# Code Smell and Refactoring Report

## 1.1 JEdit: Detecting and analyzing code smells

To find the code smell of the JEdit project, we used the SonarLint plugin in Eclipse. First SonarLint was installed using the Eclipse MarketPlace and then restarted Eclipse. Right click on the project jEdit and select SonarLint -> Analyze. This will generate the SonarLint report. Double clicking on each of the code smells in the report, the concerned class with the line of concern highlighted will be opened. A SonarLint Rule Description is also when clicked on the code smell. This description will give an idea about the code smell and a general way on how to remove this code smell.

It returned almost 61,000 code smells by processing 1949 java files and applying all kinds of Java rules. From these 61,000 code smells, most of them were mere warnings and some were inactive or old rules. By deactivating such rules and by differentiating rules from complexity metrics, we have reduced the number to around 2500. Now by going through all the code smells and categorizing them, we have identified the below three code smell types and refactored them to remove the code smell.

- Dead Code
- Long Method
- Duplicated Code

- **Dead Code:**

When a variable, parameter, field, method or class is no longer used, they are categorized as dead code. Also, the conditional expressions which are always true or false can lead to dead code. This is the case that we have used. Such dead code should never be used in production as it increases code size and reduces maintainability. Dead code is code smell type under the category “Dispensables”.

In this type, the class *org.gjt.sp.jedit.search.SearchAndReplace.java* is inspected and modified. The code that we have decided to modify is in the method *hyperSearch()* and is given below :

```
if(selection)
{
        s = view.getTextArea().getSelection();
        if(s == null)
        {
                results.searchFailed();
                return false;
        }
}
```

Here there is a null check for the variable ‘s’, which is returned from *getSelection()* as seen above. But the method *getSelection()* from class *org.gjt.sp.jedit.textarea.TextArea.java* is having a Nonnull annotation (as shown below) because of which null value is never returned from this method. So, this is a case where the conditional expression is never true and hence comes under Dead Code type and needs to be removed.

```
@Nonnull
public Selection[] getSelection()
{
        return selectionManager.getSelection();
}
```

- **Long Method:**

A method that contains too many lines of code is considered as a Long Method. This type of code smell is categorized under “Bloaters”, that represents something that has grown so large that it cannot be effectively handled. Such large methods should not be used as these codes are prone to have bugs and are difficult to maintain. Large methods tend to aggregate too many responsibilities, which is not a good practice.

In this category, the initialization method “*_init()*” from class *org.gjt.sp.jedit.options.StatusBarOptionPane.java* is inspected and modified. This method has a total of 140 lines describing the initialization of different modules. SonarLint has the authorized number of lines of codes in a method as 75, thereby categorizing this method as a code smell. We can divide this method *_init()* into smaller methods, each for different modules like setting the color panel, creating buttons in the Widget section, generating checkboxes etc. This improves readability and testability.

- **Duplicate Code:**

This is the case when a block of code is repeated across the source code. Code that is redundant needs to be removed and this type is under the category “Dispensables”. This code smell increases the code size, there by reducing testability and maintainability.

In class *org.gjt.sp.jedit.search.HyperSearchOperationNode*, the method *insertTreeNodes()* has some duplicated code. This comes under the rule “two branches in a conditional structure should not have exactly the same implementation”. Having two branches in an if chain with the same implementation is a case of duplicated code and these if-cases could be combined. In this specific case, the code is repeated as shown below.

```
if (topPathNdx == -1)
{
        topPathNdx = paths.length;
        topPathTmp = paths;
}
else if (paths.length < topPathNdx)
{
        topPathNdx = paths.length;
        topPathTmp = paths;
}
```

This could be combined to remove the code smell as below:

```
if(topPathNdx == -1 || paths.length < topPathNdx)
{
        topPathNdx = paths.length;
        topPathTmp = paths;
}
```

## 1.2 PDFsam: Detecting and analyzing code smells

We have done an analysis with the PDFsam project by evaluating the classes with the highest number of rule violations reported by JDeodorant[1]. The number of violations for the entire project are not so high as we tried running static code analysis tools for this project. The highest number which we have observed either falls under the God Class or Long Method type of code smells. The Feature Envy has not been observed in the JDeodorant and very few detections for the Type Checking have been found in the pdfsam project. Other types of code, like code duplication, commented code, empty classes have been there but we have used Sonarqube[2], to detect that. We also tried following tools for the static analysis of the PDFsam project: PMD[3], DesigniteJava[4].

To reach to decide code smells for the fix, we have gone through the entire analysis using those tools and studied three classes having unique rules violations. However, we have fixed more than 2 classes for this assignment. The details of the same can be found below.

| Module/ Class | Long Method | God Class | Feature Envy | Type Checking | Sonarqube | PMD | Designite Java |
|---|---|---|---|---|---|---|---|
| SelectionChangedEvent.java | No | Yes | No | Yes | No | CommentRequired, MethodArgumentCouldBeFinal, RedundantFieldInitializer | N/A |
| WindowStatusController.java | Yes | Yes | No | No | No | LawOfDemeter, MethodArgumentCouldBeFinal, CommentRequired,ImmutableField | N/A |
| pdfsam-split-by-size | Yes | Yes | No | No | CommentedCodeShouldBeRemoved, Duplicate code | N/A | Magic Number |

1 Table(2)

**Table(2):** Summary of the code smells detected by different tools. Yes indicates the code smell is detected by JDeodorant and No signifies not reported by the tool. N/A signifies that

[1] https://github.com/tsantalis/JDeodorant
[2] https://docs.sonarqube.org/latest/setup/get-started-2-minutes/
[3] https://pmd.github.io/
[4] http://www.designite-tools.com/designitejava/

we have not run/considered that tool for this file/module. Sonarqube, PMD and Designite Java's violations are listed here based on the high priority and also not all if priority is the same.

