

CS515-Software Maintenance and Evolution

Assignment 4 - Code Smells and Refactoring

Team name:DeepMind-Ujwal Srinivasa and Nisha Rani

Tool Used for Detecting Code Smells

There are many tools available in eclipse that automatically help you detect code smells. We were able to detect code smells using the tool JDeodorant. JDeodorant is a plugin found in the eclipse marketplace. JDeodorant helps identify all the design related problems within the software. These are called code smells. There are five different types of code smell that are identified by the JDeodorant tool.

1. God class
2. Long Method
3. Type checking
4. Feature Envy
5. Duplicated Code

Code Smells in Pdfsam:

The packages in Pdfsam consist of many different code smells. However the feature envy code smell is not found in any of the packages. There are two Type Checking code smells found in the Pdfsam-Fx package. Long Method is found in almost all packages. There are packages like Pdfsam-gui and Pdfsam-fx with more than 50 long method code smells. God Class is found in every package except for Pdfsam-simple split. The highest number of god class bad smells is found in the Pdfsam-gui package which is 14. Duplicated Codes were not found anywhere in Pdfsam.

Pdfsam-Rotate(Long Method):

There are two types of code smell found here: Long Method and God Class. The class **RotateParameterBuilder** incorporated this bad smell due the modification made to integrate the change request 3 in assignment 2. The method in which this smell was found is the **void addinput** method. The type of the bad smell found is the **long method(Bloaters)**.

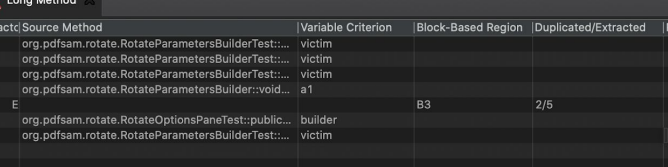
The lines of code in this method before incorporating the change was 5, after integrating the change request 3, the lines of code changed to 30 within this method. By the rule of thumb a method is said to have a Long Method smell if it has more than 10 lines of code (Lecture videos). When implementing the change request 3, we performed an override regarding the page range to implement EVEN and ODD functionality within the page range. After using JDeodorant we were able to figure out that the above implementation caused a long method code smell. JDeodorant suggests that we only define an if statement that allows you to access the EVEN or ODD functionality from the class Pag selection. The change we implemented was prone to errors and changes. The maintainability and the ability to understand the change without refactoring is difficult. Therefore i agree this to be a code smell.

```
victim.output(output);
```

These changes do not create a huge impact with respect to the lines of code. Therefore It is a little hard to agree with them as code smells.

Refactoring:

(Pdfsam-Rotate->RotateParametersBuilder.java->Method-addinput) :



Refactor	Source Method	Variable Criterion	Block-Based Region	Duplicated/Extracted	Rate it!
▶	org.pdfsam.rotate.RotateParametersBuilderTest:...	victim			
▶	org.pdfsam.rotate.RotateParametersBuilderTest:...	victim			
▶	org.pdfsam.rotate.RotateParametersBuilderTest:...	victim			
▶	org.pdfsam.rotate.RotateParametersBuilder:void...	a1			
E			B3	2/5	
▶	org.pdfsam.rotate.RotateOptionsPaneTest:public...	builder			
▶	org.pdfsam.rotate.RotateParametersBuilderTest:...	victim			

JDeodorant has a feature that allows you to refactor the code automatically. We

used the same feature to resolve the above defined code smell. This feature performed a series of changes within the addinput method. The series of changes are as follows.

- The previously implemented override function was removed. This can be viewed in line 63 of the class RotateParameterBuilder.
- The arraylist a1 is added as one of the parameters to the inputs.add() method. This was done by replacing the rotate parameter with a1.
- Then page range is specified as an argument within the array. By doing this we eliminate the need to extract EVEN pages using the for loop and just implement it using simple if condition.
- The above was repeated for ODD_PAGES and ALL_PAGES.

This refactoring simplified the steps in the change request 2 by reducing the lines of code from 30 to 10. The change request previously had not affected any other modules(detailed report in assignment 2). Therefore, refactoring this method did not produce any effect on other classes. This refactoring performed by JDeodorant only made changes within the method addinput in the RotateParameterBuilder class. The below figure shows the elimination of the code smell a1 after refactoring.

