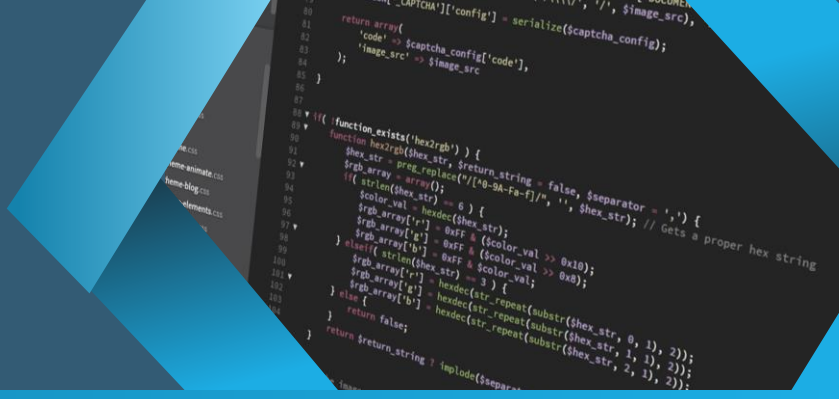
Clean CaDET

Clean Code and Design Educational Tool



REPORT ON THE SELECTED TARGET SMELLS

Executive Summary

Code smells are structures in code that signal the need for refactoring to avoid degrading the software's quality attributes. Our project aims to develop a tool that automatically detects code smells, notifies the developer, and educates them on the possible strategies for resolving the identified issues. There is a large variety of cataloged code smells, and addressing all smell types is a challenging task due to their varied nature. Thus, in this document, we have analyzed the existing empirical studies on code smells to determine each smell type's prevalence and negative impact. This analysis helped us select our first-priority target code smell types: *God Class, Long Method, and Feature Envy*. In this document, we describe the characteristics of each chosen target code smell.

Key Takeaways

1. *Method-level smells should be prioritized, as they can cause infection of an entire class*
2. *Code snippets affected by multiple code smells at once are especially problematic*
3. *Based on their prevalence and harmfulness, our first-priority target smells are God Class, Long Method, and Feature Envy*

Contact
Jelena Slivka
slivkaje@uns.ac.rs
https://clean-cadet.github.io/

1 Introduction

Martin Fowler and Kent Beck [1] defined code smells as structures in code that signal the need for refactoring to avoid degrading the software's quality attributes. As Sharma et al. [2] point out, smells negatively impact both the quality of the software product and the developers' productivity. Our project aims to develop a tool that automatically detects code smells, notifies the developer, and educates them on the possible strategies for resolving the identified issues.

There is a large variety of cataloged code smells. Fowler and Beck [1] introduced 24 code smells that range from straightforward software coding problems, such as the use of repeated *switch statements*, to complicated structural issues, such as the occurrence of *Shotgun Surgery*. Since then, authors have explored smells in different domains and focus areas, and came up with various code smell catalogs, neatly summarized in Sharma et al. [2].

Addressing all smell types is a challenging task due to the varied nature of different smells. While we hope to develop suitable solutions for all significant smells incrementally, our first goal is to prioritize and address the most severe ones. To identify these smells, we explored which types of smells are prevalent and have a high negative impact.

In this document, we first explain our choice of the selected smells in Section 2. Then, in Sections 3-5, we examine the definition of each selected code smell, offer an illustrative example, note its prevalence, and describe its negative impact on the software.

2 Selected target code smells

As recommended by [3], we chose the code smells that satisfy two conditions:

1. The smell has a high prevalence (i.e., developers frequently create this code smell while programming)
2. The smell has a high negative impact (e.g., the code smell impairs developer performance, the source code's maintainability, or the software's reliability).

Palomba et al. [4] conducted an extensive empirical investigation on both code smells prevalence, and the relationship between the presence of code smells and source code change- and fault-proneness. They conclude that the removal of seven highly diffused smells[1] provides a high benefit in terms of change-proneness. Thus, these smells should be removed to improve the overall maintainability of the code. Palomba et al. [4] further conclude that removing other smells seems less relevant from a practical perspective. However, we note that their results suggest that removing *Inappropriate Intimacy* and *Feature Envy* smells has a higher effect on fault-proneness than all other smells.

