Assignment 4 : Refactoring

Team Members :

Prashant K Thakur, Bibek Raj Shrestha, Sadaf Ghaffari

1. Manual Refactoring

1.1. jEdit

1.1.1. Smell Implementation Code (Empty catch clause in org.jedit.options.OptionGroupPane)

The first manual refactoring was done for the class “OptionGroupPane”. The code smell we try to address is “Implementation Code Smell” as there is Empty catch clause in the code.

```
        else if (obj instanceof String)
        {
            String pane = (String) obj;
            if (pane.equals(name) || name == null)
            {
                path.add(pane);
                TreePath treePath = new TreePath(path.toArray());
                paneTree.scrollPathToVisible(treePath);
                try {
                    paneTree.setSelectionPath(treePath);
                }
                catch (NullPointerException npe) {}

                return true;
            }
        }
    }
```

Figure 1: Implementation Code Smell with Empty catch clause

Description on class functionality and refactoring

The snapshot (figure 1) is from the “OptionGroupPane” class under method “selectPane” which is used to create a panel for global options and plugin options as shown in the figure 2. Only two class within the same package is dependant on this class and it is not directly used which makes it a segregated class. If we analyze the usage of the entire package (figure 4) of which “OptionGroupPane” is part of, we see the module has less effect. Since most of the methods are private and not exposed for customization for JUnit test, we try to compile the program and see the behavior in the GUI to check for the functionality. In conclusion, we took for the validation of change is manual. Additionally, we can see that there is less effect of the changes we did to the program behavior as we only logged the exception rather than silently swallowing the exception (which is of course a bad programming practice). While testing the GUI, we went

over different functionality available to simulate the “NullPointerException” but we were not able to simulate it.

