

Assignment 4: Code Smells & Refactoring

Team:

Team Name: Grim Reapers

Team Members: Pooria Taheri and Sushmitha Jayaramaiah

For both of the applications i.e., Jedit and PdfSam, we used the **JDeodorant** tool to identify the code smells. JDeodorant identifies bad smells/design problems in software and resolves them by applying appropriate refactoring. It identifies five types of bad smells, namely:

- Duplicated Code
- Feature Envy
- God Class
- Long Method
- Type Checking

In addition to identifying bad smells, it also provides visualization under the 'Code Smell Visualization view.'

1) Detecting and Analyzing Code Smells

Code smells are violations of fundamental software development principles that decrease the quality of code. In short, they express the symptoms of bad quality of code. There are many types of code smells defined. For this assignment, we are going to identify and discuss only those given by JDeodorant.

jEdit

For jEdit we identified three types of bad smells: Feature Envy, Duplicated Code, and God Class.

- 1. Feature Envy:** The whole point of OO programming is that object packages data and actions that manipulate that data. If a method in a class is focused on manipulating the data of other classes rather than its own, then this method is said to exhibit bad smell of type Feature Envy. Feature envy comes under a broader type of bad smell called Couplers. Couplers indicate a high coupling between classes, which makes code changing difficult due to the ripple effect.

In jEdit, JDeodorant identified many methods that exhibited feature envy code smell. Below is a picture displaying the same:

Refactoring Type	Source Entity	Target Class	Source/Target accessed members
Move Method	org.gjt.sp.jedit.bsh.BSHAllocationExpression::constructOb	org.gjt.sp.jedit.bsh.CallStack	0/1
Move Method	org.gjt.sp.jedit.gui.DockingLayoutManager::getCurrentEdit	org.gjt.sp.jedit.View	0/1
Move Method	org.gjt.sp.jedit.indent.BracketIndentRule::getBrackets(org	org.gjt.sp.jedit.buffer.JEditBuffer	0/1
Move Method	org.gjt.sp.jedit.pluginmgr.PluginDetailPanel::getDepends(c	org.gjt.sp.jedit.pluginmgr.ManagePanel.En	0/1
Move Method	org.jedit.keymap.EmacsUtil::makeBufferPropertyName(or	org.gjt.sp.jedit.Buffer	0/1
Move Method	org.gjt.sp.jedit.textarea.SelectionManager::getSelectionSt	org.gjt.sp.jedit.textarea.TextArea	1/8
Move Method	org.gjt.sp.jedit.textarea.TextArea::addExplicitFold(int, int,	org.gjt.sp.jedit.buffer.JEditBuffer	1/7
Move Method	com.ultramixer.jarbundler.JarBundler::addConfiguredDocu	com.ultramixer.jarbundler.DocumentType	1/5
Move Method	org.gjt.sp.jedit.textarea.ElasticTabstopsTabExpander::get	org.gjt.sp.jedit.textarea.ColumnBlock	1/5
Move Method	org.gjt.sp.jedit.textarea.TextArea::joinLineAt(int):void	org.gjt.sp.jedit.buffer.JEditBuffer	1/5
Move Method	org.gjt.sp.jedit.textarea.StructureMatcher.Highlight::getOf	org.gjt.sp.jedit.textarea.TextArea	1/4
Move Method	org.gjt.sp.jedit.textarea.TextArea::setText(java.lang.String	org.gjt.sp.jedit.buffer.JEditBuffer	1/4
Move Method	org.gjt.sp.jedit.textarea.TextAreaMouseHandler::getSelect	org.gjt.sp.jedit.textarea.TextArea	1/4
Move Method	com.ultramixer.jarbundler.JarBundler::setJvmversion(java.	com.ultramixer.jarbundler.AppBundleProp	1/3
Move Method	org.gjt.sp.jedit.PluginJAR::startPlugin():void	org.gjt.sp.jedit.EditPlugin	1/3
Move Method	org.gjt.sp.jedit.buffer.UndoManager.CompressedReplace::	org.gjt.sp.jedit.buffer.UndoManager.Repla	1/3
Move Method	org.gjt.sp.jedit.textarea.ExtensionManager::paintScreenLi	org.gjt.sp.jedit.textarea.TextArea	1/3
Move Method	org.gjt.sp.jedit.textarea.TextArea::getSelectedText(org.gjt	org.gjt.sp.jedit.textarea.Selection	1/3

Fig.: Results of Feature Envy on jEdit

The image has 4 columns as shown:

- Refactoring Type: The type of refactoring to be done.
- Source Entity: Source code where the code smell was identified.
- Target Class: If refactored, the target class where the method will be moved to.
- Source/Target accessed members: Count of members from the source accessing the target members.

For this code smell, we chose the *joinLineAt()* method from the *TextArea* class under *org.gjt.sp.jedit.textarea* package.

```
private void joinLineAt(int line)
{
    if (line >= buffer.getLineCount() - 1)
        return;
    int end = getLineEndOffset(line);
    CharSequence nextLineText = buffer.getLineSegment(line +
1);
    buffer.remove(end -
```

```

1,StandardUtilities.getLeadingWhiteSpace(
    nextLineText) + 1);
    if (nextLineText.length() != 0)
        buffer.insert(end - 1, " ");}

```

Analyzing the method code above, we see that the *joinLineAt()* method makes five calls to *JEditBuffer* class from package *org.gjt.sp.jedit.buffer.JEditBuffer*.

If we observe the if statement below, it's evident that data returned by the *buffer* instance of *JEditBuffer* class is being modified here to compare with the *line* parameter.

```

if (line >= buffer.getLineCount() - 1)

```

Similarly, the below line has a function *getLineEndOffset* in *TextArea* which is calling the *getLineEndOffset* method from *JEditBuffer* class. Again the data returned is being modified and sent as a parameter to the method *buffer.remove*.

```

int end = getLineEndOffset(line)

```

From the code smell visualization, we can see how the *joinLineAt()* method is communicating with the *JEditBuffer* class. *TextArea* is reading a field from the source as indicated by the arrow.

