

# Code Smell Detection using Machine Learning Classification Algorithm

Law Teng Yi

Faculty of Computer Science and Information Technology, New Era University College,  
Kajang, Malaysia

DOI: <https://dx.doi.org/10.47772/IJRISS.2024.805063>

Received: 22 April 2024; Revised: 01 May 2024; Accepted: 06 May 2024; Published: 05 June 2024

## ABSTRACT

Code smell indicates a poor implementation choice that affects software quality attributes (Pérez, 2013). Fowler (1999) also describes it as an internal code-level problem where the code becomes complex, the design broken, and eventually worsens software quality. Jose (2020) has reported that most applied existing approaches for code smells detection are search-based (30.1%), metric-based (24.1%), and symptom-based approaches (19.3%). However, these existing approaches can only apply to simpler detection; the greater the complexity of code smell, the lower the results for code smell detection (Mantyla M, 2004). Kessentini (2014) also has reported that detecting the problems of code smell is difficult and the performance is not effective using the existing approaches such as search-based, symptom-based, visualization-based, probabilistic, cooperative-based, manual, metrics-based, and rule-based. As a result, many of these approaches extend to the application of machine learning classifiers in software code smell detection. Fontana (2016) reported that a supervised machine learning strategy can be used to forecast the value of the dependent variable using machine learning classifiers to address the problem. In this project, we propose a machine learning supervised Gaussian processes algorithm for JAVA open-source code smell detection. The Gaussian process is a highly interpretable supervised machine learning algorithm used in regression testing to quantify prediction uncertainty. A code smell detection application prototype will be developed to implement the proposed work. The effectiveness of the proposed work in terms of detection accuracy will be evaluated further.

**Keywords:** Code Smell, Machine Learning classifiers, regression testing, Gaussian Process

## INTRODUCTION

The phrase “smell” refers to an inherent issue in software, either at the code level (Fowler, 1999) or higher, characterizing symptoms noticed in components that impede software progress. Code smells are breaches of code design principles (Fowler, 2019), and they contribute to technical debt, impacting programmed maintenance and evolution. It is indisputable that the notion of smells was originally adopted by the agile software development community as a means of pointing out flaws or areas for improvement. This phrase is now used in the industry to describe anomalies in software components. According to Jose (2020), the most widely used existing ways for detecting code smells are search-based (30.1%), metric-based (24.1%), and symptom-based approaches (19.3%). However, current methodologies can only be used for simple detection; the larger the complexity of the code smell, the worse the results for code smell detection (Mantyla, 2004). Kessentini (2014) also reported that detecting the problems of code smells is difficult and the performance is not effective using existing approaches such as search-based, symptom-based,

visualization-based, probabilistic, cooperative-based, manual, metrics-based, and rule-based.

A search-based technique is utilized at each phase to develop a solution by selecting the local best option from a pool of possibilities. Others construct a neighborhood of viable solutions, which are obtained by modifying the search-based approach. The quality of the solutions is assessed, and a candidate solution is chosen to be the current one. When the halting conditions are met, the current solution is returned. The search-based method generates a large sequence of refactorings as one solution without explaining to developers how the various operations in the solution are dependent on each other in terms of fixing specific quality issues or improving fitness functions, which can affect developers' trustworthiness in practice.

Because the developer must either accept or reject the whole refactoring solution, the search-based method is restricted in its versatility. Furthermore, because development is halted during the refactoring process, fully automated refactoring methodologies are inappropriate for floss refactoring, where the goal is to maintain great design quality while modifying existing functionality.

The simpler detection is requiring code inspection and human judgment and this unfeasible for large software systems. The detection of simpler CS used probabilistic, metric-based, symptom-based, and search-based are achieved precision and recall the detection techniques are very high.

The major aim is to develop code smell detection application prototype to evaluate the effectiveness of detection. The code smell detection application prototype implement using gaussian process classifiers for God Class design smell detection in JSmell, which adding the gaussian process algorithms into JSmell tools.

The experiment deals with the effectiveness of detection of God Class design detection in JSmell. The experiment is based on the research question which is RQ1: The effectiveness of code smell detection application prototype and existing application code smell detection? .RQ2: How does Gaussian process classifiers affect effectiveness to detect God Class code smell. We have formulated research questions in two hypotheses denoted as H<sub>0</sub> and H<sub>a</sub>. The following specify the null and alternative hypothesis.

- H<sub>0</sub>: The equality of the effectiveness of prototype detection God Class and existing detection code smell tool.
- H<sub>a</sub>: The difference of the effectiveness of prototype detection God Class and existing detection code smell tool.

