or region of a function to determine its intended behavior. As another example not related to our selected smells, the Shotgun Surgery code smell (Fowler, 2018) could define a heuristic as "supporting a change to an existing functionality requires opening too many source files." We determine heuristics for a specific code smell by examining industry recommendations (e.g., from books (Fowler, 2018; Martin, 2009), blog posts, tutorials) and empirical research related to the engineer's perception of code smells (Hozano et al., 2018; Mäntylä et al. 2004).

We ask the annotators to assess whether heuristics apply to the analyzed code snippet to help them assess whether it suffers from a particular code smell. Notably, both the code smell and its related heuristics are inherently subjective, which means that one engineer might claim that a method is too complex, while another might not[5]. This subjectivity is why it is challenging to label heuristics automatically, and we ask the annotators to do it manually.

4.1.2 Code characteristics

Code characteristics are low-level concepts that explain why our heuristic applies to the given code snippet. They relate to exact parts of the code snippet that induced the annotator to mark the particular heuristic as applicable to this code snippet[6]. Annotating the applicable heuristics and their manifestations at the source code level provides in-depth annotator reasoning for assigning the code smell label to a particular code snippet.

---

[5] Refer to Section 1 of our supplementary material for an example illustration of this point.

[6] There is a terminological difference between the term heuristics as we use it and as Martin (2009) and a significant portion of industry best practice authors define them. Martin's heuristics focus on code characteristics instead of the engineer's perception. They are much more concrete, as most can easily map to a specific code structure. For example, Martin defines a heuristic around "magic numbers" where the goal is to identify any token with a value that is not self-describing (e.g., a literal number with a strange value) and replace it with a descriptive variable or constant. We refer to Martin's heuristics as *Code Characteristics* in our model.

For example, an annotator might find the heuristic "method does multiple things" applicable when they find several code regions in a function delimited by newline characters and comments explaining what the next region of code does. The comments and newline characters are the *code characteristics* in this case. Likewise, for the "method is too complex" heuristic, an annotator might determine that the reason behind this complexity are several sophisticated conditional expressions that are hard to process mentally. In general, these low-level concepts can be traced to specific lines of code, structural metrics, or concern metrics (Padilha et al., 2014) of the code snippet.

We differentiate two categories of code characteristics, including *Structural Indicators* tied to structural metrics and *Semantic Indicators* related to concern metrics (Padilha et al., 2014). Unlike metrics that give a concrete number (e.g., how many lines of code a function has or how many responsibilities a class has), our indicators are assessments of the contribution of the given code characteristic to the applicability of the related heuristic. As an example, an annotator might explain that a "method is too complex" because of high cyclomatic complexity combined with several long conditional expressions that use magic numbers and fields with mysterious names. When taken to the extreme, these code characteristics might be sufficient to set the Long Method's smell severity to 2 or 3, even though the function might have less than thirty lines of code.

Like heuristics, code characteristics categories stem from our analysis of developer reasonings and smell specifications from the literature. Moha et al. (2010) derived a similar conclusion – they classified key concepts extracted from specifications of four design smells and their underlying 15 code smells into three categories: measurable, lexical, and structural properties. Measurable properties are concepts related to code metrics. In our notation, code metrics fall into *structural indicators*. Moha et al. (2010) define structural properties as relationships between the systems' constituents[7]. In our notation, these properties would also fall into *structural indicators.* Lexical properties relate to the vocabulary used to name systems' constituents. In our notation, lexical properties fall into *semantic indicators* as they suggest semantics, for example, semantic relationships between classes and responsibilities (Palomba et al., 2017). Indeed, Palomba et al. (2017) showed that textual (semantics) and structural techniques of detecting code smells are complementary, which justifies our separation of these two categories.

Code characteristics we singled out can be further justified by looking at various code smell detection approaches. Most code smell detection in the literature rely on using structural and/or semantic indicators (Azeem et al., 2019; Palomba et al., 2017).

We note that, in our annotation process, code characteristics are very loosely related to our heuristic annotations through the "Reasoning" free-form description. Though annotating code characteristics would provide a complete picture of annotators' reasoning, we did not explicitly annotate these indicators, as we found it significantly reduces annotation speed. Instead, we listed notable instances of these indicators during the annotation procedure to refine it and align the annotators' understanding.

4.1.3 Smell Annotation

Once we select the code smells and determine heuristics that signal their presence, we can instantiate the label model for a set of code snippets. We use the term code snippet to define any code that can be affected by a code smell (i.e., a function for Long Method, and a class for Large Class). An annotator

[7] The examples of systems' constituents are classes, methods, fields, and relationships.

instantiates a *Smell Annotation* entity for each code snippet that is examined for a specific smell. In our annotation model, the annotator determines the presence and severity of a particular code smell.

We extend the severity scale from Fontana et al. (2016a) and Fontana and Zanoni (2017) to tie it with the maintainability issue the annotator perceives:

0. means there is no smell or that it is very mildly present and negligible. The code snippet does not require refactoring regarding this code smell. This does not mean that the code is perfect, and a code snippet can have room for minor enhancement and still have a severity of 0.
1. means there is a minor presence of the smell that slightly reduces the snippet's maintainability. Usually, one or two refactoring operations can resolve the issue. In terms of prioritizing work, we note that such code is "good enough". However, engineers should refactor such instances as part of normal development to reduce the comprehensibility strain that arises from multiple severity 1 instances.
2. means there is a significant issue that hampers maintainability. It should be resolved by applying a series of refactoring operations. We consider such refactoring a high-priority activity when the code snippet is part of a module under active development, as it negatively impacts daily tasks.
3. means there is a critical issue that severely harms the maintainability of the code snippet. Resolving this issue requires dedicated work to redesign the code snippet and entails many refactoring operations. We consider refactoring mandatory for such code snippets, provided they are part of a module under active development.

Before determining the severity of a code smell, the annotator labels any applicable heuristics regarding the smell. For each *Heuristic Annotation*, they provide reasoning why the heuristic is applicable. This reasoning provides insight into the thinking process of the annotator and helps guide the annotation procedure. For example, an annotator might apply the "method is too complex" heuristic for a code snippet. They then define the reasoning that justifies this application. For example, the annotator can state that "the method has many long expressions and message chains" or "the method has several complex conditional expressions that include literal values with unclear meaning".

Notably, the final severity is not the sum of the applicable heuristics. For example, we annotated code snippets with two applicable heuristics and severity of 3, while another had three applicable heuristics and severity of 2. This can occur because a heuristic might present a minor violation (e.g., "method does multiple things" applies because it does three things), a major violation (e.g., "method does multiple things" applies because it does 30 things), and everything in between.

We do not model the severity of each heuristic, as handling disagreements on the heuristic-level severity significantly reduces the annotation speed. Instead, each annotator assesses the severity level of all applicable heuristics when assigning the code smell severity label. Multiple annotators assign each code snippet a severity label, and disagreements are resolved as discussed in Section 5. In other words, we do not resolve disagreements on heuristic-level severity. Instead, we control the subjectivity on the code smell level severity. We recognize that labeling the heuristic-level severity would provide a more comprehensive insight into annotators' reasoning, but this is not the goal of our study – we aim to create quality code smell-level annotations. We use the reasoning field to record thoughts that would improve our model and heuristics in retrospective discussions (Section 5).

4.2 Long Method

Here we denote the literature that influenced our heuristic selection for the Long Method code smell. For each heuristic, we discuss the literature findings that support the use of the heuristic for smell identification and examine the related code characteristics. We supplement this set of code characteristics with our experience from the proof-of-concept annotation.

