This document lists the supplementary material for the paper "*Towards a systematic approach to manual annotation of code smells*". The document is divided into six sections which are referred to in the appropriate segment of the main paper.

1 Code heuristics and their subjective interpretation

Code smell heuristics are inherently subjective, which means that one engineer might claim that a method is too complex, while another might not. This subjectivity is the reason why it is challenging to apply rule engines or other automatic tools to identify and label code smells. To illustrate this point, we examine three simple code solutions that solve the same requirement in Listing 1.

```
//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);
}
```

Listing 1 Code examples that illustrate code smell heuristic subjectivity

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 [1], 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.

2 Examples of instances where the severity label does not match the overall metric values

In this section, we list several example code snippets that, through following our guidelines, we labeled as severity 1, despite the low metric values. We also list the instances that we labeled as 0, despite the high metric values. We group the examples around the Long Method and Large Class smells.

2.1 Long Method examples

- EffectsExtension1 and RegionCaptureTransparentForm2 present instances that have below average metric values for most metrics. They are simple methods with no structural complexity. However, our annotators decided to label these methods with severity 1 due to the semantic complexity arising from the used identifier names.
- DefaultCursor3 presents a method with almost 70 lines of code that follows the project's convention for instantiating UI elements. Any complexity is significantly reduced to anyone familiar with this convention, which is why we labeled it as 0.
- Init4 is a specific method that has almost 50 code lines and cyclomatic complexity of 13. However, we labeled it with 0 as it is focused on a semantically cohesive task and doesn't present comprehensibility issues.

2.2 Large Class examples

- EpisodeNfoSaver5 and Plant6 present classes that are dominated by a single Long Method. This method significantly increases various class metrics, like LOC and WMC. However, since the method is the issue and not the containing class, our annotators decide to label the classes as 0.
- ManiaPlayfield7 presents a class with moderate complexity in terms of metric values like WMC. However, because the identifier names of the class members are meaningful, the complexity is reduced enough to warrant a severity 0 label.

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 [2] 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. [3]. However, Palomba et al. [3] employed a semi-

1 https://github.com/MonoGame/MonoGame/blob/4802d00db04dc7aa5fe07cd2d908f9a4b090a4fd/MonoGame.Framework/Platform/Audio/OpenAL.cs#L762-L785

2 https://github.com/ShareX/ShareX/blob/c9a71ed00eda0e7c5a45237b9bcd3f8f614cda63/ShareX.ScreenCaptureLib/Forms/RegionCaptureTransparentForm.cs#L64-L94

3 https://github.com/ppy/osu/blob/2cac373365309a40474943f55c56159ed8f9433c/osu.Game.Rulesets.Osu/UI/Cursor/OsuCursor.cs#L69-L135

4 https://github.com/MonoGame/MonoGame/blob/4802d00db04dc7aa5fe07cd2d908f9a4b090a4fd/MonoGame.Framework/Platform/Utilities/CurrentPlatform.cs#L26-L72

5 https://github.com/jellyfin/jellyfin/blob/6c2eb5fc7e872a29b4a0951849681ae0764dbb8e/MediaBrowser.XbmcMetadata/Savers/EpisodeNfoSaver.cs#L18-L133

6 https://github.com/egordorichev/BurningKnight/blob/a55594c11ab681087356af2c129c2d493eba4bd2/BurningKnight/level/entities/plant/Plant.cs#L12-L78

7 https://github.com/ppy/osu/blob/2cac373365309a40474943f55c56159ed8f9433c/osu.Game.Rulesets.Mania/UI/ManiaPlayfield.cs#L18-L135

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. [4] 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[8]. 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 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 [2], [3], [4], [5] 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 labelling procedure (Section 1).

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. [4] 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. Hozano et al. [4] report a significant disagreement on code smell perception.

Madeyski and Lewowski [2] 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 [6], 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[9], 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. [7] 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 analysed code smells. Oliveira et al. [8] 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.

[8] 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.

[9] 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.

4 Code heuristics for the Long Method code smell

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.

4.1 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 [9], [10] advocate for short, focused functions. Likewise, engineers of various seniority use the length of the method to determine the Long Method smell [4]. A recent literature review [11] 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 methods, 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 [9] 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 [12].
- 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 Method is too complex

Literature. Fowler and Kent note that conditionals and loops can indicate the Long Method code smell [9]. 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 [4]. 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 [11] 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 [13] or the maximum number of branches helped us determine if this heuristic applies to a given method. Exceptions to this rule included methods 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” [10]. Methods 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 method arguments, a Message Chain smell [9] in conditional expressions, or nested ternary operators).