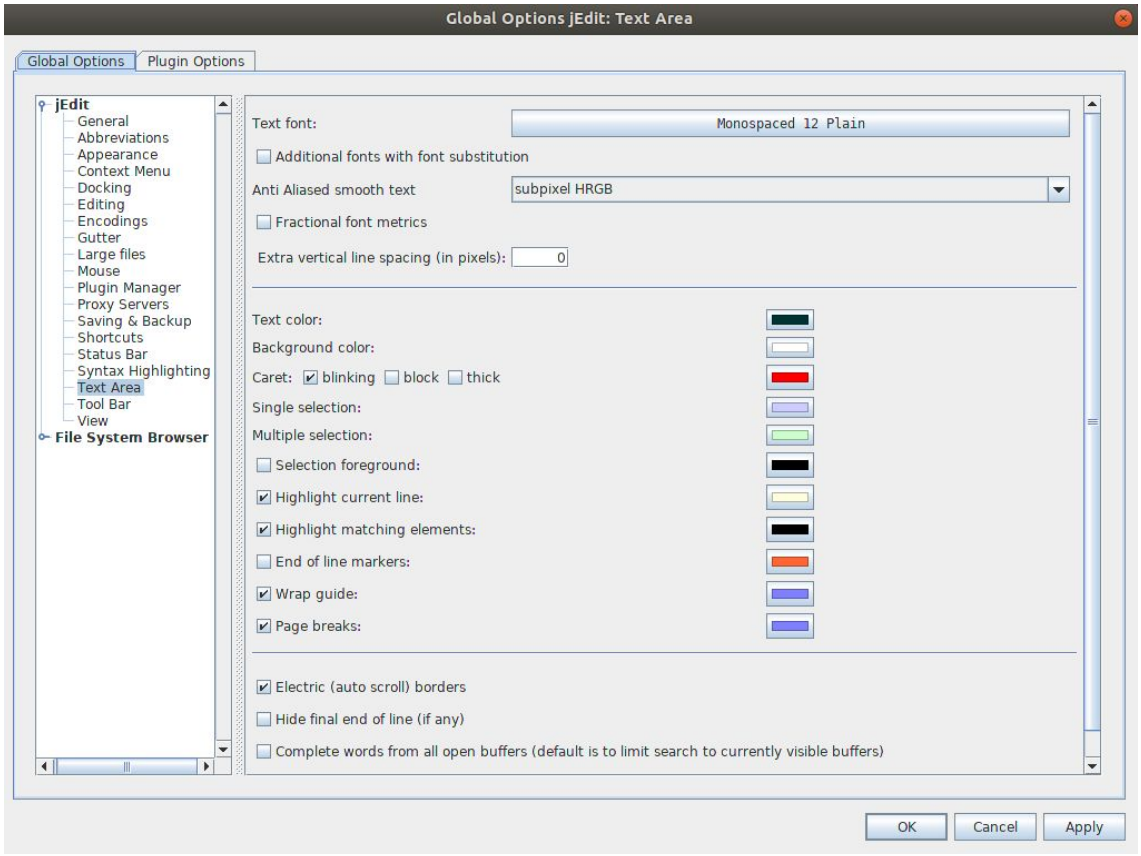

Figure 2: GUI functionality relaying on OptionGroupPane

```
        else if (obj instanceof String)
        {
            String pane = (String) obj;
            if (pane.equals(name) || name == null)
            {
                path.add(pane);
                TreePath treePath = new TreePath(path.toArray());
                paneTree.scrollPathToVisible(treePath);
                try {
                    paneTree.setSelectionPath(treePath);
                }
                catch (NullPointerException npe) {
                    Log.log(Log.WARNING, source: this, message: "Received null pointer: "+
                            npe.getMessage());
                }

                return true;
            }
        }
    }
```