For the Long Method code smell, we defined the following set of heuristics:

- Method is too long, where the annotators focus on the length of the method and the logic it performs.
- Method is too complex, where the annotators focus on code characteristics that make the method harder to comprehend.
- Method does multiple things, where the annotators focus on the semantic cohesiveness of the logic inside a method.

Our process resulted in the same three heuristics for the Long Method detection as Hozano et al. (2018). Through coding, we identified two additional heuristics: "method has expressions at different levels of abstraction" and "method has side-effects" heuristics recommended by Martin (2009). However, after performing the proof-of-concept annotation, we decide to exclude these two heuristics – we elaborate why in section 6.4.

4.2.1 Method is too long

Literature. A method's length is a simple characteristic that can quickly signal if a method requires refactoring. Experienced industry leaders (Fowler, 2018; Martin, 2009) advocate for short, focused functions. Likewise, engineers of various seniority use the length of the method to determine the Long Method smell (Hozano et al., 2018). A recent literature review (Bafandeh Mayvan et al., 2020) examined rule engines and other code smell detectors. For Long Method, they found that the most relaxed rule (with the highest number of true and false positives) checked if the function had more than 50 lines of code.

Our Findings. The annotators agreed that examining a method for which the length heuristic applied produced a sense of exploration, where the method needed to be researched to understand its purpose. Such methods required focus to track which intermediate results were relevant for later code. They also required scrolling through the functions, which were often more than two screens in height. The number of lines of code was an important structural indicator for our annotators, and a high number usually indicated the applicability of this heuristic. Exceptions to this rule included repeated logic that was not duplicate code (e.g., validation checks for many fields, the logic that transforms one data structure into another). We noted two other code characteristics that made this heuristic applicable:

- Repeated expressions related to the Duplicate Code (Fowler, 2018) smell. This usually occurred with branching control flow, where expressions were duplicated in multiple branches instead of placed before or after the branching. The occurrence of these two smells aligns with the findings of code smell cooccurrence, identified in (de Paulo Sobrinho et al., 2018).
- Redundant or unnecessary expressions. This usually accompanied redundant validation or null checks or calculations whose results were never used (e.g., variables assigned a value but never read).

4.2.2 Method is too complex

Literature. Fowler and Kent note that conditionals and loops can indicate the Long Method code smell (Fowler, 2018). Further research found that engineers examine the complexity of a method's control structure (e.g., the number of branches and loops) to determine if it suffers from the Long Method smell (Hozano et al., 2018). The cyclomatic complexity structural metric counts branches, loops, and several other code constructs to determine a method's complexity. This metric is used in several smell detectors (Bafandeh Mayvan et al., 2020) to determine the Long Method smell, where a complexity above 5 signals the presence of this smell.

Our Findings. Our annotators employed much cognitive processing to understand complex methods and the intermediate results of sophisticated expressions for which the intent was unclear. Structural indicators based on high cognitive complexity (Campbell, 2018) or the maximum number of branches helped us determine if this heuristic applies to a given function. Exceptions to this rule included functions with many short branches with simple conditional expressions (e.g., multiple null checks or simple validation rules). We defined two other code characteristics that helped us make conclusions regarding a method's complexity:

- Use of "magic numbers" (Martin, 2009). Functions that used literal values (especially numbers that were not equal to 0 or 1) increased the cognitive load required to understand the code.
- Long and complex single lines of code. This occurred when many expressions were nested or chained (e.g., arithmetic operations conducted inline of several function arguments, a Message Chain smell (Fowler, 2018) in conditional expressions, or nested ternary operators).

4.2.3 Method does multiple things

Literature. Industry leaders have championed various forms of the single responsibility principle (Martin et al., 2003), which is often associated with classes but is also applicable to functions. Clean code leaders state that a function should do one thing or focus on a single task (Martin, 2009). In the empirical research of Hozano et al. (2018), software engineers noted that they examine if a method does multiple things to determine the Long Method smell's presence. Comments and newline characters that divide the function into segments are recognized by industry leaders (Fowler, 2018) and researchers (Palomba et al., 2017) as semantic indicators that support this heuristic.

Our Findings. Determining the "things" that a method does was highly subjective due to the ambiguity of the term. Furthermore, a distinct piece of logic, such as exception handling or input validation, might be considered a separate thing in one code snippet and part of another thing in a different snippet. The main criteria for determining if a piece of logic is a thing was its semantic difference from the surrounding code and standalone complexity (where trivial code was ignored). The presence of newline characters and comments was a strong indicator of the applicability of this heuristic during our annotation. We found three other factors that contributed to the function doing multiple things:

- Feature Envy (Fowler, 2018), where a function would be charged with changing state or performing some logic that should be encapsulated in another class. The occurrence of these two smells aligns with the findings of code smell cooccurrence identified in (de Paulo Sobrinho et al., 2018).
- Duplicate Code (Fowler, 2018), where a function had repeated regions of code with slight variation.

- Different levels of abstraction (Martin, 2009), where out of place low-level expressions were nested among higher-level method calls, signaling either misplaced logic or a missing higher-level function.

Finally, some methods did not display obvious signs of doing multiple things. We had to examine and understand the function's semantic intent to separate any hidden concerns it had. Such cases were hard to discover and were brought to light by a single annotator, usually the one with the most experience in the subject matter.

4.3 Large Class

Like the previous section, we define the sources that influenced our heuristic selection for the Large Class code smell. We discuss the literature findings that support the heuristics and their related code characteristics. We supplement this set of code characteristics with our experience from the proof-of-concept annotation.

We defined the following set of heuristics:

- Class is too long, where the annotators focus on the length of the class, the number of fields, properties, and methods.
- Class is too complex, where the annotators focus on code characteristics that make the class harder to comprehend.
- Class has multiple concerns, where the annotators focus on the semantic cohesiveness of the logic inside a class.

Hozano et al. (2018) derived nine different heuristics for God Class detection. We mapped these heuristics to three[8]. Before the proof-of-concept annotation, our initial set of heuristics was larger[9]. We elaborate on why we removed some heuristics in section 6.4.

4.3.1 Class is too long

Literature. Fowler (2018) states that a Large Class has too many fields and methods, where most methods use a subset of the field set. Likewise, software engineers use the length of the class to determine the presence of the Large Class smell (Hozano et al., 2018; Padilha et al., 2014). A recent literature review (Bafandeh Mayvan et al., 2020) examined rule engines and other code smell detectors. For Large Class, they found that the most relaxed rule (with the highest number of true and false

[8] Notably, Hozano et al. (2018) map each answer to a single heuristic. In contrast, we allow several heuristics to be present in one answer. For example, Hozano et al. (2018) treat the answers (1) "By verifying the Single responsibility principle," (2) "When a class implements several features or the code is complex," and (3) "I considered the number of features and the aggregation of several independent features" as three different heuristics. In contrast, we would map answer (1) to the responsibility heuristic, answer (2) to two heuristics: responsibility and complexity, and answer (3) to two heuristics: size (number of fields) and responsibility.

[9] During the coding procedure, we considered including the coupling and cohesion heuristics. However, we decided to include these dimensions into a broader responsibility dimension. Coupling and cohesion indicate whether the responsibilities are optimally distributed over the project's classes. For example, in the catalog synthesized by Fontana et al. (2016b), the text-based definition of the Large Class smell uses the terminology related to the classes' responsibilities, while the correlated detection rule uses the measures of cohesion/coupling. Moreover, we note that each study that mentioned cohesion/coupling dimensions also emphasized the responsibility dimension, while the reverse was not the case.

positives) checked if the class had more than 100 lines of code. They also considered classes with more than 14 methods or 8 fields as Large Classes.