**1. Pdfsam-fx -> org.pdfsam.ui.selection.multiple.SelectionChangedEvent.java**

The above class has different code smells based on different tools. The code smells covered under the PMD column in Table(2) are very trivial based on other smells which we have identified using JDeodorant as significant and justifiable.

JDeodorant detected God Class and Type Checking code smells in this class file. God Class has been observed because the class has certain( three ) methods which we can move to another class. It would not be that difficult and effertious to extract them into another class and I think the change may add extra class which is not necessary so I ignored god class smell to fix. The second smell the class contains is multiple switch statements and addition or deletion of the new condition requires a lot of change. Therefore, replacing switch statement code smell that falls under **Object-Orientation Abusers** category with state/strategy technique that has been taught in the class makes more sense to refactor.

The method that consider for the refactoring:

- public boolean canMove(MoveType type)

The above method in the org.pdfsam.ui.selection.multiple.SelectionChangedEvent.java class is flagged as a method under Type Checking category. As this method has switch statements with multiple conditions it is very hard to add/delete more conditions for the future and also the modification to the current conditions may take more time to change. There are high chances of affecting/altering existing functionality if we modify the existing switch statements. Thus, we are considering the detected smell is the actual smell which may pave a way for the future enhancement, maintainability and the maintenance.

**2. Pdfsam-gui -> org.pdfsam.WindowStatusController.java**

The above class has different code smells based on different tools. The code smells covered under the PMD column in Table(2) are not taken into consideration as God Class code smell which we have identified using JDeodorant is significant and more meaningful to fix. The only one instance of Long Method has been identified but we did not consider it as a long method because the entire method is of 7-8 lines and the tool was suggesting 4 lines which can be separated into another method which is not justifiable. The PMD tool has suggested LowOfDemeter violation for method chain calls but we feel that no extra/ indirect call has been made which we can fix. We did try but most of the objects that call methods are from external packages/jar/library. Therefore, we think of leaving it as it is.

JDeodorant detected God class (**Bloaters - category**) code smell in this class file. It has detected this smell because class performs more than one responsibility of setting up the stage status and initializing ui for the application. When we ran JDeodorant the Lines of Code(L.O.C) of the class was 99. And the number of methods was six. As class has more than one responsibility and different methods which makes this class a God Class. Therefore, class flagged as a smelly class.

The following methods can be shifted to another class.

- public void defaultStageStatus()
- public void restore(StageStatus latestStatus)
- public boolean isNotMac()

After careful consideration we think that setting up stage related responsibility can be combined into one class. We decided to move the above three methods to another class which makes the single responsibility principle valid for each class. In our case, WindowStatusController.java class is extracted into WindowCtrlStageStatus.java class. This has also reduced the size of the original class WindowStatusController.

This class is having too many functionalities/responsibilities because of which the class has bloated. Therefore, this code smell is actual and splitting it up into classes will fix this code smell and also the SOLID principle of software development preserves.

**3. Pdfsam-split-by-size -> org.pdfsam.splitbysize -> SizeUnit.java, SizeUnitTest.java, SplitBySizeParametersBuilderTest.java, SplitOptionsPaneTest.java.**

The above classes have different code smells based on different tools. The code smells covered under the column(2-5) in Table(2) are there but the refactoring does not make any big difference in the classes. This is because three instances of the Long methods have been detected but automatic refactoring was suggesting to change only a couple of lines of the code which will not make a difference. Also, God Class is there but there is only one instance of that type of the class. Sonarqube suggested a couple of smells. The commented code should be removed and the duplicate code. The duplicate code has been found across different modules : pdfsam-split-by-size, pdfsam-split and pdfsam-splitbyBookmarks. Therefore,to fix this smell there is a very big change that we have to make which may include modification of the central code module or core code modules. Therefore, there are high chances of breaking the working functionality. Therefore, we have ignored that part.

DesignateJava has detected Magic Number code smell. This smell was there during A2 assignment. As we have changed constant values in A2 assignment we should have changed them into a human-readable constant which serves the purpose of live documentation as a constant would have a meaningful name which describes itself ( This is something we could have done while implementing A2 change request). As there are multiple places where constant value is being used or hardcoded it has no obvious meaning associated with it. This “anti-pattern” makes it harder to understand the program and to refactor the code. Problems will be increased when we have to make changes to this magic number. This is because, finding and replacing the number will not work because the same number may be used for a lot of different purposes in the code which leads to verification of each line of code change and test cases change to make sure it should work.

As the symbolic constant can act as a live documentation in the code and it is much easier to change the value of a constant at one place rather than searching for a number into the entire codebase and making sure that the numbers we are changing have the same purpose. Thus, maked smelly code for the given classes SizeUnitTest.java,

SplitBySizeParametersBuilderTest, SplitOptionsPaneTest.java is the actual one. One thing to take note is that, DesignateJava has not identified the smell into SizeUnit.java file as it is the enumerated constant file, it has been ignored by the tool. But the constant is used/originated into this file and this file is the place where the size conversion happens so we have replaced the Magic Number with Symbolic Constant in SizeUnit.java and all above mentioned files. This code smell falls under the Change **Preventers** category.

