CS 581A3 Software Maintenance & Evolution: Spring 2018
Assignment 4 (A4) : Refactoring OSS

Team & Role:
Smell# 1, 3, 5, 7 -> Swetha Varadarajan
Smell# 2, 4, 6, 8 -> Kartikay Sharma

In A3, we used the iPlasma tool to detect the smells for Jedit. We reported three smells: Feature envy, Shotgun surgery and Intensive coupling. For A4, we used Jdeodorant to detect the code smells and to perform automatic refactoring. The method in which feature envy was reported by iPlasma is not detected by Jdeodorant. Also, Jdeodorant does not detect Shotgun surgery and Intensive Coupling. So, we had to detect four new smells for Jedit.

For Pdfsam, we did not agree with the God class smell detected by Jdeodorant in A3. So, we detected two new smells for this assignment. We explain the newly detected smells as and when required throughout the report.

1 Manual refactoring

1.1 Smell #1: Jedit: Feature Envy

1.1.1 Class: EmacsUtil.java
1.1.2 Location: org/jedit/keymap/
1.1.3 Method: public char charAt(int i) (It is the overridden method in EmacsUtil class)

1.1.4 Smell description and analysis:
Feature envy occurs when a method seems to be focused on manipulating the data of other classes rather than its own. Here, charAt () method takes an integer as input and returns a character as output. It calls the getLength() method of buffer class and returns a zero if this length is lesser than the input integer. If not, it returns the first character of the requested text by making a call to the buffer class's parent class (JeditBuffer) method: getText. his in turn calls the getText method of ContentManager class.

Here, the charAt() method clearly does not use any data of the current class and manipulates on the data obtained from ContentManager class. Moreover there are eight instances spread over four other methods where this method is called.

1. emacsKillLine()
2. lineAt (int)
3. findEndOfSentence()
4. findBeginningOfSentence()

We agree that this is a feature envy smell and there is a need to refactor it. The *emacsKillLine()* method is called in the *Emacs_Kill_Line.bsh* file located at macros/Emacs. The analysis of this is necessary in order to verify the functionality of the application before and after refactoring. This .bsh file correspond to the functionalities found in Macros -> Emacs menu of jEdit text editor.

1.1.5 List of refactorings: Move method

1.1.6 Rationale: Moving the method to the class whose data is manipulated helps to have a better code design and enhances the readability of the software.

1.1.7 List of manual code changes:

1. Move the charAt() method implementation to Buffer class.
2. Change the invocation (at all the eight instances) of the moved method in EmacsUtil class.

1.1.8 Test cases:
We manually tested the functionality before and after refactoring. Specifically, we verified the functionality of emacsKillLine() function by using the "Emacs Kill Line" option under Macros -> Emacs menu of jEdit text editor. We had ten lines of text and performed the following tests.

Test Case #	Description	Rationale
1	**Test case:** Emacs_Kill_Line:Start of non-empty line **Inputs:** Mouse pointer is at the start of line #4 **Expected output:** On Clicking "Macros->Emacs->Emacs Kill Line" deletes the entire line#4 and rest of the lines are not shifted one line up and total number of lines should be 10.	To verify if the entire line is killed. Test case passes.
2	**Test case:** Emacs_Kill_Line:Start/End/Middle of empty line **Inputs:** Mouse pointer is at the beginning of line #4 which is now empty. **Expected output**: On clicking "Macros->Emacs->Emacs Kill Line" deletes the entire line#4 and rest of the lines are shifted one line up and total number of lines should be 9.	To verify if the blank line is killed. Test case passes.
3	**Test case:** Emacs_Kill_Line:End of non-empty line **Inputs:** Mouse pointer is at the end of line #4. **Expected output:** On clicking "Macros->Emacs->Emacs Kill Line" concatenates the line#4 with line#5 and rest of the lines are shifted one line up and total number of lines should be 9.	To verify if the whitespaces at the end of a line is killed. Test case passes.
4	**Test case:** Emacs_Kill_Line:Middle of non-empty line **Inputs:** Mouse pointer is at the middle of line #4. **Expected output:** On clicking "Macros->Emacs->Emacs Kill Line" deletes the line starting from the mouse pointer in line#4 and rest of the lines are not shifted one line up and total number of lines should	To verify if the rest of the line from the start of the mouse pointer is killed. Test case passes.