## LITERATURE REVIEW

Fowler and Beck (Fowler, 2000) identified and proposed higher levels of bad code smells taxonomy for classifying the classes. The classes are bloaters, object-orientation abusers, change preventers, dispensables, encapsulators, and couplers. Bloaters are instances of code that have grown so huge that they can no longer be handled efficiently. The Bloater category includes Long Method, Large Class, Primitive Obsession, Long Parameter List, and Data Clumps. Categories of Object-Oriented abuser have switch statements, temporary field, refused bequest, Alternative classes with different interfaces, and parallel inheritance hierarchies. Because the Alternative Classes with Different Interfaces smell lacks a common interface for closely related classes, it can also be considered a sort of inheritance abuse. The Category of Change Preventers refers to code structures that hinder modification of the software. These categories include Divergent Change and Shotgun Surgery. The crucial point is that the classes and prospective modifications must have a one-to-one connection. Category of dispensables are Lazy Class, Data Class, Duplicate Code, and Speculative Generality. These code smells denote something that should be deleted from the code.

Classes that are not contributing enough must be deleted or their responsibilities enhanced. Category of Encapsulators are the Message chains and Middle Man. The fragrances in this category are opposites, which means that lowering one will cause the other to grow. Encapsulators deal with the objects, data, or operations that are accessed. Category of couplers which are feature envy and Inappropriate Intimacy. Both code smells indicate strong coupling, which is contrary to the object-oriented design principles. Of course, we might argue that these odours belong in the Object-Orientation Abusers category, but since they both focus solely on coupling. It appears obvious that the presence of some fragrances would correlate favorably with the presence of others, while others would correlate negatively. We identified negative connections only with the Primitive Obsession fragrance (Mika Mäntylä, 2003) in our small sample research, which had the greatest ( $r > 0.575$ ) and most significant ( $p < 0.01$ ) associations between the code smell.

From Fernandes et al., only focused on tools rather than the techniques. However, Vale et al. take considers in the product lines and highlight the exist of the code smell using which techniques can be applied. The most used approaches to identifying refactoring opportunities are quality metrics-oriented, pre-condition oriented, and clustering oriented (Guilherme Lacerda, 2020). Fowler et al. define the 22 sets of symptoms of code smell that include large classes, feature envy, long parameter lists, and lazy classes (M. Fowler, 1999). In addition, Fowler provided 68 object-oriented refactoring techniques with the goal of reorganizing classes, methods, and variables for modifications and extensions for programmed maintenance (Martin Fowler, 2002).

A Literature Review on Code Smells and Refactoring is carried out, which investigates the significance of detecting and correcting code smells in terms of availability, relativity, scalability, unobtrusiveness, expressiveness, context sensitivity, and Rationality. Various code smells with varied symptoms have been explored in try to enhance detection and provide improvement suggestions. Each sort of code smell is followed by refactoring recommendations to eliminate it. The table below lists and discusses 22 distinct forms of code smells.

Long methods are seen as a code smell because the more lines of code a method has, the more difficult it is for developers to comprehend and understand what the function does. Long methods are caused by the fact that it is easier to write code than to read it, resulting in more code being added over time without removing unneeded complexity (Aleksandar Kovačević, 2022). Long methods can be solved by breaking them down into smaller, more focused methods. It can replace the entire method with a new method object and decomposing sophisticated conditional logic. Long methods are regarded an anti-pattern in software development because they break concepts of modularity and make the code harder to comprehend and maintain. The metric threshold for detecting long method code smell is MLOC greater than 30, VG greater than 4, and NBD greater than 3, indicating that the lengthy method is large, complex, and has a high number of nested blocks (Priscila P. Souza, 2017).

The code smell is detected in three steps. The first step is to use JFlex and Java to parse the source code. The second phase is Reification, which is essentially a defect definition based on the target system's meta-model (Ra'ul Marticorena, 2005), and this procedure is used to capture high-level faults, and a repository is kept track of them. The detection method is designed and implemented as visitors on the meta-model as the final phase.

Amal Alazba's research provides actual evidence for software practitioners and machine learning engineers on the use of stacking ensemble applications in identifying code smells, hence influencing the software reworking process and the associated software quality assurance duties. The findings show that specialized classifiers, such as the Gaussian Process, Multilayer Perceptron, and Decision Trees, can detect most code smells (Aliamaan, 2021).