## 2. Refactoring and Evaluation

After analysing both the projects we have found different code smells in different projects. The below two subsections will showcase the step and the components that we have selected with different smells and finally refactored them and also validated them so that it works the same before and after the change by running the tool on which the smells have been found.

## 2.1 jEdit Refactoring and Evaluation

**All Refactoring commits are present here:**
https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-jedit/commits/a4_refactoring

**Commit to refactor and remove Dead Code:**

- https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-jedit/commit/f38531a4fb0f4fb92c2e87ad3d7e961a212667f4
- https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-jedit/commit/6ed6e8c39f1959693b35303d6ddc3e1c059d5d2c

**Commit to refactor Long Method:**

- https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-jedit/commit/3856800316e99c43abffd431293ec33a46742ccb

Since there are no test classes available for jEdit, we tried creating test cases using Junit, Mockito and PowerMockito(for static methods); but we failed to execute them successfully. So we have decided on manual testing for the modified methods to validate the modifications. Initially, before the modifications, we ran the methods to understand the behavior. Then after code modification, the test cases are manually checked and validated.

- **Dead Code:**

The code to modify in class *org.gjt.sp.jedit.search.SearchAndReplace.java* is below as we have discussed in section 1.1. The code that we have decided to modify is in the method *hyperSearch()* and is given below.

```
if(selection){
        s = view.getTextArea().getSelection();
```

```
            if(s == null){
                results.searchFailed();
                return false;
            }
        }
```

The variable s gets the value from TextArea.getSelection() method, which has the annotation @Nonnull as we discussed in 1.1. So, “s” will never have a null value meaning that the above null check and the code block inside that if statement is never executed, leading to dead code.

We removed this code block, highlighted above in yellow, and the SonarLint ran automatically upon saving the class. This code smell is removed from the newly generated report. To confirm, we explicitly ran SonarLint for this class alone by right clicking on the class and choosing SonarLint -> Analyze from the menu options. This code smell of dead code is removed from the SonarLint report.

**Validation:**

We removed code smell using refactoring in the Search functionality. The following are the test cases that are used to see if the behavior before and after the refactoring remains the same.

| **Step #** | **Description/Test Cases** | **Rationale** |
|---|---|---|
| 1 | Run JEdit Application, once the JEdit text editor is opened, place any text there. The text that we have used to validate is pasted towards the end of this document. Make sure that the cursor is at the beginning of the text. Use from menu options, Search -> Find or Control + Find from the keyboard.<br>A “Search and Replace” dialog box opens, where we can enter the word to be searched. Clicking on Find, it will highlight the first occurence of that word in the text and clicking on Find again, the next occurrence will be highlighted. | Setting up the environment to perform validation. |
| 2 | ● Inputs: Go to Search -> Find, in the dialog box that opens up type “it”, click on Find. The Ignore case checkbox is checked and Whole word is not checked.<br>● Expected Output: The first occurence of the word at line 1 is to be shown.<br>● Actual Output: “it” from Jedit in the first line is highlighted.<br>● Test Result: Passed | Checking the working of the Search or Find menu. |

| | | |
|---|---|---|
| 3 | ● Inputs: Go to Search -> Find, in the dialog box that opens up type “it”, click on Find. The Ignore case checkbox is checked and Whole word is also checked.<br>● Expected Output: The first occurence of the word at line 1 is to be shown.<br>● Actual Output: “It” from Jedit in the first line is highlighted.<br>● Test Result: Passed | Checking the Search and Replace option for different options like searching for the whole word and ignoring case. |
| 4 | ● Inputs: Go to Search -> Find, in the dialog box that opens up type “It”, click on Find. The Ignore case checkbox is unchecked and Whole word is checked.<br>● Expected Output: The first occurence of the word at line 1 is to be shown.<br>● Actual Output: “It” from Jedit in the first line is highlighted.<br>● Test Result: Passed | Checking the Search and Replace option for different options like searching for the whole word and making it case sensitive. |
| 5 | ● Inputs: Go to Search -> Find, in the dialog box that opens up type “It”, click on Find. The Ignore case checkbox is checked and Whole word is checked. In Search in option, select “Selection”<br>● Expected Output: There should not be any output as we have not selected any text.<br>● Actual Output: A dialog box shows up saying, “Please select some text or deactivate the “search in selection” option.<br>● Test Result: Passed | Checking the Search and Replace option for different options of Search in. In the other test cases the Search in was selected as Current buffer or All buffers which is the default. |
| 6 | ● Inputs: Go to Search -> Find, in the dialog box that opens up type “It”, click on Find. The Ignore case checkbox is checked and Whole word is checked. A portion of the first 8 lines are selected. In Search in option, select “Selection”<br>● Expected Output: Should show all the occurences of it in this selection only.<br>● Actual Output: A hypersearch is performed and highlights only the 2 “it” present in the selected text area in HyperSearch Results Window.<br>● Test Result: Passed | Checking the Search and Replace option for different options of Search in. In the other test cases the Search in was selected as Current buffer or All buffers which is the default. |

| | | |
|---|---|---|
| 7 | ● Inputs: Select a portion from the text area. Click Control+F or Go to Search -> Find, type “it” and click on Find.<br>● Expected Output: Should show all the occurences of it in this selection only.<br>● Actual Output: A hypersearch is performed and highlights only the 2 “it” present in the selected text area in HyperSearch Results Window.<br>● Test Result: Passed | Checking the Search and Replace option for different options of Search in. In the other test cases the Search in was selected as Current buffer or All buffers which is the default. |

- **Long Method:**

The initialization method “_init()” in class *org.gjt.sp.jedit.options.StatusBarOptionPane.java* has 140 lines of code. It contains the initialization of different types of modules which could be extracted to different methods and thereby reduces the number of lines of each of these methods to a permitted number (75 in case of SonarLint). We used Eclipse IDE’s Refactor -> Extract Method from the context menu options to refactor this long method. The following steps are followed to refactor this method _init().