Figure 3: Added logging to at least know if something goes wrong

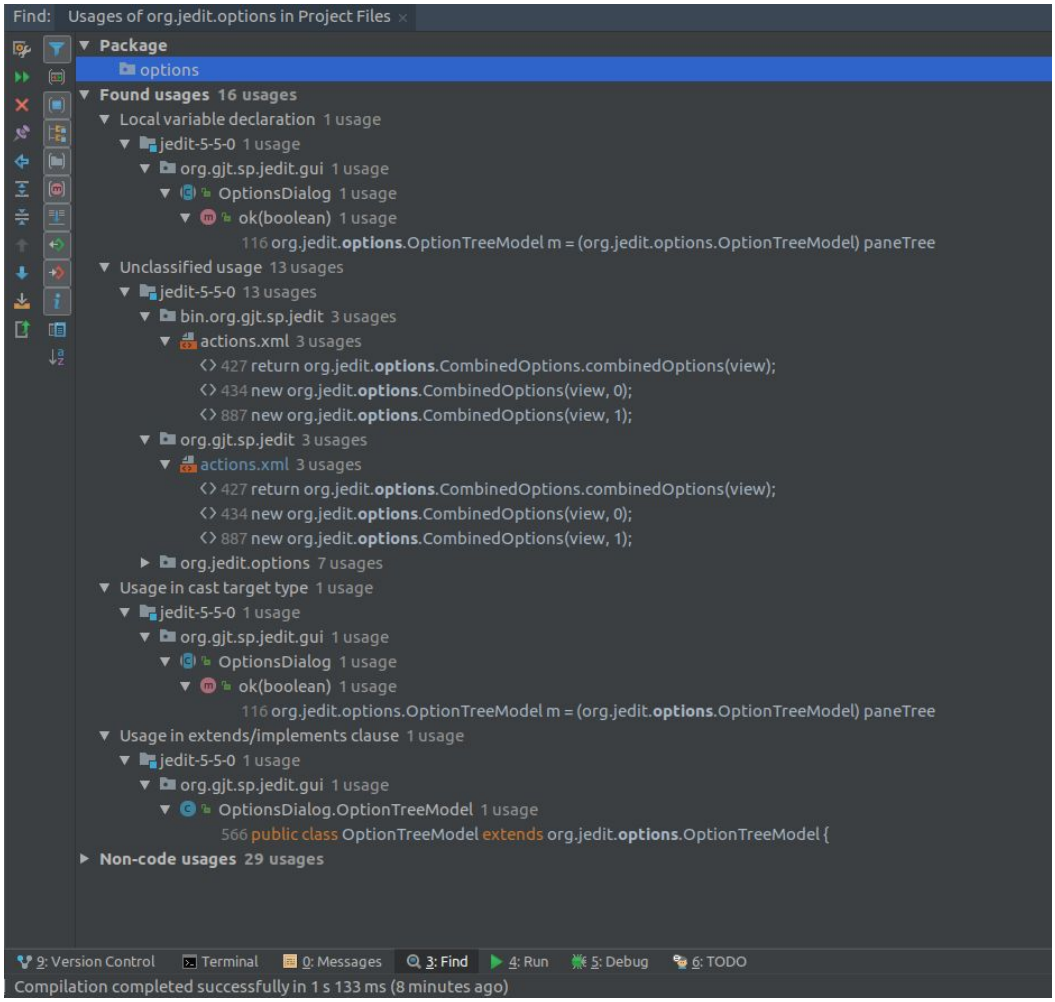

Figure 4: Usage of package org.jedit.options in the entire project.

1.1.2. Smell (Feature Envy in org.gjt.sp.jedit.textarea.TextAreaMouseHandler)

Description on class functionality and refactoring

Feature envy occurs in a code when a method calls methods on another class multiple times. This often indicates that the functionality being used is located in the wrong class. The method “doSingleDrag” in “TextAreaMouseHandler” uses TextArea methods multiple times so the method functionality has been shifted to class “TextArea”. Figure 5 shows the method change made to the method by calling the method present in “TextArea”. Since the method is calling textArea multiple times, it makes more sense to keep the method functionality inside TextArea so that the code can be clean. Figure 6 shows the implementation of “singleDragHelper” in “TextArea” class.

In order to test the program, we compiled the code to check if there is any compilation error. Since the method “doSingleDrag” still exists in the original class, any other class or method calling this functionality would still be functional. Also the code compiled assuring that the code change does not have any adverse effect. In order to check the functionality, we ran the compiled program and tried to select (single mouse drag) to check if the program generates any error or crashes. After multiple tests we confirmed that the program has no change in its behavior.

```
    //{{{ doSingleDrag() method
    private void doSingleDrag(MouseEvent evt)
    {
        dragged = true;
//...
        textArea.singleDragHelper(evt, quickCopyDrag, control, dragStart);
    } //}}}
```

Figure 5: Method functionality shifted to TextArea

```
754     void singleDragHelper(MouseEvent evt, boolean quickCopyDrag, boolean control, int dragStart ){
755 //      Log.log(Log.WARNING, this, "PKT::::drag...");
756         TextAreaPainter painter = getPainter();
757
758         int x = evt.getX();
759         int y = evt.getY();
760         if(y < 0)
761             y = 0;
762         else if(y >= painter.getHeight())
763             y = painter.getHeight() - 1;
764
765         int dot = xyToOffset(x,y,
766                 round: (!painter.isBlockCaretEnabled()
767                         && !isOverwriteEnabled())
768                         || quickCopyDrag);
769         int dotLine = getLineOfOffset(dot);
770         int extraEndVirt = 0;
```

Figure 6: Implementation of singleDragHelper method in TextArea

<u>Similar change for another method:</u>

The method “setText()” in “TextArea” was calling buffer for all its functionality so we moved this method to “JEditBuffer” class so that the functionality can be implemented in native class rather than calling it from another class. Figure 7 shows the code section which was commented as it was calling buffer methods to implement the functionality. In order to fix this code smell we moved this method to the original class as shown in figure 8. The correctness of the program was tested manually by compiling the program and checking if the text area functionality works as expected. There were no errors generated by manipulating text areas in jEdit.

