**Report of Code Smells and Refactoring**

**Code smells**

A code smell indicates that there is a deeper problem in the source code which doesn’t change the functionality of the program’s output, but it is important for us to identify them and know why they are there. Code smells are helpful in improving the code structure and code quality.

**Common code smells:**

1. Duplicate code
2. Long method
3. God class
4. Type checking
5. Feature envy

**PDFsam**

In this report, we are going to discuss about three code smells in PDFsam project namely Long method, God class and Type checking.

**1. Long Method Code smell:**

If a method/function has too many lines of code or the procedure gets grown up, long method creates a separate method/function to reduce the code and bug proneness. Generally, a method keeps getting updated with additional set of lines and attributes. We tend to focus on writing it but not reading it. So, what Long Method does is it observes the classes/methods/attributes and wherever there is a chance to create a new method, it creates a new method. The longer a method, the more complex it becomes.

**Long method code smell component observation:**

Project used: Rotate

Class: RotateParametersBuliderTest

Function: buildRanges()

1. Firstly, we had checked Long method code smell for class RotateParametersBuliderTest and found this smell in function buildRanges(). In this function, there are set of lines that are highlighted because these lines do not involve in the functional behavior of the ongoing method and could be separated from the method. Our code smell is successful in finding the extra lines of code that can be separated in order to remove this smell.
2. In buildRanges() function, the smelled two lines of code doesn’t perform anything related to the methods functionality and even if these lines are separated into a new method, the functional behavior of the code doesn’t change. So, long method code smell identified these lines of code as smelly.
3. Yes, the detected smell is a code smell because the method is long, and it is associated with high bug proneness. We had checked the method by separating it and we ran the code and we are successful in removing it. The code smell is also removed after re-checking the execution of source code.

A screenshot of a social media post

Description automatically generated

Fig 1: Long method code smell for Rotate project

**2. God class code smell:**

A class may contain large number of variables, methods and lines of code. Classes will get bloated with more data and the program grows huge in the same class. When a class consists of too many functions in it, then it is better to split some data into sub classes that do not change the functionality of the class. A class can be split into separate components, sub classes, domains. This way remembering many attributes will be less. It also avoids duplication of code which is a big threat to the code.

**Observation:**

Project used: Rotate

Class: RotateSelectionPaneTest

1. In RotateSelectionPaneTest class, we found God class bad smell where it has several methods, features and cases involved in a single class. In this case, the main class has methods required for the implementation of a particular use case. The class is responsible for the correct functionality of the use cases and it would be wrong to separate them to a whole new class. So, it is easy to create a sub class and we can do it through refactoring method or manually. The methods this class has are considered as a special case and are to be separated as a sub class.
2. This class is considered as smelly because it has certain features which are used only in special cases. So, we must create a sub class and use them in these cases. Also, it has long methods, so the part of the behavior of the large class can be considered as a sperate component and extract to a new class.
3. Yes, the detected smell is considered as a God class because the class gets bloated with so much data and the program get expanded and hence it becomes a problem for the new programmer to identify the functionality with ease. Any change to a method or a feature results in changing the entire classes code method. So, we need to split the data into a separate component without changing the functional behavior of the class. This class has long method and in these rare cases, we tend to separate it into a new sub class.

A screenshot of a computer

Description automatically generated

Fig 2: God class code smell for Rotate project

**3. Type checking Code smell:**

Type checking code smells arises when the code has switch-case statements or if-else statements. In switch-case statements, similar switch statements are created throughout the program and if you add a clause or remove a clause from a switch, we need to find and repair other switch-cases too throughout the source code. This will happen by using Type checking code smell. This smell is usually caused to create new classes or use polymorphism. In order to remove the switch-cases we need to extract method and then move the method.

**Observation:**

Project used: pdfsam-fx

Class: RememberingLatestFileChooserWrapper

1. In RememberingLatestFileChooserWrapper class, we found out the smell is present because of the switch-cases in it. Switch cases are created in various parts of the code and it makes the new programmers to get confused. Similarly, the highlighted lines of code produced by the type checking code smell needs to be extracted and moved into a better process.
2. The class is considered as smelly because it has complex switch-cases which are to be modified to a better version to make the code understandable for the programmer or else it could create complex problems in the future. So, we must get rid of these switch-cases and are to be extracted to a different method.
3. Yes, the detected smell is considered as Type checking because the class is getting boated with switch-cases and thereby making the source code complex. Basically, type checking code smell arises when the code contains switch-cases and if-else statements. In order to fix this, we can replace the type code with subclasses or can replace the conditional with polymorphism. Sometimes polymorphism alone can’t fix the code smell instead, we can extract the method into smaller methods.

