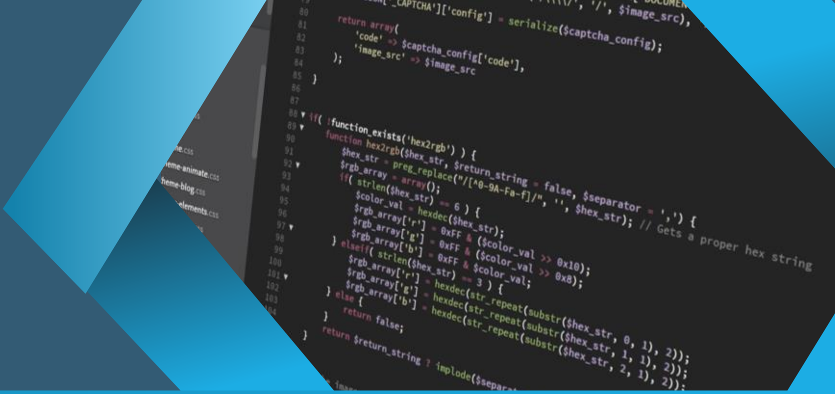
Clean CaDET

Clean Code and Design Educational Tool



REPORT ON THE SELECTED TARGET SMELLS - Y2

Executive Summary

After an initial analysis of existing empirical studies on code smells, we selected the first-priority target smells (God Class, Long Method, and Feature Envy) based on their prevalence and negative impact. Due to the lack of datasets for Feature Envy, we focused on the remaining two smells in the project's first year. Due to the high negative impact of the Feature Envy code smell, we will focus on labeling the novel Feature Envy code smell in the second year of the project. This dataset will allow us to develop automatic Feature Envy detection strategies. Towards the end of the first year of work on the project, we analyzed 12 papers regarding the impact of code smells. Based on the collected data and analysis of prevalence and impact (fault-proneness and change-proneness), we selected two additional target code smells: Data Class and Refused (Parent) Bequest. In this document, we describe the characteristics of each chosen target code smell.

Key Takeaways

1. *A well-annotated dataset is necessary for the detection of code smells. After annotating a quality dataset, we can refocus on detecting Feature Envy code smell.*
2. *Based on prevalence and harmfulness (fault- and change-proneness), our new target smells are Feature Envy, Data Class and Refused Bequest.*
3. *Feature Envy and Data Class code smells are strongly dependent.*

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

1. Selected target code smells

Table 1 shows the papers we reviewed to analyze the prevalence and impact of code smells on changes and faults in a software project. We have selected articles that represent empirical studies of how developers perceive code smells.

Table 1. Papers used for analysis of the prevalence and impact of code smells

Year	Authors	Paper
2007	Li et al.	An empirical study of the bad smells and class error probability in the postrelease object-oriented system evolution. [1]
2008	Lozano et al.	Assessing the effect of clones on changeability. [2]
2010	Rahman et al.	Clones: What is that smell? [3]
2012	Romano et al.	Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes. [4]
2012	Khomh et al.	An exploratory study of the impact of antipatterns on class change- and fault-proneness. [5]
2014	Ban et al.	Recognizing Antipatterns and Analyzing their Effects on Software Maintainability. [6]
2014	Hall et al.	Some Code Smells Have a Significant but Small Effect on Faults. [7]
2016	Soh et al.	Do Code Smells Impact the Effort of Different Maintenance Programming Activities? [8]
2017	Elish	On the Association between Code Cloning and Fault-Proneness: An Empirical Investigation. [9]
2017	Islam et al.	A Comparative Study of Software Bugs in Clone and Non-Clone Code. [10]
2018	Palomba et al.	On the diffuseness and the impact on maintainabilityof code smells: a large scale empirical investigation. [11]
2018	Cairo et al.	The Impact of Code Smells on Software Bugs: A Systematic Literature Review. [12]

The book [13] shows the dependence of several different code smells. Figure 1 shows that the code smells Data Class and Feature Envy are correlated: Methods that suffer from Feature Envy code smell often use classes that suffer from Data Class code smell. In this case, the Data Class provides data to the method affected by Feature Envy, and the method envies the class. When we find the Feature Envy method, it is rather probable that we will also find Data Classes among the classes from which that method accesses data. We can conclude that it is good to analyze, detect, and even refactor these two code smells together. Refactoring a case of Feature Envy could lead to a positive effect towards repairing an envied Data Class.