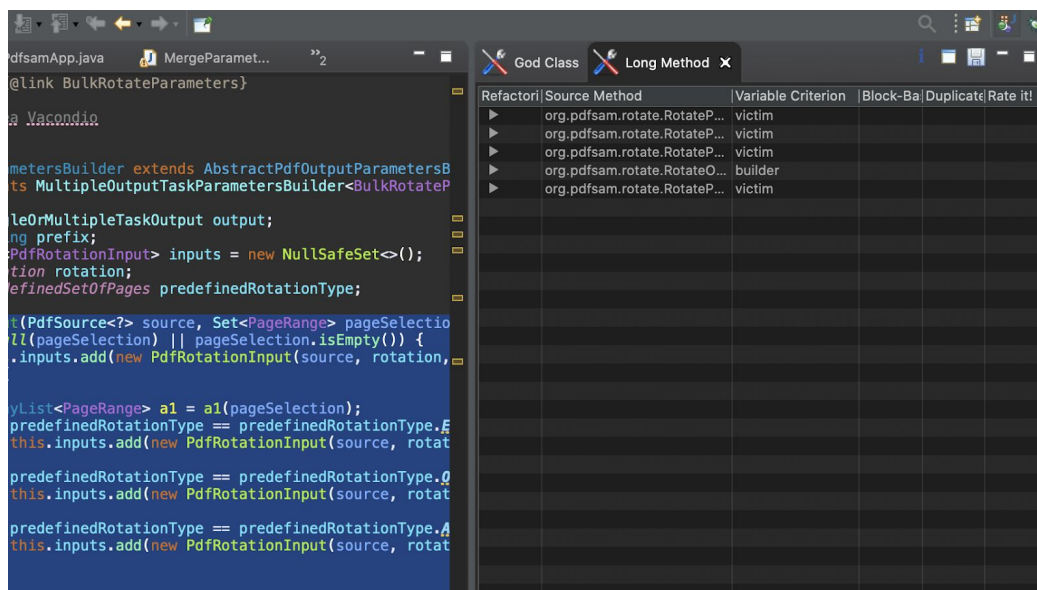


Fig 2: Code smell a1 eliminated after refactoring

You can see from the above figure the Long method Code smell is eliminated after refactoring. The other 5 Long method smells are left unchanged.

Testing:

The testing cases are already present in the Pdfsam-rotate->src/test/java. Since we did not implement any new methods or classes for this refactoring, we used the same pre-existing test cases and was able to build the software successfully. A screenshot of the build success is given below.

```
terminated: New configuration [maven build] /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java (Apr 9, 2020, 2:29:07 AM)
[INFO] --- maven-install-plugin:2.4:install (default-install) @ pdfsam-basic ---
[INFO] Installing /Users/ujwals/eclipse-workspace/cs515-001-s20-DeepMind-Pdfsam/pdfsam-basic/target/pdfsam-basic-4.0.5.jar to
[INFO] Installing /Users/ujwals/eclipse-workspace/cs515-001-s20-DeepMind-Pdfsam/pdfsam-basic/pom.xml to /Users/ujwals/.m2/repo
[INFO] Installing /Users/ujwals/eclipse-workspace/cs515-001-s20-DeepMind-Pdfsam/pdfsam-basic/target/pdfsam-basic-4.0.5-source:
[INFO]
[INFO] -----< org.pdfsam:pdfsam-docs >-----
[INFO] Building PDFsam docs 4.0.5 [15/15]
[INFO] -----[ pom ]-----
[INFO]
[INFO] --- maven-source-plugin:3.1.0:jar-no-fork (attach-sources) @ pdfsam-docs ---
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ pdfsam-docs ---
[INFO] Installing /Users/ujwals/eclipse-workspace/cs515-001-s20-DeepMind-Pdfsam/pdfsam-docs/pom.xml to /Users/ujwals/.m2/repo:
[INFO]
[INFO] Reactor Summary for PDFsam 4.0.5:
[INFO]
[INFO] PDFsam ..... SUCCESS [ 0.569 s]
[INFO] PDFsam i18n ..... SUCCESS [ 2.954 s]
[INFO] PDFsam core ..... SUCCESS [ 3.072 s]
[INFO] PDFsam JavaFx views ..... SUCCESS [ 4.165 s]
[INFO] PDFsam service ..... SUCCESS [ 1.081 s]
[INFO] PDFsam merge module ..... SUCCESS [ 0.576 s]
[INFO] PDFsam simple split module ..... SUCCESS [ 0.520 s]
[INFO] PDFsam split by size module ..... SUCCESS [ 0.467 s]
[INFO] PDFsam split by bookmarks module ..... SUCCESS [ 0.568 s]
[INFO] PDFsam alternate mix module ..... SUCCESS [ 0.292 s]
[INFO] PDFsam rotate module ..... SUCCESS [ 0.501 s]
[INFO] PDFsam extract module ..... SUCCESS [ 0.388 s]
[INFO] PDFsam desktop client application ..... SUCCESS [ 2.472 s]
[INFO] PDFsam Basic Edition application ..... SUCCESS [ 1.386 s]
[INFO] PDFsam docs ..... SUCCESS [ 0.016 s]
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 19.435 s
[INFO] Finished at: 2020-04-09T02:29:28-06:00
[INFO]
```

Fig 3: Pdfsam Rotate successfully passed the test cases.