4.3 Method does multiple things

Literature. Industry leaders have championed various forms of the single responsibility principle [14], 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 [10]. In the empirical research of Hozano et al. [4], 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 [9] and researchers [15] 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 [9], 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 [13].
- Duplicate Code [9], where a function had repeated regions of code with slight variation.
- Different levels of abstraction [10], where out of place low-level expressions were nested among higher-level method calls, signalling 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.

5 Code heuristics for the Large Class code smell

Here we define the sources that influenced our heuristic selection for the Large Class code smell and their related code characteristics. 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.

5.1 Class is too long

Literature. Fowler [9] 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 [4], [16]. A recent literature review [11] 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, 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 favoured 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.

5.2 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 [17]. They state that a Large Class is too complex or that some of its methods are too complex [7]. Bafandeh Mayvan et al. [11] found that rule engines detect the Large Class smell by considering the total cyclomatic complexity of the class's methods.

Our Findings. A significant portion of the labelled classes had their complexity stem from one or more complex methods. Notably, we avoided labelling 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 labelled 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's complexity:

- Inner classes, in general, contributed to the complexity of a class, especially when there were multiple non-trivial inner classes.
- Mysterious names [9] 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.

5.3 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 [14]. Software engineers consider this principle when identifying Large Classes [4]. They try to summarize the class's responsibility in a sentence while avoiding conjunctions that uncover multiple concerns [4], [10], [17]. Concern metrics, such as concern diffusion over lines of code, are considered good indicators of the Large Class smell [16]. Notably, "being concerned/responsible" for a piece of logic means knowing the details of that logic [14]. 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 [7], [17].

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.

6 Details of the statistical tests conducted to test the annotation consistency

This Section provides the details of the MANOVA and ANOVA tests conducted to test the code smell annotation consistency regarding the code metrics (Section 4.1.2 of the manuscript).

We used the ANOVA (Analysis of Variance) and MANOVA (Multivariate Analysis of Variance) tests from the Statsmodels library [18]. As dependent variables, we used 25 class-level metrics and 18 method-level listed in Table 1. We chose the metrics that researchers used for Large Class and Long Method detection in heuristic-based approaches [11] and ML-based approaches [19]. Many of these metrics are form CK metrics suite [20] that is widely adopted in the studies researching code smell detection.

Regarding the interpretation of test results, 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).

Table 1 The list of metrics calculated by our Clean CaDET platform[10]

Level	Acronym	Definition
Class	CLOC	Lines of code of a class
	CELOC	Effective lines of code of a class
	LCOM, LCOM3, LCOM4	Lack of cohesion of methods
	NMD	Number of methods declared
	NAD	Number of attributes defined
	NMD_NAD	Number of methods and attributes
	WMC, WMC_NO_CASE	Weighted methods per class WMC_NO_CASE does not count Cases
	ATFD	Access to foreign data
	TCC	Tight class cohesion
	CNOR	Total number of return statements from all class members
	CNOL	Total number of loops from all class members
	CNOC	Total number of comparison operators from all class members
	CNOA	Total number of assignments from all class members
	NOPM	Number of private methods
	NOPF	Number of protected fields
	DIT	Depth inheritance hierarchy
	DCC	Direct class coupling
	CMNB	Max nested blocks from all class members
	RFC	Response for a class, counts the number of unique method invocations
	CBO	Number of dependencies
	NIC	Number of inner classes
Method	CYCLO, CYCLO_SWITCH	Cyclomatic (McCabe's) complexity. Also denoted as: VG or CC CYCLO_SWITCH doesn't count every case label, only the switch label
	MLOC	Lines of code in a method
	MELOC	Effective lines of code of a method, excluding comments, blank lines, function header and function braces
	NOP	Number of parameters. Also denoted as: PAR or PL
	NOLV	Number of local variables
	NOTC	Number of try catch blocks
	MNOL	Number of loops in method
	NOR	Number of return statements in method
	MNOC	Number of comparison operators in method
	MNOA	Number of assignments in method
	NONL	Number of numeric literals
	NOSL	Number of string literals
	NOMO	Number of math operations
	NOPE	Number of parenthesized expressions
	NOLE	Number of lambda expressions
	MMNB	Max nested blocks in the method
	NOUW	Number of unique words

[10] The list of metrics and their definitions is available at https://github.com/Clean-CaDET/platform/blob/c4acff95ec00ff6c25fa62dde4818c1f40e39d39/CodeModel/CaDETModel/CodeItems/CaDETMetrics.cs

6.1 The annotation consistency of the single annotator

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 favourable result, in this case, would be to reject the null hypothesis, as this would mean that metrics do influence[11] 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[12]. 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 2, 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 2 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

6.2 The annotation consistency between annotators

The annotation consistency between annotators was analysed 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:

- 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.

[11] 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.

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

The favourable 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 3 indicate no statistically significant differences between groups in most cases, which is favourable 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 3 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 4 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. For example, different annotators assigned severity 1 to the Long Methods inconsistently regarding the MLOC (method lines of code) metric.

Table 4 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 3 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.

7 References

[1] A. Kovačević, J. Slivka, D. Vidaković, K. Grujić, N. Luburić, S. Prokić and G. Sladić, "Automatic detection of Long Method and God Class code smells through neural source code embeddings," *Expert Systems with Applications,* vol. 204, p. 117607, 2022.

[2] L. Madeyski and T. Lewowski, "MLCQ: Industry-relevant code smell data set," in

Proceedings of the Evaluation and Assessment in Software Engineering, 2020.

[3] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering,* vol. 23, no. 3, pp. 1188-1221.

[4] M. Hozano, A. Garcia, B. Fonseca and E. Costa, "Are you smelling it? Investigating how similar developers detect code smells. Information and Software Technology," vol. 93, pp. 130-146, 2018.

[5] G. Rasool and Z. Arshad, *A lightweight approach for detection of code smells. Arabian J. for Science and Engineering,* vol. 42, no. 2, pp. 483-506, 2017.

[6] A. Kovačević, J. Slivka, D. Vidaković, K. Grujić, N. Luburić, S. Prokić and G. Sladić, "Automatic detection of Long Method and God Class code smells through neural source code embeddings. Expert Systems with Applications," vol. 204, p. 117607, 2022.

[7] J. Santos, J. Rocha-Junior and M. de Mendonça, "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.*

[8] R. Oliveira, R. de Mello, E. Fernandes, A. Garcia and C. Lucena, "Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers," *Information and Software Technology,* vol. 120, p. 106242 , 2020.

[9] M. Fowler, Refactoring: Improving The Design of Existing Code, Addison-Wesley Professional, 2018.

[10] R. C. Martin, Clean Code: a Handbook of Agile Software Craftsmanship, Pearson Education, 2009.

[11] B. Bafandeh Mayvan, A. Rasoolzadegan and A. Javan Jafari, "Bad smell detection using quality metrics and refactoring opportunities," *Journal of Software: Evolution and Process,* vol. 32, no. 8, p. e2255 , 2020.

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

[13] G. A. Campbell, "Cognitive complexity: An overview and evaluation," in *Proceedings of the 2018 International Conference on Technical Debt*, 2018.

[14] R. C. Martin, J. Newkirk and R. S. Koss, Agile Software Development: Principles, Patterns, and Practices, vol. 2, Prentice Hall, 2003.

[15] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering,* vol. 44, no. 10, pp. 977-1000, 2017.

[16] J. Padilha, J. Pereira, E. Figueiredo, J. Almeida, A. Garcia and C. Sant'Anna, "On the effectiveness of concern metrics to detect code smells: An empirical study," in *International Conference on Advanced Information Systems Engineering*, 2014.

[17] J. Schumacher, N. Zazworka, F. Shull, C. Seaman and 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*, 2010.

[18] S. Seabold and J. Perktold, “Statsmodels: Econometric and statistical modeling with python,” in *Proceedings of the 9th Python in Science Conference*, 2010.

[19] M. Azeem, F. Palomba, L. Shi and Q. Wang, “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis,” *Information and Software Technology,* vol. 108, pp. 115-138, 2019.

[20] M. Aniche, “Java code metrics calculator (CK),” 2015. [Online]. Available: https://github.com/mauricioaniche/ck/. [Accessed 3 9 2022].

[21] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto and A. De Lucia, “Do they really smell bad? a study on developers' perception of bad code smells,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014.