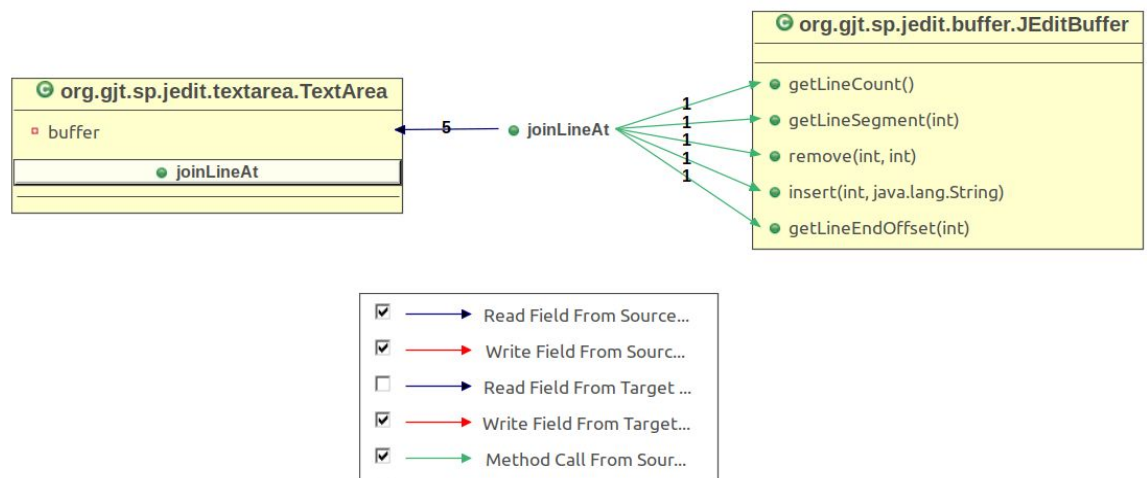


Fig.: Code Smell Visualization for Feature Envy

Although there is no direct *JEditBuffer* attributes modification in the *joinLineAt* method, it does modify the data returned by the methods of *JEditBuffer* making it reduce the quality of code.

As explained above, the *joinLineAt* method modifies the data returned by the *JEditBuffer* class, hence making it smelly. It is a feature envy code smell because the data returned from another object is being modified in this method.

We believe the following method is smelly because data returned by another class is getting manipulated here. By the definition of feature envy, if any class modifies the data of another object then it exhibits a code smell. If there was no feature envy, ideally *JEditBuffer* class should format the data and return it to the *joinLineAt* method. That is why we consider this a smelly class.

2. God Class: God Class / Large Class is a type of Bloaters code smell. God Classes usually display the symptoms of :

- Very large class with many methods and attributes.
- Tightly coupled functionalities
- The SOLID principle is not being followed

The major cause of God Class is that developers keep appending more functionality to the class until it grows too big.

In jEdit, JDeodorant identified many such classes that exhibited God Class code smell. Below is a picture displaying the same:

Problems Progress Duplicated Code Feature Envy God Class Code Smell Visualization			
Refactoring Type	Source Class/General Concept	Extractable Concept	Source/Extracted accessed members
▶	org.gjt.sp.jedit.syntax.ParserRuleSet		0/10
▶	org.gjt.sp.jedit.io.VFS		0/8
▶	org.gjt.sp.jedit.pluginmgr.PluginList		0/5
▶	org.gjt.sp.jedit.browser.VFSBrowser		0/4
▶	installer.CBZip2InputStream		0/3
▼	org.gjt.sp.jedit.Buffer		0/3
	[flag]		
Extract Class		[flag]	0/3
▶	org.gjt.sp.jedit.gui.SplashScreen		0/3
▶	org.gjt.sp.jedit.gui.StatusBar		0/3
▶	org.gjt.sp.jedit.gui.statusbar.ErrorsWidgetFactory		0/3
▶	org.gjt.sp.jedit.print.BufferPrintable		0/3
▶	org.gjt.sp.jedit.search.SearchBar		0/3
▶	org.gjt.sp.util.PropertiesBean		0/3

Fig.: Results of God Class on jEdit

In the above picture, there are 4 columns displayed for results:

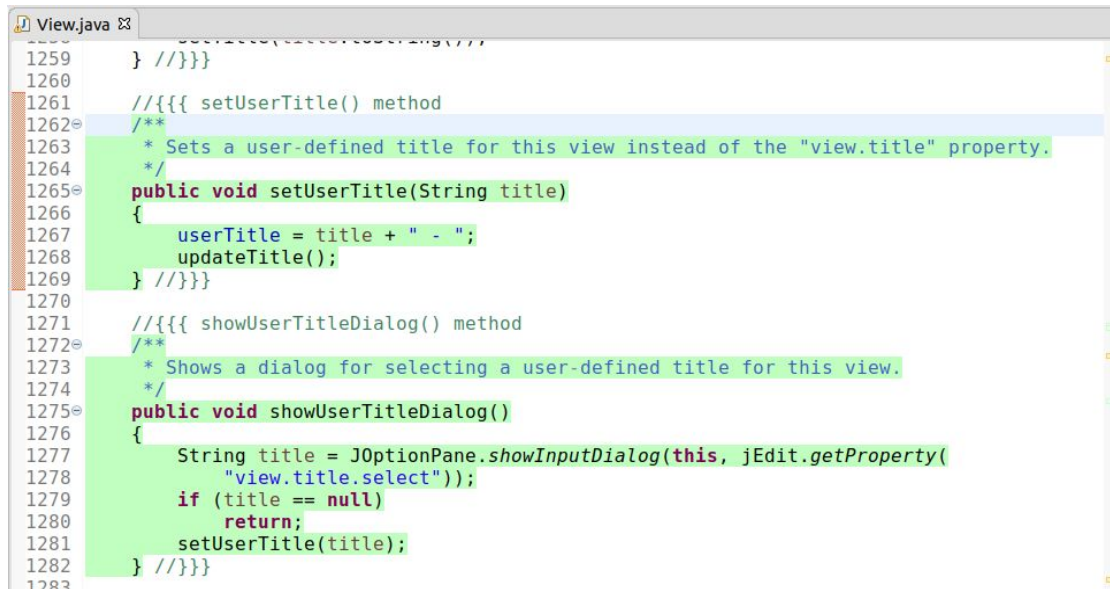
- Refactoring Type: The type of refactoring to be done
- Source Class/General Entity: Source Class where the code smell was identified
- Extractable Concept: If refactored, the part of the class that could be extracted
- Source/Extracted accessed members: Source access members and the extracted access members

For this code smell, we chose the *View* class from the *org.gjt.sp.jedit* package. This *View* class has almost 2350 lines of code and more than 30 attributes and methods. A *View* is jEdit's top-level frame window. It contains an *EditPane* that in turn contains a *JEditTextArea* that displays a *Buffer*. From this we can guess that it is a big class with many responsibilities and tight coupling with other objects. The figure below shows the result of God Class on *View* class.