A screenshot of a computer screen

Description automatically generated

Fig 3: Type checking code smell for pdfsam-fx project

**Code Refactoring**

Code refactoring is a process of reconstruction an existing source code without changing its behavior. Generally, we perform refactoring to improve the codes design, structure and functionality. It makes the software easy to understand and helps in finding software bugs. In this report, we are going to discuss refactoring of two code smells that we had discussed above namely Long method and God class.

**1. Refactoring of Long Method:**

Class: RotateParametersBuilderTest

* Basically, what long method code smell means, if a class has long method with extra lines of codes or functions, there is a chance of increasing bug proneness, so it extracts a new method and moves the method.
* In the class RotateParametersBuliderTest, we found Long method bad smell which means the function buildRanges() has more than required information in a single method.
* Our goal is to alter the changes without changing the functionality of the class.
* After observing the code smell lines, there are two lines which doesn’t perform anything related to the overall functional behavior.
* So, the next step is refactoring the code. At the time of refactoring, we created a new set of method (init\_range\_output()) in the buildRanges() function and separated it from the current method and extracted to a new method.
* This resulted in the reduction of code from the previous method and the separated method has a new behavior. This is the actual meaning of Long method.
* After refactoring, we checked to see the code smell is removed or not and we are successful in removing the smell.
* We ran the entire code again and we got not break-down or crashes in the functionality of the project.
* The program ran efficiently without any bugs and without code smell.
* The longer the method, the more complex it gets and could produce more error proneness.
* The changes in the code after refactoring is shown below in fig 4.

A screenshot of a computer

Description automatically generated

Fig 4: Class RotateParametersBuliderTest after refactoring

**2. Refactoring of God class:**

Class: RotateSelectionPaneTest

Newly created Class: RotateSelectionPaneTestCore

* If a class consists of too many functions in it, they can be extracted into a sub class in order to decrease the complexity without changing the functional behavior of the source code.
* In the class RotateSelectionPaneTest, God class code smell detected several lines of code with a greater number of functions, methods and attributes.
* Our job is to refactor the smell and extract a new class and commit the changes without changing the functionality of it and test it.
* After refactoring, we created a whole new class and named it as RotateSelectionPaneTestCore and applied the changes.
* Functions which are present in the old class (RotateSelectionPaneTest) moved to the new class.
* We renamed it as RotateSelectionPaneTestCore.
* In the next step we checked if the code smell is removed or not and we are successful in removing it.
* After that, we checked the rotate project and try building it.
* Build is successful and there are no new errors and the new class is created successfully.
* There is no need to create a new test case for the newly created class because it doesn’t directly change the functionality of the RotateSelectionPane class and since the new class is a part of the old test class.
* Then we tried running the new class and no functionality has been changed thereby extracting only the bloated functions to the new class.
* No new errors are detected after running the entire project.
* The successful execution of the new changes is shown in the below fig 7.
* The screenshots of the refactored classes are shown in the below figures 5 and 6.

A screenshot of a computer

Description automatically generated

Fig 5: RotateSelectionPaneTest class after extracting its functions

A screenshot of a computer

Description automatically generated

Fig 6: Newly created RotateSelectionPaneTestCore class

A screenshot of a computer

Description automatically generated

Fig 7: Console of Build success after refactoring two bad smells

**jEdit**

In this report, we are going to discuss about three code smells in jEdit project namely God class, Type checking and Feature Envy.

**God Class:**

A God Class is a class that has grown to be excessively complex, which gathers too much non cohesive functionality and heavily manipulates data members from other classes. It means it is a huge class in terms of number of lines. It creates tight coupling and increases challenges in code maintainability. The SOLID principles of object-oriented programming include Single Responsibility principle, Open/Closed principle, Liskov Substitution principle, Interface Segregation principle, Dependency Inversion principle. The god class violates the Single Responsibility principle and the Interface Segregation principle as their main objective is to separate a large solution into smaller pieces.