Our Findings. Our annotators experienced a sense of exploration while examining classes that were too long. Such classes required significant scrolling through the code to understand their meaning and the services they offered. A high number of code lines, fields, and methods was a strong structural indicator of the applicability of this heuristic. Another indicator was the presence and length of any inner classes that contributed to the length of the outer class. Notably, some classes appeared to be long (e.g., LOC > 300) but used a coding style that liberally applied the newline character, separated object construction and method invocation across multiple lines, or favored local variables and short chains of method invocations. These examples present the pitfall of using an automated heuristic based on a simple metric such as physical LOC, while the last example highlights that logical LOC can be misleading.

4.3.2 Class is too complex

Literature. Software engineers perceive that a class is too complicated when they have difficulties in creating a mental model of how the class works (Palomba et al., 2014a). They state that a Large Class is too complex or that some of its methods are too complex (Santos & de Mendonça, 2015). Bafandeh Mayvan et al. (2020) found that rule engines detect the Large Class smell by considering the total cyclomatic complexity of the class' methods.

Our Findings. A significant portion of the labeled classes had their complexity stem from one or more complex methods. Notably, we avoided labeling a class as too complex when it only contained a single complex method and trivial code (e.g., a few fields or simple methods). However we labeled the following classes as too complex: classes with multiple complex methods, a single complex method and many fields, or classes with a single complex method and a sophisticated inner class. We found three other factors that contributed to a class' complexity:

- Inner classes, in general, contributed to the complexity of a class, especially when there were multiple non-trivial inner classes.
- Mysterious names (Fowler, 2018) played a significant role in obscuring the class's intent. On the other hand, classes with sophisticated logic that followed good naming significantly reduced the cognitive burden required to understand how they work.
- Classes that were coupled to static fields and methods (i.e., global state) were difficult to understand, as the logic was distributed. It was often unclear what the responsibility of the examined class was.

4.3.3 Class has multiple concerns

Literature. The Single Responsibility Principle states that a class should group all the things that change for a single reason or single category of requirement changes (Martin et al., 2003). Software engineers consider this principle when identifying Large Classes (Hozano et al., 2018). They try to summarize the class's responsibility in a sentence while avoiding conjunctions that uncover multiple concerns (Hozano et al., 2018; Martin, 2009; Schumacher et al., 2010). Concern metrics, such as concern diffusion over lines of code, are considered good indicators of the Large Class smell (Padilha et al., 2014). Notably, "being concerned/responsible" for a piece of logic means knowing the details of that logic (Martin et al., 2003). This means that coordinator classes that encapsulate multiple objects do not necessarily have multiple concerns provided they only know the details of the coordination logic. Classes that only

delegate minor tasks to other classes indicate a Large Class (Santos & de Mendonça, 2015; Schumacher et al., 2010).

Our Findings. Like determining what a method does, defining the concerns of a class was a highly subjective activity. Our annotators looked for semantic differences to identify subsets of fields and methods that could meaningfully be extracted into a separate class, ignoring trivial subsets such as those containing a single field or simple method. Class-level comments and whitespace between the members proved to be useful semantic indicators of the different responsibilities of the class. Likewise, shared prefixes in the names of fields and methods helped determine hidden concerns. Notably, we had to examine long and complex methods to determine if they contained a hidden class within their logic, which was challenging and time-consuming.

4.4 Generalizability of the proposed annotation model

We can classify code smells into (1) low-level smells related to particular problems in the code and (2) high-level smells regarded as equivalent to the architectural/design smells and are manifested by the composition of low-level smells (Perez, 2011; Moha et al., 2010). Our annotation model is applicable to the low-level smells that can be characterized through structural and semantic indicators.

This encompasses many types of code smells. For example, researchers showed that the following code smells can be detected using structural and semantic indicators: Feature Envy (Liu et al., 2018; Hadj-Kacem at al. 2019; Azeem et al., 2019), Data Class (Fontana et al., 2016a; Azeem et al., 2019), and Azeem et al. (2019) list many other smells that have been detected using these indicators.

We note that some code smells such as Divergent Change, Parallel hierarchies, and Shotgun Surgery can be reliably detected by analyzing the change history of a project (Palomba et al., 2014b). This indicates that the code heuristics in our annotation model might be extended by historical indicators. We did not include historical indicators in our code characteristics in this iteration, as we derived our model by examining two smells, Large Class and Long Method. Though historic indicators can be used to detect Large Classes (Barbez et al., 2019), we did not find that any code smell specification, heuristics or annotator reasoning provided in the analyzed literature analyze historical factors when deciding on whether the class suffers from this smell. Furthermore, Moha et al. (2010) showed that their domain analysis that did not include historic indicators is complete enough to describe a whole range of smells.

Nevertheless, future empirical studies on developers' perception of code smells might reveal that historical indicators play an important role in annotators' assessments for certain code smells. In such a case, our annotation model can be extended to include historic indicators. As Moha el al. (2010), we suggest our annotation framework can be extended, if required, by iterating the process we proposed to derive the initial annotation model. The largest current obstacle to refining our current framework is the limited number of empirical studies that examine developers' perception of code smells. As we discussed in Section 3.1, one of the primary reasons for choosing the Long Method and Large Class code smells is the availability of such studies.

For annotating the Long Method and Large Class code smells we only needed to considered code components in isolation without analyzing other parts of the project. Santos et al. (2017) also concluded that the tools supporting design comprehension does not affect the human perception of the Large Class. However, to annotate smells such as Feature Envy and Refused (parent) Bequest, the annotator must also consider related code components. Our future work includes building a tool that will support

the annotators in this process by eliminating trivial cases and providing easy access to related code components.

5 Data Annotation Procedure

Starting from the annotation model described in the previous section, we designed an annotation procedure and used it to create a dataset of Long Method and Large Class code smells. Figure 4 presents the main activities of the annotation procedure used to create our dataset.

The initial annotation model and procedure were constructed by three authors (NL, JS, AK). One of them (NL) then followed the annotation procedure with two other authors (SP, KGG) to create the final dataset. NL has six years of experience in the software engineering industry and is a professor on several programming and software engineering courses. SP and KGG are Ph.D. students researching code quality and code smells with several small-scale industry projects behind them, amounting to a year and a half of industry experience each. We conducted two two-hour workshops to train the annotators through theory and exercises and reach a common understanding of the selected smells and heuristics.

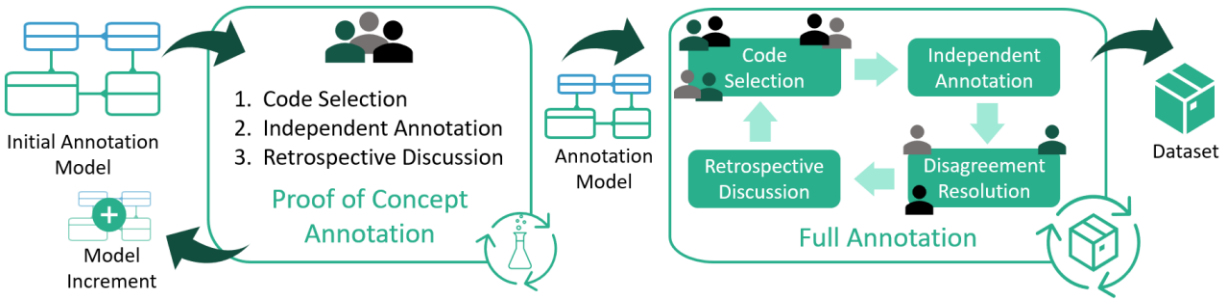

Figure 4 Annotation procedure

As part of the proof-of-concept annotation, we annotated a set of code snippets to test the chosen heuristics' validity, streamline the annotation procedure, and further develop the understanding of code smells among the annotators. This activity resulted in most of the changes to the annotation model. We describe the proof-of-concept annotation in Section 5.1. Then we performed the full annotation of code snippets to create the complete dataset. We describe the details of this activity in Section 5.2.

