Towards a systematic approach to manual annotation of code smells

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

[a] Faculty of Technical Sciences, University of Novi Sad, Serbia.
* Corresponding author, email: slivkaje@uns.ac.rs

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 perhaps 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 mostly 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 pitfalls 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, refactoring, clean code, dataset, manual annotation

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, a significant

portion 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 [1]. 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 [2], increasing the overall cost of development [1]. Researchers [1][3] and industry leaders [4][5] 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 [1][3][4][5].

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 [4] is present in functions that try to do too many things [5], requiring the programmer to analyze regions of the function to understand their intent before understanding the overall function. By addressing the Long Method smell, programmers reduce the semantic distance between what the method does and how it does it [4]. However, this smell is not strictly tied to the method's length. Notably, functions with 30 lines of code might not suffer from the Long Method smell, while functions with 10 lines might require refactoring to address this issue. As noted in [4], even a single code line might be suitable for extraction into a separate function if its intent is unclear.

Because of this ambiguity, 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 10) 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 [1]. 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. [6] 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.

Manually annotating code smells is time-consuming [6] and challenging [3], where a high disagreement exists between software engineers on which code snippets suffer from some code smell [3]. In [9], the authors 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. to Because of this, there are no large-scale manually constructed datasets.

Many larger code smell datasets are automatically labeled using heuristic-based tools [6]. Such datasets exclude instances that do not satisfy some threshold, eliminating positive instances that would otherwise be identified by an expert. While some studies manually filter the generated dataset to remove false positives, there is no way of knowing the number of false negatives [10][14]. This issue is especially relevant for code smells such as Long Method and Large Class that significantly depend on the code's semantics and not on, for example, the number of code lines.

While the lack of large 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 [6][10][12][18]. Others do not sufficiently train the annotators [7][14]. Notably, most studies start with vague definitions of what they are annotating [3] and produce datasets that are not published in a form that can be used for reliable reproduction, as pointed out in [7].

In this study, we work towards developing a systematic approach to creating a code smell dataset that is useful for training machine learning smell detectors. The primary contribution is the proposed annotation model and procedure and the annotators' experience report. By surveying the literature, we defined an initial version of the annotation model and procedure, which we the refined while building a medium-sized corpus[1] for the Long Method and Large Class code smells [4]. We report on our annotation experience while following the proposed procedure, highlighting any tips, tricks, pitfalls, and risks we identified.

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*[2] platform and supports data acquisition from C# and Java projects. We have also published documentation and tutorial videos to help other researchers benefit from our tool[3]. 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.

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 4 describes our annotation procedure and its composing steps. In Section 5, we present our findings, including the annotators' experience reports, the characteristics of the constructed dataset, and our study's limitations. Finally, Section 6 concludes our work and lists opportunities for future work.

2 Related Annotation Procedures

Experimental studies on source code usually rely on data from three sources, including commercial projects, academic projects, and open-source projects [8]. Researchers favor open-source projects, as the study results are relatively easy to reproduce, validate, and compare with other studies [10]. We reviewed the studies that produced code smell datasets from open-source code snippets and analyzed their annotation procedure.

Walter et al. [10] developed a dataset from 92 Java open-source projects, which are part of the *qualitas corpus* [11] 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-

[1] The dataset is available at to be defined.
[2] The Clean CaDET Platform can be examined at https://github.com/Clean-CaDET/platform
[3] The Dataset Explorer tools, along with the documentation, is available at https://github.com/Clean-CaDET/platform-documentation/wiki/Dataset-Explorer

based tools can be useful as a training set. However, manual annotation is necessary to produce better results.

Fontana et al. [12] also worked on the *qualitas corpus* [11]. 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 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 [13]. Concretely, the dataset instance distribution responsible for the model's high performance significantly differs from a realistic software project [16].

Palomba et al. [14] 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 [3][15].

Another study conducted by the same group [16] 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 [3][4][5] and classes with LOC above 100 [17] might be affected by a smell. Furthermore, even functions with LOC above 10 can impose a readability issue if the code is sufficiently complex [4][5].

Madeyski and Lewowski [7] 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.

To the best of our knowledge, Rasool and Arshad [18] present the only study that includes 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 studies is the starting premise – the definition of a code smell, which is used to train the annotators. Most studies briefly define a code smell, usually with a few high-level sentences, and in the best case with an illustrative example. Furthermore, 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 [15], produces high disagreement among experienced software engineers [3][9], and significantly differs between the scientific literature [17] and industry best practices [4][5]. 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 [7]. Finally, almost all available datasets focus on the Java programming language.