Pdfsam-Gui(God Class):

There are two types of bad smells found in this package: Long Method and God class. There are about 60 Long method code smells and 14 god class code smells. Most of the long methods are found within the test classes that are used to test the GUI, like PreferenceAppearancePaneTest, PreferenceOutputPaneTest,...etc. Most of the classes in this module have methods that exceed more than 10 lines of code, by the rule of the thumb it is a Long Method Bad smell. However there are some methods whose lines of code do not exceed more than 10. They were identified as code smells because of the unused or extra defined variables. Refactoring these methods would not reduce the lines of code significantly, just reduce it by 1 or 2 lines. In such cases it is tough to argue that it is a code smell. Eg: `button = new Button("show");`

```
Scene scene = new Scene(new VBox(button));
stage.setScene(scene);
stage.show();
```

The above code was highlighted by JDeodorant for refactoring in the class `OpenWithDialogueTest`. The JDeodorant suggests extracting the method `stage` and creating a new class "stage", this may remove the long method code smell but it later creates another smell in the form of god class. In this case this cannot be accepted as a Bad smell as refactoring it will pave way for another bad smell in future.

In class **PdfsamApp** there are about 3 Long Methods identified. Out of which two of them are identified in the same method, `start`. The third Long method is found in the method `startLogAppender`. This method has about 14 lines of code, only 4 lines longer than normal. These 4 lines create a new local object, set variables for the object and then instantiate a new object for the class using the local variable. The shortening of this method will help us to understand the code even better and increases overall maintainability. Therefore we agree this to be a code smell. The second type of smell identified in Pdfsam-Gui is the God class. There are 14 different god class smells found in Pdfsam-Gui in classes like `QuickbarModuleProviderTest`, `NotationsControllerTest`, `SummaryTab`, `ControlPane`...etc.

PdfsamApp class has no Long Method but an instance of God class smell. User related features are held in the variable `UserContext`. This variable is initiated and saved via method invocations in the said class in PdfsamApp. This is identified as a God class by JDeodorant. The fix for this Bad smell is discussed in detail below in refactoring. This code smell can be agreed upon as a bad smell as this variable because this class has a few misplaced functionalities and these functionalities along with the variable `UserContext` does require another class. By doing this we are reducing the workload of the god class and reducing the proneness for duplication of code. This further improves the overall maintainability.

The following figure gives the visualization of the code smell in the class PdfsamApp.

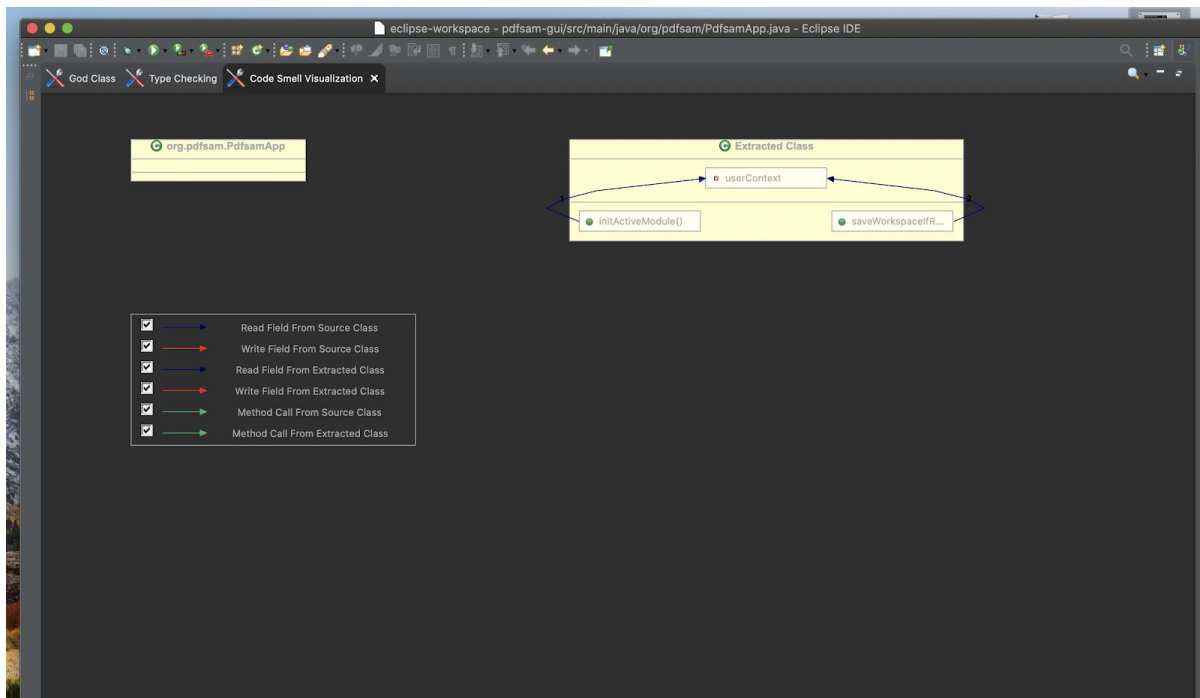


Fig 4 : Code smell visualization.