5.1 Proof of Concept Annotation

We used the proof-of-concept annotation to test the annotation model and procedure and gather insight for their improvement. We conducted the proof-of-concept annotation in three rounds over four software projects listed in Table 1.

At the start of each round, we *selected code* that we would annotate. We chose a simple student project developed by four third-year undergraduate students as part of their software engineering semester project for the first round. For the second round, we selected a random subset of code snippets from an open-source project that was much larger than the student projects. Finally, we selected a random subset of code snippets from two open-source projects for the third round. In the second and third rounds, we chose a random 10% of classes and functions, excluding any functions that had less than 5 lines of code to avoid trivial code snippets, as recommended in (Madeyski & Lewowski, 2020). We also excluded test-related classes and functions (i.e., integration and unit tests) to focus on functional code smells and not test smells.

Table 1 Summary of selected projects for the proof-of-concept annotation

Name	Software type	Selected classes	Selected methods and constructors
Student Project	Administrative application	81	263
BurningKnight	Video game	130	408
ShopifySharp	Integration library	25	20
Core2D	2D diagram editor	30	120

Three annotators *independently annotated* the presence of a code smell, its severity, and applicable heuristics for each code snippet. When annotating a single instance of our dataset, we adhered to the following algorithm:

1. For each heuristic, determine if it is applicable. For example, declare if a "method is too long" by answering the question, "In your opinion, does the method's length harm its readability?".
2. For each applicable heuristic, provide brief reasoning behind the decision. We used the reasoning to understand which code characteristics are related to each heuristic. We describe the findings we made through this reasoning in the previous section.
3. Considering the applicable heuristics and the code snippet's overall structure, determine the presence and severity of a particular code smell.

During each round, the annotators met several times to discuss their progress and observations regarding the annotation procedure and align their understanding. Additionally, each round ended with a *retrospective discussion*. Here we summarized these findings to enhance the annotation model and streamline the procedure for the next round. For example, the student project's simplistic nature guided us to select larger open-source projects and exclude the student project from the dataset. Then, to increase the variety of examined coding styles, we opted to select a smaller percentile of random code snippets from a single project to have time to cover more projects.

By the end of the third round, each annotator labeled 155 classes and 709 methods and constructors. We refined the annotation procedure, conceptualized the final annotation model, and achieved a common understanding of the code smells and heuristics between the annotators. We made a final review of the labels and contributed the annotations related to the open-source projects to the final dataset.

5.2 Full Annotation

We conducted a full annotation to build a medium-sized dataset. Here we followed a similar approach for the full annotation as with the third round of the proof-of-concept annotation with several differences.

We expanded the *code selection* strategy to exclude classes with less than two methods or four fields. We determined that such classes are too trivial to suffer from the Large Class code smell. Furthermore, we searched GitHub for sufficiently sophisticated projects developed in C#. Using the advanced search, we looked for moderately popular projects (i.e., over 5000 stars) that had undergone sufficient development (i.e., over 1000 commits) and had development activity within the past six months. This search criterion helped us avoid simplistic and under-developed projects that might not represent a typical active open-source project. Table 2 lists the selected projects for the full annotation. Finally, we

selected different types of projects (e.g., video games, graphic frameworks, AI frameworks, security libraries, media systems…) to cover a wide variety of coding styles and flavors of code smells.

Table 2 Summary of selected projects for the full annotation

Name	Software type	Selected classes	Selected methods and constructors
ShareX	Screen capture and media sharing library	81	194
OpenRA	Strategy game engine	219	441
Jellyfin	Software media system	129	519
MonoGame	Video game development framework	70	280
osu!	Video game	236	593

We divided the snippets into subsets, where two of the three annotators *independently annotated* each subset. The third annotator examined code snippets where the two annotators were not in agreement regarding the presence of the code smell or its severity. Without looking at the individual annotations, the third annotator would submit a third opinion for the code snippet. This *disagreement resolution* is like the cross-check performed in (Madeyski & Lewowski, 2020). If a code snippet was annotated with three different severity scores, we extensively discussed the snippet and the reasoning behind our severity score. Often the initial disagreement would be the result of an oversight by an annotator, and they would modify their score after more in-depth analysis.

Annotating each project ended with the *retrospective discussion*. The annotators discussed new code characteristics that helped them determine the presence and severity of a smell, expanding the previous section's findings.

5.3 Comparing our Annotation Procedure with Related Work

In our procedure, we selected code samples randomly from the projects according to the Independent and Identically Distributed (I.I.D) assumption typically posed by ML models. Madeyski and Lewowski (2020) used the same sample selection process.

Ideally, to ensure the naturally occurring ratio of smells to non-smells in practice, we would annotate whole projects as Palomba et al. (2018a). However, Palomba et al. (2018a) employed a semi-automatic procedure to deal with many code samples. Annotating code samples is time-consuming and tedious, especially since we cross-checked each instance and conducted retrospective discussions after each round. Thus, we opted to annotate code samples from multiple projects to make our dataset diverse instead of fully annotating only a few projects.

Hozano et al. (2018) employed a different code sample selection procedure. They selected samples reported to suffer from code smells in previous studies and introduced additional metric-based constraints to ensure sample variability[10]. However, the goal of our study is different from theirs. They investigated how developers detected code smells while we strived to develop a code smell dataset for

[10] The authors considered the metrics typically used to detect a particular smell type. For example, the authors selected code samples ranging from 43 to 194 lines of code for the Long Method code smell. However, we do not impose such constraints as methods with few lines of code might be considered Long Methods if those lines are extraordinarily complex.

training a high-quality ML model for code smell detection. Thus, we need both negative and positive samples in an approximately naturally occurring ratio to train the quality ML model.

The limitation of our study is that we employ just three annotators. This limitation was necessary as we required considerable involvement from each participant. Many previous studies (Fontana et al. 2016; Palomba et al. 2015; Palomba et al., 2018a; Rasool and Arshad 2017) aimed at constructing code smell datasets employed 1-3 annotators and, in contrast to our study, did not include precise guidelines and a proof-of-concept annotation. Furthermore, many of these studies also perform a semi-automatic labeling procedure (Section 2).

Empirical studies on code smell perception typically include more annotators but purposefully omit guidelines and annotator training to derive their unbiased opinion. Hozano et al. (2018) consider a broader set of smell types and include more developers from different backgrounds in the process. On the other hand, they annotated only 15 samples per smell while we developed a medium-sized corpus (Section 6.1). Hozano et al. (2018) report a significant disagreement on code smell perception.

Madeyski and Lewowski (2020) included 26 participants in their empirical study that produced a medium-sized corpus but did not impose annotator training and cross-checking of each instance. In our earlier work (Kovačević et al., 2021), we trained multiple ML-based and heuristic-based models on the MLCQ dataset. We found that the tested code smell detectors had low *F*-measure[11], especially for God Class detection. We performed error analysis to derive the underlying reasons and found that inconsistency of the annotations and the fact that many instances were not cross-checked (and therefore, these annotations are susceptible to subjectivity) may be the underlying reasons.

Santos et al. (2017) showed that training the annotators improves Large Class detection. They recommended training the annotators using "golden" examples and conducting discussions between them to align their conceptualization of the analyzed code smells. Oliveira et al. (2020) showed that collaborative smell identification significantly increases code smell detection precision and recall. We employed these recommendations in our approach. We trained the annotators, cross-checked each instance, and performed retrospective discussions to harmonize the annotator understanding of the code smell annotation task.