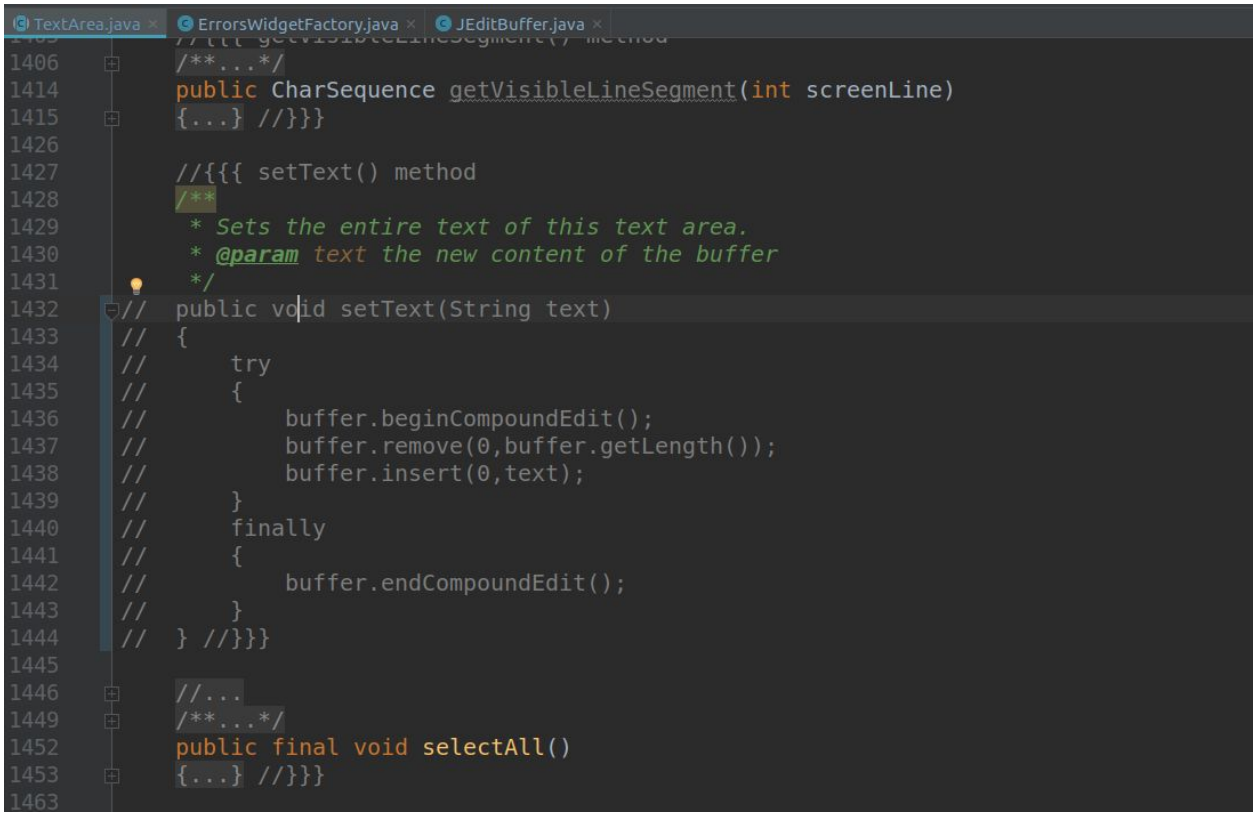

Figure 7: Commented setText in TextArea class

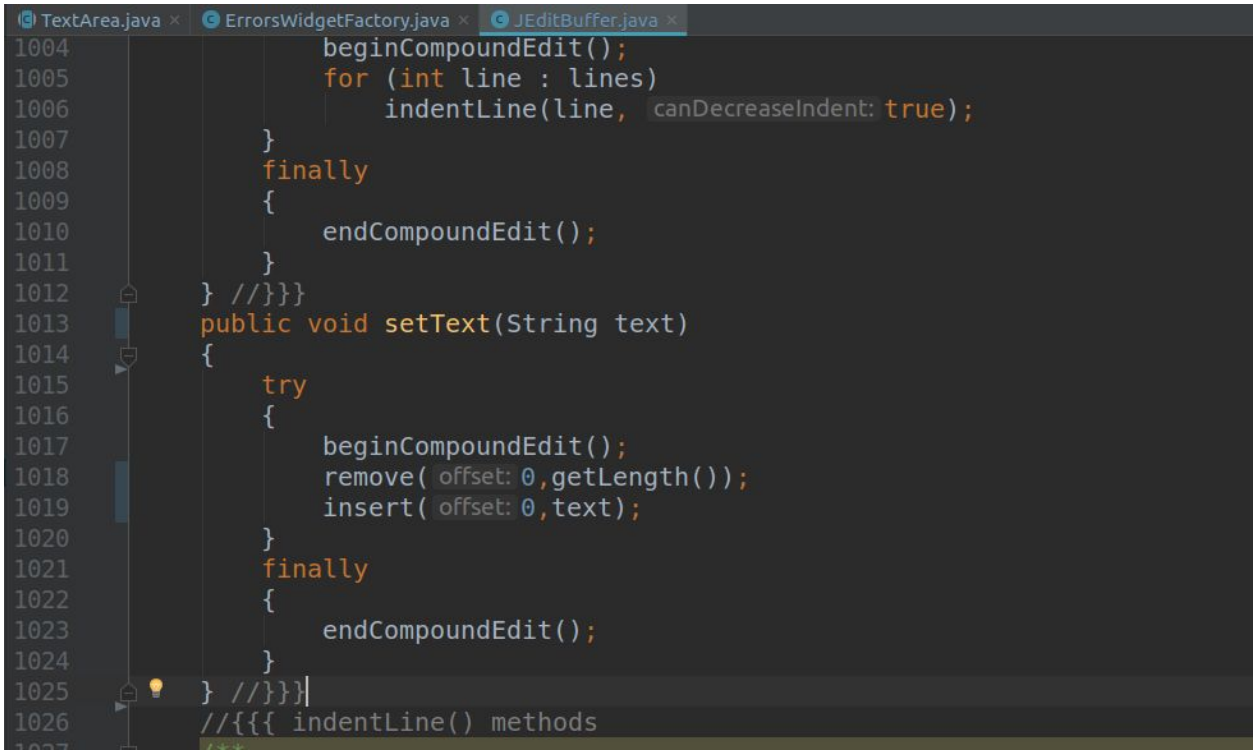

Figure 8: Moved setText method to JEditBuffer class

1.2. PDFsam

1.2.1: Long Method

The method `private void initTopSectionContextMenu(ContextMenu contextMenu, boolean hasRanges)` in the class "pdfsamfx/src/main/java/org/pdfsam/ui/selection/multiple/SelectionTable.java" shows Long Method bad smell. It contains more than 20 lines of code. So, we refactored this bad smell by creating an private method named `hasRangesRefactored` within the SelectionTable class and moved a part of the functionality of original method `initTopSectionContextMenu`. In doing so, we decreased the size of the two methods to less than 10 lines of code.

The task of this newly created method: `hasRangesRefactored` is to perform steps such as calculate setPageRangesItem, etc if the hasRages is true.

```
153
154     private void initTopSectionContextMenu(ContextMenu contextMenu, boolean hasRanges) {
155         MenuItem setDestinationItem = createMenuItem(DefaultI18nContext.getInstance().i18n("Set destination"),
156                 MaterialDesignIcon.AIRPLANE_LANDING);
157         setDestinationItem.setOnAction(e -> eventStudio().broadcast(
