

CS 581A3 Software Maintenance & Evolution  
Spring 2018  
Assignment 4: Refactoring OSS

## Contributions

We divided the work in such a way that each person gets to know both the software and gets a feel of both types of refactoring (manual and automated). Here is a breakup of the tasks:

<i>Task</i>	<b>jEdit</b>	<b>PDFSam</b>
<i>Automated Refactoring</i>	Abhimanyu	Gururaj
<i>Manual Refactoring</i>	Gururaj	Abhimanyu

### 1.1 jEdit

In this part of the assignment, we have performed refactoring operations on the smelly code which we had identified in the previous assignment.

#### Automated Refactoring for jEdit

For the automated refactoring, we have used either the suggestions from Eclipse plugins like JDeodorant or Eclipse's own refactoring tools.

#### 1. Smell: God Class

We had identified the class “org/gjt/sp/jedit/textarea/TextArea.java” inflicted with God Class smell. This class has 270 methods and 62 attributes and thus is a very large class.

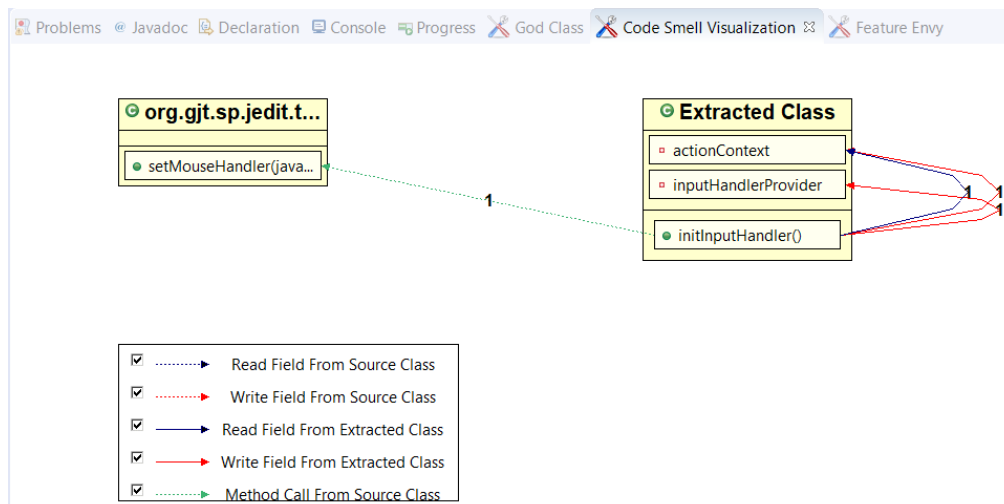


Figure 1. Extracted class suggestion from Automated Refactoring of a God Class

**Refactoring Details:** In this refactoring, we have used the suggested refactoring operations from the JDeodorant plugin in Eclipse to get rid of this smell. The refactoring took a bunch of methods from the `TextArea` class and carved out new classes with those methods. This reduced the size of the `TextArea`

class and got rid of the God Class smell. Figure 1 shows one of the extracted classes `TextAreaInitInputHandlerProduct.java`

For this refactoring we did not have to make any additional changes other than those performed by JDeodorant.

**Rationale:** The class `TextArea` has 270 methods in the original codebase, and as per the definition of a God Class, this class is a God Class. Therefore, we have chosen to refactor this class by extracting classes with some of the methods of the original class while keeping the behavior unchanged.

## 2. Smell: Feature Envy

The `getSelectionPivotCaret` method in the class “`org/gjt/sp/jedit/textarea/TextAreaMouseListener.java`” was identified as having Feature Envy bad smell.

**Refactoring Details:** In this case also, we have used the refactoring operations from JDeodorant.

Refactoring Type	Source Entity	Target Class	Source/Target acce ^
Move Method	<code>org.gjt.sp.jedit.textarea.TextArea:deletePrevCodePoint(int):void</code>	<code>org.gjt.sp.jedit.buffer.JEditBuffer</code>	1/2
Move Method	<code>org.gjt.sp.jedit.textarea.TextAreaMouseListener:getSelectionPivotCaret():int</code>	<code>org.gjt.sp.jedit.textarea.TextArea</code>	1/2
Move Method	<code>org.jedit.keymap.EmacsUtil:charAt(int):char</code>	<code>org.gjt.sp.jedit.buffer.JEditBuffer</code>	1/2
Move Method	<code>installer.TarEntry:writeEntryHeader(byte[]):void</code>	<code>installer.TarHeader</code>	2/13

Figure 2. Feature Envy smell detected by JDeodorant

Figure 2 shows the Feature Envy smell in the class `TextAreaMouseListener` and JDeodorant suggests moving the culprit method `getSelectionPivotCaret` to the class `TextArea`.