	be 10. The text before the mouse pointer in line#4 should not be deleted.	
5	Repeat the test cases (1-4) when there is a single whitespace space in the line. There should not be any change in the expected behaviour.	Test case passes.
6	Repeat the test cases (1-4) when there are multiple whitespace space in the line. There should not be any change in the expected behaviour.	Test case passes.
7	Repeat the test cases (1-4) when there are multiple sentences in the line (period followed by whitespace). There should not be any change in the expected behaviour.	Test case passes.

1.1.9 Jdeodorant before refactoring: Marks chatAt() method as feature envy.
1.1.10 Jdeodorant after refactoring: Feature Envy Smell for this method is removed. We verified that there no new smell introduced are in the moved method.

1.2 Smell #2: Jedit: Duplicated code

1.2.1 Class: TextArea.java
1.2.2 Location: org/gjt/sp/jedit/textarea/

1.2.3 Smell description and analysis:
This is a newly identified smell because we used Jdeodorant in this assignment instead of iPlasma, which was used for A3. So we describe it briefly. Duplicated code is a code smell that can result from having two or more pieces of code that look same / perform the same functionality. So, on the basis of this definition, let's analyse the duplicated code smell we found in Jedit project.

Methods *toUpperCase()* and *toLowerCase()* are exactly same except for the just one statement, the statement which changes the case of the selections;
*setSelectedText(s, getSelectedText(s).**toUpperCase()**);* in toUpperCase() method
and
*setSelectedText(s, getSelectedText(s)**.toLowerCase()**);* in toLowerCase() method

This method looks like a classic case of copy and paste.

1.2.4 List of refactorings: Extract Method - *changeCase from toUpperCase() & toLowerCase()*

1.2.5 Rationale: Both methods perform the exact same functionality and differ only in one statement, as mentioned earlier. So any change in the common functionality has to be made in both methods. So we extract the duplicated code into a new method and call that method in place of the duplicated code.

1.2.6 List of manual code changes:

1. Create a new method and extract duplicated code

we extracted a method, named *changeCase(boolean toUpper)* that is same as both methods, but it uses the boolean parameter to determine if the selections are to be converted into lowercase or uppercase as follows:

```
if(toUpper){
        setSelectedText(s, getSelectedText(s).toUpperCase());
}else{
        setSelectedText(s, getSelectedText(s).toLowerCase());
}
```

2. Call the extracted method in place of the duplicated code

And we called this extracted method in both *toUpperCase()* and *toLowerCase() methods*

```
public void toUpperCase(){
   changeCase(true);
}
public void toLowerCase(){
   changeCase(false);
}
```

1.2.7 Test cases included: We performed the following manual test cases.

Test Case	Description	Rationale
1	**Precondition**: Type some text in the buffer area like "A quick BROWN FoX jumPS oVer A LazY dOG" **Test case**: Test toUpperCase functionality on selected text **Steps:** 1. Select the text 2. Right Click -> click "To UpperCase" **Expected output:** Text should transform into "A QUICK BROWN FOX JUMPS OVER A LAZY DOG"	To verify "To Uppercase" functionality works as expected. Test case passes.
2	**Precondition**: Type some text in the buffer area like "A quick BROWN FoX jumPS oVer A LazY dOG" **Test case**: Test toLowerCase functionality on selected text **Steps:** 1. Select the text 2. Right Click -> click "To LowerCase" **Expected output:** Text should transform into "a quick brown fox jumps over a lazy dog"	To verify "To Lowercase" functionality works as expected. Test case passes.

1.2.8 Jdeodorant before refactoring: We could not make any of the required code clone tool results that Jdeodorant uses. We used Nicad4 to find cloned methods and these (*toUpperCase* and *toLowerCase*) were listed in results with 95% similarity. We then performed the refactoring manually**.**

1.2.9 Jdeodorant after refactoring: The methods are not listed in Nicad4 results with high similarity score after refactoring.