Problems Progress Duplicated Code Feature Envy God Class Code Smell Visualization			
Refactoring Type	Source Class/General Concept	Extractable Concept	Source/Extracted accessed members
▼	org.gjt.sp.jedit.View [titl, user]		1/2
Extract Class		[titl, user]	1/2

Fig.: God Class result on *View* class

The figure shows that the concept [titl, user] i.e., the methods *setUserTitle* and *showUserTitleDialog* could be implemented as a new concept/class outside the *View* class. To further see what it means, we analyzed the highlighted code as shown below.



```

1259     } //}}}
1260
1261     //{{{ setUserTitle() method
1262     /**
1263      * Sets a user-defined title for this view instead of the "view.title" property.
1264      */
1265     public void setUserTitle(String title)
1266     {
1267         userTitle = title + " - ";
1268         updateTitle();
1269     } //}}}
1270
1271     //{{{ showUserTitleDialog() method
1272     /**
1273      * Shows a dialog for selecting a user-defined title for this view.
1274      */
1275     public void showUserTitleDialog()
1276     {
1277         String title = JOptionPane.showInputDialog(this, jEdit.getProperty(
1278             "view.title.select"));
1279         if (title == null)
1280             return;
1281         setUserTitle(title);
1282     } //}}}
1283

```

Although related to *View* class, they seem to be performing a functionality that utilizes user interaction. The methods are responsible for setting a new view title in the application and don't seem to interact with any other methods and class attributes of *View*. The presence of these methods could clearly decrease class cohesion and might reduce reusability.

From the above characteristics explained with respect to class, this class could be considered a smelly class of type God Class.

We certainly agree that it is smelly, mainly because *View* being such a big class is performing many responsibilities of maintaining the view and the methods seem to be all dependent on each other. But these two methods firstly have a function of interacting with the user and updating the view title, and secondly, they seem to be not cohesive w.r.t. to other methods in the class. If they are extracted as a new class, it might increase class cohesion, decrease complexity, and increase reusability. Therefore, this class certainly exhibits Large Class symptoms.

The code smell visualization can be seen as below if a new class is extracted from this concept:

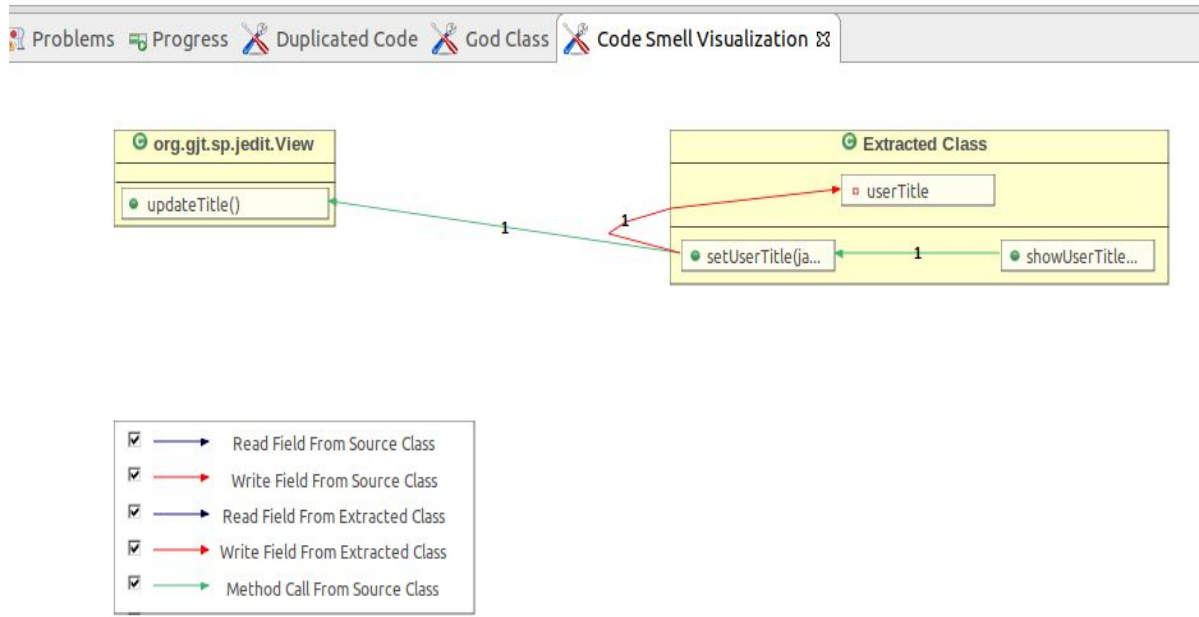


Fig.: Code smell visualization for View Class

3. **Duplicated Code:** As the name suggests, this smell is all about code duplication. When two code fragments look almost identical, they are called as duplicated code. Duplication may occur for many reasons such as,
 - a. multiple programmers working on different parts of the same program at the same time
 - b. copying and pasting the relevant code in a hurry to meet deadlines
 Duplication may also occur when specific fragments of code look different but in fact, perform the same operation.

In jEdit, JDeodorant has an option to import results from many clone detection tools such as CCFinder, Deckard, NiCad, and so on. It analyses the results and highlights the location of the smell. Alternatively, if the developer is aware of the duplication location, JDeodorant offers the option of refactoring the selected instance methods.

To analyze the duplicated code, we used the [NiCad 5.2](https://www.txl.ca/txl-nicaddownload.html)¹ tool to compute the duplication. It gives the result in the form of an XML file. When we tried to import the results using JDeodorant, it threw an error and there was no proper documentation to resolve this error. Hence, we used the result from

¹ <https://www.txl.ca/txl-nicaddownload.html>

the NiCad tool and analyzed the code smell manually. Here's a snapshot of the results file from NiCad.

The below code snippet is the result of the NiCad tool in the XML format.

```
<class classid="27" nclones="2" nlines="13" similarity="92">
  <source
file="systems/cs515-001-s20-grimreapers-jedit/org/gjt/sp/jedit/textarea/JEdit
tTextArea.java" startline="226" endline="243" pcid="1405">
  </source>
  <source
file="systems/cs515-001-s20-grimreapers-jedit/org/gjt/sp/jedit/textarea/Text
Area.java" startline="3359" endline="3376" pcid="1926">
  </source>
</class>
```

The **classid** represents the index in the result file. **nclones** shows the number of clones found w.r.t to the **file** in this **class** tag. **nlines** show the number of lines that are identical. Based on the similarity of the duplicated code, it displays the percentage which is shown as **similarity**. **file** shows the source code classes in which there is duplication, whereas **startline** and **endline** display the duplicated code lines starting point and ending point respectively.