Refactoring:

2) God Class Code Smell:

(Pdfsam-Gui -> Org.Pdfsam -> PdfsamApp.java)

Out of the 14 different God class smells in Pdfsam-Gui, we will be refactoring the God class smell detected by JDeodorant in the PdfsamApp Class. This class is identified as smelly because the presence of the variable UserContext which holds user data and is needed when the application is started or exited. By using this variable, which has potential functionality within the same class, is making the class PdfsamApp difficult to maintain and also understand. Therefore JDeodorant identified this as a smell and fixed this issue by creating a new class. This new class holds the reference to the instance variable and also all the required methods. The code smell was refactored using the automatic refactoring feature present in the JDeodorant. The following figure shows the God Class smell before refactoring.

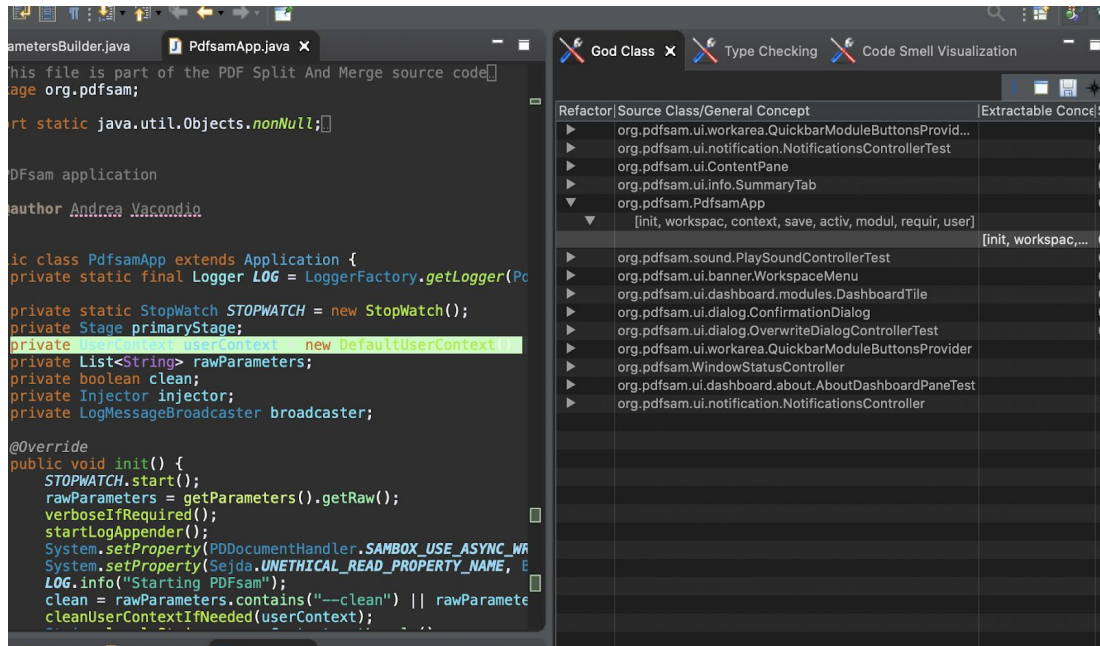


Fig 5: PdfsamApp God class smell before refactoring.

We can see from the above figure there are 14 different God class smells, one of which is the PdfsamApp class. The variable UserContext is highlighted to indicate what part of the code will be moved to a different class. The following steps are followed to eliminate the code smell.

- New class named PdfsamAppMod is created. This class holds the reference to the UserContext variable. This class is defined as public and is created alongside the PdfsamApp class.
- A new method is created for private UserContext to retrieve data. A new logger is added which will be used for logging the method.
- Three methods are added to the class PdfsamAppMod, they are:
 - SaveWorkSpaceIfRequired()
 - initActiveModule()
 - LoadWorkSpaceIfRequired()
- A reference is created in the PdfsamApp class to support the PdfsamAppMod class.
- References to the UserContext and the methods within the PdfsamApp class are updated.

The LoadWorkSpaceIfRequired was not copied to the new class, we had to manually copy this method. The rest of the work was done by the refactoring feature in JDeodorant. You can also perform the steps above manually, it is not difficult. By doing the above steps either manually or through use of tools, the God class smell will be removed from PdfsamApp. The following figure shows you the elimination of the bad smell after refactoring.