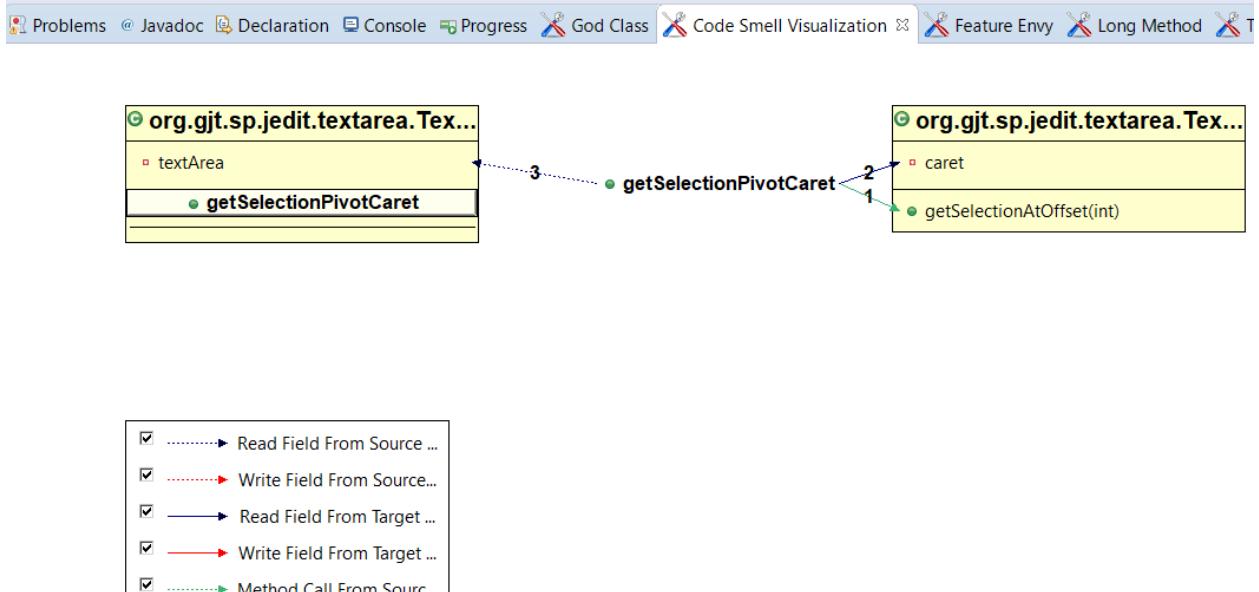


Figure 3. Suggested refactoring from JDeodorant

After performing the suggested refactoring, and running JDeodorant again, we observed that the Feature Envy smell got eliminated. We did not have to make any changes do manually, apart from the changes performed with JDeodorant’s support.

**Rationale:** The method uses attributes of the class `TextArea` and does not use any attributes from its own class. This indicates that this method should be moved to the `TextArea.java` class. Therefore, we decided to move this method to the `TextArea.java` class to get rid of the Feature Envy bad smell.

## Manual Refactoring for jEdit

In manual refactoring process, we analyzed the code with a careful consideration to the bad smells and then removed the bad smells by manually refactoring the necessary parts without using any plugins or Eclipse’s tools.

### 1. Smell: God Class

We identified the class “`org/gjt/sp/util/StringModel.java`” inflicted with God Class bad smell. This is a God Class because it tends to centralize code base and therefore the intelligence of the software into a single class. This class has 5 methods which perform too much of work on their own causing it to be a God Class.

Refactoring Type	Source Class/General Concept	Extractable Concept	Source/Extracted accessed members
▼	org.gjt.sp.util.StringModel		0/1
▼	[text, listen]		
Extract Class	.. .. .	[text, listen]	0/1

Figure 4. God Class smell detected by JDeodorant

**Refactoring Details:** We used extract class approach to extract some of the methods from this large class to another newly created class called “`org/gjt/sp/util/StringModelExtracted.java`”. This new class contains two extracted methods from the original class “`org/gjt/sp/util/StringModel.java`”.