For this smell, we chose the *goToMatchingBracket()* method from *TextArea* and *JEditTextArea* class under *org.gjt.sp.jedit.textarea* package based on the above mentioned results. These two methods seem to have 92 % of duplication which seems to be very high.


```
//{{{ goToMatchingBracket() method
/**
 * Moves the caret to the bracket matching the one before the caret.
 * Also sends PositionChanging if it goes somewhere.
 * @since jEdit 4.3pre18
 */
public void goToMatchingBracket()
{
    if(getLineLength(caretLine) != 0)
    {
        int dot = caret - getLineStartOffset(caretLine);

        int bracket = TextUtilities.findMatchingBracket(
            buffer, caretLine, Math.max(0, dot - 1));
        if(bracket != -1)
        {
            EditBus.send(new PositionChanging(this));
            selectNone();
            moveCaretPosition(bracket + 1, false);
            return;
        }
    }
    javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
} //}}}
```

Fig.: *goToMatchingBracket()* in JEditTextArea

```
//{{{ goToMatchingBracket() method
/**
 * Moves the caret to the bracket matching the one before the caret.
 * @since jEdit 2.7pre3
 */
public void goToMatchingBracket()
{
    if(getLineLength(caretLine) != 0)
    {
        int dot = caret - getLineStartOffset(caretLine);

        int bracket = TextUtilities.findMatchingBracket(
            buffer, caretLine, Math.max(0, dot - 1));
        if(bracket != -1)
        {
            selectNone();
            moveCaretPosition(bracket + 1, false);
            return;
        }
    }
}
```

Fig.: *goToMatchingBracket()* in TextArea.

It is evident from the above two code snippets that except for two lines of code they are identical and also seem to perform the same functionality based on the comments.

Also from the comments we can see that each method has been created for a different version of jEdit implying that the new developers might have overlooked the existing method and added the duplicate as a new functionality. Upon testing in the jEdit application by commenting the method in either of these classes, we found that each of them perform the same function in a text area. Commenting the *goToMatchingBracket()* method in a one class, say *TextArea* didn't break the application's functionality and similarly, when the method was commented in just *JEditTextArea*, the functionality worked as expected.

We completely agree that these two methods strongly exhibit "Duplicated Code" smell. From the above code snippets, NiCad results, and our manual analysis it can be clearly concluded that these fragments of code have code smells and might need a refactoring.

2) Removing code smells via refactoring

Refactoring is a process of improving code without creating a new functionality that can transform a smelly code into an efficient and a maintainable one.

jEdit

Out of the three code smells identified for jEdit, we chose to refactor the smelly methods/classes identified in Feature Envy and Duplicated Code category. JDeodorant suggested the refactoring for smells and the same was applied after careful review.

Feature Envy Refactoring: For feature envy, we selected the same method that was identified in the code smell section i.e., *joinLineAt()* method from the *TextArea* class under *org.gjt.sp.jedit.textarea* package. The main functionality of this method was to join a given set of input lines. JDeodorant suggested a refactoring called 'Move Method'.

In the Move Method refactoring, a new method is created in the class that uses the method the most, then the code is moved from the old method to the new one. Change the code of the original method into a reference to the new method in the old class or else remove it entirely.

Changes Made:

- 1) `joinLineAt()` method in the `TextArea` class under `org.gjt.sp.jedit.textarea` package was completely removed from here and moved to `org.gjt.sp.jedit.buffer.JEditBuffer` class.
- 2) Previously it used an attribute `buffer`- an instance of `JEditBuffer` class- to call methods from `JEditBuffer`. The below picture highlights the usage of `buffer` in the old method.

```
private void joinLineAt(int line)
{
    if (line >= buffer.getLineCount() - 1)
        return;
    int end = getLineEndOffset(line);
    CharSequence nextLineText = buffer.getLineSegment(line + 1);
    buffer.remove(end - 1, StandardUtilities.getLeadingWhiteSpace(
        nextLineText) + 1);
    if (nextLineText.length() != 0)
        buffer.insert(end - 1, " ");
} //}}}
//}}}
```

- 3) Since the new method was moved to `JEditBuffer` class, the reference was removed. The below picture shows the new method.

```
+ public void joinLineAt(int line) {
+     if (line >= getLineCount() - 1)
+         return;
+     int end = getLineEndOffset(line);
+     CharSequence nextLineText = getLineSegment(line + 1);
+     remove(end - 1, StandardUtilities.getLeadingWhiteSpace(nextLineText) + 1);
+     if (nextLineText.length() != 0)
+         insert(end - 1, " ");
+ }
```

- 4) Since the whole method was moved to the `JEditBuffer` class, any method call within the `TextArea` class was changed to reference the method from the new class. Since `buffer` was an instance of the `JEditBuffer` class, the new method was referenced through that. The pictures below show the old code and the modified code in `TextArea` respectively.

```

4620                                     // such as indent level or fold level.
4621 -                               joinLineAt(selection.endLine - 1);
4622                               doneForSelection = true;
4623                               }
4624                               }
4625 // If nothing selected or all selections span only
4626 // one line, join the line at the caret.
4627 if (!doneForSelection)
4628 {
4629     int end = getLineEndOffset(caretLine);
4630
4631     // Nothing to do if the caret is on the last line.
4632     if (end > buffer.getLength())
4633     {
4634         javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
4635         return;
4636     }
4637
4638 -                               joinLineAt(caretLine);

```

Fig.: TextArea code before refactoring

```

                               buffer.joinLineAt(selection.endLine - 1);
                               doneForSelection = true;
                               }
                               }
// If nothing selected or all selections span only
// one line, join the line at the caret.
if (!doneForSelection)
{
    int end = getLineEndOffset(caretLine);

    // Nothing to do if the caret is on the last line.
    if (end > buffer.getLength())
    {
        javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
        return;
    }

    buffer.joinLineAt(caretLine);
    if (!multi)
        selectNone();
    moveCaretPosition(end - 1);

```

Fig.: TextArea code after refactoring

5) No manual changes were performed.