6 Results and Discussion

In this section, we discuss the outcomes of our work and present the related findings and limitations. Section 6.1 details the characteristics of our dataset, including its datasheet and basic statistical information. In Section 6.2, we compare our results with the most similar dataset in the literature and perform a series of statistical tests on both datasets to discuss the advantages of our approach. In Section 6.3, we perform a series of statistical tests to determine the consistency of our annotations across multiple dimensions. Section 6.4 describes the annotators' observations and experience of annotating the dataset of code smells. Here we explore our recommendations regarding building the annotation model and conducting the procedure. We group these takeaways into an annotation

[11] The highest *F*-measure we achieved for the God Class detection was 0.53, and the highest *F*-measure we achieved for the Long Method detection was 0.75. Typically, models achieving *F*-measures of over 0.9 are considered reliable for production.

guideline to support researchers in building their datasets. Finally, in Section 6.5, we discuss the limitations of our study and threats to validity.

6.1 Dataset Characteristics

The datasheet of our dataset is inspired by (Madeyski & Lewowski, 2020), where each annotated instance contains the following:

- *Code Snippet ID* – the full name of the code snippet. For classes, this is the package/namespace name followed by the class name. The full name of inner classes also contains the names of any outer classes (e.g., *namespace.subnamespace.outerclass.innerclass*). For functions, this is the full name of the class and the function's signature (e.g., *namespace.class.method(param1Type, param2Type)*).
- *Link* – The GitHub link to the code snippet, including the commit and the start and end LOC.
- *Code Smell* – code smell for which the code snippet is examined.
- *Project Link* – the link to the version of the code repository that was annotated.
- *Individual annotations* – the severity score and applicable heuristics determined by each annotator.
- *Final severity* – a single severity score calculated using the algorithm described below.
- *Metrics* – a list of metrics for the code snippet, calculated by our platform (Prokić et al., 2021).

Table 3 lists the basic characteristics of our dataset. Notably, our dataset consists of 3495 annotated instances across two code smells, with a minimum of two annotations made by different annotators for each instance. Compared to a larger corpus, such as (Madeyski & Lewowski, 2020), our dataset comprises fewer projects and annotators but maintains a similar number of annotations, resulting in a medium-sized corpus.

Table 3 Basic characteristics of our dataset

Characteristics	Value
# of projects	8
# of annotated smells	2
# of annotators	3
Total # of annotated instances	3494
Total # of annotations	8202
Average # of annotations per instance	2.35 (minimum 2)
Average # of annotations per annotator	2734 (minimum 2599)

Table 4 counts individual severity labels for each smell. We note that more than 25% of the examined code displays at least minor maintainability issues. However, if we consider that severity 1 code is "good enough," we conclude that roughly 10% of the examined code requires immediate refactoring.

These results align with previous empirical research on the diffuseness of code smells (Palomba et al., 2018a). However, two things should be considered when interpreting these results in terms of software maintainability. First, most of our instances (80%) stem from popular and active open-source projects. This subset of all software (including less popular open-source projects and commercial projects) might not represent the broader distribution of code smells. Second, while a single severity 1 smell might be

"good enough," many code snippets suffering from such smells add up and harm the software's maintainability.

Table 4 Number of final severity labels for each smell

Code Smell	Total # of instances	# of final severity labels			
		0	1	2	3
Large Class	920	677 (73.6%)	145 (15.8%)	69 (7.5%)	29 (3.2%)
Long Method	2574	1924 (74.7%)	417 (16.2%)	182 (7.1%)	51 (2%)

6.2 Comparison with other datasets

We compare our dataset solely with the MLCQ dataset (Madeyski & Lewowski, 2020) for the following reasons:

- It was fully manually annotated, and its annotation procedure most closely resembles ours.
- It was a large-scale initiative that included 26 practitioners, examined code snippets from 523 projects, and produced 4019 Large Class labels and 3362 Long Method labels.

In Table 5, we examine the severity label distribution for the code smells annotated in both datasets. Notably, our dataset has twice as many Long Method annotations, while the MLCQ dataset has twice as many Large Class annotations. Interestingly, the severity label distribution is similar, especially for labels 2 and 3. We also note a higher presence of severity 1 labels in our dataset, arriving at 5.2% for Large Class and 7% for Long Method.

Table 5 Number of individual severity labels for each smell in our dataset compared to the MLCQ dataset

Code Smell	Clean CaDET Dataset					MLCQ Dataset (Madeyski & Lewowski, 2020)				
	Total # of annotations	# of individual severity labels				Total # of annotations	# of individual severity labels			
		0	1	2	3		0	1	2	3
Large Class	2130	1497 (70.3%)	393 (18.5%)	174 (8.2%)	66 (3.1%)	4019	3045 (75.8%)	535 (13.3%)	312 (7.8%)	127 (3.2%)
Long Method	6072	4200 (69.2%)	1243 (20.5%)	498 (8.2%)	131 (2.2%)	3362	2556 (76%)	454 (13.5%)	274 (8.1%)	78 (2.3%)

We examine the top five contributors of the MLCQ dataset, including reviewers with IDs 1 (7.5 years as a professional software developer), 3 (6.25 years), 4 (19 years), 5 (2 years), 20 (15 years). These contributors created 74% of the Large Class and 74% of the Long Method labels. As expected, the label severity distribution of these contributors varies only slightly from the distribution presented in Table 5.

Interestingly, the severity label distribution does not correlate with years of professional experience. Reviewer 3 stands out from the rest of the contributors as having a stricter labeling strategy. For Large Class, reviewer 3 contributed 25% of the total labels, including 39% of severity 1 labels, 49% of severity 2 labels, and 74% of severity 3 labels. On average, each of the other four reviewers contributed 12% of the total labels, including 9% of severity 1 labels, 6% of severity 2 labels, and 2% of severity 3 labels. We find a similar but less extreme trend for the Long Method annotations.

In the MLCQ developer survey, reviewer 3 stands out from the rest of the top contributors as someone who: (1) contributes to open-source projects, (2) uses code smell names in team discussions and code reviews, and (3) considers both Large Class and Long Method a significant (5 out of 5) code smell. These findings indicate a higher interest and possibly expertise in code quality, explaining the different labeling strategies.

6.3 Statistical analysis of the annotation process

As Fowler (2018) states, "no set of metrics rivals informed human intuition" meaning that human judgement is the ultimate authority when assessing code smell severity. Human judgement is, however, subjective. We strived to reduce the subjectivity of the code smell annotation process through annotation guidelines, annotator training and retrospective discussions. To evaluate our success, we evaluate the consistency of code smell annotations in our dataset.

To evaluate the consistency of our annotation, like (Mäntylä and Lassenius, 2006) we use code metrics that are non-subjective code characteristics. We used the ANOVA (Analysis of Variance)[12] and MANOVA (Multivariate Analysis of Variance)[13] tests from the Statsmodels library (Seabold & Perktold, 2010) to calculate:

- The annotation consistency of the single annotator - The goal of the statistical tests we conducted is to determine whether annotations of a single annotator are consistent regarding code metrics.
- The annotation consistency between annotators - In this case, we want to determine whether different annotators are mutually consistent in the annotation process, i.e., whether they follow the same guidelines as measured by code metrics when annotating.
- The significance of code metrics in the annotation process – The final step in our statistical analysis is to identify individual metrics that indicate differences in the understanding of the code smell.

From the results of statistical tests, we can identify potential rules or patterns that exist and thus improve the annotation process. The results of metric-based statistical tests that indicate annotations' inconsistency can help annotators see where they went wrong when annotating. For example, suppose the annotators assigned the same severity to instances that differ according to a particular metric. In that case, the heuristics may not cover the metric-expressed aspect of the code smell. However, note that not all code metrics are relevant for all smells. Thus, a domain expert should examine this result - if a metric indicating annotation inconsistency is irrelevant for a code smell, it should be ignored.