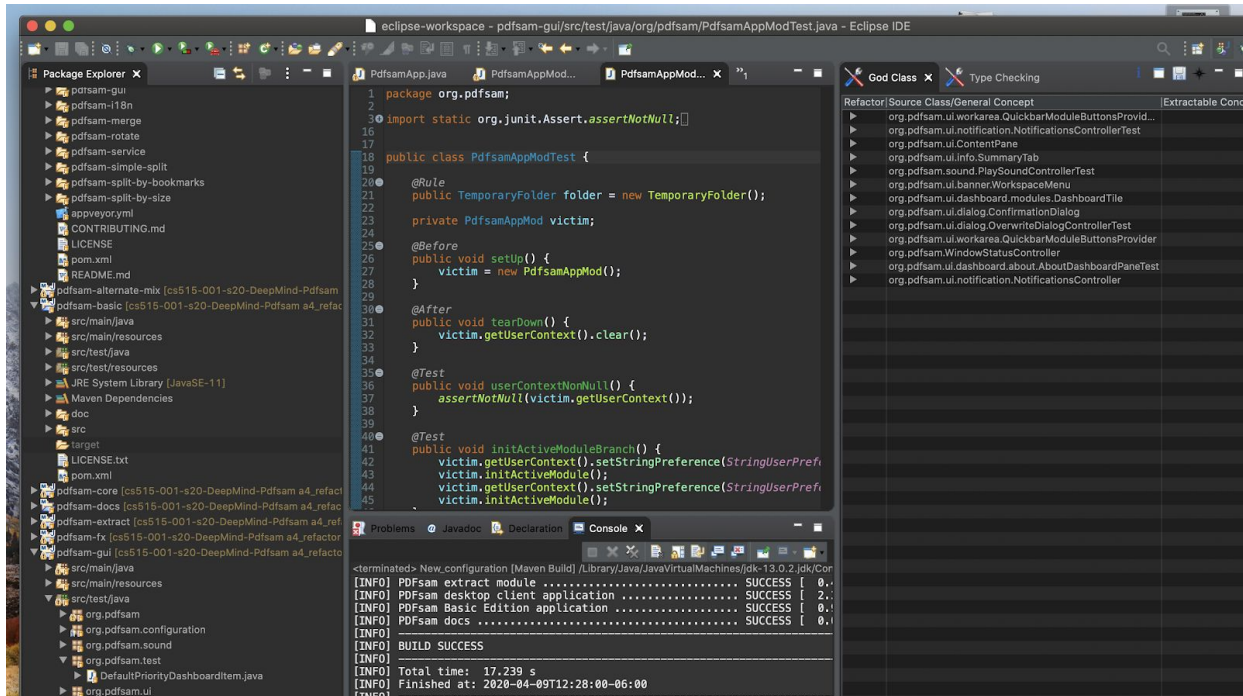


Fig 5: PdfsamApp God class smell eliminated.

Testing:

There were no test classes found within the Pdfsam-gui package to test the above code smell. We had to create a new test class within the pdfsam-gui package in src/test/java. The new test class is named as PdfsamAppModTest. The following describes the steps to take to implement a new test class. We used the Junit tool to create test classes along with manual effort.

- Create a new class PdfsamAppModTest.
- Defined functions for setUp(),tearDown() and UserContextNull(). We created these methods by referring to the other test classes within the package.
- Two new methods are created for testing the newly added methods in the new class PdfsamAppMod
 - initActiveModuleBranch()
 - SaveWorkSpaceIfRequired()
- For testing we used the same logic that was used by the other testing modules in the PdfsamGui package.

All the test cases were passed and we were able to build the software successfully. The following screenshots show the success status after passing all the test cases.