This extracted class performs the same functionality as the original class through following methods:

```
public void removeTextListener(TextListener text1)
public void fireTextChanged(StringModel stringModel)
```

Therefore, this refactoring operation delegates some of the work of the original class to the newly created class while keeping the behavior of the software unchanged. This removed the God Class bad smell from this class.

**Rationale:** The class `StringModel` has 5 methods in the original codebase. According to the definition of a God Class, this number constitutes to make it a God Class. Therefore, we decided to create another class that extracts some methods while keeping the behavior unchanged. We extracted two methods into the new class as described above and added one more method `public LinkedList<TextListener> getListeners()`.

**Manual Changes:** 1. Created a new class `StringModelExtracted.java`

2. Extracted method into extracted class

3. Created an instance of this extracted class in the original class `StringModel` and called the methods of this instance in place of the corresponding original methods.

## 2. Smell: Feature Envy

The method `public String getCurrentEditMode()` from the class “`org/gjt/sp/jedit/gui/DockingLayoutManager.java`” was identified to exhibit Feature Envy bad smell because the method behaves as if it is more interested in the data and attributes of other classes than that of their own class. In this case, the other class is `View.java`. This method uses the `Buffer` attribute (`Buffer buffer = view.getBuffer()`) of the view instance of the `View.java` class.

Refactoring Type	Source Entity	Target Class	Source/Target accessed n
Move Method	<code>org.gjt.sp.jedit.View:closeDuplicateBuffers(org.gjt.sp.jedit.msg.EditPaneUpdate):void</code>	<code>org.gjt.sp.jedit.msg.EditPaneUpdate</code>	0/1
Move Method	<code>org.gjt.sp.jedit.browser.BrowserCommandsMenu:createPluginMenu(org.gjt.sp.jedit.browser.VFSBrowser):javax.s...</code>	<code>org.gjt.sp.jedit.browser.VFSBrowser</code>	0/1
Move Method	<code>org.gjt.sp.jedit.bsh.BSHAllocationExpression:constructObject(java.lang.Class, java.lang.Object[], org.gjt.sp.jedit.b...</code>	<code>org.gjt.sp.jedit.bsh.CallStack</code>	0/1
Move Method	<code>org.gjt.sp.jedit.gui.DockingLayoutManager:getCurrentEditMode(org.gjt.sp.jedit.View):java.lang.String</code>	<code>org.gjt.sp.jedit.View</code>	0/1

Figure 5. Feature Envy smell detected by JDeodorant

**Refactoring Details:** We decided to move/relocate method approach to relocate the method `getCurrentEditMode()` to the `View.java` class because this method uses attributes of the `View.java` class. Therefore, relocating this method does not change the functionality and behavior of the software while removing Feature Envy bad smell.

**Rationale:** This method uses attributes of the class `View.java` and does not use any attributes from its own class. This is a sign that this method might have been misplaced and it should be moved to the `View.java` class, attributes of which this method accesses. Therefore, we decided to move this method to `View.java` to get rid of the Feature Envy bad smell.

**Manual Changes:** 1. Removed the method `private String getCurrentEditMode(View view)` from the class `DockingLayoutManager.java` and added it to class `View.java` as `public String getCurrentEditMode()`.

2. Changed the `String mode` attribute in the method `public static void loadCurrentModeLayout(View view)` from `String mode = instance.getCurrentEditMode(view)` to `String mode = view.getCurrentEditMode()` to reflect the relocated method.

## 1.2 PDFSam

### Automated Refactoring for PDFSam

For the automated refactoring, we have used either the suggestions from Eclipse plugins like JDeodorant or Eclipse’s own refactoring tools.

#### 1. Smell: God Class

We observed that the class “`pdfsam-  
fx/src/main/java/org/pdfsam/ui/selection/multiple/SelectionChangedEvent.java`” is afflicted with the God Class smell. This class has 7 methods and is a very large class.

Refactoring Type	Source Class/General Concept	Extractable Concept	Source/Extracted accessed members
>	org.pdfsam.ui.commons.ValidableTextField		0/1
>	org.pdfsam.ui.io.BrowsableFileFieldTest		0/1
∨	org.pdfsam.ui.selection.multiple.SelectionChangedEvent		1/4
	[select]		
Extract Class	[select]	[select]	1/4
Extract Class	[select]	[select]	2/3

Figure 6. God Class smell detected by JDeodorant

**Refactoring Details:** In this refactoring, we have used the suggested refactoring operations from the JDeodorant plugin in Eclipse to get rid of this smell. In this refactoring operation some of the methods were extracted to a new class, `SelectionChangedEventProduct.java`

We did not have to make any changes do manually, apart from the changes performed with JDeodorant's support.