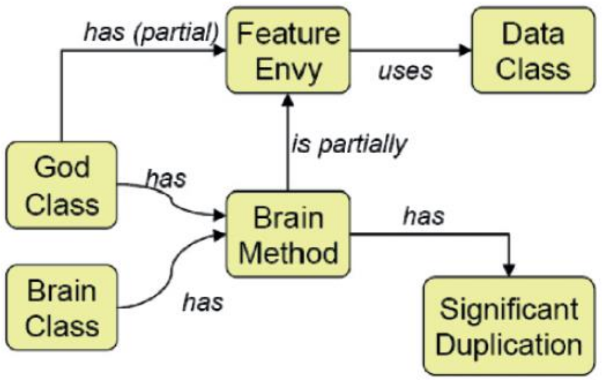


Figure 1 Correlation between code smells

After considering all evidence on the prevalence and harmfulness of the code smells (summarized in Table 2), we have decided that in the second phase of the project, we should target the following code smells:

- At the class-level, the *Data Class* and *Refused (Parent) Bequest* smell,
- At the method-level, the *Feature Envy* smell.

Table 2. Summary of reasons for choosing Data Class and Refused Bequest as new target smells

Target smell	Reasons for addressing the target smell
Data Class	• Reduces the maintainability, testability and understandability of a system [13][5]. • Breaks the principles of encapsulation which is paramount to obtain object-oriented design [13]. • Frequently co-occurs with other code smells (Feature Envy, Large Class, Duplicate Code and Long Method) [13][14].
Refused Bequest	• Fairly diffused code smell [11]. • Classes affected with this smell are more change-prone [11]. • Introduces duplication and in general class interfaces that become incoherent and non-cohesive [13].

We described the Feature Envy code smell in the previous report[1]. The following chapters will explain the new target code smells in more detail, i.e., their definitions, examples, and refactoring methods.

2. Data Class

Data Classes represent data holders without complex functionality. The lack of functionally relevant methods may indicate that related data and behavior are not kept in one place, which is a sign of a non-object-oriented design. Data Classes are the manifestation of a lacking encapsulation of data and poor data-functionality proximity. Other classes rely on Data Classes because they see and possibly manipulate their data [13].

[1] https://github.com/Clean-CaDET/Deliverables/blob/main/work_package_1/D1.1.%20Report%20on%20the%20selected%20target%20smells.pdf

Listing 1 shows an example of the class affected by the Data Class smell. In this example, the class `Phone` does not contain any methods that implement logic. It only has fields, accessor and mutator methods (marked with different font colors in Listing 1):

(1) Fields.
(2) Accessor methods (getters) - If these methods are called frequently from other classes, it is a signal that we could transfer the corresponding data to those classes.
(3) Mutator methods (setters) - Frequently called from other classes, they signal that class is being manipulated in far too much detail by other classes.

```
public class Phone(){
  private String number;
  private String areaCode;
  private String prefix;

  public Phone(){}
  public String getNumber(){
    return this.number;
  }
  public String getAreaCode(){
    return this.areaCode;
  }
  public String getPrefix(){
    return this.prefix;
  }
  public void setNumber(String number){
   this.number = number;
  }
  public void setAreaCode(String areaCode){
   this.areaCode = areaCode;
  }
  public void setPrefix(String prefix){
   this.prefix = prefix;
  }
 }
```

Listing 1 An example of the class affected by the Data Class code smell. Besides fields, accessor and mutator methods (marked with different font colors), there are no other methods in the class that contain logic.

The basic idea of refactoring a Data Class proposed in [13] is to put together the data and operations on that data and provide services to the former clients of the public data instead of allowing them direct access to this data. There are several ways to accomplish this:

(1) Identify pieces of functionality that could be extracted and moved as services to the Data Class.
(2) If the Data Class has no functionality and has only one or few clients, we can move the data in classes that use the data. Then, we can remove the Data Class entirely from the system.
(3) If the Data Class is a rather large class with some functionality and many exposed attributes, maybe only a part of the class needs refactoring. Some parts of the class could be extracted to a separate class, while pieces of functionality could be extracted from the data clients as services provided by the new class.