We used 25 metrics for Large Class code smell and 18 metrics for Long Method code smell[14]. Regarding the interpretation of test results, it is important to emphasize that we compared the p-value with the significance level α of 0.05 (this value indicates a 5% risk assuming a significant relationship between the dependent and independent variable where the relationship does not exist).

---

[12] Statsmodels library https://www.statsmodels.org/stable/anova.html. Accessed August 9, 2021

[13] Statsmodels library https://www.statsmodels.org/stable/generated/statsmodels.multivariate.manova.MANOVA.html. Accessed August 9, 2021

[14] https://github.com/Clean-CaDET/platform/blob/c4acff95ec00ff6c25fa62dde4818c1f40e39d39/CodeModel/CaDETModel/CodeItems/CaDETMetrics.cs

6.3.1 The annotation consistency of the single annotator

The annotation consistency of the single annotator was analyzed using the MANOVA test. Our goal was to conclude whether the annotations of one annotator are consistent. Ideally, an annotator should assign the same severity to instances similar in terms of code metrics values and different severity to instances different in terms of code metrics values.

The data preparation for the test involves separating annotated instances into groups (a group for each severity). The annotation (severity) is an independent variable, and the metrics are dependent variables. We executed the MANOVA test for each annotator and code smell separately to test the following null hypothesis and its alternative:

- Null hypothesis H_0: There is no significant code metrics difference in different severity groups.
- Alternative hypothesis H_1: There is a significant code metrics difference in different severity groups.

The favorable result, in this case, would be to reject the null hypothesis, as this would mean that metrics do influence[15] annotators' severity annotation.

One of the conditions for performing the MANOVA test is that the number of elements in each group must be greater than the number of the dependent variables[16]. Since this condition was not met on most individual projects, we ran the MANOVA test on the entire data set (on all annotated instances). Based on the p-values from Table 6, the annotation consistency of the individual annotators was satisfactory. The assigned annotations are consistent with metric indicators (annotators marked code snippets with similar metrics' values with the same severity).

Table 6 P-values obtained from MANOVA test results for a single annotator

Code smell	Annotator		
	1	2	3
Large Class	0.000	0.000	0.000
Long Method	0.000	0.000	0.000

6.3.2 The annotation consistency between annotators

The annotation consistency between annotators was analyzed using the MANOVA test. Our goal was to conclude whether the instances annotated with particular severity differ in metrics' values. Ideally, different annotators should assign the same severity to instances similar in terms of their code metrics values.

We prepared the data for the test by separating annotated instances into groups for each annotator, where the annotator is an independent variable and the metrics are dependent variables. We executed the MANOVA test for each severity and code smell separately to test the following null hypothesis and its alternative:

[15] Here we mean implicitly influence. The annotators do not decide the smell severity by looking at the metric values. Rather, the underlying cause (smell) influences the annotators' assessment and metric values.

[16] One-way MANOVA in SPSS Statistics https://statistics.laerd.com/spss-tutorials/one-way-manova-using-spss-statistics.php. Accessed August 9, 2021

- Null hypothesis H_0: There is no significant difference in metric values of instances annotated with the same severity.
- Alternative hypothesis H_1: The metrics' values of instances annotated with the same severity are different.

The favorable result, in this case, would be *not* to reject the null hypothesis. Rejecting the null hypothesis means that the annotator assigned the same severity to very different instances (in terms of metric values).

As in testing the consistency of a single annotator, the condition for the size of the data set of individual projects was not met here, so we ran the MANOVA test on the entire data set. P-values shown in Table 7 indicate no statistically significant differences between groups in most cases, which is favorable in this case. The absence of a difference between the groups indicates that the metrics' values of the instances annotated with the same severity by different annotators are similar. The test showed that in some cases (severities 1 and 2 for Long Method code smell), different annotators annotated instances with non-similar metrics' values with the same severity (cells marked with *). After obtaining the results of the MANOVA test, we executed the ANOVA test for each metric separately to determine which metrics influenced the inconsistency between annotators.

Table 7 P-values obtained from MANOVA test results between annotators

Code smell	Severity			
	0	1	2	3
Large Class	0.7072	0.3973	0.9994	0.9838
Long Method	0.7989	0.000*	0.0014*	0.6832

Table 8 shows the metrics for which the p-values from the ANOVA test were less than the significance level (given the number of metrics, we do not display a table with all p-values). The results indicate that instances annotated with the same severity have statistically significant differences in these metrics.

Table 8 Long method metrics with the least impact on the annotations

Severity	No. of metrics	Metrics
1	12 out of 18	CYCLO, CYCLO_SWITCH, MLOC, MELOC, NOLV, MNOL, MNOC, MNOA, NONL, NOMO, MMNB, NOUW
2	10 out of 18	CYCLO, CYCLO_SWITCH, MLOC, MELOC, NOLV, MNOL, MNOC, NOSL, MMNB, NOUW

Following this analysis, we discuss why instances annotated with the same severity differ in the values of these metrics. We performed a subsequent manual inspection of the inconsistent subset of our annotations. In some cases, particularly in our earlier annotations, there were human errors. We corrected these annotations to improve the correctness of our dataset. However, there were cases where we kept a nonzero severity label due to the particularity of the code, even though metric values would signal that there is no code smell (see Supplementary material, Section 2 for examples). Based on such instances, we assume that not all inconsistencies between the values of the metrics and the annotated severity are indicators of incorrect annotations.

6.4 Annotator Observations

Due to the ambiguity of the subject matter, annotators were instructed to pay close attention to the procedure and annotation model and write down all observations they made along the way. High-level observations were concerned with the procedure's format and workflow, which we examined during the retrospective discussions to make them more effective. Low-level observations were related to the annotation model and particular code snippets. We systematically examined the *Reasoning* fields (described in Section 4.1.3) during each retrospective discussion and expanded our annotation model (summarized in Sections 4.1.3 and 4.3) when appropriate.

Apart from expanding the annotation model, frequent discussions enabled us to define how we treat certain code constructs. For example, we agreed to treat anonymous inline functions as part of the containing method. We also agreed that inner classes could be independently annotated while contributing to the outer class's complexity and length.

> **Recommendation 1**: While frequent discussions and retrospectives are helpful for any data annotation, they are essential for data with ambiguous meaning, such as code smells. We recommend that annotators take the time to align their understanding of the subject matter and discuss all observations, especially in the starting rounds of the annotation procedure and after any lengthy break from the annotation work.

While annotating various projects, we discovered that annotation experience could affect the labels. Once we started annotating a new project, the novel domain, coding style, and constructs introduced cognitive overhead that might increase the severity score by a grade. As we got familiar with the project, the code snippets were generally easier to understand, affecting the heuristics related to complexity.

> **Recommendation 2**: Engineers familiar with the code might have a different perception of the presence and especially the severity of a code smell than somebody who has never seen the code (Taibi et al., 2017). This phenomenon affects annotators as well. We recommend annotators consider this familiarity factor and discard, give less weight to, or revisit their first annotations for a given project.

Regardless of previous experience, all three annotators had to go through several projects (about 100 code snippets per smell) to stabilize their labeling strategies and severity scores. Each annotator had to look at different projects, coding styles, and smell severities through the heuristics lens and to contemplate and discuss their applicability to develop a consistent mental model. The initial lack of consistency is why we excluded the first project we annotated (the student project) from the dataset and why we reexamined the second project at the end of the proof-of-concept annotation.

> **Recommendation 3**: While software engineering experience contributes to the quality of code smell detection and resolution (Madeyski & Lewowski, 2020; Taibi et al., 2017), annotating code smells appears to be a loosely related skill. We recommend that annotators label snippets from several different projects and coding styles to stabilize their labeling strategy. They should then discard, give less weight to, or revisit earlier annotations once they become confident in their labeling consistency.