3 Annotation Model

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. We explored the industry's guidelines regarding code smells from books, whitepapers, and blogs authored by notable subject matter experts [4][5][23] and tool vendors that specialize in code quality analysis [25]. We noted what engineers look for when doing code reviews and which metrics the quality analysis tools calculate for their smell detection engines. Next, we explored the scientific literature for related research, including:

- **Studies of automated smell detection**. We examined studies gathered in the systematic literature review on the topic [17] to extract the domain knowledge researchers strived to encode in their algorithms.
- **Studies that examine engineer perception of code smells.** We examined these studies [3][9][21][22][24][26] to extract the guidelines stemming from human intuition regarding code smell annotation and smell severity.
- **Dataset annotation studies**. We examined the literature described in the previous section to extract any annotation guidelines useful to our context.

From this literature we developed a conceptual model of code smells, which we describe in Section 3.1. To provide a concrete example of this abstraction and precisely define what we annotated, we examine the annotation model for the Long Method (Section 3.2) and Large Class (Section 3.3) code smells. We focus on the Long Method and Large Class code smells due to their prevalence in all software types [16], extensive research body from which we can derive heuristics [1][6], and the negative impact they have on software quality attributes [19]. We consider severe instances of the Large Class code smell to be akin to the God Class [24].

3.1 Conceptual Model

Figure 1 describes our code smell annotation conceptual model, where we denote the entities and their relationships.

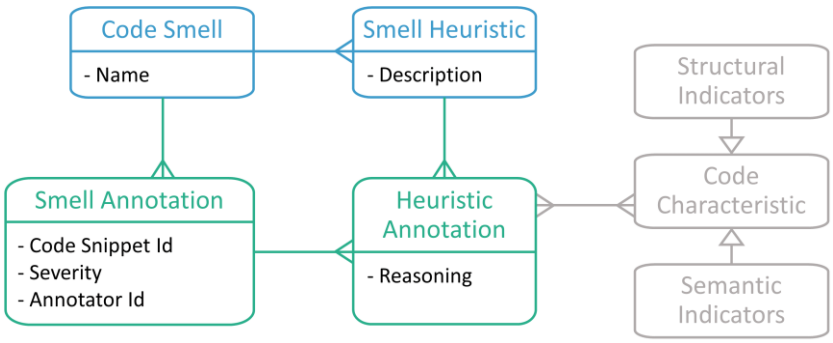

Figure 1 Code smell annotation conceptual model

3.1.1 Code smell model

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 [4].

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 [3]. 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. For the Shotgun Surgery code smell [4], a heuristic might be "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 [4][5], blog posts, tutorials) and empirical research related to the engineer's perception of code smells [3][15].

Importantly, both the code smell and its related heuristics are inherently subjective, which means that

```
//Example 1
private List<CaDETParameter> GetMethodParams1()
{
  List<CaDETParameter> memberParams = new List<CaDETParameter>();
  var paramLists = cSharpMember.DescendantNodes().OfType<ParameterListSyntax>().ToList();
  if (!paramLists.Any()) return memberParams;

  var parameters = paramLists.First().Parameters;
  foreach (var parameter in parameters)
  {
    var symbol = semanticModel.GetDeclaredSymbol(parameter);
    memberParams.Add(new CaDETParameter { Name = symbol.Name });
  }

  return memberParams;
}

//Example 2
private List<CaDETParameter> GetMethodParams2()
{
  List<CaDETParameter> memberParams = new List<CaDETParameter>();
  var paramLists = cSharpMember.DescendantNodes().OfType<ParameterListSyntax>().ToList();
  if (!paramLists.Any()) return memberParams;

  var parameters = paramLists.First().Parameters;
  memberParams.AddRange(parameters
    .Select(parameter => semanticModel.GetDeclaredSymbol(parameter))
    .Select(symbol => new CaDETParameter { Name = symbol.Name }));

  return memberParams;
}

//Example 3
private List<CaDETParameter> GetMethodParams()
{
  var paramLists = cSharpMember.DescendantNodes().OfType<ParameterListSyntax>().ToList();
  if (!paramLists.Any()) return new List<CaDETParameter>();

  return CreateCaDETParams(paramList);
}
```

one engineer might claim that a method is too complex, while another might not. To illustrate this point, we examine three simple code solutions that solve the same requirement in Listing 1.