1. We selected the initial few lines that are used for creating checkboxes and right click on it to show the menu options. From this we chose Refactor -> Extract Method. A new dialog box is opened where we are prompted to give a name to this extracted method. Upon giving the new method name as *createCheckboxPanel()* and clicking on OK, a new method is created and a new method call to this method is created in _init() as below.

   createCheckboxPanel();

2. We selected the next few code lines from _init(), that is used for creating the properties of the Abstract Options Panel. Then we used the Extract Method as described above and created the new extracted method *createAbstractOptionsPanel()*. This method returns optionsPanel and the same variable optionsPanel is used in _init() by creating an object of *createAbstractOptionsPanel()* as below.

   AbstractOptionPane optionsPanel = createAbstractOptionsPanel();

3. The next few code lines we have extracted using the above Extract Method,is used to initialize the properties of this AbstractOptions Panel. We named this new extracted method *initializeAbstractOptionsPanel()* and the refactor option in Eclipse, automatically created a call for this method in _init() as below.

   initializeAbstractOptionsPanel();

4. Now we select the code lines that is used to create buttons in the Option Panel and using Extract Method feature, a new method with these code lines are created and a variable is created in _init() as below:

   JPanel buttons = createButtons();

We refactored this long method by reducing the total number of lines from 140 to 31. When we save this class, the SonarLint report is automatically calculated and the long method code smell for this method _init() is removed from the new report.

**Validation:**

| Step # | Description/Test Cases | Rationale |
|---|---|---|
| 1 | Run JEdit Application, once the JEdit text editor is opened, place any text there. The text that we have used to validate is pasted towards the end of this document. | Setting up the environment to perform validation. |
| 2 | ● Inputs: Do not paste any text on the text editor.<br>● Expected Output: The status bar options should be line, column, caret offset, caret length, word offset, word length are shown as 1,1 (0/0)(0/0) at the bottom left.<br>● Actual Output: The status bar shows 1,1 (0/0)(0/0) at the bottom left.<br>● Test Result: Passed | Checking if Abstract Options Panel is correctly created and initialized without text. |
| 3 | ● Inputs: Paste the test on the text editor. Click on any word. Clicked on the word “MAC" in line 26.<br>● Expected Output: The status bar options should be 26,39 (1636/6033)(236/936) at the bottom left.<br>● Actual Output: The status bar shows 26,39 (1636/6033)(236/936) at the bottom left.<br>● Test Result: Passed | Checking if Abstract Options Panel is correctly created and initialized with text. |
| 4 | ● Inputs: Paste the test on the text editor (optional). From the menu options, go to Utilities -> Global Options -> jEdit -> Status Bar.<br>● Expected Output: The Options menu here should show color selection for Status bar text, Status bar background, Memory indicator foreground and Memory indicator background.<br>● Actual Output: The Options menu shows the color selection for Status bar text, Status bar background, Memory indicator foreground and Memory indicator background. | Checking if the new method createAbstractOptionsPanel() created from _init(), generates the color panel as before refactoring. |

| | | |
|---|---|---|
| | • Test Result: Passed | |
| 5 | • Inputs: Paste the test on the text editor (optional). From the menu options, go to Utilities -> Global Options -> jEdit -> Status Bar.<br>• Expected Output: The Options menu here should show 7 different Caret position display options with checkboxes.<br>• Actual Output: The Options menu shows the different Caret position display options with checkboxes, which are, “Show caret line number”, "Show caret offset from start of line", "Show caret virtual offset from start of line", "Show caret offset from start of file", "Show length of file", "Show word offset from start of file" and "Show number of words in the file".<br>• Test Result: Passed | Checking if the new methods createCheckboxPanel() and initializeAbstractOptionsPanel() created from _init(), generates the checkboxes as before refactoring. |
| 6 | • Inputs: Paste the test on the text editor (optional). From the menu options, go to Utilities -> Global Options -> jEdit -> Status Bar.<br>• Expected Output: The Widgets menu here should show the different Widgets that could be edited with the buttons or arrows present.<br>• Actual Output: The Widgets menu shows the different Widgets that could be edited by clicking on the arrows or the buttons available in the Widgets menu.<br>• Test Result: Passed | Checking if the new method createButtons() created from _init(), generates the buttons and arrows as before and the widgets could be edited using these arrows like before refactoring. |

## 2.2 PDFSam Refactoring and Evaluation

**All the commits for the pdfsam code smells can be found here:**
**https://github.com/torakiki/pdfsam/compare/master...SwathyAravind:a4_refactoring**

**Pdfsam-fx -> org.pdfsam.ui.selection.multiple.SelectionChangedEvent.java**

**Code Location In Github:**

**Commit 1: Code smell Fix**

**https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-pdfsam/commit/4820976754d38639cb464ac60f520a0000318693**

The **Type Checking** code smell is detected by JDeodorant in SelectionChangedEvent.java class as discussed in section **1.2**. We have used the automated refactoring support and the Extract Class technique to resolve this code smell. The below steps were taken to fix Type Checking code smell.