A recent tertiary systematic review [5] concluded that the code smells that most affect different software quality attributes are *God Class/Large Class, Long Method*, and *Feature Envy*. . *God Class/Large Class* has the most considerable negative effect on quality attributes**.** It affects maintainability, complexity, evolvability, stability, performance, readability, reusability, and changeability. *Long Method* also highly affects complexity, understandability, readability,

[1] Inappropriate Intimacy, Long Method, Spaghetti Code, Speculative Generality, God Class, Complex Class, and Refused Bequest

reusability, maintainability, changeability, and testability. Instances suffering from *God Class/Large Class* and *Feature Envy* smells, besides negatively affecting quality attributes, are more bug-prone than others.

Another recent systematic literature review of papers that analyze the relationship of code smells and fault-proneness [6] has confirmed that code smells are positively correlated with software defects. Thus, code smells are useful predictors for fault detection models. However, not all types of smells are equally useful. The exceptionally effective types of smells are *God Class*, *Long Method*, and *Message Chains*.

Another important finding is that the interaction of code smells can increase the change-proneness [4] and inhibit maintainability [7] of source code. The possible reason for this is that the co-occurrence of smells strongly hinders the developers' ability to understand the source code [8]. Alarmingly, in another large-scale empirical study, Palomba et al. [9] found that the smell co-occurrence is a widespread phenomenon. They identified six pairs of frequently co-occurring smells and found that the co-occurring smells tend to be removed together.

Another important conclusion from the study of Palomba et al. [9] is that the presence of method-level smells "induces" code smells affecting the entire class. Thus, they recommend that more attention should be dedicated to method-level smell detection.

After considering all this evidence (summarized in Table 1), we have decided that in the first phase of the project, we should target the following code smells:

- At the class-level, the *God Class* smell,
- At the method-level, the *Long Method* and the *Feature Envy* smell.

Table 1. Summary of reasons for choosing Long Method, Feature Envy, and God Class as primary targets

Target smell	Reasons for addressing the target smell
Long method	• Highly prevalent [4] • Provides high benefits in terms of improving software quality attributes [4][5] • The classes affected by the Long Method smell are more bug-prone then others [10][11][12] • The occurrence of this smell implies more effort in maintenance task [12] • It should be prioritized as it frequently co-occurs with other smells [9][13]
Feature envy	• Classes affected with this smell are more bug-prone [4][5] • It should be prioritized as it frequently co-occurs with other smells [9][13]
God Class	• Highly prevalent [4] • Has the highest effect on quality attributes [4][5] • Classes affected with this smell are more bug-prone [5][6]

3 Long Method

A method affected by the Long Method code smell is a lengthy method that should be decomposed into multiple shorter methods to improve software readability and reusability [1][14]. These methods are complex. They tend to process a large number of data and implement the main functionality and other auxiliary functions that should be placed in other methods [1]. Such complex methods are hard to understand.

Listing 1 shows an example of the method affected by the Long Method smell. In this example, the method `printOwing()`, which prints the amount of money the customer owes, has three diverse responsibilities marked with different font colors in Listing 1:

(1) it prints the banner,
(2) it calculates the amount of money the customer owes,
(3) it prints the details about the customer and owed money.

These three code segments can be extracted in three different methods. Then, the `printOwing` method would use these methods to implement its functionality (Listing 2).

```
void printOwing(){
  Enumeration e = _orders.elements();
  double outstanding = 0.0;

  //Print banner
  System.out.println("**************");
  System.out.println("******Customer*******");
  System.out.println("**************");

  //Calculate outstanding
  while(e.hasMoreElements()){
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }

  //Print details
  System.out.println("name " + _name);
  System.out.println("amount " + outstanding );
}
```

Listing 1 An example of the method affected by the Long Method code smell. Diverse responsibilities are marked with different font colors