3. Refused (Parent) Bequest

The relation between a parent class and its children is intended to be more special than the collaboration between two unrelated classes. This unique collaboration is based on a category of members (methods and data) especially designed by the base class to be used by its descendants. If a child class refuses to use this special bequest prepared by its parent, it is a sign that something is wrong within that classification relation [13].

The Refused Bequest affects the subclass when it inherits functionalities that are never used [15]. A class is considered to be affected by Refused Bequest when it redefines most of its inherited methods, signaling the wrong hierarchy [16].

Listing 2 shows an example of the class `Child` affected by the Refused Bequest smell. In this example, the class `Child` overrides `method1` (marked with purple) and does not use inherited methods `method2` and `method3` (marked with green).

```
class Parent {

  public void method1(){ // does something }
  public void method2(){ // does something }
  public void method3(){ // does something }

 }

class Child extends Parent {

  @Override
  public void method1(){ // does something else }

 }
```

Listing 2 An example of the class Child affected by the Refused Bequest code smell.

Figure 2 represents the detailed process of inspection and refactoring of a Refused Bequest code smell given in [13]. The figure has three areas (labeled A, B, and C) corresponding to the three identified causes for a Refused Bequest (some of the causes might co-exist):

(A) False Child Class - The child class does not belong in the hierarchy.

(B) Irrelevant Bequest - The space of inheritance-specific members is over-populated with methods and attributes that have no relevance in the context of the inheritance relation.

(C) Discriminatory Bequest - When the parent class has many child classes, and the bequest offered by it is relevant only for some of these siblings, but not for the class affected by Refused Bequest.