1. First of all the **boolean canMove(MoveType type)** method has been changed and created a new private method having multiple switch statement **getTypeObject(MoveType type)** of type **DirectionalMove** and created two new getters **int getBottom()** and **int getTop()** to return the value of the bottom and top variables.
2. A new abstract super class **DirectionalMove.java** was created under org.pdfsam.ui.selection.multiple package and it has an abstract method which the subclasses have to override this method.
    a. **public abstract boolean canMove(SelectionChangedEvent selectionChangedEvent)**
3. Four child classes **Bottom.java, Top.java, Down.java, Default.java** were created under org.pdfsam.ui.selection.multiple package. All the classes override the superclass method and returns value based on the condition
4. Inside the **getTypeObject(MoveType type) method** based on the conditions new child classes which are created in step 3 are instantiated.
5. Four new test class files **BottomTest.java, TopTest.java, DownTest.java, DefaultTest.java** were created and also we made sure that existing test cases should pass.
6. We also did manual testing to verify the functionality should be preserved
7. All the test cases along with the newly added test cases are passed and the build completed successfully.
8. We ran the JDeodorant again. Code smell has not been visible after the change.

The refactoring was suggested by JDeodorant's automatic refactoring tool and we did that. However, the name of the classes and unit test cases have been added manually. The one change apart from automatic refactoring that we implemented is the default case return in switch statement. Automated tool returned null but to preserve the original functionality we created the Default.java class and returned the instance of that class as a default case and after that all the existing test cases passed.

**Some Manual Validation and Testing:**

| Step # | Description | Rationale |
|---|---|---|
| 1 | Test case defined: Manual Functional Test<br>● Build and run the mvn project.<br>● Open the Merge/Alternate Mix module from all the available options.<br>● Add 3 pdf (Valid size) into the "Drag and Drop PDF files here" area. | All the buttons should set their state as per expected output. They should be functional.<br>The Move Up button should move pdf 1 step up.<br>The Move Down button should move pdf 1 step down. |

| | | |
|---|---|---|
| | ● Select 1st pdf and verify enabled/disabled button in the top bar( Move Up, Move Down)<br>● Select 2rd pdf and verify enabled/disabled button in the top bar( Move Up, Move Down)<br>● Select 3rd pdf and verify enabled/disabled button in the top bar( Move Up, Move Down)<br>Note: Rest of the buttons in the toolbar (Add/Clear/Remove) will not be affected due to this change.<br><br>Expected output:<br>● Select 1st pdf : The "**Move Down**" button should be enabled and functional, the "**Move Up**" button should be disabled.<br>● Select 2nd pdf : The "**Move Down**" and "**Move Up**" buttons should be enabled and functional.<br>● Select 3nd pdf : The "**Move Down**" button should be disabled and the "**Move Up**" button should be enabled and functional. | Test result:<br><br>After the code smell fix, all the buttons' state is appearing as it was previously before the change.<br>The Move Up and Move Down buttons also work as expected. ( They are moving pdf 1 step up and down )<br><br>Test result: Passed |
| 2 | Test case defined: Manual Functional Test<br>● Build and run the mvn project.<br>● Open the Merge/Alternate Mix module from all the available options.<br>● Add 4 pdf (Valid size) into the "Drag and Drop PDF files here" area.<br>● Select 2rd pdf and verify enabled/disabled button in the top bar( Move Up, Move Down)<br>● After selecting , 2rd pdf click on Remove button and verify the selected pdf and buttons on the top bar( Move Up, Move Down)<br>●<br><br>Note: Rest of the buttons in the toolbar (Add/Clear/Remove) will not be affected due to this change.<br><br>Expected output:<br>● Select 2rd pdf : The "**Move Down**" and "**Move Up**" buttons should be enabled and functional. | To verify selection change event broadcast works as expected before and after the code smell fix.<br><br>Test Result:<br><br>When we selected the 2rd pdf "**Move Down**" and "**Move Up**" buttons were enabled.<br><br>After Clicking on the "**Remove**" button selection moved to 1st pdf and the "**Move Down**" button was enabled and the "**Move Up**" button was disabled. |

| | | |
|---|---|---|
| | • After removing 2rd pdf - The selection should move to the 1st pdf and the "**Move Down"** button should only be enabled and functional. The "**Move Up"** button should be disabled | The test passed. |
| 3 | Test case defined: Unit Test case<br><br>Added Four Test files **BottomTest.java, TopTest.java,DownTest.java,DefaultTest.java**<br>• Added two test cases in each file<br>• 1st to check weather pdf can move<br>• 2nd to check pdf can not move<br><br>Note: Apply to above mentioned all the test files.<br><br>Expected output:<br>All the test cases should pass with provided values in the function select. | All the test cases related to this should be passed after the fix for the code smell. There should be no change in functionality.<br><br>The tests are passed. |
| 4 | All the existing tests cases should pass after the fix.<br><br>Expected output:<br>All test cases should pass. | As there is no change in the functionality all the test cases should pass. Which indicates functionality has not been changed.<br><br>Test Result:<br><br>All the existing test cases are passed after the modification.<br><br>All the tests are passed. |

**Pdfsam-gui -> org.pdfsam.WindowStatusController.java**

**Commit- 1: Fixed code smell**

https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-pdfsam/commit/aa6fe74f90abfa48586e00f06df3fd645c78fb90

**Commit- 2: Made Class/Variable name small**

https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-pdfsam/commit/1475b4cbb96b3e1547984e2992b2a3556c5d9116

The **God Class** code smell is detected by JDeodorant in **WindowStatusController.java** class as discussed in section **1.2.** We have used the automated refactoring support and the Extract Class technique to resolve this code smell. To fix this smell we extracted a new class called **WindowCtrlStageStatus.java** from the existing class mentioned in the previous statement. The below steps were taken to fix the God Class code smell.