158                 requestDestination(getSelectionModel().getSelectedItem().descriptor().getFile(), getOwnerModule()),
159                 getOwnerModule()));
160         setDestinationItem.setAccelerator(new KeyCodeCombination(KeyCode.O, KeyCombination.ALT_DOWN));
161
162         selectionChangedConsumer = e -> setDestinationItem.setDisable(!e.isSingleSelection());
163         contextMenu.getItems().add(setDestinationItem);
164
165         if (hasRanges) {
166             MenuItem setPageRangesItem = createMenuItem(DefaultI18nContext.getInstance().i18n("Set as range for all"),
167                     MaterialDesignIcon.FORMAT_INDENT_INCREASE);
168             setPageRangesItem.setOnAction(e -> eventStudio().broadcast(
169                     new SetPageRangesRequest(getSelectionModel().getSelectedItem().pageSelection.get()),
170                     getOwnerModule()));
171             setPageRangesItem.setAccelerator(new KeyCodeCombination(KeyCode.R, KeyCombination.CONTROL_DOWN));
172             selectionChangedConsumer = selectionChangedConsumer
173                     .andThen(e -> setPageRangesItem.setDisable(!e.isSingleSelection()));
174             contextMenu.getItems().add(setPageRangesItem);
175         }
176         contextMenu.getItems().add(new SeparatorMenuItem());
177     }
```