**Rationale:** The class `SelectionChangedEvent` has 7 methods in the original codebase, and it classifies as a God Class. Therefore, we have chosen to refactor this class by extracting some functionality out of this class as an extracted class.

## 2. Smell: Type Checking

The class `pdfsam-fx/src/main/java/org/pdfsam/ui/selection/multiple/SelectionChangedEventProduct.java` was identified as having the Type Checking code smell. In this case also, we have used the refactoring operations from JDeodorant.

Refactoring Type	Type Checking Method	Abstract Method Name	System-Level ...	Class-Level Oc...	Average #stat...	Rate it!
∨	constant variables: [BOTTOM, DOWN, TOP]		1	1.0	1.0	
Replace Type Code with State/Strategy	org.pdfsam.ui.selection.multiple.SelectionChangedEventProduct:public boolean canMo...	canMove		1	1.0	
∨	constant variables: [SAVE]		1	1.0	1.0	
Replace Type Code with State/Strategy	org.pdfsam.ui.io.RememberingLatestFileChooserWrapper:public java.io.File showDialo...	showDialog		1	1.0	

Figure 7. Type Checking smell detected by JDeodorant

**Refactoring Details:** In this refactoring, we have used the suggested refactoring operations from the JDeodorant plugin in Eclipse to get rid of this smell. In this refactoring operation the switch case method in the class has been extracted as a bunch of classes according to the clauses in the switch-case statements. The classes have been named after the switch-case statements, `Top.java`, `Bottom.java`, `Down.java` and `Type.java`.

For this refactoring we did not have to make any additional changes other than those performed by JDeodorant.

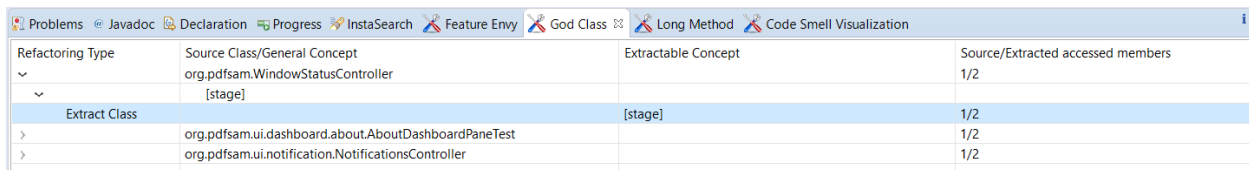
**Rationale:** The class `SelectionChangedEventProduct` has a switch-case method, which may cause code duplication if similar switch case statements are found elsewhere in the code base. And also, if one changes or adds a switch-case clause, one has to do the similar change everywhere else, which makes maintaining code very difficult. Therefore, we have chosen to refactor this class by extracting the switch-case method in separate classes. Now if any class needs this switch-case block it can simply call the methods from these newly extracted classes.

## Manual Refactoring for PDFSam

In manual refactoring process, we analyzed the code with a careful consideration to the bad smells and then removed the bad smells by manually refactoring the necessary parts without using any plugins such as JDeodorant or Eclipse's tools.

### 1. Smell: God Class

The class `pdfsam-gui/src/main/java/org/pdfsam/WindowStatusController.java` shows a God Class bad smell. This is a God Class because it tends to centralize a lot of code work in a single class. This class has 6 methods that perform too much of work on their own causing it to be a God Class.



Refactoring Type	Source Class/General Concept	Extractable Concept	Source/Extracted accessed members
▼	org.pdfsam.WindowStatusController		1/2
▼	[stage]		
Extract Class		[stage]	1/2
>	org.pdfsam.ui.dashboard.about.AboutDashboardPaneTest		1/2
>	org.pdfsam.ui.notification.NotificationsController		1/2

Figure 8. God Class smell detected by JDeodorant

**Refactoring Details:** For this case as well, we used extract class approach to extract some of the methods from this large class to another newly created class that we named “`pdfsam-gui/src/main/java/org/pdfsam/WindowStatusControllerExtracted.java`”. This new class contains four extracted methods from the original class “`pdfsam-gui/src/main/java/org/pdfsam/WindowStatusController.java`”.