1. We have created a new class **WindowCtrlStageStatus.java** that contains the that set the Stage value and performs the task of setting up stage for the WindowStatusController.
2. After that we have moved related methods which can do all the Stage setup functionality. The methods are **public void defaultStageStatus(), public void restore(StageStatus latestStatus) and public boolean isNotMac()** have been added to the newly created class.
3. We have created the instance of the newly created class into the original class. And initialized the object during the time of declaration in **WindowStatusController** class.
4. The reference to the methods in the **WindowStatusController** class has been updated and replaced by the newly created object's reference as we have moved three methods into the new class **WindowCtrlStageStatus,** we have to change that to support method invocation with newly created object.
5. After that we have verified that all the existing test cases should pass.
6. Also created a new test file **WindowCtrlStageStatusTest.java for the** newly added class and added test cases as well.
7. We have done manual testing to verify all the existing functionality should work in the same way after the change and it was working fine as well.
8. All the test cases along with the newly added test cases are passed and the build completed successfully.
9. We ran the JDeodorant again. Code smell has not been visible after the change.

The refactoring was suggested by JDeodorant's automatic refactoring tool and we did that. However, the name of the classes and unit test cases have been added manually. Most of the work has been done by tool but this refactoring took a lot of time in testing and for the verification of the existing test cases should work. As all the test cases passed which indicates that functionality before and after the change remained the same.

**Manual Verification and Testing:**
As the change is introduced in the class which sets the stage for the application on the start up. Before the change I debugged that part of code using Eclipse IDE by putting breakpoints manually to check workflow. After fixing the code it works the same way and the application starts and works exactly the same way. No change has been observed. I added test cases for the new class and I also made sure that the existing test case works fine. All the test cases are working well.

**Workflow to reach to a code can be found here:**
When application starts, under the **org.pdfsam.basic** package the **App.java** class's **main** method gets called. Inside the main **javafx.application.Application** package launches **PdfsamApp.java** class which extends **Application** class and overrides **public void**

**start(Stage primaryStage)** method which takes a **Stage(primaryStage)** parameter. Inside the **PdfsamApp.java** class's start method will be called. It then calls **initWindowsStatusController(primaryStage)** which delegates call to **setStage** inside a **WindowStatusController.java** class. This class is refactored and extracted into a new class **windowCtrlStageStatus.java**. Once the setStage method is called, the public void **initUi()** method gets called inside the **WindowStatusController.java** class file. Inside the **initUi()** method a newly instantiated class's method **public void restore(StageStatus latestStatus)** or **public void defaultStageStatus() gets** called based on the **latestStatus** value.
**StageStatus latestStatus = service.getLatestStatus();**
If **latestStatus** is not null and pdfsam's UI is enabled (Boolean.getBoolean(PDFSAM_DISABLE_UI_RESTORE) should be false) then **restore(latestStatus)** method will be called.
If **latestStatus is null or** pdfsam's UI is disabled (Boolean.getBoolean(PDFSAM_DISABLE_UI_RESTORE) should be true) then **defaultStageStatus()** method will be called.

The same workflow with the same value has been observed before and after the change and there is no change in the functionality of the application before and after the fix.

**Pdfsam-split-by-size -> org.pdfsam.splitbysize -> SizeUnit.java, SizeUnitTest.java, SplitBySizeParametersBuilderTest.java, SplitOptionsPaneTest.java**

**Code Location In Github**

**Commit 1: Replaced hard coded number with symbolic constant**

https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-pdfsam/commit/69ed4811ef72a5018de246dcc02799e5e485c592

**Commit 2: Created constant at one place and used in entire module**

https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-pdfsam/commit/e79394d3f98804af679d587393a121c436d65808

**Commit 3: Removed unused commented code and import**

https://github.com/SwathyAravind/cs515-001-s20-Swathy-Rachit-pdfsam/commit/43490886fa6c42ad7826a9a6c7fa2636e9a6c16c

The **Magic Number** code smell is detected by the DesignateJava tool under the **Pdfsam-split-by-size** module as discussed in section **1.2**. To fix this test case we have performed manual refactoring.

The below steps were taken to fix the **Magic Number** code smell.

1. First of all the commented code in the **SizeUnit.java** which was there as a part of the change request we implemented in A2. It was removed along with unused imports in the **SizeUnit.java**
2. Using search functionality of the Eclipse IDE we have identified all the places where the constant is used.

3. We have identified almost seven places where this constant is hardcoded.
4. After searching we have checked the purpose of each constant that has been used and why it has been used.
5. After looking at those we got to know that all of them used to convert the size of the pdf from Kilo-Bytes to Bytes.
6. We declare constant named **KB_TO_BYTES** in the **SizeUnit.java**
7. We used this constant in **SizeUnitTest.java, SplitBySizeParametersBuilderTest.java, SplitOptionsPaneTest.java** files and also changed other variables which do not have meaningful names.
8. Verify all the testcases and build completed successfully.
9. We ran DesigniteJava again and the code smell is no longer visible.

This refactoring was manual one and we did this change to replace hard coded number value with symbolic constant to increase readability of the code also in future if someone wants to change the conversion/kio-byte-to-bytes he only needs to change at one place. We also changed other constants like **sizeToSplitAt, splitSizeInputVal,** etc. to increase readability and maintainability of the entire **Pdfsam-split-by-size** module.