**Observation:**

Package: org.gjt.sp.jedit.textarea

Class: TextArea

Function: addLeftOfScrollBar(), removeLeftOfScrollBar()

We checked for the god class code smell using JDeodorant and found that it exists in three of the methods of this class. We selected the addLeftOfScrollBar() method and removeLeftOfScrollBar() method to analyze this code smell for the purpose of this assignment.

1. TextArea class contains several methods, features and cases which should ideally be split into several different classes. Using JDeodorant we found that there were 3 parts of code where the God Class smell was detected. First in the hideScrollBar() method, second in the addLeftOfScrollBar() and removeLeftOfScrollBar() methods and the third in the initInputHandler() method. The TextArea class has 6735 lines of code. Lines of code is one of the aspects of detecting the God Class code smell.
2. From the methods addLeftOfScrollBar() and removeLeftOfScrollBar(), we can notice that they perform operations relating to the Left Scroll Bar of the TextArea. These methods could be refactored into a new class as they have functionality relating to one component of TextArea in order to avoid the god class code smell.
3. Yes, the detected smell is considered as God Class as this class TextArea is bloated with several different methods and features which should ideally split into different individual classes.

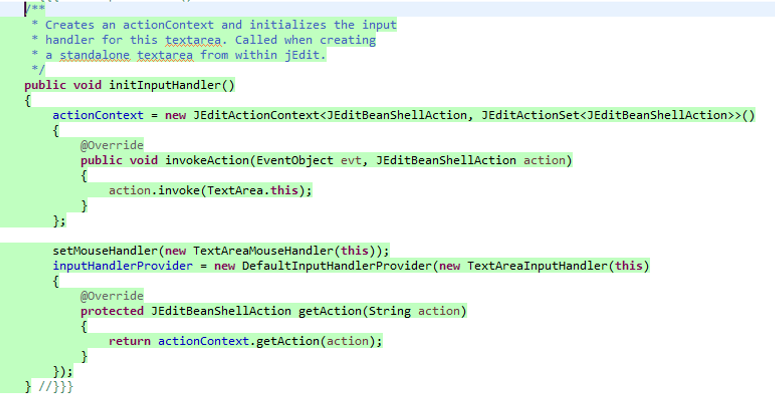


Figure 1. God Class TextArea

**2. Type checking code smell:**

A type checking code smell occurs when a class contains complicated conditional statements which makes the code difficult to understand and maintain.

**Observation:**

Package: org.gjt.sp.jedit.gui

Class: VariableGridLayout

1. In the VariableGridLayout class, we found the smell is present in the following lines of code. The conditional statement accesses the variable FIXED\_NUM\_ROWS. Therefore, the highlighted lines of code produced by the type checking smell which is needed to be modified into a better way of execution.
2. The class is considered as smelly because in the conditional if statement as highlighted is flagged as complicated. In this case, the conditional if/else statement accesses FIXED\_NUM\_ROWS (no. of rows and columns) despite creating different functions of rows and columns.
3. Yes, the detected smell is considered as Type Checking code smell as it contains a conditional statement which is complicated the fields like strInsert, strRemove, offset and thereby making the source code complex.

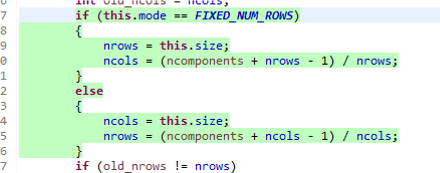


Figure 2. Type Checking for UndoManager class

**3. Feature envy Code smell:**

It’s a code smell when an object accesses fields of another object to execute some operation, instead of just telling the object what to do.

**Observation:**

Project used: jedit

Class: CompressedReplaced

1. In CompressedReplaced class, we found out the smell is present in the add method. Here in this add method, it is accessing fields of Replace class. Therefore, the highlighted lines of code produced by the feature envy code smell need to be modified into a better way of execution.
2. The class is considered as smelly because in this add method, an object is accessing the fields of another object (rep.strInsert, rep.strRemove, rep.offset) to execute some operation.
3. Yes, the detected smell is considered as Feature envy as its access the fields like strInsert, strRemove, offset and thereby making the source code complex.