Few engineers would apply the "method is too complex" heuristic to the third example. However, based on the engineer's experience, knowledge, and even cognitive traits such as working memory capacity [20], the first or second snippet might be labeled with this heuristic. For example, an engineer not familiar with language-integrated queries might prefer no or simple queries and suffer cognitive load when faced with multiple *Select* statements, like in the second example. On the other hand, engineers with higher working memory capacity and familiarity with the language might label all three examples as non-smelly code.

3.1.2 Label model

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 piece of code that can be affected by a code smell, such as functions and classes. An annotator instantiates a **smell annotation** entity for each code snippet that is examined for a specific smell.

The annotator determines the presence and severity of a particular code smell. We used the severity scale defined in [12], where:

0. means there is no smell or that 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 readability. Usually, one or two refactoring operations can resolve the issue. In terms of prioritizing work, we note that such code is "good enough", and effort should be funneled to other development activities.
2. means there is a notable issue that hampers readability. 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 readability 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 while giving reasoning 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 that have two applicable heuristics with a severity of 3, while another had four applicable heuristics and a 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 thirty things), and everything in between. We do not model the severity of each heuristic, as this overly complicates the annotation procedure.

3.1.3 Code characteristics

Significantly, our smell heuristics differ from the heuristics defined by Martin [5] and a significant portion of industry best practice authors. 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. In general, these low-level concepts can be traced to specific lines of code, structural metrics, or concern metrics [21] of the code snippet.

We model Martin's heuristics as **code characteristics**. These low-level concepts can explain why our heuristic is applicable for the given code smell. For example, an annotator might find the heuristic "method does multiple things" applicable when they find several code regions in a function that are delimited by newline characters and comments that explain what the next region of code does. In this case, the comments and newline characters are the code characteristic. 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.

We differentiate two categories of code characteristics, including **structural indicators** tied to structural metrics and **semantic indicators** related to concern metrics [21]. 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 subjective 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.

Notably, we relate these code characteristics to our heuristic annotations very loosely, through the "Reasoning" free-form description. We do not explicitly annotate these indicators, as that would significantly reduce the speed of annotation. Instead, we list notable instances of these indicators during the annotation procedure to refine it and align the annotators' understanding.

3.2 Long Method

Here we denote the sources 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.
- Method is too complex.
- Method does multiple things.

3.2.1 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 [4][5] advocate for short, focused functions. Likewise, engineers of various seniority use the length of the method to determine the Long Method smell [3]. A recent literature review [17] 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 this 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 function, 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 [4] smell. This usually occurred with branching control flow, where expressions were duplicated in multiple branches instead of placed before or after the branching.
- Redundant or unnecessary expressions. This usually accompanied redundant validation or null checks or calculations whose results were never used (e.g., variables that were assigned a value, but never read).

3.2.2 Method is too complex

Literature. Fowler and Kent note that conditionals and loops can be indicators of the Long Method code smell [4]. Further research found that engineers examine the complexity of a method's control structure (e.g., number of branches and loops) to determine if it suffers from the Long Method smell [3]. 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 [17] to determine the Long Method smell, where a complexity above 5 signals the presence of this smell.

Our Findings. Methods for which this heuristic applied required much cognitive processing to understand the intermediate results of sophisticated expressions for which the intent was not clear. Structural indicators based on high cognitive complexity [25] or the maximum number of branches helped us determine if this heuristic is applicable for 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" [5]. 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/or 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 [4] in conditional expressions, or nested ternary operators).

3.2.3 Method does multiple things

Literature. Industry leaders have championed various forms of the single responsibility principle [23], 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 [5]. In the empirical research presented in [3], 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 [4] and researchers [22] 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 [4], where a function would be charged with changing state or performing some logic that should be encapsulated in another class.
- Duplicate Code [4], where a function had repeated regions of code with slight variation.
- Different levels of abstraction [5], 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 usually brought to light by a single annotator, usually the one with the most experience in the subject matter.

3.3 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.
- Class is too complex.
- Class has multiple concerns.

3.3.1 Class is too long

Literature. Fowler [4] states that a Large Class has too many fields, where most methods use a subset of the field set. Likewise, engineers use the length of the class to determine the presence of the Large Class smell [3][21]. A recent literature review [17] 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 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 and 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.

3.3.2 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 [26]. They state that a Large Class is a class that is too complex or that some of its methods are too complex [24]. [17] found that rule engines detect the Large Class smell by considering the total cyclomatic complexity of the class' methods. The most relaxed threshold for this metric (known as WMC – weighted methods per class) was 47.