The Long Method code smell is treated using the Extract Method refactoring. Listing 2 shows the previous example after refactoring. For every diverse segment of code, a new method has been created that has a single responsibility. Functionality (1) is now implemented in the `printBanner()` method, (2) is implemented in the `calculateOutstanding()` method, and (3) in the `printDetails()` method. Now, the `printOwing()` method is only concerned with calling other methods in the defined order.

```
void printOwing(){
  this.printBanner();
  double outstanding = this.calculateOutstanding();
  this.printDetails(outstanding);
}

void printBanner(){
  System.out.println("**************");
  System.out.println("******Customer*******");
  System.out.println("**************");
}

double calculateOutstaning(){
  Enumeration e = _orders.elements();
    double outstanding = 0.0;
  while(e.hasMoreElements()){
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }
  return outstanding;
}

void printDetails(double outstanding){
  System.out.println("name " + _name);
  System.out.println("amount " + outstanding );
}
```

Listing 2 Fixing the Long Method code smell shown in Listing 1 by applying the Extract Method refactoring

4 Feature envy

A method suffering from Feature Envy code smell is "a function in one module that spends more time communicating with functions or data inside another module than it does within its own module" [1][4][16]. Feature Envy affected methods tend to call methods and use many attributes of other classes[2].

Feature Envy smell is characterized by a high degree of coupling[3] between the affected method and envied classes. Methods affected with this smell typically change when the envied classes change, instead of changing when the class they belong to changes [17].

Listing 3 shows an example of the Feature Envy code smell. In the example, the `getMobilePhoneNumber()` method class (belonging to the `Customer` class) uses methods from the (envied) `Phone` class and does not use any method from the `Customer` class.

[2] considering also attributes accessed through accessor methods (i.e., getters and setters)
[3] coupling refers to the interdependencies between modules.

```
public class Phone {
    private final String unformattedNumber;
    public Phone(String unformattedNumber) {
        this.unformattedNumber = unformattedNumber;
    }
    public String getAreaCode() {
        return unformattedNumber.substring(0,3);
    }
    public String getPrefix() {
        return unformattedNumber.substring(3,6);
    }
    public String getNumber() {
        return unformattedNumber.substring(6,10);
    }
}
public class Customer {
    private Phone mobilePhone;
    public String getMobilePhoneNumber() {
        return "(" +  mobilePhone.getAreaCode() + ") " +
            mobilePhone.getPrefix() + "-" + mobilePhone.getNumber();
    }
}
```

Listing 3 An example of the method affected by the Feature Envy code smell

Feature Envy code smell is solved using the *move method* refactoring technique: the method is moved from its original class to the class whose attributes or methods it uses the most. Listing 4 shows an example of the move method refactoring that fixes the Feature Envy smell shown in Listing 3. In the refactoring example, code located within the `getMobilePhoneNumber()` method was moved to the `Phone` class within the newly created `toFormattedString()` method. That way, instead of calling three methods from the external `Phone` class, `getMobilePhoneNumber()` calls only one which provides better encapsulation.

```
public class Phone {
    private final String unformattedNumber;
    public Phone(String unformattedNumber) {
        this.unformattedNumber = unformattedNumber;
    }
    private String getAreaCode() {
        return unformattedNumber.substring(0,3);
    }
    private String getPrefix() {
        return unformattedNumber.substring(3,6);
    }
    private String getNumber() {
        return unformattedNumber.substring(6,10);
    }
    public String toFormattedString() {
        return "(" + getAreaCode() + ") " + getPrefix() + "-" + getNumber();
    }
}
public class Customer {
    private Phone mobilePhone;
    public String getMobilePhoneNumber() {
        return mobilePhone.toFormattedString();
    }
}
```

Listing 4 Fixing the Feature Envy code smell shown in Listing 3 by applying the Move Method refactoring

5 God Class

A significant principle of object-oriented programming is that each component in the system should have one purpose and one responsibility. The disregard of this principle leads to the emergence of components with multiple responsibilities. The God Class or Large Class code smell represents a class that implements many responsibilities and contains a large part of the system's logic. Such classes are usually characterized by many attributes and methods that are not interconnected, which means that the class has low cohesion[4]. Also, God Class is a class that is associated with many other components in the system, which indicates high coupling. Authors of papers [18][19][4] provide similar definitions of the God Class code smell.

An example of God Class is class *org.apache.tools.ant.Project* from the *Apache Ant* project (release 1.8.3)[5]. The comments above the class declaration show that the class implements several functionalities. Considering the value of the WMC [6] metric of 224 (given by the RepositoryMiner[7]) and the fact that only 4 of the 832 other classes have a higher WMC value, it can be concluded that this class is challenging to develop and maintain. This class has 105 methods, which is the largest number of class methods in this project. Additionally, this class has 2476 lines of code, and only three other classes in the system have more lines than this class.