This extracted class performs the same functionality as the original class through following methods:

```
public void defaultStageStatus()  
  
public void setStage(Stage stage, WindowStatusController  
windowStatusController)  
  
public boolean isNotMac()  
  
public void restore(StageStatus latestStageStatus)
```

Therefore, this refactoring operation delegates some of the work of the original class to the newly created class while keeping the behavior of the software unchanged. This removed the God Class bad smell from this class.

**Rationale:** The class `WindowStatusController.java` has 6 methods in the original codebase. According to the definition of a God Class, this number constitutes to make it a God Class. Therefore, we decided to create another class that extracts some methods while keeping the behavior unchanged. We extracted 3 methods into the new class as described above.

### Manual Changes:

1. Created a new class `WindowStatusControllerExtracted.java`.

2. Extracted four methods into extracted class.

3. Created an instance of this extracted class in the original class `WindowStatusController.java` and called the methods of this instance in place of the corresponding original methods.

## 2. Smell: Long Method

The method `private void initTopSectionContextMenu (ContextMenu contextMenu, boolean hasRanges)` in the class “`pdfsam-fx/src/main/java/org/pdfsam/ui/selection/multiple/SelectionTable.java`” shows Long Method bad smell because it has 20 lines of code. So, we removed this bad smell by creating another method in the same class that performs a part of the task that this method does.

```
151
152 private void initTopSectionContextMenu(ContextMenu contextMenu, boolean hasRanges) {
153     MenuItem setDestinationItem = createMenuItem(DefaultI18nContext.getInstance().i18n("Set destination"),
154         MaterialIcon.FLIGHT_LAND);
155     setDestinationItem.setOnAction(e -> eventStudio().broadcast(
156         requestDestination(getSelectionModel().getSelectedItem().descriptor().getFile(), getOwnerModule()),
157         getOwnerModule()));
158     setDestinationItem.setAccelerator(new KeyCodeCombination(KeyCode.O, KeyCodeCombination.ALT_DOWN));
159
160     selectionChangedConsumer = e -> setDestinationItem.setDisable(!e.isSingleSelection());
161     contextMenu.getItems().add(setDestinationItem);
162
163     if (hasRanges) {
164         MenuItem setPageRangesItem = createMenuItem(DefaultI18nContext.getInstance().i18n("Set as range for all"),
165             MaterialIcon.TOC);
166         setPageRangesItem.setOnAction(e -> eventStudio().broadcast(
167             new SetPageRangesRequest(getSelectionModel().getSelectedItem().pageSelection.get()),
168             getOwnerModule()));
169         setPageRangesItem.setAccelerator(new KeyCodeCombination(KeyCode.R, KeyCodeCombination.CONTROL_DOWN));
170         selectionChangedConsumer = selectionChangedConsumer
171             .andThen(e -> setPageRangesItem.setDisable(!e.isSingleSelection()));
172         contextMenu.getItems().add(setPageRangesItem);
173     }
174     contextMenu.getItems().add(new SeparatorMenuItem());
175 }

151
152 private void initTopSectionContextMenu(ContextMenu contextMenu, boolean hasRanges) {
153     MenuItem setDestinationItem = createMenuItem(DefaultI18nContext.getInstance().i18n("Set destination"),
154         MaterialIcon.FLIGHT_LAND);
155     setDestinationItem.setOnAction(e -> eventStudio().broadcast(
156         requestDestination(getSelectionModel().getSelectedItem().descriptor().getFile(), getOwnerModule()),
157         getOwnerModule()));
158     setDestinationItem.setAccelerator(new KeyCodeCombination(KeyCode.O, KeyCodeCombination.ALT_DOWN));
159
160     selectionChangedConsumer = e -> setDestinationItem.setDisable(!e.isSingleSelection());
161     contextMenu.getItems().add(setDestinationItem);
162
163     /* Extracted a method called hasRangesMethod() from this Long Method containing more than 20 lines of code */
164     if (hasRanges) hasRangesMethod(contextMenu);
165
166     contextMenu.getItems().add(new SeparatorMenuItem());
167 }

168
169 public void hasRangesMethod(ContextMenu contextMenu) {
170     MenuItem setPageRangesItem = createMenuItem(DefaultI18nContext.getInstance().i18n("Set as range for all"),
171         MaterialIcon.TOC);
172     setPageRangesItem.setOnAction(e -> eventStudio().broadcast(
173         new SetPageRangesRequest(getSelectionModel().getSelectedItem().pageSelection.get()),
174         getOwnerModule()));
175     setPageRangesItem.setAccelerator(new KeyCodeCombination(KeyCode.R, KeyCodeCombination.CONTROL_DOWN));
176     selectionChangedConsumer = selectionChangedConsumer
177         .andThen(e -> setPageRangesItem.setDisable(!e.isSingleSelection()));
178     contextMenu.getItems().add(setPageRangesItem);
179
180 }
181
```