**Some Manual Validation and Testing:**

| Step # | Description | Rationale |
|---|---|---|
| 1 | Test case defined: Unit Test case<br><br>Added constant in sizeUnit.java<br>● We added a new symbolic constant **KB_TO_BYTES** rather than hard coded value 1000.<br>● We added **sizeToSplitAt** constant to improve readability.<br><br>Unit Test case in SizeUnitTest.java.<br>● We changed the unit test case to conversion constant from hardcoded 1000 in toBytes() to symbolic constant **KB_TO_BYTES** method's assert statement to ensure that file should not exceed the size of the given threshold and existing unit test should pass.<br><br>Expected output:<br>It should pass the test case with the given threshold during the conversion to maintain the functionality. | All the test cases related to this should be passed after the change in the sizeUnit.java and corresponding changes into SizeUnitTest.java file. There should be no change in functionality.<br><br>The test passed. |
| 2 | Test case defined: | The test cases related to this should be passed while performing apply |

| | | |
|---|---|---|
| | Unit Test cases in SplitOptionsPaneTest.java.<br>● We changed the unit test case to use **KB_TO_BYTES** constant into the **public void apply()** method to ensure that file should not exceed the size of the given threshold for the apply function and existing unit test should pass.<br><br>Expected output:<br>It should pass the test case with the given threshold during the conversion while running the apply function. | function which gets called on the footer.runButton click. SplitOptionsPaneTest.java file has an apply function which does this functionality of splitting the pdf and therefore all the test cases should be passed after adding constants.<br><br>The test passed. |
| 3 | Test case defined: Manual Functional Test<br>● Build and run the mvn project.<br>● Open the Split by Size module from all the available options.<br>● Add a pdf file of valid size. let's say ( 10 MB) Split pdf at 1 MB.<br><br>Expected output: Splitted pdfs size should not exceed the threshold size except for the single page. | The newly generated pdfs should not exceed the given split size.<br><br>All the pdfs have size < threshold except a single page pdf whose size exceeds.<br><br>Test result: Passed |
| 4 | Test case defined: Manual Functional Test<br>● Provide files as a input to the updated version ( For example: file size 505, 509, 514 KB )<br>● Add a split size of 500 KB and click on the Run button.<br>● Validate that files have been created and all of them should be below threshold.<br><br>Expected Output:<br>All the files should be less than the given threshold values. | Generated files should be less than the specified split size.<br><br>The test passed. |

**NOTE:** We fixed the above mentioned Magic Number code smell as well.
Extra Code smell that we have fixed is found below.

**Pdfsam-merge -> Org.pdfsam.merge.MergeSelectionPane.java**
**Glthub Location for the Code:**
**Commit 1: Code smell and Test cases related Change**
**https://github.com/torakiki/pdfsam/commit/0b9c57284d026ede9211d6f68ed14296e4952f0e**

The smell which we fixed here is **LawOfDemeter** - **Method chain calls** violation falls under **Couplers category** that has been detected while we ran PMD tool in Eclipse IDE.
**public void apply(MergeParametersBuilder builder, Consumer<String> onError)** method makes too many calls from this module to another and increases complexity of the code. Which can be simplified by reducing method chain calls. If we reference methods which are necessary using local reference rather than referencing each time we can reduce that complexity and increase readability of the code. To fix that smell we changed calling methods via referencing the local variable **selectionData** of type **SelectionTableRowData** and iterated over the loop. All the methods we referenced previously on A()->B()->C() now we call methods using local variable reference. Also added another local variable on which methods can be referenced.
**PdfDocumentDescriptor pdfDocDes = selectionData.descriptor();**

Changing these has resulted in no violation of method chain calls. We refactored the method and also made sure that test cases pass. Also manually verified the functionality before and after the fix. Both observed the same. However, during the code fix we omitted
**if (!builder.hasInput()) {**
**onError.accept(DefaultI18nContext.getInstance().i18n("No PDF document has been selected"));**

**}**
Above if condition from the violation. This is because we did try to make changes that will fix the Method chain calls violation but it is the interface method which the super class implements and this functionality shared across the application so we decided not to change.

**Manual Verification and Testing:** As functions affect merging functionality we added test cases of that functionality. Also automated test cases are working fine as well.

| Step # | Description | Rationale |
|---|---|---|
| 1 | Test case defined:<br>● Run the mvn project.<br>● Open the merge menu.<br>● We have added a pdf having 5 pages and specify intersecting ranges comma separated.<br>● Inputs: We specify the range as 1-4,2-4 | This is the regular expected behavior.<br><br>The test passed.<br><br>On the click of the run button we got the output with pdf having 1,2,3,4,2,3,4 pages. |

| | | |
|---|---|---|
| | Expected output: The merged document should have pages 1,2,3,4,2,3,4 | Test result: Passed |
| 2 | Test case defined:<br>● Run the mvn project.<br>● Open the merge menu.<br>● We have added pdf with 8 pages and specify the range 2-7 and then intersection range without specifying end range<br>● Inputs: We specified the range with 2-7, 4-.<br><br>Expected Output: document should have 2,3,4,5,6,7,4,5,6,7,8 | We want all the pages from start - end of the file to be appended to the new pdf.Also, for only one input (4-) we want pages from 4 to the end of the file should be appended.<br><br>Application works as expected<br><br>The test passed. |
| 3 | Test case defined:<br>● Run the mvn project.<br>● Open the merge menu.<br>● We add two different pdfs, and both input had intersection range<br>● Inputs: The range for the first pdf was 7-9, 8. The second pdf range was 1-6, 4<br><br>Expected output:<br>Final pdf should have pages 7,8,9,8 for the first pdf followed by pages 1,2,3,4,5,6,4 for the second pdf. | For multiple documents that we are inserting, it should work if we provide an intersecting range for them.<br><br>Merge pdf should have all the ranges.<br><br>The test passed. |