After overcoming the initial labeling inconsistency and considering the familiarity factor, we found two more factors influencing our labeling consistency. First, all three annotators reported a maximum of two

hours per day spent on labeling. It was not easy to maintain focus after that, and the quality and speed of the annotation significantly declined. Second, all three annotators reported a subjective feeling of "annotating too fast, at the cost of quality" after annotating code snippets over a more extended period (e.g., two-three weeks).

> **Recommendation 4**: Annotating code smells following our annotation model is mentally taxing and becomes tedious when practiced over an extended period (i.e., over a few hours a day or several weeks in a row). We recommend annotators spread out their dataset construction and integrate periods of downtime to refresh their perspective and patience.

Before the proof-of-concept annotation, our initial set of Long Method and Large Class heuristics was larger than the one reported in Section 3. For Long Method, we examined the applicability of the "method has expressions at different levels of abstraction" and "method has side-effects" heuristics. We quickly found that the first heuristic always applies when the heuristic "method does multiple things" applies. As it was a subset, we declared the "abstraction levels" heuristic to be a code characteristic of the "multiple things" heuristic. Regarding side-effects, we found it was too difficult to identify and consistently annotate this heuristic. Similarly, the Large Class contained two additional heuristics that we discarded because they were too difficult to examine and consistently label.

> **Recommendation 5**: It is not easy to define a good set of heuristics due to the ambiguity of code smells. We recommend that annotators remain flexible with their annotation model. They should be aware of heuristics that rarely get selected and remove them. They should also look for heuristics that are tightly correlated with other heuristics and consider demoting them to code characteristics of the superset heuristic. Finally, while our experience did not include such cases, annotators should examine if any heuristic could benefit from being divided into multiple heuristics.

Over time, we refined rules that would exclude trivial code snippets that had no chance of suffering from a code smell. We inherited one rule from (Madeyski & Lewowski, 2020) by excluding methods with less than five lines of code (including method header). We then modified this rule to exclude methods with less than three effective lines of code to eliminate short methods that were expanded because of whitespace or comments. Likewise, we introduced a rule to eliminate class code snippets with less than two methods or four fields.

> **Recommendation 6**: Over time, annotators will discover that certain combinations of code characteristics will always result in a zero-severity score for some smell. We recommend that annotators filter out such instances while providing a thorough justification for this exclusion. Importantly, annotators should avoid the pitfall made by previous studies that use too high thresholds (e.g., LOC > 100), as discussed in Section 2.

6.5 Limitations

Here we discuss the limitations of our study and the identified risks inherent to the study of code smells.

The first threat to our study's validity is related to our annotators and their expertise. Firstly, the three annotators had modest experience in software engineering, where one had six years of prior experience, while two had a year and a half. However, our annotators have spent a significant portion of the past two years studying software maintainability and code smells. Furthermore, we showed how

industry experience does not correlate with a practitioner's labeling strategy in the MLCQ dataset (Section 6.2). We also saw examples where engineers of the same experience level had significantly different criteria for determining the severity of a smell. The experience factors are mitigated with the fact that we cross-checked each instance and performed retrospective discussions. Oliveira et al. (2020) found that even novice developers can effectively identify code smells if working collaboratively, except when there is a need for identifying too complex and scattered smell types.

Another limitation related to the annotators is their unfamiliarity with the annotated code. While this problem affects all the results, it is especially prominent when annotators are not familiar with the software's domain. In those situations, the identifier names were less helpful, and there was an increase in cognitive load. Furthermore, different projects adhere to different coding styles and use various language features, patterns, and conventions. An annotator with a strong preference for one coding style might grade another more severely. To combat these tendencies, we trained our annotators to keep these difficulties in mind and separate the source of any identified issues. We examined situations when a code smell resulted from some meaningful heuristic or unfamiliarity with the domain or coding style during our retrospective discussions. However, it is possible that we graded some projects more severely due to the unfamiliarity factors. On the other hand, Oliveira et al. (2020) found that developers unfamiliar with the system performed the code smell detection task as effectively as those with higher familiarity with the system.

The final limitation regarding our annotators is the possibility of human error. Annotating a dataset, in general, is repetitive work prone to mislabeling. Annotating code smells is even more challenging due to the complexity of the instances and ambiguity of the smells. The purpose of our annotation procedure is to combat this difficulty and reduce the chance of human error. We mitigated the risk of mislabeling through our disagreement resolutions and retrospective discussions. Oliveira et al. (2020) showed that collaborative code smell identification reduces subjectivity and significantly improves code smell detection precision and recall. We also introduced validation logic in our supporting tool (Prokić et al., 2021) to avoid trivial mistakes such as forgetting to set the severity or providing invalid data. However, it is still possible that some instances remain mislabeled.

A significant limitation of our annotation procedure stems from the way we examine code snippets. Methods are examined in isolation, without significantly considering the other methods in a class, while classes are graded without understanding the broader package and software design. Without understanding the broader context, it might be challenging to determine if a class or method has too many responsibilities. In software engineering, we often reveal issues that a module has when working on its clients (the code that calls the module) (Fowler, 2018; Martin, 2009). Because of this, an annotator torn between two severity grades might give a lower grade without examining the broader context and a higher grade if they understand how the client code works. Notably, a severity 0 long method cannot be a severity 3 long method in a different context. Apart from raising our annotators' awareness of this issue, we could not address it as it would significantly slow down the annotation process.

While not the focus of our work, the produced dataset is limited to open-source C# projects, where most of the code snippets come from popular solutions. The selected code snippets might have a different distribution of code smells than open-source solutions developed by smaller teams or commercial projects. This selection might affect the generalizability of any machine learning models trained on our dataset.

7 Conclusion

We can confirm that manually annotating code smells is time-consuming and challenging, aligning with earlier findings (Hozano et al., 2018; Azeem et al., 2019). Due to the ambiguity of code smells, annotators must invest the effort to understand a code smell, the heuristics used to define and identify them, and the related code characteristics that can be weighed to determine the presence and severity of the code smell. However, understanding the code smell is only the first step. A systematic procedure must be established to mitigate the mislabeling risks arising from human error, unfamiliarity with the software's domain, and the used coding styles. Finally, consistent annotating requires practice. Annotators should label a significant number of instances for each smell (e.g., over a hundred) before considering their annotations valid.

We developed an annotation model and procedure as a step towards a systematic approach to manual annotation of code smells. We refined our procedure while building a medium-sized corpus for the Long Method and Large Class code smells. We developed a supporting tool for the procedure and made it available to researchers to aid their annotating efforts. We analyzed our resulting dataset through statistical tests and compared it with similar datasets to validate our procedure. Finally, we prescribed a set of recommendations to help annotators perform more efficient and effective code smell annotation.

In future work, we plan to explore other prominent code smells and discover their heuristics and significant code characteristics. Furthermore, we look to understand better the Large Class and Long Method code smells and explore how we can decompose them to align with the low-level code characteristics analyzed by static code analysis and other related tools more closely. Notably, the concern heuristics present a challenge as the automated semantic understanding of code remains an open issue.

Acknowledgments

This research was supported by the Science Fund of the Republic of Serbia, Grant No 6521051, AI-Clean CaDET.

References

Abbes, M., Khomh, F., Gueheneuc, Y. G., & Antoniol, G. (2011, March). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15Th european conference on software maintenance and reengineering* (pp. 181-190). IEEE.

AbuHassan, A., Alshayeb, M., & Ghouti, L. (2021). Software smell detection techniques: A systematic literature review. *Journal of Software: Evolution and Process*, *33*(3), e2320.