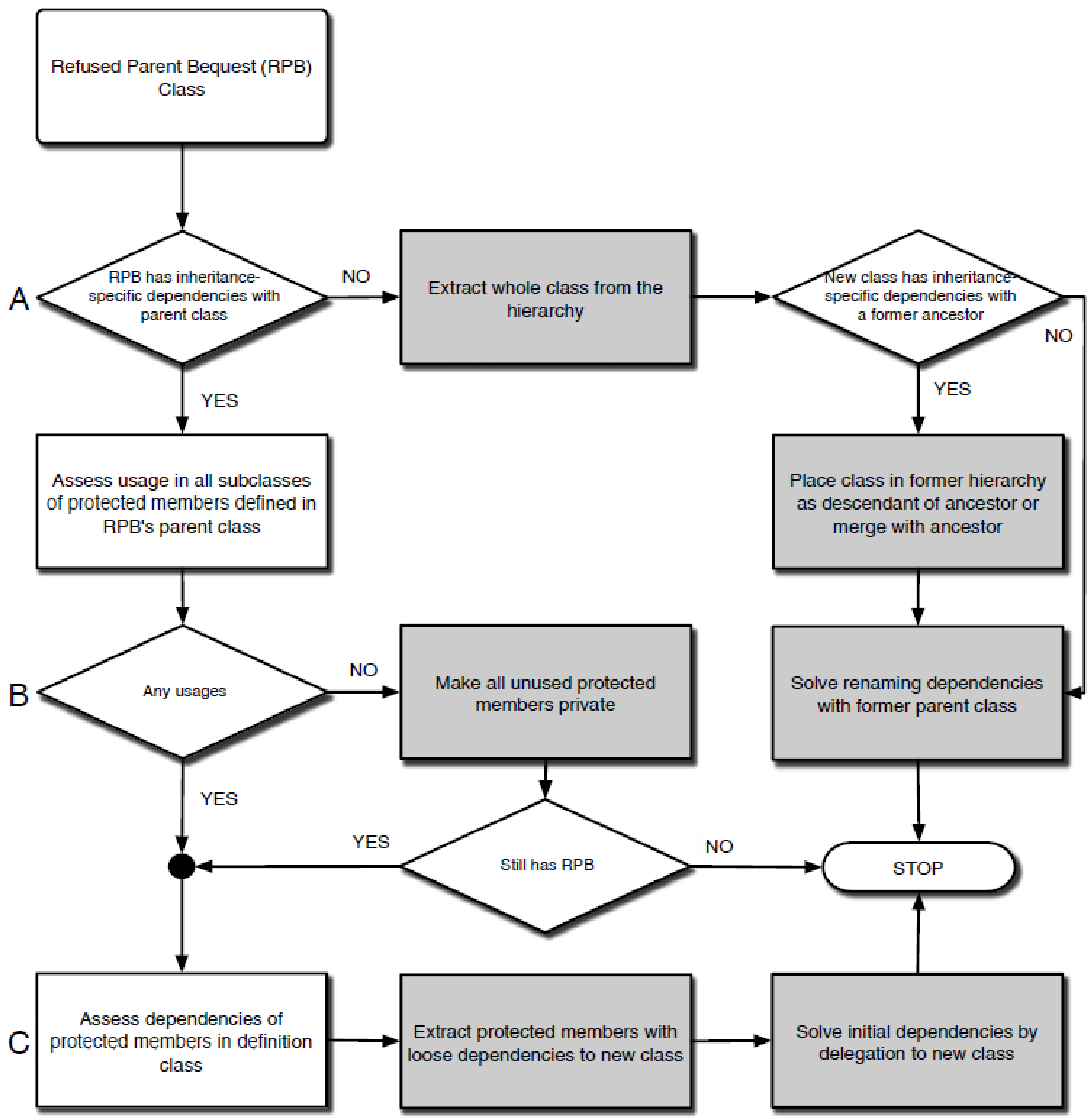

Refused Parent Bequest (RPB) Class
A
RPB has inheritance-specific dependencies with parent class
NO
Extract whole class from the hierarchy
New class has inheritance-specific dependencies with a former ancestor
NO
YES
YES
Assess usage in all subclasses of protected members defined in RPB's parent class
Place class in former hierarchy as descendant of ancestor or merge with ancestor
B
Any usages
NO
Make all unused protected members private
Solve renaming dependencies with former parent class
YES
YES
Still has RPB
NO
STOP
C
Assess dependencies of protected members in definition class
Extract protected members with loose dependencies to new class
Solve initial dependencies by delegation to new class

Figure 2 Inspection and refactoring process of the Refused Bequest code smell

4. References

[1] 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.

[2] Lozano, A., Wermelinger, M. 2008. Assessing the effect of clones on changeability. 2008 IEEE International Conference on Software Maintenance

[3] Rahman, F., Bird, C., Devanbu, P. 2010. Clones: What is that smell? 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)

[4] Romano, D., Raila, P., Pinzger, M., Khomh, F. 2012. Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes. 2012 19th Working Conference on Reverse Engineering.

[5] Khomh, F., Di Penta, M., Gueheneuc, Y., Antoniol, G. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empir Software Eng **17,** 243–275 (2012).

[6] Ban, D., Ferenc, R. 2014. Recognizing Antipatterns and Analyzing Their Effects on Software Maintainability. ICCSA 2014: Computational Science and Its Applications – ICCSA 2014 pp 337-352

[7] Hall, T., Zhang, M., Bowes, D., Sun, Y. 2014. Some Code Smells Have a Significant but Small Effect on Faults. ACM Transactions on Software Engineering and Methodology 23(4):1-39

[8] Soh, Z., Yamashita A., Khomh, F., Gueheneuc, Y. 2016. Do Code Smells Impact the Effort of Different Maintenance Programming Activities? 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)

[9] Elish, M. 2017. On the association between code cloning and fault-proneness: An empirical investigation. 2017 Computing Conference

[10] Islam, J., Mondal, M., Roy, C., Schneider, K. 2017. A Comparative Study of Software Bugs in Clone and Non-Clone Code. The 29th International Conference on Software Engineering and Knowledge Engineering

[11] 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.

[12] Cairo, A., Carneiro, G., Monteiro, M. 2018. The Impact of Code Smells on Software Bugs: A Systematic Literature Review.

[13] Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems.

[14] Sobrinho, E.., De Lucia, A., de Almeida Maia, M. 2021. systematic literature review on bad smells – 5 W's: which, when, what, who, where. IEEE Transactions on Software Engineering (Volume: 47, Issue: 1, Jan. 1 2021)

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

[16] Pecorelli, F., Di Nucci, D., De Roover, C., De Lucia, A. 2020. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. Journal of Systems and Software, Volume 169.November 2020.

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