There are a few ways for refactoring and removing God Class code smell [1]. The class affected by God Class code smell implements various responsibilities that can be identified and extracted in a new class or a subclass. The idea is to group coherent attributes and methods, but the exact way of doing it depends on the refactored code. Data such as attributes and parameters that are usually used together represent suitable candidates for extraction in a new class. Similarly, the methods that use only a small subset of attributes of the God Class should be moved to a suitable class. Besides the class elements, developers should consider which parts of the system use the class, i.e., whether different clients use different sets of methods. The sets of methods used by different clients can be grouped and extracted into separate interfaces.

6 References

[1] Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.

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

[3] 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.

[4] 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.

[5] Lacerda, G., Petrillo, F., Pimenta, M. and Guéhéneuc, Y.G., 2020. Code smells and refactoring: a tertiary systematic review of challenges and observations. Journal of Systems and Software, p.110610.

[4] Cohesion describes the extent to which the functions within a module are related.
[5] The class was annotated as God Class in the dataset provided by Palomba et al. [4]. It can be found at https://github.com/apache/ant/blob/rel/1.8.3/src/main/org/apache/tools/ant/Project.java
[6] Weighted Method Count (WMC) is a sum of the class methods complexities, and it serves as a good indicator of how much time and effort it will take to develop and maintain a given class [20].
[7] RepositoryMiner is a project/API that allows us to extract the history of source code metrics. Available at https://github.com/antoineBarbez/RepositoryMiner/

[6] Piotrowski, P. and Madeyski, L., 2020. Software defect prediction using bad code smells: A systematic literature review. In Data-Centric Business and Applications (pp. 77-99). Springer, Cham.

[7] A. Yamashita, L. Moonen, Exploring the impact of inter-smell relations on software maintainability: an empirical study, International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 682–691.

[8] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, CSMR '11, IEEE Computer Society (2011), pp. 181-190

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

[10] Li W and Shatnawi R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of Systems and Software 2007; 80(7): 1120–1128.

[11] Shatnawi R and Wei L. An investigation of Bad Smells in object-oriented design. Third International Conference on Information Technology: New Generations (ITNG 2006), 2006; 161–165.

[12] Cairo, Aloisio S., Glauco Carneiro, A. M. P. Resende, and F. Brito e Abreu. "The influence of god class and long method in the occurrence of bugs in two open source software projects: an exploratory study." *The influence of god class and long method in the occurrence of bugs in two open source software projects: an exploratory study* (2019): 199-204.

[13] Palomba, F., Oliveto, R. and De Lucia, A., 2017, February. Investigating code smell co-occurrences using association rule learning: A replicated study. In 2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE) (pp. 8-13). IEEE.

[14] Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y., & Zhang, L. (2020). Deep Learning Based Code Smell Detection. IEEE Transactions on Software Engineering, 1–1.

[15] Martin, R.C., 2009. Clean code: a handbook of agile software craftsmanship. Pearson Education.

[16] Dietz, L.W., Manner, J., Harrer, S. and Lenhard, J., 2018. Teaching clean code. In Proceedings of the 1st Workshop on Innovative Software Engineering Education.

[17] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. Mining version histories for detecting code smells. Software Engineering, IEEE Transactions on, vol. 41, no. 5, pp. 462–489, May 2015.

[18] Pantiuchina, J., Bavota, G., Tufano, M. and Poshyvanyk, D., 2018. Towards Just-In-Time Refactoring Recommenders. ACM/IEEE 26th International Conference on Program Comprehension.

[19] Barbez, A., Khomh, F. and Guéhéneuc, Y.G., 2019. Deep Learning Anti-patterns from Code Metrics History. IEEE International Conference on Software Maintenance and Evolution (ICSME) pp. 114-124.

[20] Chidamber, S. R., Kemerer, C. F., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, June 1994.

The Project is funded by the Science Fund of the Republic of Serbia, Grant No. 6521051, AI - Clean CaDET

Science Fund of the Republic of Serbia