Figure 9:The code snippet showing code smell with long method (line 154 -177)

```
153
154     private void initTopSectionContextMenu(ContextMenu contextMenu, boolean hasRanges) {
155         MenuItem setDestinationItem = createMenuItem(DefaultI18nContext.getInstance().i18n("Set destination"),
156                 MaterialDesignIcon.AIRPLANE_LANDING);
157         setDestinationItem.setOnAction(e -> eventStudio().broadcast(
158                 requestDestination(getSelectionModel().getSelectedItem().descriptor().getFile(), getOwnerModule()),
159                 getOwnerModule()));
160         setDestinationItem.setAccelerator(new KeyCodeCombination(KeyCode.O, KeyCombination.ALT_DOWN));
161
162         selectionChangedConsumer = e -> setDestinationItem.setDisable(!e.isSingleSelection());
163         contextMenu.getItems().add(setDestinationItem);
164
165         if (hasRanges) {
166             hasRangesRefactored(contextMenu);
167         }
168         contextMenu.getItems().add(new SeparatorMenuItem());
169     }
170
171     private void hasRangesRefactored(ContextMenu contextMenu){
172
173         MenuItem setPageRangesItem = createMenuItem(DefaultI18nContext.getInstance().i18n("Set as range for all"),
174                 MaterialDesignIcon.FORMAT_INDENT_INCREASE);
175         setPageRangesItem.setOnAction(e -> eventStudio().broadcast(
176                 new SetPageRangesRequest(getSelectionModel().getSelectedItem().pageSelection.get()),
177                 getOwnerModule()));
178         setPageRangesItem.setAccelerator(new KeyCodeCombination(KeyCode.R, KeyCombination.CONTROL_DOWN));
179         selectionChangedConsumer = selectionChangedConsumer
180                 .andThen(e -> setPageRangesItem.setDisable(!e.isSingleSelection()));
181         contextMenu.getItems().add(setPageRangesItem);
182     }
183
```

Figure 10: Refactored code to remove long method

1.2.2: Magic Number (from Assignment 3)

The class MergeSelectionPane contains magic number so the type of refactoring implemented in this part was “Extract Constant”. The hard-coded number within the code fragment had been replaced with symbolic constant. The constant is set as final static class variable named `BASE_SIZE`.

```java
/**
 * Selection panel for the merge module.
 *
 * @author Andrea Vacondio
 *
 */
public class MergeSelectionPane extends MultipleSelectionPane
        implements TaskParametersBuildStep<MergeParametersBuilder> {
    private static final Logger LOG = LoggerFactory.getLogger(MergeSelectionPane.class);

    public MergeSelectionPane(String ownerModule) {
        super(ownerModule, true, true,
                new SelectionTableColumn<?>[] { new LoadingColumn(ownerModule), FileColumn.NAME, LongColumn.SIZE,
                        IntColumn.PAGES, LongColumn.LAST_MODIFIED, new PageRangesColumn(DefaultI18nContext.getInstance()
                                .i18n("Double click to set pages you want to merge (ex: 2 or 5-23 or 2,5-7,12-)")) });
    }

    @Override
    public void apply(MergeParametersBuilder builder, Consumer<String> onError) {
        try {
            table().getItems().stream().filter(s -> !Objects.equals("0", trim(s.pageSelection.get())))
                    .map(i -> new PdfMergeInput(i.descriptor().toPdfFileSource(), i.toPageRangeSet()))
                    .forEach(builder::addInput);
            if (!builder.hasInput()) {
                onError.accept(DefaultI18nContext.getInstance().i18n("No PDF document has been selected"));
            }
        } catch (ConversionException e) {
            LOG.error(e.getMessage());
            onError.accept(e.getMessage());
        }
    }
}
```