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, classes that had multiple complex methods, a single complex method with many fields, or a single complex method with a sophisticated inner class were labeled as too complex. Inner classes in general contributed to the complexity of a class, especially when there were multiple non-trivial inner classes. We found two other factors that contributed to a class' complexity:

- Mysterious names [4] played a significant role in obscuring the class' 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 and it was unclear what the responsibility of the examined class was.

3.3.3 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 [23]. Software engineers consider this principle when identifying Large Classes [3]. They try to summarize the class' responsibility in a sentence while avoiding conjunctions that uncover multiple concerns [3][5][27]. Concern metrics, such as concern diffusion over lines of code, are considered good indicators of the Large Class smell [21]. Notably, "being concerned/responsible" for a piece of logic means knowing the details of that logic [23]. 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 are an indicator of a Large Class [24][27].

Our Findings. Like determining the things that 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 Data Annotation Procedure

Starting from the annotation model described in the previous section, we created a dataset of Long Method and Large Class code smells, where we annotated C# code snippets from 10 open-source projects. Figure 2 presents the main activities of the annotation procedure used to create our dataset.

The initial annotation model and procedure was constructed by three authors of the paper (NL, JS, AK). One of them (NL) then followed the annotation procedure, along with two other authors (SP, KGG) to create the final dataset. NL has five years of experience in the software engineering industry and is a professor on several clean code courses. SP and KGG are Ph.D. students researching code quality and code smells with several small-scale industry projects behind them. From the initial annotation model, we conducted two one-hour workshops to further train the annotators through theory and exercises and reach a common understanding of the smells and heuristics.

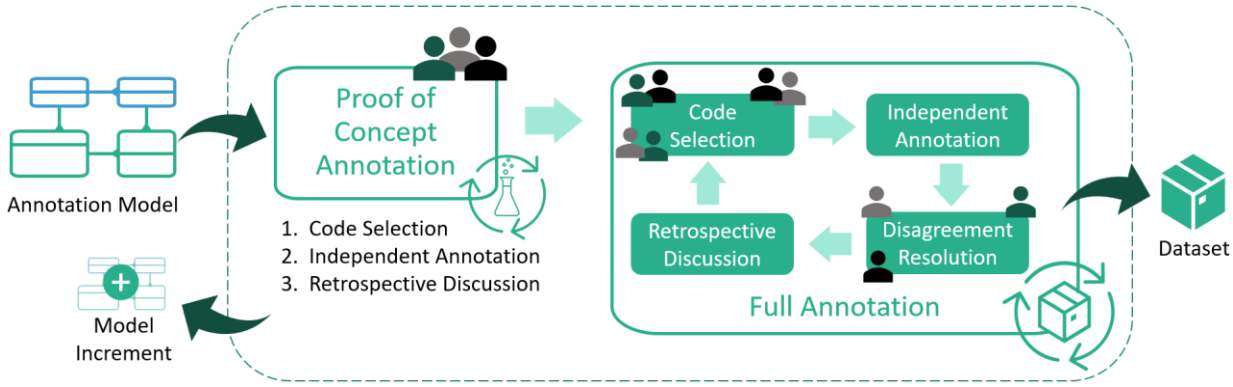

Figure 2 Annotation procedure

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 4.1. Then we performed the full annotation of code snippets to create the complete dataset. We describe the details of this activity in Section 4.2.

4.1 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. 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 [7]. 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	Total classes	Selected classes	Total methods and constructors	Selected methods and constructors
Student Project	Administrative application	81	81	263	263
BurningKnight	Video game	1626	79	6669	305
ShopifySharp	Integration library	254	25	450	20
Core2D	2D diagram editor	304	30	1850	121

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. They also used an instant messaging application to communicate observations and align understanding in real-time. 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. 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 (excluding the student project).

4.2 Full Annotation

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 less than 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	Total classes	Selected classes	Total methods and constructors	Selected methods and constructors
ShareX	Screen capture and media sharing library	810	81	3180	194
OpenRA	Strategy game engine	2192	219	8215	441
Jellyfin	Software media system	TODO			
MonoGame	Video game development framework				
osu!	Video game				
Ocelot	API gateway and security proxy				
MachineLearning	Machine learning framework				

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 [7].

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 Discussion

In this section, we discuss the outcomes of our work and present the related findings and limitations. Section 5.1 describes the annotators' observations and experience of annotating the dataset of code smells. We explore tips, pitfalls, and risks regarding building the annotation model and conducting the procedure. We group these takeaways into an annotation guideline to support researchers in building their datasets. Section 5.2 details the characteristics of our dataset, including its structure and basic statistical analysis. Finally, in Section 5.3, we discuss the limitations of our study and potential threats to validity.