As explained in the code smell section, the method in the old class was calling too many of *JEditBuffer* methods and also was modifying the data returned. Most of the functionality this method required seemed to be in *JEditBuffer* class. This increases coupling between these two classes and hence makes it hard to maintain or reuse.

Moving the method to *JEditBuffer* makes more sense since it will reduce the number of calls made from *TextArea* to *JEditBuffer*. Also data manipulation will happen in the same object not violating any fundamental software principle. Although joining lines happens in the given text area, any functionality w.r.t the lines in a text area buffer seems to be implemented in *JEditBuffer*. Hence it would be more apt to have the *joinLineAt()* method in *JEditBuffer*.

After the changes were made, we re-ran JDeodorant to check if the smell still existed and the results did not show this code smell anymore.

Test Cases: jEdit did not seem to have any test suite for the classes that were changed. To generate jUnit test classes for the new or modified classes/methods, we used [EVOSUITE](#)², a tool that automatically generates test cases with assertions for classes written in Java. This tool takes a bytecode class name and its classpath as arguments, and generates two classes where one is the actual test class and the other is a scaffolding file that contains JUnit annotations which ensure that these methods are executed before/after execution of each individual test.

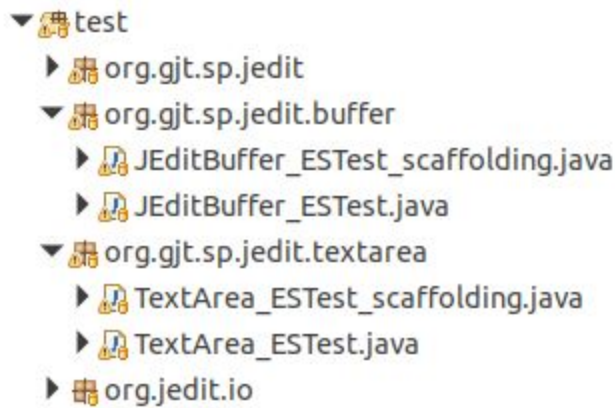
To build our project successfully, we downloaded the ecosuite dependencies through Ivy and added @Ignore annotation for all the scaffolding classes. The below picture shows the dependencies in the ivy file.

```
<dependency org="org.bouncycastle" name="bcpg-jdk16" rev="1.46" conf="scripting"/>

<dependency org="com.google.code.findbugs" name="jsr305" rev="3.0.0"/>
<dependency org="org.evosuite" name="evosuite-runtime" rev="1.0.6"/>
<dependency org="org.evosuite" name="evosuite-standalone-runtime" rev="1.0.6"/>
<dependency org="org.evosuite" name="evosuite" rev="1.0.6">
  <artifact name="evosuite" type="pom" ext="pom"/>
</dependency>
```

² G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, New York, NY, USA, 2011, pp. 416-419.

Using this tool we generated test cases for the modified class where the method was moved to i.e., *JEditBuffer* as well as the old class *TextArea*. The picture below shows the test classes under the test folder.



The test ran successfully for all the generated test cases. Below is a test case in *JEditBuffer_ESTest* class for the *joinLineAt* method.

```
* This file was automatically generated by EvoSuite[.]

package org.gjt.sp.jedit.buffer;

import org.junit.Test;

@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true)
public class JEditBuffer_ESTest extends JEditBuffer_ESTest_scaffolding {

    @Test(timeout = 4000)
    public void testjoinLineAt() throws Throwable {
        JEditBuffer jEditBuffer0 = new JEditBuffer();
        jEditBuffer0.joinLineAt(930);
        assertTrue(jEditBuffer0.isEditable());
    }
}
```

We also performed manual testing to make sure the functionality was working as expected.

Step #	Description	Rationale
1	<p>Test Case 1: Check if the two lines join on selecting the option "Join Lines" from menu</p> <p>Steps: open jEdit application → Point the caret on some line in text area → go to Edit Menu → select Text → select Join Lines</p>	<p>This is the expected behaviour.</p> <p>Test passed!</p>

	<p>Input: Line 1: This is a test for joining lines Line 2: for A4 assignment Line 3: for cs515 course</p> <p>Expected Behavior: If the caret is at Line 1, then it should join with Line 2 only.</p>	
2.	<p>Test Case 2: Check if a selection of lines join on selecting the option from menu</p> <p>Steps: open jEdit application → Select some lines of text in the text area → go to Edit Menu → select Text → select Join Lines</p> <p>Input: Line 1: This is a test for joining lines Line 2: for A4 assignment Line 3: for cs515 course</p> <p>Expected Behavior: If all the lines are selected, then the output should be a single line as follows: "This is a test for joining lines for A4 assignment for cs515 course"</p>	<p>This is the expected behaviour.</p> <p>Test passed!</p>
3	<p>Test Case 3: Check if any lines are joined if the caret is positioned at the last line</p> <p>Steps: open jEdit application → Point the caret on the last line in text area → go to Edit Menu → select Text → select Join Lines</p> <p>Input: Line 1: This is a test for joining lines Line 2: for A4 assignment Line 3: for cs515 course</p> <p>Expected Behavior: If the caret is at Line 3, then no lines are joined.</p>	<p>This is the expected behaviour.</p> <p>Test passed!</p>

Github Commit : [rf1-Feature Envy Refactoring](#)³, [Test Classes for Feature Envy](#)⁴

Duplicated Refactoring: For Duplicated Code refactoring, we chose the same class that was identified during code smell i.e., *goToMatchingBracket()* method from *TextArea* and its subclass *JEditTextArea* class under *org.gjt.sp.jedit.textarea* package. The identified smell pointed at two duplicated methods with the same name and functionality- *goToMatchingBracket*- which performed the function of finding a matching bracket. JDeodorant suggested a refactoring type Extract and Pull Up Method.