Figure 11: The class that contains hard-coded constant (Code Smell with Magic Number)

```java
/**
 * Selection panel for the merge module.
 *
 * @author Andrea Vacondio
 *
 */
public class MergeSelectionPane extends MultipleSelectionPane
        implements TaskParametersBuildStep<MergeParametersBuilder> {
    private static final Logger LOG = LoggerFactory.getLogger(MergeSelectionPane.class);

    private static final String BASE_SIZE = "0";

    public MergeSelectionPane(String ownerModule) {
        super(ownerModule, true, true,
                new SelectionTableColumn<?>[] { new LoadingColumn(ownerModule), FileColumn.NAME, LongColumn.SIZE,
                        IntColumn.PAGES, LongColumn.LAST_MODIFIED, new PageRangesColumn(DefaultI18nContext.getInstance()
                                .i18n("Double click to set pages you want to merge (ex: 2 or 5-23 or 2,5-7,12-)")) });
    }

    @Override
    public void apply(MergeParametersBuilder builder, Consumer<String> onError) {
        try {
            table().getItems().stream().filter(s -> !Objects.equals(BASE_SIZE, trim(s.pageSelection.get())))
                    .map(i -> new PdfMergeInput(i.descriptor().toPdfFileSource(), i.toPageRangeSet()))
                    .forEach(builder::addInput);
            if (!builder.hasInput()) {
                onError.accept(DefaultI18nContext.getInstance().i18n("No PDF document has been selected"));
            }
        } catch (ConversionException e) {
            LOG.error(e.getMessage());
            onError.accept(e.getMessage());
        }
    }
}
```

Figure 12: Implementing Extract Constant instead of hard-coded value (Refactoring Code Smell with Magic Number)

2. Automatic Refactoring

Utilizing Eclipse's refactoring tool, we have done the automated refactoring based on the availability of refactoring functions in the tool.

2.1. jEdit

The first smell refactored in jEdit, was the "Complex Conditional" in the "read" method of "Native2ASCIIEncoding" class (jedit-5-5-0/org/jedit/io/Native2ASCIIEncoding.java). In this refactoring, we only used Eclipse's support. The type of refactoring implemented in this part was "Extract Method". The "if" condition and "nested if" had been extracted so that the method would be cleaned up from lengthy, cluttered and over-complicated statements and expressions. As a result of this change, that part of code would be more readable. After running the *Designite for Java,* we made sure the smell was removed. Besides, after each change, we confirmed that our change did not affect the component's behavior.

The second smell refactored in jEdit, was the "Complex Method" in "valueChanged" method "OptionGroupPane" class (jedit-5-5-0/org/jedit/options/OptionGroupPane.java). In this refactoring, we only used Eclipse's support. The type of refactoring implemented in this part was "Extract Method". This method had lots of "if" and "else if" conditions. The inner ones and those could be independent, had been extracted. As a result of this change, that part of code would be more readable. After running the *Designite for Java*, we made sure the smell was removed. Besides, after each change, we confirmed that our change did not affect the component's behavior.

2.2. PDFsam

The first smell refactored in PDFsam, was the "Magic Number" in the "MergeOptionsPane" method of "MergeSelectionPane" class (pdfsam/pdfsam-merge/src/main/java/org/pdfsam/merge/MergeSelectionPane.java). In this refactoring, we only used Eclipse's support. The type of refactoring implemented in this part was "Extract Constant". The numbers within the code fragment had been replaced with symbolic constants. After running the *Designite for Java,* we made sure the smell was removed. Besides, after each change, we confirmed that our change did not affect the component's behavior by compiling the PDFsam with its own implemented test suites.

The second smell refactored in PDFsam, was the "Data Class" in the "AddNotificationRequestEvent"(pdfsam/pdfsam-fx/src/main/java/org/pdfsam/ui/notification/AddNotificationRequestEvent.java)

). In this refactoring, we only used Eclipse's support. The type of refactoring implemented in this part was "Encapsulate Field". The class was exposing a significant amount of data in its public interface, either via public attributes or public accessor methods. Utilizing "Encapsulate Field", we did hide them from direct access and require that access be performed via getters and setters. After running the *InCode*, we made sure the smell was removed. Besides, after each change, we confirmed that our change did not affect the component's behavior by compiling the PDFsam with its own implemented test suites.

3. Conclusion

The manual refactoring was difficult to approach as we have to think of applying the changes and there is a high chance we might introduce some new bug. Additionally, we needed to be sure of how to solve the smell with some other ways of implementation like creation of new class, method etc. However, it gives better understanding of the source code and it also gives us an opportunity to add documentation so that it can be helpful in future. The Automated refactoring was relatively easy to make. We needed to locate where the refactoring should had happened and after that the tool was taking care of refactoring and if there had been any chance of mistakes the tool itself would give us warning.