Aniche, M., Treude, C., Zaidman, A., Van Deursen, A., & Gerosa, M. A. (2016, October). SATT: Tailoring code metric thresholds for different software architectures. In *2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM)* (pp. 41-50). IEEE.

Azeem, M. I., Palomba, F., Shi, L., & Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, *108*, 115-138.

Bafandeh Mayvan, B., Rasoolzadegan, A., & Javan Jafari, A. (2020). Bad smell detection using quality metrics and refactoring opportunities. *Journal of Software: Evolution and Process*, *32*(8), e2255.

Barbez, A., Khomh, F. and Guéhéneuc, Y.G., 2019, September. Deep learning anti-patterns from code metrics history. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 114-124). IEEE.

Campbell, G. A. (2018, May). Cognitive complexity: An overview and evaluation. In Proceedings of the 2018 international conference on technical debt (pp. 57-58).

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., & De Lucia, A. (2018, March). Detecting code smells using machine learning techniques: are we there yet?. In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)* (pp. 612-621). IEEE.[dataset] Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016a). Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering, 21(3), 1143-1191.

Fontana, F. A., Dietrich, J., Walter, B., Yamashita, A., & Zanoni, M. (2016b, March). Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)* (Vol. 1, pp. 609-613). IEEE.

Fontana, F. A., & Zanoni, M. (2017). Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, *128*, 43-58.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Hofmeister, J., Siegmund, J., & Holt, D. V. (2017, February). Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)* (pp. 217-227). IEEE.

Hovy, E. (2010, July). Annotation. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*.

Hozano, M., Garcia, A., Fonseca, B., & Costa, E. (2018). Are you smelling it? Investigating how similar developers detect code smells. *Information and Software Technology*, *93*, 130-146.

Ide, N., & Pustejovsky, J. (Eds.). (2017). *Handbook of linguistic annotation* (Vol. 1). Berlin: Springer.

Kaur, A. (2020). A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. *Archives of Computational Methods in Engineering*, *27*(4), 1267-1296.

Kovačević, A., Slivka, J., Vidaković, D., Grujić, K. G., Luburić, N., Prokić, S., & Sladić, G. (2021). Automatic detection of Long Method and God Class code smells through neural source code embeddings.

Lacerda, G., Petrillo, F., Pimenta, M., & Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, *167*, 110610.

[dataset] Lenarduzzi, V., Saarimäki, N., & Taibi, D. (2019, September). The technical debt dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering* (pp. 2-11).

[dataset] Madeyski, L., & Lewowski, T. (2020). MLCQ: Industry-relevant code smell data set. In *Proceedings of the Evaluation and Assessment in Software Engineering* (pp. 342-347).

Malhotra, R. (2019). Empirical research in software engineering: concepts, analysis, and applications. Chapman and Hall/CRC.

Mäntylä, M., Vanhanen, J., & Lassenius, C. (2003, September). A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* (pp. 381-384). IEEE.

Mäntylä, M. V., Vanhanen, J., & Lassenius, C. (2004, September). Bad smells-humans as code critics. In 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. (pp. 399-408). IEEE.

Mäntylä, M. V. (2005, November). An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *2005 International Symposium on Empirical Software Engineering, 2005.* (pp. 10-pp). IEEE.

Mäntylä, M. V., & Lassenius, C. (2006). Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, *11*(3), 395-431.

Martin, R. C., Newkirk, J., & Koss, R. S. (2003). *Agile software development: principles, patterns, and practices* (Vol. 2). Upper Saddle River, NJ: Prentice Hall.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Oliveira, R., de Mello, R., Fernandes, E., Garcia, A., & Lucena, C. (2020). Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers. *Information and Software Technology*, *120*, 106242.

Padilha, J., Pereira, J., Figueiredo, E., Almeida, J., Garcia, A., & Sant'Anna, C. (2014, June). On the effectiveness of concern metrics to detect code smells: An empirical study. In *International Conference on Advanced Information Systems Engineering* (pp. 656-671). Springer, Cham.

Piotrowski, P. and Madeyski, L., 2020. Software defect prediction using bad code smells: A systematic literature review. *Data-Centric Business and Applications*, pp.77-99.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., & De Lucia, A. (2014a, September). Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution* (pp. 101-110). IEEE.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., & De Lucia, A. (2014b). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, *41*(5), 462-489.

[dataset] Palomba, F., Di Nucci, D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D., & De Lucia, A. (2015, May). Landfill: An open dataset of code smells with public evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (pp. 482-485). IEEE.

Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., & De Lucia, A. (2017). The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, *44*(10), 977-1000.

[dataset] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018a). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering, 23(3), 1188-1221.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018b). A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology*, *99*, 1-10.

Pecorelli, F., Palomba, F., Khomh, F., & De Lucia, A. (2020, June). Developer-driven code smell prioritization. In *Proceedings of the 17th International Conference on Mining Software Repositories* (pp. 220-231).

Pérez, J. (2011). Refactoring planning for design smell correction in object-oriented software. *Escuela Técnica Superior de Ingeniería Informática*.

Prokić, S., Grujić, K.G., Luburić, N., Slivka, J., Kovačević, A., Vidaković, D., & Sladić, G. (2021). Clean Code and Design Educational Tool. In *2021 44th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (In Press). IEEE.

de Paulo Sobrinho, E. V., De Lucia, A., & de Almeida Maia, M. (2018). A systematic literature review on bad smells—5 W's: which, when, what, who, where. *IEEE Transactions on Software Engineering*.

[dataset] Rasool, G., & Arshad, Z. (2017). A lightweight approach for detection of code smells. *Arabian Journal for Science and Engineering*, *42*(2), 483-506.

Sae-Lim, N., Hayashi, S., & Saeki, M. (2018). An investigative study on how developers filter and prioritize code smells. *IEICE TRANSACTIONS on Information and Systems*, *101*(7), 1733-1742.

Santos, J. A. M., & de Mendonça, M. G. (2015, April). Exploring decision drivers on god class detection in three controlled experiments. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (pp. 1472-1479).

Santos, J.A.M., Rocha-Junior, J.B. and de Mendonça, M.G., 2017. Investigating factors that affect the human perception on god class detection: an analysis based on a family of four controlled experiments. *Journal of Software Engineering Research and Development*, *5*(1), pp.1-39.

Schumacher, J., Zazworka, N., Shull, F., Seaman, C., & Shaw, M. (2010, September). Building empirical support for automated code smell detection. In *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement* (pp. 1-10).

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, *25*(4), 557-572.

Sharma, T., & Spinellis, D. (2018). A survey on software smells. *Journal of Systems and Software*, *138*, 158-173.

[dataset] Sharma, T., & Kessentini, M. (2021). QScored: A Large Dataset of Code Smells and Quality Metrics. *Methods*, *73*, 16-072.

Seabold, S., & Perktold, J. (2010, June). Statsmodels: Econometric and statistical modeling with python. In *Proceedings of the 9th Python in Science Conference* (Vol. 57, p. 61).

Tahir, A., Dietrich, J., Counsell, S., Licorish, S., & Yamashita, A. (2020). A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. *Information and Software Technology*, *125*, 106333.

Taibi, D., Janes, A., & Lenarduzzi, V. (2017). How developers perceive smells in source code: A replicated study. Information and Software Technology, 92, 223-235.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., ... & Noble, J. (2010, December). The Qualitas Corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference* (pp. 336-345). IEEE.

Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, *86*(6), 1498-1516.

Yamashita, A., & Moonen, L. (2013, May). Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)* (pp. 682-691). IEEE.

[dataset] Walter, B., Fontana, F. A., & Ferme, V. (2018). Code smells and their collocations: A large-scale experiment on open-source systems. *Journal of Systems and Software*, *144*, 1-21.

Wohlin, C. (2014, May). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering* (pp. 1-10).