1.3 Smell #3: Pdfsam: God class

1.3.1 Class: ModuleDescriptorBuilder.java
1.3.2 Location: pdfsam-core/src/main/java/org/pdfsam/module/
1.3.3 Method: This smell is for the whole class. Not a particular method.

1.3.4 Smell description and analysis:
The God class code smell identifies classes that implement multiple responsibilities and should be refactored. It is a class that centralizes the intelligence in the system. It is large, complex, has low cohesion and uses data from other classes. In this case, the ModuleDescriptorBuilder class is used to set and return the description of a module in Pdfsam. Specifically, it returns the category, input type, name, description, supportURL and priority of a given module.

This class has private variables for all the six description parameters and corresponding setter methods. The only difference is that the initial value of the priority parameter is obtained from ModulePriority class whereas the other parameters are set using the setter methods. The class has two overloaded method to set the priority parameter. One of the method takes ModulePriority object as an input whereas the other takes an integer as an input. When the input is an integer, the ModuleDescriptor Builder class just sets the value to its priority value. When the input is a ModulePriority object, it sets the value obtained from getPrioirty method of ModulePriority class.

From the discussion, we can say that the ModuleDescriptorBuilder class is performing two tasks:

1. Setting and returning the six description parameters of a module
2. Getting the priority parameter from getPriority method of ModulePrioirty class.

In order to follow the single responsibility design principle, there is a need to separate the functionality. So, we conclude that this smell is a true GodClass smell.

1.3.5 List of refactorings: Extract method

1.3.6 Rationale: Extracting the methods that perform the second functionality as described above will ensure the single responsibility design principle.

1.3.7 List of manual code changes:

1. Create a new class: ModuleDesscriptorBuilderPrioirty.java in the same location.
2. Copy the two overloaded methods: priority()
3. Inspect the data this method operates and resolve the dependencies of the method.
    a. Input data: Either integer or ModulePriotity object. No need to alter the code as the module package (where the ModulePriority class is located) was automatically created when creating the class.
    b. Output data: It returns an object of ModuleDescriptorBuilder class. There is a need to input this object and return it. Although we do not manipulate any aspect of this object,

there is a necessity to maintain the interface of the GodClass. So, we modify the function signature to take ModuleDescriptorBuilder class as input.

   c. Data inside the method: It manipulates the priority variable. So, there is a need to remove declaration of priority variable in ModuleDescriptorBuilder class and include a declaration of this in ModuleDescriptorBuilderPrioirty class.

4. Inspect the responsibility of the extracted methods and resolve the dependencies outside the method.
   a. The responsibility of the extracted methods is to set the variable priority and return the ModuleDescriptorBuilder class object.
   b. The ModuleDescriptorBuilder class returns this priority variable (in its supportURL() method). There is a need to write a new getter method in to return the priority variable. So, we introduce a new method getPriority() in ModuleDescriptorBuilderPriority class.
5. Maintain the interface of God class (ModuleDescriptorBuilder): We do not change any format of this class except for the removal of the priority variable declaration.
6. Inside the extracted methods in the God Class, remove the original set of lines and delegate calls to the newly extracted class: We make two calls to the newly created class:
   a. priority (int) method calls priority(int, ModulePrioirtyBuilder)
   b. priority (ModulePriority) method calls prioirty(ModulePrioirty, ModulePrioirtyBuilder)

1.3.8 Test cases: There are test cases already present in ModuleDescriptorBuilderTest class. Specifically, it builds the description for MERGE module for various values of the six description parameters. We see that in one of the test cases (shown at the end of this paragraph) , it verifies if the priority variable obtained from ModuleDescriptorBuilder class is equal to the one obtained ModulePriority class. Because of this, we didn't see the necessity to introduce another test case to verify the getPriority functionality of ModuleDescriptorBuilderPrioity class. We verified that these test cases pass before and after refactoring.