## JEdit Text that we used for Validation:

*Features*
*jEdit includes syntax highlighting that provides native support for over 200 file formats. Support for additional formats can be added manually using XML files. It supports UTF-8 and many other encodings.*

*It has extensive code folding and text folding capabilities as well as text wrapping that takes indents into account.*

*The application is highly customizable and can be extended with macros written in BeanShell, Jython, JavaScript and some other scripting languages.*

*Plug-ins*
*There are over 150 available jEdit plug-ins for many different application areas.*

*Plug-ins are used to customize the application for individual use and can make it into an advanced XML/HTML editor, or an integrated development environment (IDE), with compiler, code completion,*
*context-sensitive help, debugging, visual differentiation and language-specific tools.*

*The plug-ins are downloaded via an integrated plug-in manager which finds and installs them along with any dependencies. The plugin manager will track new versions and can download associated updates automatically.[5]*

*Some available plug-ins include:*

*Spell checker using Aspell*
*Syntax and style checkers for various languages[6]*
*Text auto-complete*
*Accents plugin that converts character abbreviations for accented characters as they are typed.*
*XML plugin that is used for editing XML, HTML, JavaScript and CSS files. In the case of XML, the plug-in does validation. For XML, HTML and CSS, it uses auto-completion popups for elements, attributes and entities.[7]*
*Reception*
*In general jEdit has received positive reviews from developers.*

*Rob Griffiths wrote in April 2002 for MAC OS X HINTS saying he was "very impressed" and naming it "pick of the week". He cited its file memory upon reopening, its ability to notice if an open file was*
*changed on disk by another program, syntax coloring, including that users can create their own colour schemes, split windows feature, show line number feature, convertible tabs to soft-tabs and view sidebars.*
*He also praised its customization possibilities using the extensive preferences panel and the "on the fly" search engine, which searches while typing. Griffiths noted that the application has a few drawbacks,*
*such as that it is "a bit slow at scrolling a line at a time" and that because it is a Java application it doesn't have the full Aqua interface.[8]*

*Also reviewing the application in April 2002, Daniel Steinberg writing for O'Reilly Media said "The strength of jEdit for Java developers comes from the plug-ins contributed by the community...For*
*the most part, there's nothing here that couldn't be done with BBEdit or even with Emacs or vi. jEdit packages the capabilities much more nicely and makes it easy to call often-used functionality using the plug-ins.*

*Where I saw NetBeans as overkill, others may see jEdit as underkill for an IDE or overkill for a text editor. I find it Mac friendly and easy to use. I don't expect too much from it, so I tend to be pleased with what I get."[9]*

*Scott Beatty reviewing jEdit on SitePoint in 2005 particularly noted the application's folding feature along with its search and replace and PHP syntax highlighting capabilities.*
*He recommended the use of the PHPParser plug-in. PHPParser is a sidebar that checks for PHP syntax errors whenever a PHP code file is loaded or saved.*
*He noted that downloading jEdit is simple, but that getting and installing the plug-ins to customize it for individual use can be a complex process:*
*"Beware that a full setup requires a series of downloads, and that this process can take time."[10]*

*Writing in December 2011, reviewer Rares Aioanei praised jEdit's versatility, stating "jEdit's design allows you to use it as a simple editor, but also use it as an IDE and expand its functionality via plugins*
*so that it becomes exactly what you want it to be for the task or language at hand." but also adding that "jEdit is not, however, an IDE with everything but the Christmas tree, like Eclipse or Microsoft Visual Studio.*
*Rather, it's a compact application for editing code, providing practical tools along with basic IDE features."[11]*

*DESCRIPTION*
*Artificial intelligence (AI) has been fluidly defined, but, broadly, definitions have focused on either replicating human-like thought/behavior or creating rational thought/behavior. Historically, the goal of AI was to create a general intelligence (GI). when it became clear GI was out of reach, AI split into numerous subfields like computer vision and natural language processing, to name a few. Within the last decade, each of these subfields has achieved breakthrough results, facilitating AI applications like autonomous vehicles and home assistants. These successes have caused some to speculate the replication of GI in a computer is just around the corner. However, the reality of the situation is quite the opposite: these applications, while groundbreaking, are not indicative progress toward the original goals of intelligent thought or action. In this course, we will discuss the limits and potentials of various approaches to modern AI applications, as well as opportunities to overcome these limitations. In this vein, this course will be a tour of both classic ideas and state-of-the-art methods in AI.*

*LEARNING OUTCOMES*
*Upon successful completion of the course, students will be able to:*

*Understand the philosophical underpinnings of the field.*

*Discuss which problems that are solvable with today's AI algorithms and others that require novel solutions.*
*Deploy general search algorithms that can be applied to a wide variety of tasks.*
*Formulate decision making processes that can be used for planning and classification purposes.*
*Build intelligent agents that perform simple tasks in an autonomous fashion.*
*Learn task-specific models from large collections of labeled training data samples using algorithms that are optimized using numeric solvers.*
*Analyze the successes and limitations of an AI system, with an emphasis on real-world applications and ethics.*