Figure 9. Refactoring done to get rid of Long Method

**Refactoring Details:** To remove the long method bad smell, we create a method named as `public void hasRangesMethod(ContextMenu contextMenu)` in the same class. Functionality of this newly created method is to perform steps such as calculate `setPageRangesItem`, etc if the `hasRages` flag is true. This new method contains 10 lines of code and reduces the original number of lines in the long method to half.

**Rationale:** The method `initTopSectionContextMenu` contains 10 lines of code which is more than the threshold for a method to be a long method. Therefore, we decided to split the functionality of this method into two methods while keeping its overall behavior the same. We identified a conditional statement that contained major functionality of the original method. Therefore, we created another method that performed this conditional functionality given the attributes `ContextMenu contextMenu`.

**Manual Changes:** 1. Identify the part of the code that can be used to create another method.

2. Create a method called `hasRangesMethod()` that performs the conditional part of the original method.

3. Refactored the original method so that it calls this new method where the conditional part of the method was located before.

## Testing:

For this assignment we have employed manual testing for both JEdit and PDFSam.

### *JEdit*

Steps we followed:

1. Before refactoring, we built and ran the software. JEdit has some built-in tests which are run during the ant build. We noted the number of tests passed and total number of tests.
2. Opened a text file and used the software.
3. Identified the code to be refactored and performed the refactoring described in the previous section.
4. After refactoring, we again built and ran the software. We noted any build errors, and the number of built-in tests passed.
5. Opened the same text file and used the software to verify correct functionality.
6. As shown in Figure 10, there was no test case failed after refactoring, confirming correct behavior.

```
test:
[delete] Deleting directory C:\Users\abhim\eclipse-workspace\cs581A3-f18-a2-gitrams-jedit\build\test\raw-reports
[mkdir] Created dir: C:\Users\abhim\eclipse-workspace\cs581A3-f18-a2-gitrams-jedit\build\test\raw-reports
[junit] Running org.jedit.io.Native2ASCIIEncodingTest
[junit] Tests run: 108, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.938 sec
```

Figure 10. Built-in tests for JEdit, showing all tests passed

### *PDFSam*

Steps we followed:

1. Before refactoring, we built the software, making sure the tests option is selected in the Maven build. PDFSam has a full-featured test suite which tests the functionality during the build. We noted the number of tests passed and the total number of tests.



2. Ran the software and opened up a couple of PDF files. We performed some tasks with these PDF files like merging etc.
3. Identified the code to be refactored and performed the refactoring as described in the previous sections.
4. After refactoring, we built the software again. We tracked any build errors, and observed if any test failed during the build.
5. Opened the same PDF files as before and performed the exact same task, like merging. We carefully observed if there is any change in the behavior, and found that all the tests passed and the software behaved the same way as it did before refactoring.

## Manual vs. Automatic Refactoring:

For this assignment we performed both automated and manual refactoring. For automated refactoring we mostly used the Eclipse plugin JDeodorant. We found that doing automated refactoring was quite easy. It was as easy as clicking some buttons. All the intricate details of method names, class names, etc. are handled by the tool. We do not have to worry whether we have made all the required changes at every place in the entire codebase.

However, in the case of manual refactoring, we felt that it is a bit difficult to keep a track of all the changes that need to be done. And one must do the necessary import statements if needed. In case of small refactorings, it is not very difficult to do all the changes manually, but for large and complex refactorings the whole process becomes tough to manage.

On the other side, the manual refactoring has its benefits too. It allows a finer control over what is being changed and where. This could help in making sure that the refactored code is optimized and does not affect performance. And in some cases where tools do not offer any refactoring suggestions, it might be still possible to perform refactoring if we do it manually.