Line 70 of ModuleDescriptorBuilderTest class:

assertEquals(ModulePriority.DEFAULT.getPriority(), victim.getPriority());

We added a new test case:

Test Case	Description	Rationale
1	**Test case**: public void **buildNullUrl**() **Input:** use null URL when creating ModuleDescriptorBuilder object **Expected output:** 1. Test Case should not throw illegal argument exception when creating the object 2. It should throw a NoSuchElementException when accessing the URL	Null is a valid argument, it should not throw an exception Test case passes.

1.3.9 Jdeodorant before refactoring: Marks ModuleDescriptorBuilder class as God Class Smell.
1.3.10 Jdeodorant after refactoring: God class Smell for this class is removed. We verified that there no new smell introduced are in the newly created class.

1.4 Smell #4: Pdfsam: Duplicated code

1.4.1 Class: ConversionUtils
1.4.2 Location: org.pdfsam.support.params

1.4.3 Smell description and analysis:
To Implement change request 3 for A2, we added a method named *toPageRangeList(), in ConversionUtils.java,* to preserve duplicate page ranges in a merge request. But in the implementation we basically copied the functionality of another method named *toPageRangeSet()* and changed it to return a list instead of a set. These methods perform the same functionality

1. Create a list / set
2. Go over the all page ranges
    a. Converts the string page range to PageRange object
    b. Validate the page range
    c. Add the object to list / set
3. Return the list / set

So, these methods were listed in Nicad4 results with 85% similarity.

1.4.4 List of refactorings: Extract Method *addPageRanges from toPageRangeList() & toPageRangeSet()* methods

1.4.5 Rationale: Duplicated code make the changes propagate to other duplicated areas. Any change in one section has to be made in all duplicated sections. So, in this case we extracted the duplicated code into a new method and call this new method in place of the duplicated code

1.4.6 List of manual code changes:

1. Create a new method and extract the duplicated code, i.e. subpoint 2 in previous section which iterates over page ranges and adds it to the list / set,
    - *private static Collection<PageRange> addPageRanges(String selection, Collection<PageRange> collection)*

    The method uses Collection interface so that it is compatible with List and Set interfaces. This method performs the following functionality

    Go over the all page ranges
    1. Converts the string page range to PageRange object
    2. Validate the page range
    3. Add the object to collection
    4. Return the collection
2. Call this method in *toPageRangeList()* and *toPageRangeSet()* methods

1.4.7 Test cases included:

Pdfsam's test suite already has a Test class named *ConversionUtilsTest.java,* which tests toPageRangeSet() method. And as a part of A2. we also created 8 test cases for toPageRangeList() method. All of these test cases pass before and after the change.

Interestingly, our first implementation, to remove this smell, was to call *toPageRangeList()* in *toPageRangeSet()* and convert the list to a set before returning. In that case too all of the test cases passed in *ConversionUtilsTest.java*, but *validInput()* test in *AlternateMixSelectionPaneTes*t class failed because the conversion from list to set did not preserve the order of the page ranges. So we had to try another implementation.

We created 2 new test cases in *ConversionUtilsTest.java*

Test Case	Description	Rationale
1	**Test case**: public void multiple3() **Input:** Page selection "2-4, 2, 2-4" **Expected output:** 1. Result should contain 3 page ranges 2. First range should start from 2 3. First range should end at 4	To verify that intersecting and duplicate page ranges are preserved Test case passes.
2	**Test case**: public void multiple4() **Input:** Page selection "2-4, 2-4" **Expected output:** 1. Result should contain only 1 page range for duplicated page ranges 2. First range should start from 2 3. First range should end at 4	To verify that duplicate page ranges are not preserved and only one element remains. Test case passes.

1.4.8 Jdeodorant before refactoring: We could not make any of the required code clone tools results work with Jdeodorant. We used Nicad4 to find cloned methods and these (*toPageRangeList* and *toPageRangeSet*) were listed in results with 85% similarity. We then performed the refactoring manually**.**

1.4.9 Jdeodorant after refactoring: The methods are not listed in Nicad4 results after refactoring.

2. Automated refactoring

2.1 Smell #5: Jedit: Type checking

2.1.1 Class: HyperSearchRequest.java
2.1.2 Location: /org/gjt/sp/jedit/search
2.1.3 Method: doHyperSearch(Buffer, int , int)

2.1.4 Smell description and analysis:
When we see a type check smell, it is telling us that the function should be split into multiple functions that handle a single type or that we should rely on JavaScript dynamic nature and abuse duck-typing and type casting/coercion. In this example, the set of lines marked as type checking smell is a simple if-else loop. The if condition compares if the object "matcher" is an instance of BoyerMooreSearchMatcher class. Here, the matcher object is in fact an instance of SearchMatcher, parent of BoyerMooreSearchMatcher class. We believe that this is a valid smell because the two inherited class- BoyerMooreSearchMatcher and PatternSearchMatcher can be considered as a type of SearchMatcher class and the code can be modified to better utilize OOPS design principles.

2.1.5 List of refactorings: Replace conditional with Polymorphism

2.1.6 Rationale: Replacing the conditionals with Polymorphism makes sure that the code is neat, readable and follows the design principles.

2.1.7 List of automatic code changes:

1. Replace the conditional structure in doHyperSearch method with polymorphic method invocation.
2. Add abstract method in SearchMatcher class
3. Add concrete method in BoyerMooreSearchMatcher and PatternSearchMatcher classes.

2.1.8 Test cases: The doHyperSearch method is called in serachInSelection method of the same class which in turn is called by the overridden run method in the same class. doHyperSearch is an overloaded method. The two overloaded methods take the buffer(string to be searched), starting and ending positions as input. The difference is in the type of selection we make: rectangle or none. The method in which we made the change is where the selection is none. In other words, when we place a mouse pointer at anywhere in the text editor and do a hyper search, it gives results (searches, lists and highlights the query in all the lines where the query is located, highlights the complete line in which the original query is located and the reports of number occurrences of the query) for the immediate next word from the start of the mouse pointer. To verify this functionality, we performed the following test cases manually.

Test Case #	Description	Rationale
1	Test case defined: hyperSearch_Test1	To verify if the

	Inputs: Mouse pointer is at the start of the word “hyperSearch” at line 145 in SearchBar.java file opened with jEdit. In the Search menu of jEdit, following choices were made: 1. “Whole Word” option checked 2. “Ignore Case” option unchecked 3. “Regular Expressions” unchecked Expected output on the click of “Search->HyperSearch for word”: -> Highlights the complete 145th line -> Highlight “hyperSearch” in all the lines where it appears -> total number of occurrences: 18 in 17 lines	doHyperSearch functionality works. Test case passes.
2	Test case defined: hyperSearch_Test2 Inputs: Mouse pointer is at the start of the word “hyperSearch” at line 145 in SearchBar.java file opened with jEdit. In the Search menu of jEdit, following choices were made: 1. “Whole Word” option unchecked 2. “Ignore Case” option checked 3. “Regular Expressions” unchecked Expected output on the click of “Search->HyperSearch for word”: -> Highlights the complete 145th line -> Highlight “hyperSearch” in all the lines where it appears -> total number of occurrences: 26 in 23 lines	To verify if the doHyperSearch functionality works. Test case passes.
3	Test case defined: hyperSearch_Test3 Inputs: Mouse pointer is at the start of the word “hyperSearch” at line 145 in SearchBar.java file opened with jEdit. In the Search menu of jEdit, following choices were made: 1. “Whole Word” option checked 2. “Ignore Case” option checked 3. “Regular Expressions” checked Expected output on the click of “Search->HyperSearch for word”: -> Highlights the complete 145th line -> Highlight “hyperSearch” in all the lines where it appears -> total number of occurrences: 24 in 22 lines	To verify if the doHyperSearch functionality works. Test case passes.
4	Test case defined: hyperSearch_Test4 Inputs: Mouse pointer is at the start of the word “hyperSearch” at line 145 in SearchBar.java file opened with jEdit. In the Search menu of jEdit, following choices were made: 1. “Whole Word” option unchecked 2. “Ignore Case” option unchecked 3. “Regular Expressions” checked Expected output on the click of “Search->HyperSearch for word”: -> Highlights the complete 145th line -> Highlight “hyperSearch” in all the lines where it appears	To verify if the doHyperSearch functionality works. Test case passes.

	-> total number of occurrences: 18 in 17 lines	
5	Test case defined: hyperSearch_Test5 Inputs: Mouse pointer is at the start of the word “hyperSearch” at line 145 in SearchBar.java file opened with jEdit. In the Search menu of jEdit, following choices were made: 1. “Whole Word” option unchecked 2. “Ignore Case” option unchecked 3. “Regular Expressions” unchecked Expected output on the click of “Search->HyperSearch for word”: -> Highlights the complete 145th line -> Highlight “hyperSearch” in all the lines where it appears -> total number of occurrences: 18 in 17 lines	To verify if the doHyperSearch functionality works. Test case passes.

2.1.9 Jdeodorant before refactoring: Marks the lines 225-228 as type checking smell in doHyperSearch method of HyperSearchRequest class.

2.1.10 Jdeodorant after refactoring: Type checking smell for these particular lines of code is removed. We wondered if there would be a feature envy smell in the concrete methods of BoyerMooreSearchMatcher and PatternSearchMatcher classes as their only functionality is to modify the Cancellable attribute of HyperSearchRequest class. But, when running Jdeodorant, we didn’t notice any additional smells for the newly introduced methods.

2.2 Smell #6: Jedit: God Class

2.2.1 Class: FilesChangedDialog.java
2.2.2 Location: org/gjt/sp/jedit/gui/

2.2.3 Smell description and analysis:
This is a newly identified smell because we used Jdeodorant in this assignment instead of iPlasma, which was used for A3. FilesChangedDialog class is listed as a God Class. On code analysis we can see that this class not only creates the UI elements of the the “Files Changed” dialog, but it also implements the controller functionality for the dialog box like action handlers for the click buttons, selection functionality of the files that are listed on the dialog, representing the files using TreePath objects etc. Ideally this should be implemented via MVC framework. So, this is a genuine smell.

2.2.4 List of refactorings: Extract class - *FilesChangedDialogUpdateButtons* from *FilesChangedDialog* class

2.2.5 Rationale:

The Extracted class contains the functionality to update the *reload* and *ignore* buttons in the dialog box. This update, to the *ignore* and *reload* buttons, depends on whether or not some of the listed files, in the dialog, have been selected. Extracting this code from the *FilesChangeDialog* class makes it more cohesive than it was before as it separates some of the code that handles the GUI part from the controller part. It is still not fully cohesive because it extracts only a part of the GUI code and keeps functionality for *selectAll*, *cancel*, *view* etc. features still in *FilesChangeDialog*. Ideally the extraction should have completely separated the action handlers and other controller functionality from the GUI code.

2.2.6 List of code automatic and manual changes:

Automatic Changes

1. Create FilesChangeDialogUpdateButtons class
2. In FilesChangedDialog class:
    a. Create a field holding a reference to the extracted class
    b. Remove extracted fields
    c. Remove extracted method
    d. Change access of extracted method in:
        i. FilesChangedDialog's constructor
        ii. selectAll() method
        iii. ActionHandler inner class
        iv. TreeHandler inner class

Manual Changes

1. Change the method call in *filesChangedDialogUpdateButtons.updateEnabled(**this.bufferTree**);* to *filesChangedDialogUpdateButtons.updateEnabled(**bufferTree**);* in *valueChanged()* method of the *TreeHandle*r inner class.

2.2.7 Test cases included:

Test Case	Description	Rationale
Precondition (For all test cases)	1. Click on **Global Options** in **Utilities** menu 2. Go to "General" 3. Select "Prompt" from the dropdown menu for "If files are changed on disk" option	
1	**Test case**: Test *Reload* and *Ignore* buttons remain disabled when no files selected **Steps:** 1. In Jedit, write some text in a new file and save it. 2. Open the file in some other text editor and save it. 3. Come back to Jedit 4. The FilesChangeDialog Should Appear 5. Click "Changed on Disk" **Expected output:** *Reload* and *Ignore* buttons are disabled.	To verify that *Reload* and *Ignore* buttons are disabled when no files are selected. Test case passes.
2	**Test case**: Test *Reload* and *Ignore* buttons become enabled when *Select All* is clicked **Steps:** 1. In Jedit, write some text in a new file and save it. 2. Open the file in some other text editor and save it.	To verify that *Reload* and *Ignore* buttons are enabled when all files are selected.

	3. Come back to Jedit 4. The FilesChangeDialog Should Appear 5. Click “Select All” **Expected output:** *Reload* and *Ignore* buttons are enabled.	Test case passes.
3	**Test case**: Test *Reload* and *Ignore* buttons become enabled when *Filename* is clicked **Steps:** 1. In Jedit, write some text in a new file and save it. 2. Open the file in some other text editor and save it. 3. Come back to Jedit 4. The FilesChangeDialog Should Appear 5. Click “Changed on Disk” 6. Click on the filename you are working with. **Expected output:** *Reload* and *Ignore* buttons become enabled when file is clicked.	To verify that *Reload* and *Ignore* buttons are enabled when a particular file is selected. Test case passes.

2.2.8 Jdeodorant before refactoring:
org/gjt/sp/jedit/gui/FilesChangedDialog.java is listed as God Class in the JDeodorant result.

2.2.9 Jdeodorant after refactoring: Both *org/gjt/sp/jedit/gui/FilesChangedDialog.java and org/gjt/sp/jedit/gui/FilesChangedDialogUpdateButtons.java* are not listed in the JDeodorant result.

2.3 Smell #7: Pdfsam: Type Checking

We are using the smell detected in A3.

2.3.1 Class: RememberingLatestFileChooserWrapper.java
2.3.2 Location: org.pdfsam.ui.io
2.3.3 Method: public File showDialog(Window ownerWindow, OpenType type)

2.3.4 List of refactorings: Replace Type Code with State/Strategy

2.3.5 Rationale:
Replacing the SAVE and OPEN type with a user defined type makes it easier to add additional type in the future and stick with the design principles of object oriented programming. Here, the two types are already put together in a enum variable named OpenType. Instead of checking the type of each constant in the OpenType,, a strategy to create methods for this data type is wise. For example, we have methods for default data types like int and string. Similarly, we can define methods (showDialog is the method in our case) for each of the constant in the OpenType datatype. Automatic refactoring does not implement the showDialog method for OPEN type because there is no implementation of this function in the code. As the principle of refactoring is not to change the functionality of the code, we did not add any new functionality (the showDialog for OPEN type) to the code. Whereas we have the implementation (showDialog method) for the SAVE type.

2.3.6 List of code automatically changes:

1. Create getTypeObject method to return the specific type for the enum variable OpenType.
2. Replace switch-case structure with polymorphic method invocation: This invocation first calls the getTypeObject method and then the showDialog method for that specific data type.
3. Create Type.java class in the same package: To create an abstract method showDialog for all the constants used in the OpenType datatype
4. Create Save.java class in the same package: This class extends the Type class and has its own specific implementation of the showDialog method.

2.3.7 Test cases:
The showDialog method with the two input arguments (the identified smelly method) is used in BrowsableFileField class. Specifically, it is used in a private class named BrowseEventHandler which extends the event handler class (is a callback routine that operates asynchronously and handles inputs received into a program). There are GUI test cases for these methods in the same package. These test cases pass before and after refactoring.

2.3.8 Jdeodorant before refactoring: Marks the lines 85-91 (of the master branch since these lines would have change in the a4_refactoring branch) as type checking smell in showDialog method of RememberingLatestFileChooserWrapper class.
2.3.9 Jdeodorant after refactoring: Type checking smell for these particular lines of code is removed. Running Jdeodorant again, we didn't notice any additional smells for the newly introduced methods.

2.4 Smell #8: Pdfsam: Long method

2.4.1 Class: SelectionTable
2.4.2 Location: org.pdfsam.ui.selection.multiple

2.4.3 Smell description and analysis:
A method that contains too many lines of code is considered to have the long method smell. The showPasswordFieldPopUp method contains twelve lines of code and has a nested loop depth of two. The functionality of this method is to get the coordinates of the editor and show a password window. Jdeodorant flags this as the long method smell and suggests to refactor the lines of the inner-most loop. We believe that having more than ten lines of code is considered as a long method.

2.4.4 List of refactorings: Extract Method: passwordPopupDisplay -> extracted from -> passwordPopup from -> extracted from -> showPasswordFieldPopup

2.4.5 Rationale:
Long methods typically mean that the method contains too many statements. If we subgrouped these statements and extract them as smaller methods there is a good possibility that those methods might perform some functionality that other methods may also need. So it would leads to a better reuse of the code and can reduce duplicated code.

We had to recursive apply this refactoring twice as the extracted method was also listed by Jdeodorant in subsequent run. After both refactorings, the original method(*showPasswordFieldPopup*) calls the intermediate extracted method (*passwordPopup*) after variable initialization and null checks. Similarly, *passwordPopup calls passwordPopupDisplay after* variable initialization and null checks. So the extracted method, which calculates the X and Y coordinate anchors and showing the popup anchored at those points, has a more narrower focus in terms of its responsibilities than the original one.

2.4.6 List of code changes automatically:

1. Create passwordPopup (extracted) method
2. Modify showPasswordFieldPopup method

2.4.7 Test cases included:

The org.pdfsam.ui.selection.multiple.SelectionTableTest class already has testcases namely onSaveWorkspaceEncryptedPwdStored and onSaveWorkspaceEncryptedNoPwdStored which uses the password popup in its test cases. We verified through regression testing that the test cases pass after the refactoring operation is performed.

2.4.8 Jdeodorant before refactoring:

Jdeodorant highlights *showPasswordFieldPopup as a Long Method*. Once extract method refactoring is applied, it also highlights the extracted method as a long method.

2.4.9 Jdeodorant after refactoring:

None of the original method and the extracted methods are highlighted by Jdeodorant after refactoring.

---

3. Comparison of Automated and Manual Refactoring

	Automated	**Manual**
Advantages	**1. Fast**: It's all automated, so barely takes any time. There might be some errors after refactoring which have to be removed manually. But overall it's far less than manual refactoring. **2. Easy**: In most cases it's just takes a few clicks. **3. Reliable**: Changes are applied to all appropriate places by itself. **4**. **Design Knowledge not required**: The automated tools can perform the required changes in all places and do a good job in refactoring smells like God Class, Type Checking which involve introducing new classes. Such refactoring, if done	**1**. A manual refactoring is more likely to be a thorough smell removal process than the automated one. The drawbacks of using automated refactoring in section 2.2 could be avoided if manual refactoring is used. **2.** More likely that a thorough analysis will lead to a better design as compared to automated refactoring.

	manually, would require someone to have a good understanding of the design to successfully make the changes.	
Disadvantages	**1.** As seen in section 2.2, automated refactoring may not fully remove with the smell. **2.** It is quite likely that automated refactoring, for smells like God class, might degrade the quality of the design when the extraction is not as thorough as discussed in the previous point	**1**. Requires a good understanding of the project design **2.** It is hard to know the change propagation. **3.** So, it is more time consuming than automated refactoring. **4**. It is quite likely that some of the changes could be missed altogether
Difficulties	It is time consuming to understand the code and create test cases for Jedit. Whereas for pdfsam, the already existing test cases helped to create new one wherever needed.	It is challenging to figure out the change propagation for the given refactoring. But, with the practice of doing refactoring on 8 smells, we were able to realize that refactoring is done on small scale code and we can adopt techniques (such as preserving the interface of the smelly class and delegating calls to the refactored code) to mitigate this challenge.