5.1 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 3.1.2) during each retrospective discussion and expanded our annotation model (summarized in Sections 3.2 and 3.3) 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 also contributing to the outer class's complexity and length.

> **Recommendation 1**: While frequent discussions and retrospectives are useful for any data annotation, they are essential for data with ambiguous meaning, such as code smells. We

recommend 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.

5.1.1 Determining code smell severity

While annotating various projects, we discovered that annotation experience could affect the labels. Familiarity with the project affected our labeling. 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 [9]. 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 one reason 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 [7][9], annotating code smells appears to be a loosely related skill. We recommend 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 that influenced 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 extended periods of downtime to refresh their perspective and patience.

5.1.2 Code smell heuristics

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 [7] 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 annotators filter-out such instances while providing thorough justification behind this exclusion. Importantly, annotators should avoid the pitfall made by previous studies that use too generous thresholds (e.g., LOC > 100), as discussed in Section 2.

5.2 Data Set Characteristics

We will provide the Data sheet, major statistical characteristics, and discussion once the rest of the datasets are annotated and run through our procedure.

5.3 Limitations

We will complete the threats to validity discussion once the rest of the datasets are annotated and run through our procedure.

6 Conclusion

Our concluding remarks once the rest of the datasets are annotated and run through our procedure.

Acknowledgments

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

References

[1] Sharma, T. and Spinellis, D., 2018. A survey on software smells. Journal of Systems and Software, 138, pp.158-173.

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

[3] Hozano, M., Garcia, A., Fonseca, B. and Costa, E., 2018. Are you smelling it? Investigating how similar developers detect code smells. Information and Software Technology, 93, pp.130-146.
[4] Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.
[5] Martin, R.C., 2009. Clean code: a handbook of agile software craftsmanship. Pearson Education.
[6] Azeem, M.I., Palomba, F., Shi, L. and Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology, 108, pp.115-138.
[7] Madeyski, L. and Lewowski, T., 2020. MLCQ: Industry-Relevant Code Smell Data Set. In Proceedings of the Evaluation and Assessment in Software Engineering (pp. 342-347).
[8] Malhotra, R., 2016. Empirical research in software engineering: concepts, analysis, and applications. CRC Press.
[9] Taibi, D., Janes, A. and Lenarduzzi, V., 2017. How developers perceive smells in source code: A replicated study. Information and Software Technology, 92, pp.223-235.
[10] Walter, B., Fontana, F.A. and Ferme, V., 2018. Code smells and their collocations: A large-scale experiment on open-source systems. Journal of Systems and Software, 144, pp.1-21.
[11] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H. and 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.
[12] Fontana, F.A., Mäntylä, M.V., Zanoni, M. and Marino, A., 2016. Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering, 21(3), pp.1143-1191.
[13] Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A. and 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.
[14] Palomba, F., Di Nucci, D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D. and 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.
[15] Mantyla, M.V., Vanhanen, J. and Lassenius, C., 2004, September. Bad smells-humans as code critics. In 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. (pp. 399-408). IEEE.
[16] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R. and De Lucia, A., 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering, 23(3), pp.1188-1221.
[17] Bafandeh Mayvan, B., Rasoolzadegan, A. and Javan Jafari, A., 2020. Bad smell detection using quality metrics and refactoring opportunities. Journal of Software: Evolution and Process, p.e2255.
[18] Rasool, G. and Arshad, Z., 2017. A lightweight approach for detection of code smells. Arabian Journal for Science and Engineering, 42(2), pp.483-506.
[19] 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), pp.1267-1296.

[20]Hofmeister, J., Siegmund, J. and 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.

[21]Padilha, J., Pereira, J., Figueiredo, E., Almeida, J., Garcia, A. and 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.

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

[23]Martin, R.C., 2002. Agile software development: principles, patterns, and practices. Prentice Hall.

[24]Santos, J.A.M. and 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).

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

[26]Palomba, F.; Bavota, G.; Penta, M.D.; Oliveto, R.; Lucia, A.D. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, 2014.

[27]J. Schumacher, N. Zazworka, F. Shull, C. Seaman, M. Shaw, Building empirical support for automated code smell detection, in: Proceedings of the 2010 ACM–IEEE International Symposium on Empirical Software Engineering and Measurement.