Extract and Pull Up Method is applied when the clones are located in methods belonging to different subclasses within the same inheritance hierarchy. JDeodorant introduces a functional interface as a parameter to the extracted method. The original methods containing the clones pass a Lambda expression as an argument for the introduced parameter when calling the extracted method.⁵

Changes Made:

- 1) JDeodorant created a functional interface and extracted the method to the superclass *TextArea*. The below picture shows the interface and the extracted method.

```
/** Moves the caret to the bracket matching the one before the caret.
    Created for A4_ Refactoring*/

@FunctionalInterface
protected interface goToMatchingBracketInterface {
    void apply();
}

protected void goToMatchingBracketExtracted(goToMatchingBracketInterface positionChange) {
    if (getLineLength(caretLine) != 0) {
        int dot = caret - getLineStartOffset(caretLine);
        int bracket = TextUtilities.findMatchingBracket(buffer, caretLine, Math.max(0, dot - 1));
        if (bracket != -1) {
            positionChange.apply();
            selectNone();
            moveCaretPosition(bracket + 1, false);
            return;
        }
    }
    javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
}
```

³ https://github.com/sushmithaj/cs515-001-s20-grimreapers-jedit/tree/a4_refactoring

⁴ <https://github.com/sushmithaj/cs515-001-s20-grimreapers-jedit/commit/6898e4b55d537dc598319f1fee1b38ef22c53357>

⁵ Mazinianian, Davood & Tsantalos, Nikolaos & Stein, Raphael & Valenta, Zackary. (2016). JDeodorant: Clone Refactoring.

10.1145/2889160.2889168.

- 2) The *goToMatchingBracket* method in the *TextArea* class was updated to call the newly extracted method as a lambda expression as shown below.

```
public void goToMatchingBracket()
{
    goToMatchingBracketExtracted(() -> {
    });
}
```

- 3) The *goToMatchingBracket* method in the *JEditTextArea* class was updated to call the newly extracted method by passing the `EditBus.send(new PositionChanging(this));` line as the interface parameter in the lambda expression as shown below.

```
*/
// A4_ Duplicated Code Refactoring
public void goToMatchingBracket()
{
    goToMatchingBracketExtracted(() -> {
        EditBus.send(new PositionChanging(this));
    });
}
```

- 4) Only manual changes included were changing the interface name and the parameter to make the code more readable.

Since both the superclass and the subclass had the same method, to avoid duplication it is necessary to eliminate the redundancy. Since *JEditTextArea* is a subclass of *TextArea*, the method was extracted to the latter so that it could be easily inherited.

JDeodorant created a functional interface because originally, the method in *TextArea* class did not implement the position changing functionality. But it was implemented in *JEditTextArea* which given is by the line;

```
EditBus.send(new PositionChanging(this));
```

Implementing a functional interface by giving the above line as a parameter to the extracted method makes sure that the functionalities are not changed and allows the flexibility of using the method.

Test Cases: Using EVOSUITE tool we generated test cases for the modified classes and methods. The picture below shows the test classes under the test folder.



The test ran successfully for all the generated test cases. Additionally, we performed manual testing as well to make sure the functionality was as working as expected.

Step #	Description	Rationale
1	<p>Test Case 1: Check if the caret moves to the bracket matching the one before the caret.</p> <p>Steps: open jEdit application → Point the caret to the closing bracket → go to Edit Menu → select Source → select Go to Matching Bracket</p> <p>Input: Line 1: { Line 2: This is a test for joining lines Line 3: for A4 assignment Line 4: for cs515 course Line 5: }</p> <p>Expected Behavior: If the caret is at Line 5, then it should move to Line 1.</p>	<p>This is the expected behaviour.</p> <p>Test passed!</p>
2.	<p>Test Case 2: Check if a caret moves to the correct matching bracket when there are many brackets</p> <p>Steps: open jEdit application → Point the caret</p>	<p>This is the expected behaviour.</p> <p>Test passed!</p>

	<p>to the closing bracket at Line 7 or Line 6→ go to Edit Menu → select Source → select Go toMatching Bracket</p> <p>Input: Line 1:{ Line 2:{ Line 3: This is a test for joining lines Line 4: for A4 assignment Line 5: for cs515 course Line 6: } Line 7: }</p> <p>Expected Behavior: 1) If the caret is at Line 7, then it should go to Line 1. 2) If the caret is at Line 6, then it should go to Line 2.</p>	
3.	<p>Test Case 3: Check when there are no brackets</p> <p>Steps: open jEdit application →Point the caret anywhere in the buffer→ go to Edit Menu → select Source → select Go toMatching Bracket</p> <p>Input: Line 1: This is a test for joining lines Line 2: for A4 assignment Line 3: for cs515 course</p> <p>Expected Behavior: Nothing should happen.</p>	<p>This is the expected behaviour.</p> <p>Test passed!</p>

Github Commit: [rf3-Duplicated Code Refactoring](https://github.com/sushmithaj/cs515-001-s20-grimreapers-jedit/commit/14dbeac6126a330b62995edee77fe562133702e6)⁶

⁶ <https://github.com/sushmithaj/cs515-001-s20-grimreapers-jedit/commit/14dbeac6126a330b62995edee77fe562133702e6>

PDFSAM

Detecting and Analyzing Code Smells and Refactoring

For this project, we identified three types of code smells God Class, Long Method, and Type checking.

In this project, we decided to investigate two packages that we modified (Merge and Rotate) to see how our modification had an effect on the PDFSAM code to get a better understanding of where did we go wrong when we modified the code.

1) Long Method:

The extract method is one of the most important refactoring technics since it can be used for decomposing large methods and can be applied in combination with other refactoring methods.

Rules:

The extracted code should contain the complete computation of a given variable declared in the original method

- The extracted code should contain the complete computation of a given variable declared in the original method
- The behavior of the program should be preserved after the application of the refactoring
- The extracted code should not be excessively duplicated in the original method

When we ran the Long method analyze with JDeodorant for Merge package:

Refactoring Type	Source Method	Variable Criterion	Block-Based Region	Duplicated/Extracted	Rate it!
▼	org.pdfsam.merge.MergeParametersBuilder::void addInput(... inputs				
Extract Method			B2	0/3	
Extract Method			B3	0/2	
►	org.pdfsam.merge.MergeParametersBuilderTest::public voi... input				

After running the JDeodorant, automatically parts of the code that causes the method to belong and has the ability to be extracted will be highlighted:

```

void addInput(PdfMergeInput input) {
    for(PageRange pagerange : input.getPageSelection()) {
        for(int pagenumber : pagerange.getPages(pagerange.getEnd())) {
            PdfMergeInput newInput = new PdfMergeInput(input.getSource(), Set.of(new PageRange(pagenumber, pagenumber)));
            this.inputs.add(newInput);
        }
    }
}

```

This is the part we modified. We are using 2-dimensional for loop in *addInput* method inside *MergeParametersBuilder* class and we ignored the 10 LoC rule for this method.