Heuristic techniques use detection criteria based on software metrics to discover code smells. The typical technique used by such approaches consists of two steps: identifying the primary symptoms that characterize a code smell and mapping them to a set of thresholds based on structural metrics. The second stage is to combine these symptoms to arrive at the final rule for detecting the code smell. Kreimer suggested a prediction model based on code metrics as independent variables that can lead to high levels of accuracy, as well as a decision tree to identify two Code smells (J. Kreimer, 2005).

JSmell is a software tool that helps programmers to automatically detect bad smells in Java code (Moha, 2007). This tool also displays a tree structure of the system's source code breakdown. This tool assists the user in comprehending source code at a high level. JSmell (Roperia, 2009) is a C#-based Java fragrance detector. "Data Class," "Message Chain," "Primitive Obsession," "Speculative Generality," "Parallel Inheritance Hierarchy," "Duplicate Code," and "Comments" are among the seven code scents identified. It parses the code file with the ANTLR (ANOther Tool for Language Recognition) parser and collects statistical information to identify the Smells.

To detect a smell, JSmell comprises two steps. During the first phase, it parses all Java source code files and collects necessary information such as method declarations, variable declarations, and class names. It then utilizes this statistical data and parses all of the code again in the second phase to detect the scents present in each of them. This work aims to implement the Gaussian process classifiers for God Class design smell detection in JSmell, which adds the Gaussian process algorithms into JSmell tools. A God Class, sometimes known as a Blob, is a class that tends to centralize the majority of the system's intellect and takes on a large number of tasks. It is distinguished by the existence of several characteristics, methods, and dependencies on data classes.

The most of available approaches either reverse engineer the source code to discover design patterns or create tools to detect design patterns in source code (Detten, 2010). Although it is very simple to extract structural pieces from source code such as classes, attributes, methods, and translate them into graphs or other representations, but it has low accuracy and fails to forecast most design patterns successfully (Dongjin Yu, 2018). Simultaneously, extracting semantic (lexical) information from source code is difficult and has not yet been extensively tried in finding design patterns.

Code smell detection is the process of finding code fragments that may violate the structure or semantic properties related to coupling and complexity. For example, in this scenario, it requires the internal attributes to define these properties and capture through software metrics and properties that can be expressed for following metrics. According to Fowler's research, the code smell has been identified with the software refactoring. From a collection of previous work to detect the code smell that can using machine learning-based code approach and the hybrid-based code approach for software system. Due to the extended issues, parameter tuning in machine learning technique applied by using hybrid-based code approach to solve the issue. The hybrid-based code combines with meta-heuristic technique and algorithm to find the code smell occur.

Machine learning technique can help to differentiate the characteristics of code smell and non-code smell of source code. To exist of machine learning technique, the proposed of the multi-label classification methods to find the code smell for affected by the multiple type of code smell (Guggulothu, 2020). The author uses the dataset that converted into multi-label dataset from the two-code smell dataset, and it show the positive correlation among code smell are the long method and feature envy. The manual static code analysis code smell in Machine learning application and using 74 from open-source project and run on Pylint by using manual analysis to show the code smell are detected in duplicate occurs occurs (vanOort, 2021).

The multi-criteria approach provided here is designed to facilitate generalization to other scents, but efforts

should be taken to validate and adjust to those other smells. Establishing precedence among different types of code smell is outside the scope of this investigation. The method is broken down into five steps.

In the initial phase, we randomly selected a group of Java software projects, with the source code acting as the approach's input. The projects chosen differed in size and scope. They were taken from the Source Forge repository (Khalid Alkharabsheh, 2022), which is one of the most well-known open-source repositories. The next step is Multiple smell detectors are employed to analyze the source code of the target version of the software. Several tools were employed to detect God Classes. We chose a collection of tools that were frequently referenced and used in the context. Furthermore, they have a high precision of God class detection (Khalid Alkharabsheh, 2022).

The output of each tool will be incorporated into the preliminary list of God classes received from all tools (union set) in this stage after the software systems have been analyzed in the previous step. In the fourth stage, some of the parameters for each God Class in the union set for each of the three criteria are computed, which is regarded as the heart of the proposed approach. The examined aspects focus on code stability, maintainability, and developer evaluation. First, the historical information criterion is used to assess the stability of the classes on the target version. For this reason, a sample of prior versions should be studied. Furthermore, the maintainability element is evaluated by taking into account the density and severity of odors in the target version. Finally, the God class list is prioritized and retrieved.

To choose the tools, we used a list of criteria that included being available and free, being common in God class identification, analyzing Java source code, and having good detection accuracy. The chosen tools were the most referenced in works relevant to the activity of design scent detection, according to our systematic mapping study on design smell detection, released in 2019 (Class, 2018). To detect the God class, the selected tools used a variety of methodologies. The strategies were developed using specific definitions of the God class. The use of different detection algorithms will raise the total number of God classes in the dataset while minimizing the threats to construct validity.

## METHODOLOGY

The open-source tool Designite Java includes code smells categorized into two categories: design smells and implementation smells. Design smells cover 17 categories, while implementation smells cover 10 to detect the code smells. Designite Java comes with a command line to input the command to run the application.

```
λ java -jar DesigniteJava-1.1.2.jar -i input_f -o output_f
output_f
detect_comments.txt
Parsing the source code ...
Resolving symbols...
Extracting metrics...
Extracting code smells...
Done.
```

Figure 3.1. Designite Java Process

In the third step of extracting metrics, the thresholds of each code smell are extracted. After extracting metrics, code smells are extracted by using the thresholds and saved to a CSV file record. The process is done with 4 steps.

Designite Java application detects the 10 implementation code smells that do not include comments code smell. We add two code smells which are comments code smell and God Class code smell. Comments code smell can detect the single-line and multiple-line comments in a Java file. To detect comments code smells, we add a regular pattern to match specific types of comments. The regular pattern is shown in the table.

Single Comments	//[^\\r\\n]*
Multi Comments	/\\*[[\\s\\S]?\\*/
String Comments	\\\"(?:\\\\\\\\. [^\\\\\\\\\\\"\\r\\n])*\\\"
Character Comments	'(?:\\\\\\\\. [^\\\\\\\\\\'\\r\\n])+'
Any of comments	[\\s\\S]

Table3.1.Regular Pattern of Comments Code Smell

The comments code smell did not need have threshold and formula to detect comments. To apply all pattern, we use the java compile function to detect all regular pattern and match to write it to output file.

```

// System.out.println(path);
else {
    if (f.getName().endsWith(suffix: ".java")){
        String ab_file=f.getAbsolutePath();
        System.out.println(ab_file);
        try {
            String contents = read(new File(ab_file));

            String slComment = "//[^\\r\\n]*";
            String mlComment = "/\\*[[\\s\\S]?\\*/";
            String strLit = "\\\"(?:\\\\\\\\.|[^\\\\\\\\\\\"\\r\\n])*\\\"";
            String chLit = "'(?:\\\\\\\\.|[^\\\\\\\\\\'\\r\\n])+'"';
            String any = "[\\s\\S]";
            String timeStamp = new SimpleDateFormat(pattern: "ddMMyyyy_HHmm").format(Calendar.getInstance().getTime());

            Pattern p = Pattern.compile(
                String.format(format: "(%s)|(%s)|%s|%s|%s", slComment, mlComment, strLit, chLit, any)
            );

            Matcher m = p.matcher(contents);

```

Figure 3.2. Compile Library Function

Compile function is built in java function to compile the pattern of regular pattern. To try read java file first and compare the regular pattern, the true Boolean which match with pattern, it list in text file.

Below is the equation showing the formula and states the thresholds for detecting the God class (GC).

$$GC(C) = \begin{cases} 1, & ((WMC(C) \geq 47) \wedge (TCC(C) < 0.3) \wedge (ATFD(C) > 5)) \\ 0, & else \end{cases} \quad (1)$$

Figure 3.3. God Class Thresholds Formula

WMC is the weight method count which sum of the cyclomatic complexity of all methods in class. TCC is the relative number of directly connected to methods in class. ATFD is the accessed to foreign data which the number of attributes of foreign classes or via the accessor methods.

To calculate the tight class cohesion which formula of below:

$$TCC = NDC / NP$$

NP is the maximum number for possible connections which N is numbers of methods. The number of methods is calculated as the equation of  $N * (N - 1)$ . The NDC denotes the number of direct connections, which corresponds to the number of edges in the connection graph. A pair of public methods shares an attribute directly if both methods reference the attribute and transitively if one of the two methods does not reference the attribute but directly or transitively calls a method that does. The open-source code of Designite Java application is a command-based line. To run the application using command to debug whole application. After debugging, we need to click the output to review results.

```
private int deepHierarchy = 6;
private int wideHierarchy = 10;

private int brokenModularizationLargeFieldSet = 5;
private int hubLikeModularizationLargeFanIn = 20;
private int hubLikeModularizationLargeFanOut = 20;
private int insufficientModularizationLargePublicInterface = 20;
private int insufficientModularizationLargeNumOfMethods = 30;
private int insufficientModularizationHighComplexity = 100;

// thresholds for god class
private int weight_method_count = 47;
private double tight_class_cohesion=0.3;
private int access_foreign_data = 5;

//set method and get method

public int getWeight_method_count(){
    return weight_method_count;
}

public double getTight_class_cohesion(){
    return tight_class_cohesion;
}
```

Figure 3.4. Threshold God Class

The threshold for God Class defined in threshold class and it call the function getter and setter for weight method count, tight class cohesion and access foreign data.

```

// if(sum >= thresholdsDTO.getWeight_method_count()){

int sum = typeMetrics.sum_extractWeightedMethodsPerClass();

// System.out.println(typeMetrics.getInheritanceDepth());
// System.out.println(calculateTCC());
if(sum >= thresholdsDTO.getWeight_method_count()
    && calculateTCC() >thresholdsDTO.getTight_clas
    && typeMetrics.extractNumOfMethodsMetrics2())thresholdsDTO.getAccess_foreign_data()){
int
Designite.smells.ThresholdsDTO.getAccess_foreign_data()
//System.out.print("God class detect!!");

```

Figure 3.5. Compare Number of God Class

The number of weight method count, tight class cohesion, and access foreign data compare the threshold. The number is exceeding the threshold, it counts as God Class categories.

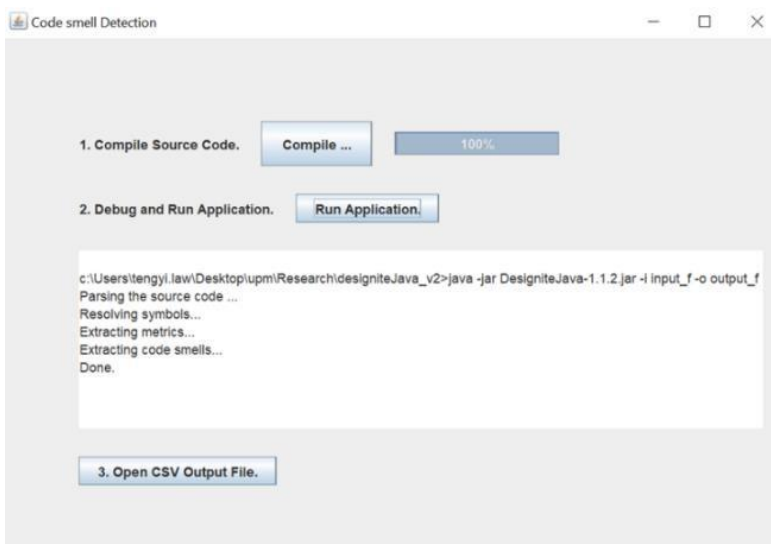


Figure 3.6. Graphic User Interface Code Detection

The graphics user interface shows the steps to click button from steps 1 to 3. First step to compile applications. Compile program to install and build jar application. The progress bar shows the completed of debug applications. Step 2 is run application and show the process for extracting code smells. The done message show completed process after parsing source code, resolving symbols, extracting metrics, and extracting code smells. Last step, we did not need to open output result in folder. The click button of open CSV file will open file to show implementation code smells.

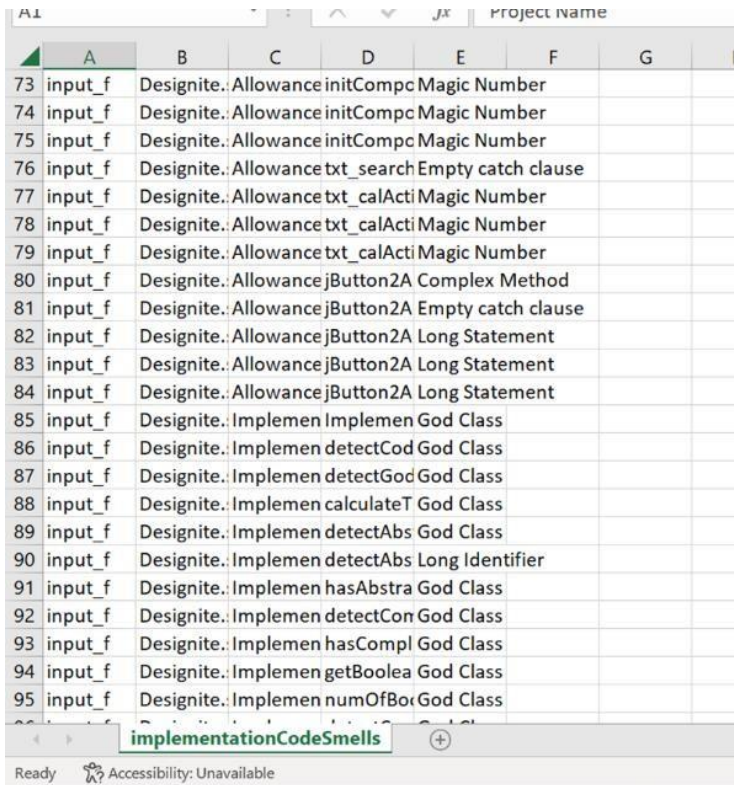
## RESULTS

After running the application, the comments code smell detected and saved the information in text file format.

The detection of comments code smell detects single and multiline of comments in Java file.

The output of detection God Class code smell save in CSV file format. The detection of code smell detects line by line of code which list the method, package, and the class of java.





Line	Severity	Description	Category
73	input_f	Designite..Allowance initCompc Magic Number	Magic Number
74	input_f	Designite..Allowance initCompc Magic Number	Magic Number
75	input_f	Designite..Allowance initCompc Magic Number	Magic Number
76	input_f	Designite..Allowance txt_search Empty catch clause	Empty catch clause
77	input_f	Designite..Allowance txt_calActi Magic Number	Magic Number
78	input_f	Designite..Allowance txt_calActi Magic Number	Magic Number
79	input_f	Designite..Allowance txt_calActi Magic Number	Magic Number
80	input_f	Designite..Allowance jButton2A Complex Method	Complex Method
81	input_f	Designite..Allowance jButton2A Empty catch clause	Empty catch clause
82	input_f	Designite..Allowance jButton2A Long Statement	Long Statement
83	input_f	Designite..Allowance jButton2A Long Statement	Long Statement
84	input_f	Designite..Allowance jButton2A Long Statement	Long Statement
85	input_f	Designite..Implemen Implemen God Class	God Class
86	input_f	Designite..Implemen detectCod God Class	God Class
87	input_f	Designite..Implemen detectGod God Class	God Class
88	input_f	Designite..Implemen calculateT God Class	God Class
89	input_f	Designite..Implemen detectAbs God Class	God Class
90	input_f	Designite..Implemen detectAbs Long Identifier	Long Identifier
91	input_f	Designite..Implemen hasAbstra God Class	God Class
92	input_f	Designite..Implemen detectCon God Class	God Class
93	input_f	Designite..Implemen hasCompl God Class	God Class
94	input_f	Designite..Implemen getBoolea God Class	God Class
95	input_f	Designite..Implemen numOfBo God Class	God Class

Figure 5.2. God Class Code Smell Output

Besides that God class code smell, the other code smell is detected and recorded.

## ANALYSIS

To evaluate the effectiveness of detection of God Class, we use the open-source project version. Below shows the list of open-source project with version and the number of God class detect with existing detection tool.

Each project was assigned two versions: the target version and the previous version (Khalid Alkharabsheh, 2022). The table’s last column displays the number of classes added to or removed from the previous version of the software projects. This number may be used to show code improvement by deleting classes or separating complicated or huge classes (God Class) into two or more classes, hence reducing the overall number of classes.

Project Name	Version	Number of God Class Detected
Angry IP Scanner	3.5	4
Apeiron	2.94	9
Check style	8.0.0	9
Digi Extractor	2.5.2	36
Free mind	1.1.0	62

Table 6.1 Latest Version of Project Source Code.

We use the same version of open-source to test the prototype of code smell detection to detect the number of

God class.

The data has been collected to test the hypothesis as listed in table.

The evaluation is shown in the table below, with the negative numbers representing distinct numbers detected on the God class. The existing detection technique can only detect the class, but prototyping detection can identify the god class. The difference in number is derived by subtracting the prototype detection tools from the existing detection tools. Negative values indicate that the God class has been detected. To calculate the effectiveness, divide the prototype detection number by the existing detection number plus the prototype detection.

As an example,  $(1/5) * 100$  equals 20%. The rest of the class is at 80 percent. 20 percent is the detection of god class, and thus demonstrates the usefulness of scanning the complexity of the Java source code file.

Project Name	Angry IP Scanner	Apeiron	Check style	Digi Extractor	Free mind
Existing Detection Tools	4	9	9	36	62
Prototype Detection Tools	1	2	46	3	58
Different	-3	-7	37	-33	-4

Table 6.3 Different number of God Class Detection

The paired-sample t-test, also known as the dependent sample t-test, is a statistical process used to determine if the mean difference between two sets of data is zero.

**Distribution: T(df:4)**

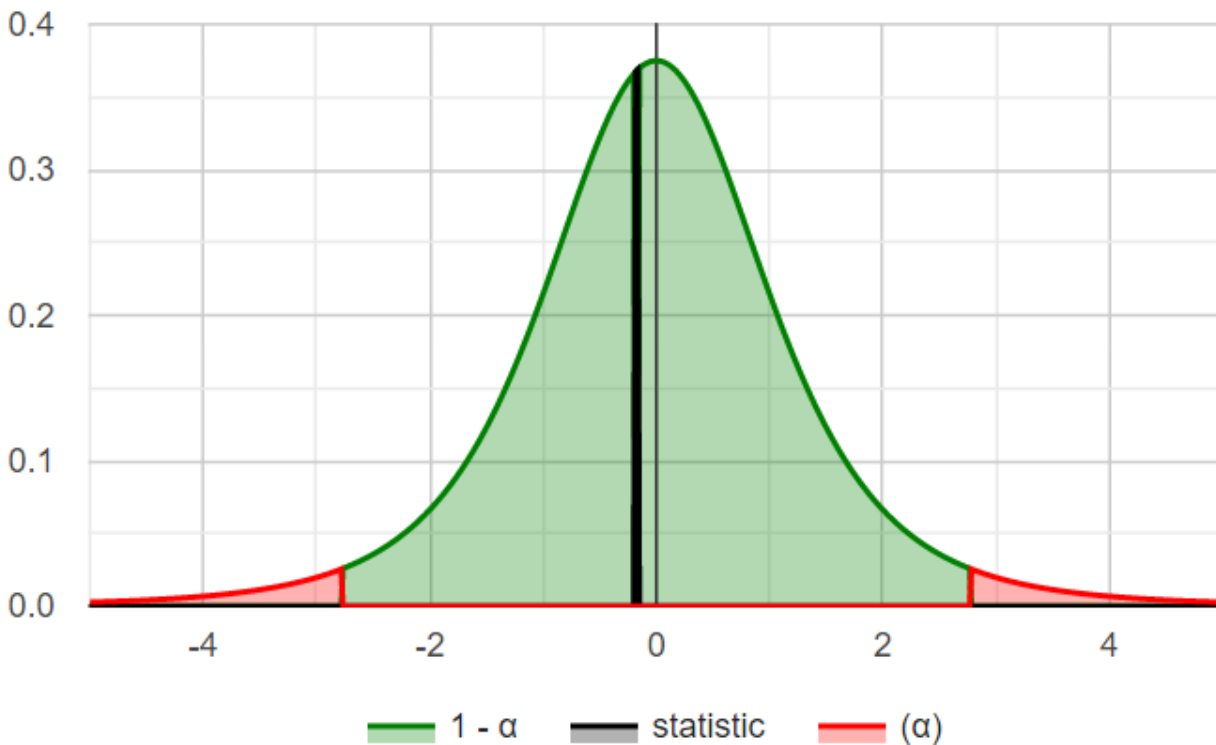


Figure 6.1. Distribution of Two Tails

The p-value equals 0.867,  $(P(x \leq -0.1785) = 0.4335)$ . The larger the p-value the more it supports  $H_0$  and it show the high percentage. Since the p-value  $> \alpha$ ,  $H_0$  cannot be rejected.

The paired sample t-test, also known as the dependent sample t-test, is a statistical process used to determine if the mean difference between two sets of data is zero. The p-value equals 0.867, ( $P(x \leq -0.1785) = 0.4335$ ). The larger the p-value the more it supports  $H_0$  and it show the high percentage. Since the p-value  $> \alpha$ ,  $H_0$  cannot be rejected.

## CONCLUSION AND DISCUSSION

The prototype detection application did not cover all 22 code smells, only 11 of them. The output contains a list of method names, class names, and code smell. The goal is to detect the God class code smell and classify the number of God classes using a machine learning algorithm. Aside from the God class code smell, the comments code smell is included, as is the use of the pattern to discover multiline and single comments. The comments code smell is saved as a text file, while the God class code smell is saved as a csv file.

After running and debugging the application, the information saved in the csv file is overwritten. The prototype application improved command line-based applications by adding a graphical user interface. The limitation is this prototype application only can detect java file or java source code and this application build in java platform. The open-source project code compares existing detection tools and prototype detection tools to the most recent version. Because the runtime is longer and stuck to detect a large number of files and directories, the open-source project code is downloaded from source forge and GitHub repository and input into prototype detection tools one by one.

We comparison with existing approaches to code smell detection for highlighting the strengths and weaknesses.

### Existing Approach:

Strengths:

- Can detect a large number of code smells in a file, providing comprehensive coverage.
- Faster runtime, presumably due to simpler detection methods.

Weaknesses:

- Detection accuracy is lacking, potentially leading to false positives or negatives.
- Only shows the numbers of code smells without providing detailed information about the classes affected.

### Current Approach (With Classifiers Algorithm):

Strengths:

- Can detect code smells at both the directory and file levels, offering a broader perspective on code quality.
- Provides detailed information, including class names and the number of code smells, in a CSV file, aiding in further analysis.

Weaknesses:

- Longer runtime compared to the existing approach, likely due to the file and directory.

The file should include the java source code, which has the extension java file. To build the code, it should look for duplicate classes in another java file. Because it can detect the god class in many files and duplicate the class, hence increasing the number of detections.

The major aim is to develop code smell detection application prototype to evaluate the effectiveness of detection God Class. The hypothesis testing as below:

$$H_0 : U_{Existing} - U_{prototype} = 0$$

$$H_a : U_{Existing} - U_{prototype} \neq 0$$

As the results show the p-value equals 0.867, ( $P(x \leq -0.1785) = 0.4335$ ). The larger the p-value the more it supports null hypothesis and it show the high percentage. Conclusion that the existing and prototype detection tools are equal effectiveness and also show that strong evidence that cannot reject null hypothesis.

## REFERENCES

1. Aliamaan, A.A. (2021). Code smell detection using feature selection and stacking ensemble: An empirical investigation. *Information and Software Technology*.
2. Apostolos Ampatzoglou, S. C. (2013). Research state of the art on GoF design patterns: A mapping study. *The Journal of Systems and Software*.
3. Class, I.o. (2018). Khalid Alkharabsheh, Shahed Almobydeen, Yania Crespo, José A. Taboada. *International Computer Sciences and Informatics Conference*. Amman Arab.
4. Detten, M.M. (2010). Reverse engineering with the reclipse tool suite. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 299-300.
5. Dongjin Yu, P. Z. (2018). Efficiently detecting structural design pattern in stances based on ordered sequences. *Journal of Systems and Software*, 35-56.
6. F.A, F. M. (2016). Comparing and experimenting matching learning techniques forced smell detection. *Empir. Sofw. Eng*, 1143-1191.
7. Fowler, M. (1999). *Refactoring: Improving the Design of existing Code*. Addison-Wesley.
8. Fowler, M. (2019). *Refactoring: Improving the Design of Existing code*. Addison-Wesley.
9. Fowler, M. a. (2000). "Bad Smells in Code", *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. Guggulothu, T. &. (2020). Code smell detection using multi-label classification approach. *Software quality journal*, 1063-1086.
10. Guilherme Lacerda, F.P. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *The Journal of Systems and Software*.
11. Khalid Alkharabsheh, S. A. (2022). Prioritization of god class design smell: A multi-criteria based approach. *Journal of King Saud University-Computer and Information Sciences*, 9332-9342.
12. Lincke, R. (2007). Compendium of Software Quality Standards and Metrics-Version 1.0. *IEEE Std 1061-1992*.
13. M.Fowler,K.B.(1999).*Refactoring: Improving the Design of Existing Code Reading*. MA, USA: Addison Wesley.
14. Mantyla M, V. J. (2004). Bad smells- humans as code critics. *20th IEEE International conference on Software Maintenance*, (pp. 399–408).
15. Martin Fowler, K. B. (2002). *Refactoring: Improving the*. Addison-Wesley Professional.
16. Mika Mäntylä, J. V. (2003). A Taxonomy and an Initial Empirical Study of Bad Smells in Code. *Proceedings of the International Conference on Software Maintenance*, 1063-6773.
17. Moha, N. (2007). Decor: A tool for detection of design defect.
18. Ra'ul Marticorena, C.L. (2005). Parallel Inheritance Hierarchy: Detection from a static view of the system. *6th international workshop on object oriented reengineering*. van Oort, B.C. (2021). The prevalence of code smells in machine learning projects. *IEEE/ACMI st work shop on AI Engineering software engineering for AI*, 1-8