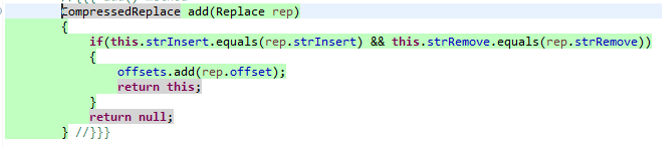


Figure 3. Feature Envy for CompressedReplace class

**Code Refactoring**

**Code Refactoring**

Code refactoring is a process of rebuilding an existing source code without changing its behavior. Refactoring is performed to improve the codes design, structure and functionality. In this report, we are going to discuss refactoring of two code smells namely type checking and feature envy.

**Refactoring of type checking:**

Class: VariableGridLayout

* Type checking code smell means, how complicated conditional statements are and that would make code difficult to understand and maintain.
* toString() method manifests FIXED\_NUM\_ROWS in a single method instead of creating different functions of rows and columns.
* Our goal is to alter the changes without changing the functionality of the class.
* So, the next step is refactoring the code. The if/else statements were replaced by switch statement in the mentioned class and refactored code by creating new classes i.e. Mode.java, FixedNumRows.java that extends class Mode.java. These later two classes implement getMode(), update() of Mode.java class.The existing VariableGridLayout class uses getter and setter methods called the Mode.java class and to return the attributes.
* This resulted in the less complex version of code from the previous method.
* We did not perform any code manually as the tool performed all required changes.
* After refactoring, we checked to see if the code smell is removed or not and we are successful in removing the smell.
* We ran the entire code again and we did not break-down or crash in the functionality of the jedit project.

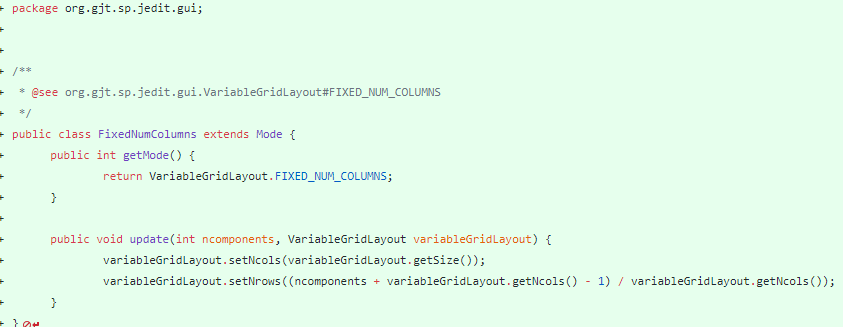


Figure 4. Code Refactoring of FixedNumColumns class

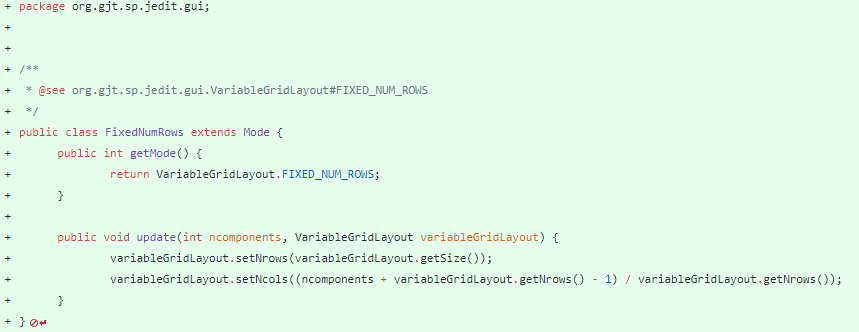


Figure 5. Code Refactoring of FixedNumRows

**Refactoring of feature envy:**

Class: CompressedReplace

* Basically, what feature envy code smell means, if a class accesses fields of another object to execute some operation, instead of just telling the object what to do.
* In the class CompressedReplace, we found an add method with feature envy code smell which accesses the fields of the replace class.
* Our goal is to alter the changes without changing the functionality of the class.
* So, the next step is refactoring the code. At the time of refactoring, we refactored the add() method in the compressedReplace class to the following code.

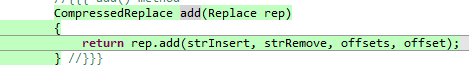


Figure 6. Code Refactoring of CompressedReplace class

* This resulted in the reduction of code from the previous method and the separated method does not directly access the features of another class.
* After refactoring, we checked to see if the code smell is removed or not and we are successful in removing the smell.
* We ran the entire code again and we did not break-down or crash in the functionality of the jedit project.