After refactoring *addInput* contains a 1-dimensional for loop and the extracted method called *addInputRefactored* contains the other one.

```

54 void addInput(PdfMergeInput input) {
55     for(PageRange pagerange : input.getPageSelection()) {
56         addInputRefactored(input, pagerange);
57     }
58 }
59
60
61 private void addInputRefactored(PdfMergeInput input, PageRange pagerange) {
62     for (int pagenumber : pagerange.getPages(pagerange.getEnd())) {
63         PdfMergeInput newInput = new PdfMergeInput(input.getSource(),
64             Set.of(new PageRange(pagenumber, pagenumber)));
65         this.inputs.add(newInput);
66     }
67 }
68

```

As you can see it still has complexity, so we ran JDeodorant again and refactored it again and this is the result:

```

54 void addInput(PdfMergeInput input) {
55     for(PageRange pagerange : input.getPageSelection()) {
56         addInputRefactored(input, pagerange);
57     }
58 }
59
60
61 private void addInputRefactored(PdfMergeInput input, PageRange pagerange) {
62     for (int pagenumber : pagerange.getPages(pagerange.getEnd())) {
63         addInputlv12(input, pagenumber);
64     }
65 }
66
67 private void addInputlv12(PdfMergeInput input, int pagenumber) {
68     PdfMergeInput newInput = new PdfMergeInput(input.getSource(), Set.of(new PageRange(pagenumber, pagenumber)));
69     this.inputs.add(newInput);
70 }

```

Much better!

- 2) **God Class:** We increased the Loc for the above particular class to have shorter methods, but we have a bigger class now.

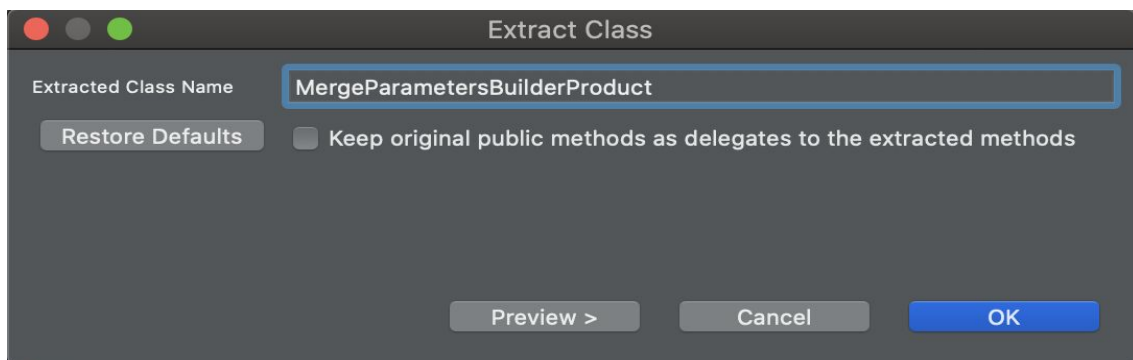
We performed God Class analysis for this package again and bingo the very same class now have the God Class problem.

```

43 //private Set<PdfMergeInput> inputs = new NullSafeSet<>();
44 private ArrayList<PdfMergeInput> inputs = new ArrayList<>();
45 //private Multiset<PdfMergeInput> inputs = new HashMultiSet<>();
46 private OutlinePolicy outlinePolicy = OutlinePolicy.RETAIN;
47 private boolean blankIfOdd;
48 private boolean footer;
49 private boolean normalize;
50 private AcroFormPolicy formsPolicy = AcroFormPolicy.MERGE;
51 private ToCPolicy tocPolicy = ToCPolicy.NONE;
52 private FileTaskOutput output;
53
54 void addInput(PdfMergeInput input) {
55     for (PageRange pagerange : input.getPageSelection()) {
56         addInputRefactored(input, pagerange);
57     }
58 }
59
60
61 private void addInputRefactored(PdfMergeInput input, PageRange pagerange) {
62     for (int pagenumber : pagerange.getPages(); pagerange.getEnd(); ) {
63         addInputlvl2(input, pagenumber);
64     }
65 }
66
67 private void addInputlvl2(PdfMergeInput input, int pagenumber) {
68     PdfMergeInput newInput = new PdfMergeInput(input.getSource(), Size of new PageRange(pagenumber, pagenumber));
69     this.inputs.add(newInput);
70 }
71
72 boolean hasInput() {
73     return inputs.isEmpty();
74 }

```

We extracted a class and called it *MergeParameterBuilderProduct*:



This is the refactored class:

```

1 package org.pdfsam.merge;
2
3
4 import java.util.ArrayList;
5
6
7 public class MergeParametersBuilderProduct {
8     private ArrayList<PdfMergeInput> inputs = new ArrayList<>();
9
10    public ArrayList<PdfMergeInput> getInputs() {
11        return inputs;
12    }
13
14    public void addInputlvl2(PdfMergeInput input, int pagenumber) {
15        PdfMergeInput newInput = new PdfMergeInput(input.getSource(), Set.of(new PageRange(pagenumber, pagenumber)));
16        this.inputs.add(newInput);
17    }
18
19    public boolean hasInput() {
20        return !inputs.isEmpty();
21    }
22
23    public void addInputRefactored(PdfMergeInput input, PageRange pagerange) {
24        for (int pagenumber : pagerange.getPages(pagerange.getEnd())) {
25            addInputlvl2(input, pagenumber);
26        }
27    }
28
29    public void addInput(PdfMergeInput input) {
30        for (PageRange pagerange : input.getPageSelection()) {
31            addInputRefactored(input, pagerange);
32        }
33    }
34 }

```

And the original class changed to this to refer to the newly extracted classes:

```

private MergeParametersBuilderProduct mergeParametersBuilderProduct = new MergeParametersBuilderProduct();

```

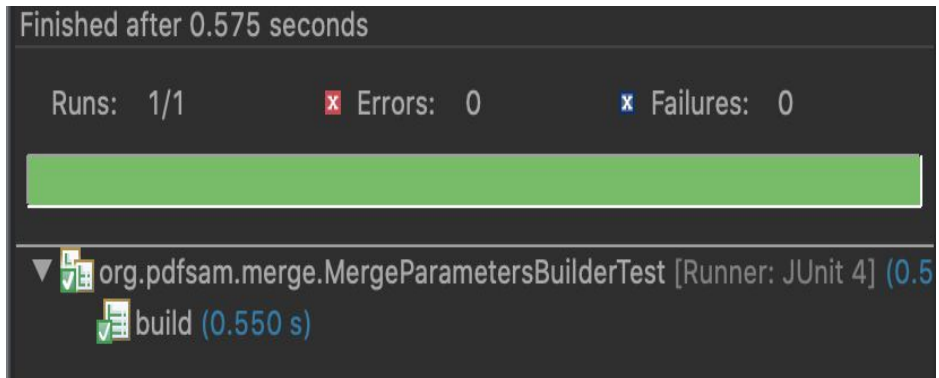
```

void addInput(PdfMergeInput input) {
    mergeParametersBuilderProduct.addInput(input);
}

boolean hasInput() {
    return mergeParametersBuilderProduct.hasInput();
}

```

Here is the test for *mergeParameterBuilder* with JUnit:



I analyzed other packages like pdfsam-core.