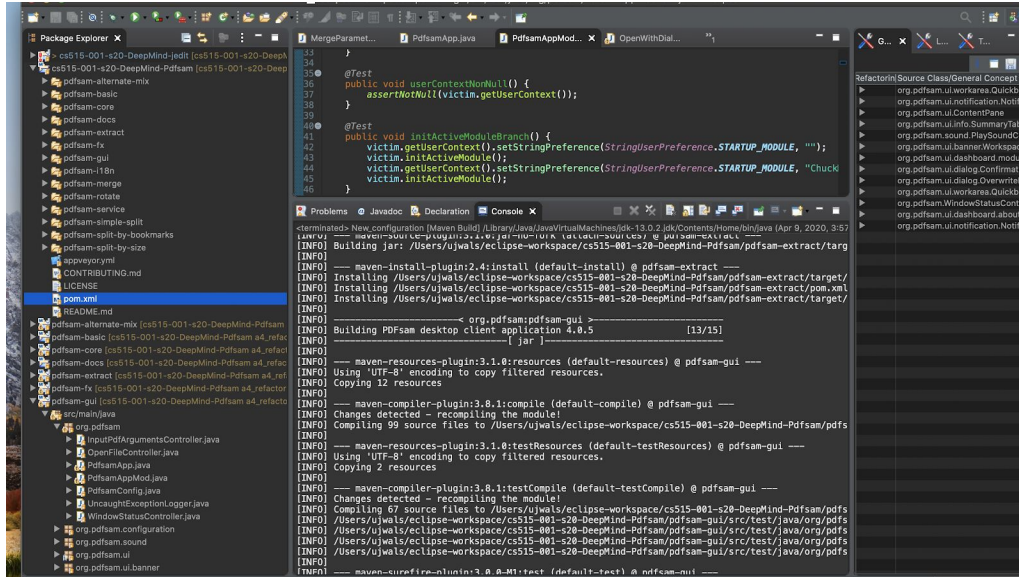


Fig 6: Test case passing in Pdfsam-gui

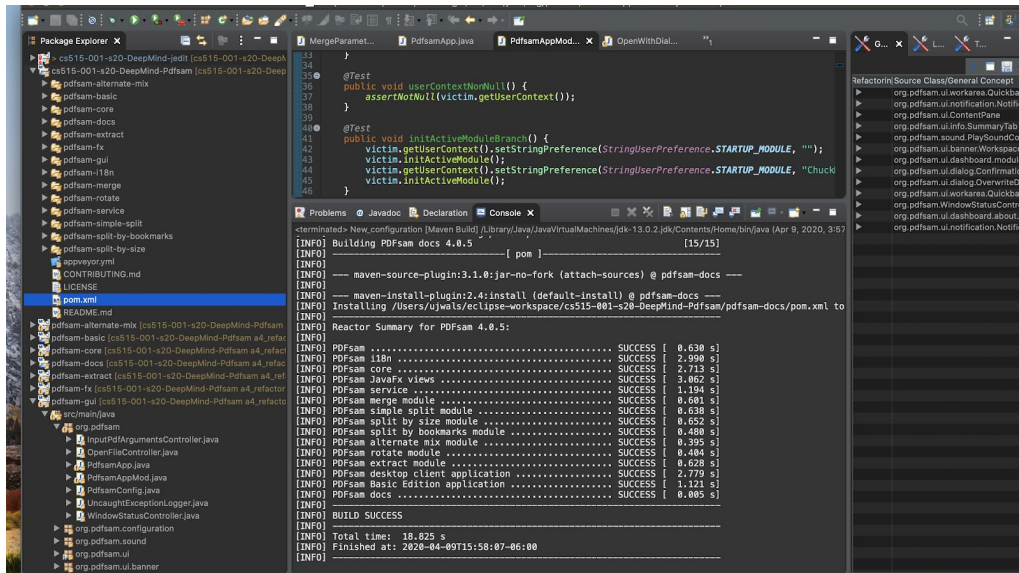


Fig 7: Build success after passing test cases.

Pdfsam-Fx(Type-Checking):

There are three types of bad smells found in this package: Type checking, Long Method and God class. There are two Type checking smells, 60 Long Methods and 8 God classes. Type checking is found in the classes RememberingLatestFileChooserWrapper and SelectionChangedEvent. Type Checking is considered a bad smell especially in statically typed languages because in a large software there will be a lot of classes inheriting other classes, so if type checking is used you may lose track of what type the object is at some point in the program. Therefore type checking is considered a code smell.

In the class SelectionChangedEvent there is type checking done in the Public method canMove(). This method uses type checking to determine Top, Bottom or Down selection. We do agree this to be a code smell as this class is inherited by various other classes in Pdfsam-fx. Therefore it is not optimal to implement type checking within this class. The JDeodorant suggests refactoring by removing the steps involved in type checking the selection within the canMove() method and creating a separate class for each type i.e Bottom, Top, and Down. By creating different classes we no longer have to worry about tracking the type. This refactoring is highly beneficial.

There is another type checking found in the class RememberingLatestFileChooserWrapper within the method showDialog. The method checks for a type SAVE. We do agree that this is a code smell because this class is extended and inherited by other classes in the module, at some point it becomes difficult to track the type. JDeodorant suggests in extracting the types and creating a separate class.

The figure Below shows the type checking code smell found in the Pdfsam fx.

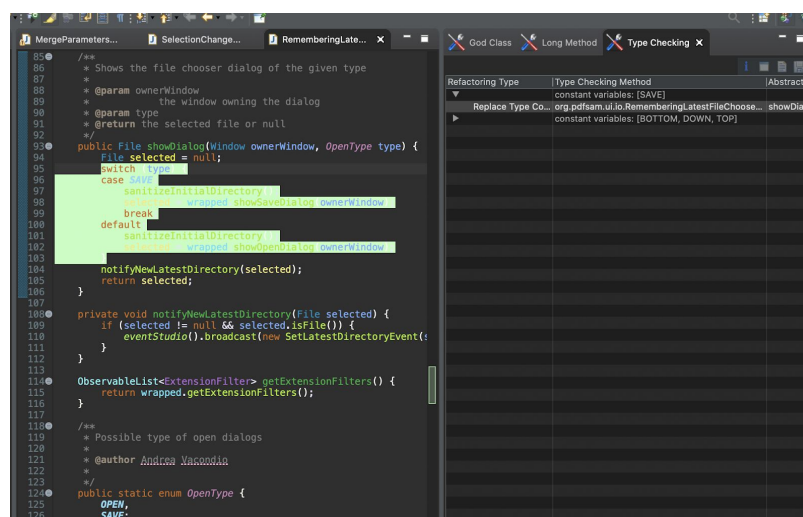


Fig 8: Type checking code smells.

JEDIT

Detecting and analyzing code smells:

1. God class:

Class: org/gjt/sp/jedit/textarea/TextArea.java

Method: Public void hideScrollBar(Boolean b)

1. Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

It violates the single responsibility principle and it controls a large number of objects implementing different functionalities. It provides tight coupling and challenges the code maintainability. It is very famous in bad programming because it creates tight coupling and increases the challenges in the code maintainability.

The solution is to extract all the methods and fields which are related to specific functionality into a separate class.

2. Explain why the class/method is flagged as smelly.

This class is flagged as smelly since it contains a large number of lines of code(270), methods(60) and attributes(6795). This class is a base class of *JEditEmbeddedTextArea* and *StandaloneTextArea*, hence it needs to be separated. There are few functions for instance, *public void hideScrollBar(Boolean b)* which can be allocated in a separate class as much of its functionality is not bound with other methods of the *TextArea.java* class.

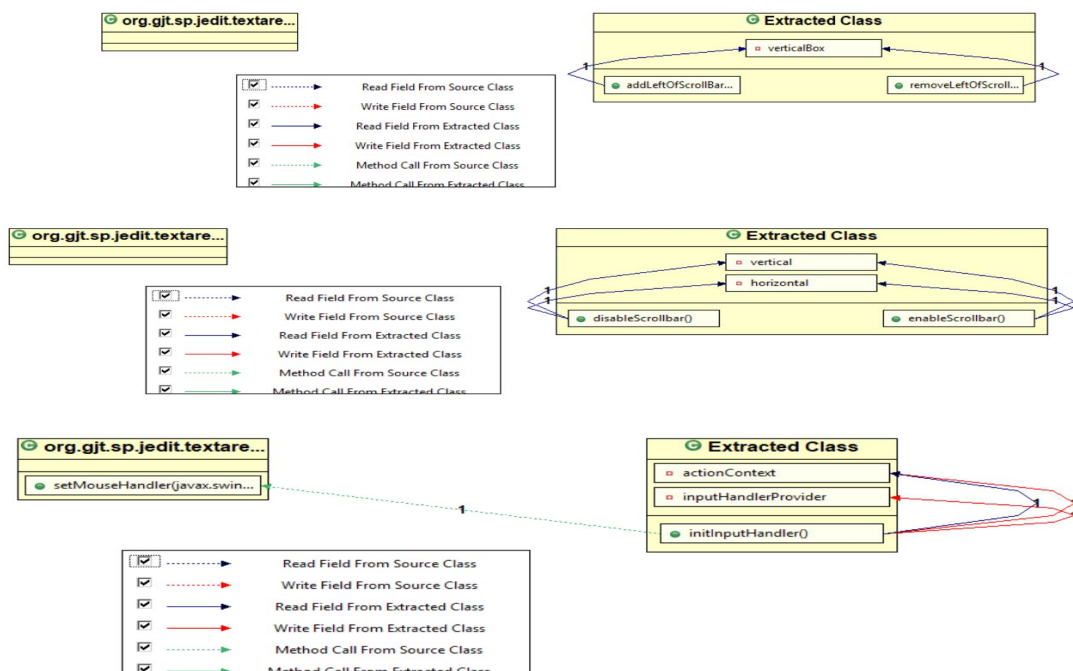


Figure1: Refactoring operations from Jdeodorant tool for God class

3. Do you agree that the detected smell is an actual smell?

Yes, we found the detected smell to be an actual smell by analyzing the code with the Jdeodorant tool. God class performs most of the work and allocates very little to other classes and the same is demonstrated here. Some of the functions such as *scrollUpPage*, *showPopupMenu*, *deleteParagraph* present in this class have little functionalities in this class and this could be placed in different classes.

2. Feature Envy:

Class: org.gjt.sp.jedit.textarea.TextAreaMouseHandler.java

Method: private int getSelectionPivotCaret()

1. Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

This occurs when a method references another class through method and fields, more often it references its own class. For a good design, the data and the methods modifying that data should stay as close together as possible.

The solution is to move the method to the class that it is most referring to and passing any parameters the new method requires.

2. Explain why the class/method is flagged as smelly.

This class flagged as smelly due to the presence of classes that use the data of other classes for instance, *getSelectionPivotCaret()* method refers to data from another class *TextArea.java* from the line `int caret = textArea.caret;` where the `caret` variable of *TextArea.java* class was accessed by the *TextAreaMouseHandler.java* class. Hence, this clearly defines this feature of bad smell where data is accessed by other classes instead of using attributes from its own class.

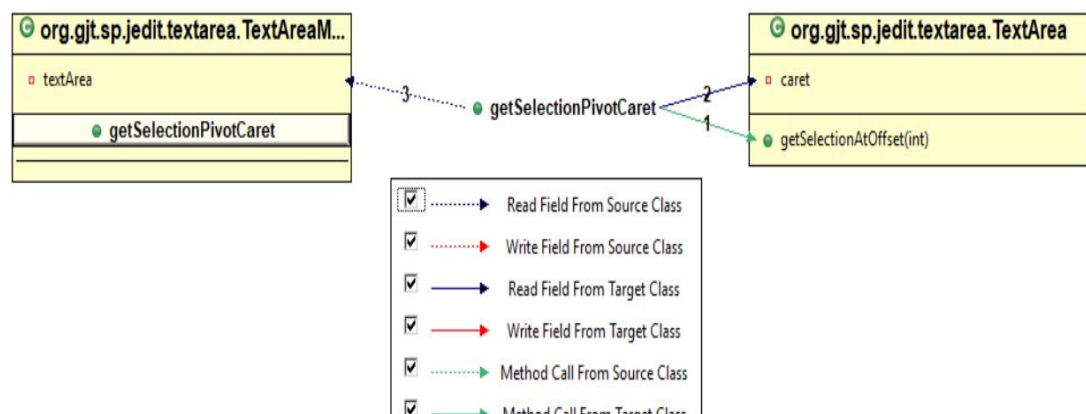


Figure2 : Refactoring operations from Jdeodorant tool for Feature Envy

3. Do you agree that the detected smell is an actual smell?

Yes, we agree the smell to be an actual code smell because the two attributes "Selection s and int caret" that belongs to other class (TextArea.java) were used by the method in the TextAreaMouseHandler.java class and does not perform on any operations on the attributes of its own class. Therefore, it can be said that this method is an example of Feature Envy bad smell.

3. Type Checking:

Class: org/gjt/sp/jedit/gui/VariableGridLayout.java

Method: public String toString()

1. Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

This type of code smell is introduced to select a variation of a code that should be executed, depending on the value of an attribute. It shows complicated conditional statements which would make code difficult to understand and maintain.

The solution is to apply refactorings to part of if/switch conditional statements that introduce inheritance and polymorphism.

2. Explain why the class/method is flagged as smelly.

The method flagged smelly due to the conditional code fragment i.e. an if/else structure where the static attributes were compared for equality with the type field. The conditional branches of the type-checking code were not moved as separate methods to the subclasses of a newly created State. In this case, *public String toString()* method manifests FIXED_NUM_ROWS (no. of rows and columns) in a single method in spite of creating different functions of rows and columns.

3. Do you agree that the detected smell is an actual smell?

We agree that the detected smell is an actual smell because here the if statement is not split into multiple functions that handle a single type. There should have been a separate class that handles each of these if conditions.

Removing code smells via refactoring

1. Feature Envy

1. List and describe in detail the refactorings (i.e., the code changes) used to remove the smell.

Using the Feature Envy bad smell we detected smell in the *getSelectionPivotCaret()* method in the class *org/gjt/sp/jedit/textarea/TextAreaMouseHandler.java*. By applying the Jdeodorant tool on jedit project, it recommends to move the *getSelectionPivotCaret()*

method to `TextArea` class and the method would be invoked by calling this class in the `TextAreaMouseHandler` class. After refactoring, the `getSelectionPivotCaret()` method got deleted and hence the Feature Envy smell for that component was removed.

2. Provide the rationale of the chosen refactoring operations.

The method does not perform any operations and it is just composed of 6 lines. The attributes used by this method were initialised in some other class and did not use any attributes which were initialized in its own class. This made no logical sense of preserving that method and hence removed it from the class and moved it to the `TextArea.java` class in order to get rid of the Feature Envy bad smell. The method in the `TextAreaMouseHandler.java` class was called using the class name(`TextArea`).

3.Explain what code changes you had to perform manually.

We did not perform any code manually as the tool performed all required changes.

2. Type Checking

1. List and describe in detail the refactorings (i.e., the code changes) used to remove the smell.

The method `public String toString()` in the class `VariableGridLayout.java` is used to fix the number of rows and columns. It uses a certain if/else condition to apply row and column functionality by using the “mode” attribute. By using JDeodorant tool, the if/else statements were broken and replaced by switch statement in th `VariableGridLayout` class and refactored the code by creating few new classes i.e. `Mode.java`, which is an abstract class and contains abstract methods, `FixedNumRows.java` that extends class `Mode.java` and `FixedNumColumns.java` that extends class `Mode.java`. These later two classes implement the two methods(`getMode()`, `update()`) of `Mode.java` class.

The existing class(`VariableGridLayout.java`) uses getter and setter methods called the `Mode.java` class and to return the attributes.

2. Provide the rationale of the chosen refactoring operations.

The method describes more complicated code by if/else statements where complex functionality is performed under a single conditional component. Hence, new abstract classes were developed in order to distribute the work. The conditional branches of the type-checking code were moved as separate methods to the subclasses of a newly created class. The subclasses should implement the abstract method of the superclass by copying the code of the corresponding conditional branch inside the body of the overridden method.

3. Explain what code changes you had to perform manually.

We did not perform any code manually as the tool performed all required changes.