▼	org.pdfsam.module.ModuleDescriptorBuilder	0/1
▼	[priority]	
Extract Class	[priority]	0/1
▶	[input, name, categori, type, descript, sup...	
▶	org.pdfsam.pdf.PdfDocumentDescriptor	0/1
▶	org.pdfsam.pdf.PdfDocumentDescriptorTest	1/2

I refactored *ModuleDescriptorBuilder* class and it looks like this now:

```
public ModuleDescriptorBuilder priority(int priority) {
    return moduleDescriptorBuilderProduct.priority(priority, this);
}

public ModuleDescriptorBuilder priority(ModulePriority priority) {
    return moduleDescriptorBuilderProduct.priority(priority, this);
}
```

```
package org.pdfsam.module;

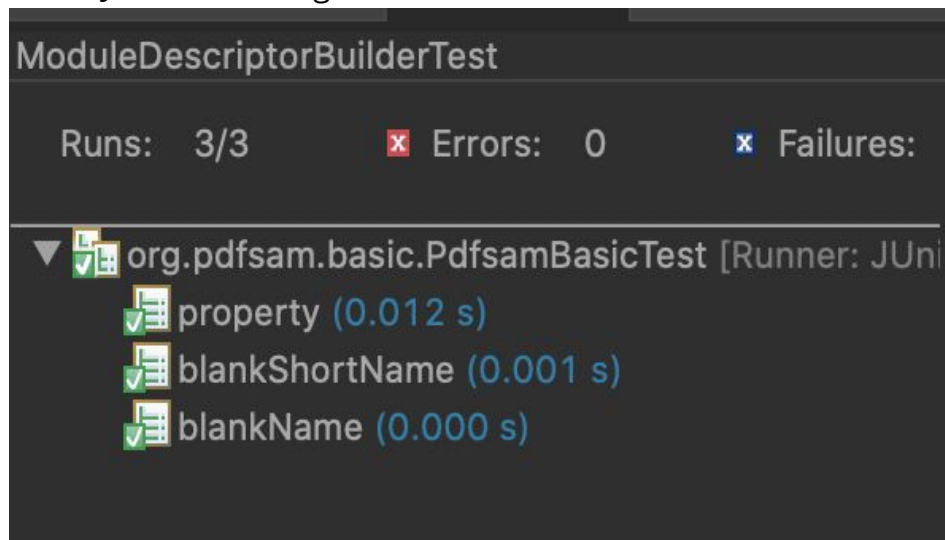
public class ModuleDescriptorBuilderProduct {
    private int priority = ModulePriority.DEFAULT.getPriority();

    public int getPriority() {
        return priority;
    }

    public ModuleDescriptorBuilder priority(int priority, ModuleDescriptorBuilder moduleDescriptorBuilder) {
        this.priority = priority;
        return moduleDescriptorBuilder;
    }

    public ModuleDescriptorBuilder priority(ModulePriority priority, ModuleDescriptorBuilder moduleDescriptorBuilder) {
        this.priority = priority.getPriority();
        return moduleDescriptorBuilder;
    }
}
```

I used Junit for testing and this is the result:



3) Type Checking:

Type checks can be identified in a scenario there is an attribute in a class that represents a state (type field).

I analyzed the Type Checking analysis for pdfsam-fx and here is the result:

▼	constant variables: [SAVE]		1	1.0	2.0
Replace Type Code with Sta...	org.pdfsam.ui.io.RememberingLatestFileChooserWrapper::public java.io...	showDialog		1	2.0
▼	constant variables: [BOTTOM, DOWN, TOP]		1	1.0	1.0
Replace Type Code with Sta...	org.pdfsam.ui.selection.multiple.SelectionChangedEvent::public boolea...	canMove		1	1.0

So I took a look at *selectionChangedEvent* class and tried to identify the smell by using JDeodorant and here is the problem:

```
public boolean canMove(MoveType type) {
    if (isClearSelection()) {
        return false;
    }
    switch (type) {
        case BOTTOM:
            return isSingleSelection() && bottom < totalRows - 1;
        case DOWN:
            return bottom < totalRows - 1;
        case TOP:
            return isSingleSelection() && top > 0;
        default:
            return top > 0;
    }
}
```

As you can see, switch cases that depend on type are the first suspect of the type check smells because in this case, the attribute represents the state.

Here are the classes that are refactored by JDeodorant:

```

1 package org.pdfsam.ui.selection.multiple;
2
3
4 /**
5  * @see org.pdfsam.ui.selection.multiple.move.MoveType#TOP
6  */
7 public class Top extends Type {
8     public boolean canMove(SelectionChangedEvent selectionChangedEvent) {
9         return selectionChangedEvent.isSingleSelection() && selectionChangedEvent.getTo
10     }
11 }

```

```

1 package org.pdfsam.ui.selection.multiple;
2
3
4 /**
5  * @see org.pdfsam.ui.selection.multiple.move.MoveType#DOWN
6  */
7 public class Down extends Type {
8     public boolean canMove(SelectionChangedEvent selectionChangedEvent) {
9         return selectionChangedEvent.getBottom() < selectionChangedEvent.getTotalRows()
10     }
11 }

```

```

1 package org.pdfsam.ui.selection.multiple;
2
3
4 /**
5  * @see org.pdfsam.ui.selection.multiple.move.MoveType#BOTTOM
6  */
7 public class Bottom extends Type {
8     public boolean canMove(SelectionChangedEvent selectionChangedEvent) {
9         return selectionChangedEvent.isSingleSelection()
10             && selectionChangedEvent.getBottom() < selectionChangedEvent.getTotalRows()
11     }
12 }

```

```

1 package org.pdfsam.ui.selection.multiple;
2
3
4 public abstract class Type {
5     public abstract boolean canMove(SelectionChangedEvent selectionChangedEvent);
6 }

```

JUnit passed these test cases:

