# Assignment 4: Refactoring

Ben Sattelberg and Matt Dragan

Team reNo

## 1 Automatic Refactoring One: jEdit

### 1.1 Note

Two of our three code smells we determined were not actually code smells, and our third was that jEdit had type checking issues, but when we tried to fix this smell using automatic refactoring it introduced several errors that were hard to track down because it was hard to determine where changes were made and what and where classes and methods were added. We consider new smells for jEdit that were not in our assignment 3.

### 1.2 Code Smell

The method addExplicitFold in TextArea.java has a feature envy smell towards JEditBuffer.java. The addExplicitFold method was using significant functionality from JEditBuffer, and the only functionality it seemed to be using from TextArea was the JEditBuffer contained in a TextArea. This smell was detected using JDeodorant.

### 1.3 Code Changes

We used JDeodorant's automatic refactoring "move method" tool to move the addExplicitFold method. The method was moved to JEditBuffer.java class. This change fixed the smell. We also experimented with the AutoRefactor plugin to compare the two, and we were interested in the fact that it automatically detected the method we were trying to move, even though it does not seem to have support for determining code smells. The AutoRefactor plugin moved the method to

JEditActionContext.java for reasons we didn't understand, but we thought it was interesting that this tool chose to move the function hear. The change made by AutoRefactor fixed the smell as well.

### 1.4 Rationale

Moving a method that is envious of the behavior of another class to that class is useful because it decreases coupling and tends to increase cohesion. In this case, we aren't quite sure why AutoRefactor moved the method to JEditActionContext, but (at least according to JDeodorant) this fixed the smell.

## 2 Automatic Refactoring Two: jEdit

### 2.1 Code Smell

The ParserRuleSet.java class is considered to be a god class by jDeodorant. This seems to be correct, as the file deals with all the behavior of adding and using rules for parsing text matching a given token.

### 2.2 Code Changes

We used jDeodorant's automated refactoring tool to do an extract class. This created a ParserRuleSetData class that contains all the member variables of ParserRuleSet and a constructor for the class which initializes two member variables. Other than the constructor this method does not have any other functionality. Somewhat surprisingly, this (according to jDeodorant) fixed the code smell. We thought that this behavior was interesting because many of the methods in the ParserRuleSet class are getters and setters, so the class seems to be mostly a data class anyways.

### 2.3 Rationale

We chose to use extract class as a way to break up the functionality of ParserRuleSet. However, the automated behavior of jDeodorant does not seem to match what we would desire for actually fixing this code smell. Pulling the member variables to another class certainly splits up the ParserRuleSet,

but it still does too much. It would likely be better to create separate classes for storing and applying the rules.

# 3 Automatic Refactoring Three: PDFsam

## 3.1 Code Smell

The first smell in PDFsam was a Complex conditional in the initUI method in the WindowStatusController class. This smell was discussed previously in assignment 3. This smell was detected using Designite tool. Complex conditionals make it difficult to understand what condition is being checked.

## 3.2 Code Changes

This method has associated tests, so no further tests were written. Since the bulk of the complexity of this method was the combined conditional used for splitting between the if and else branches, we used the AutoRefactor tool's extract method to extract the condition itself to its own method with a meaningful name.

## 3.3 Rationale

The main smell in this function is that it's difficult to tell what the branches in the if-else mean. Adding a function that computes the conditional value allows us to create a function name that gives the abstract meaning of that conditional (in this case that it is checking if the UI is restorable to a previous configuration).

# 4 Automatic Refactoring Four: PDFsam

## 4.1 Note

In assignment three, one of PDFsam's code smells was designated as not a smell. The second was fixed in refactoring three. The third was the complex method initDragAndDrop in SelectionTable. We concluded this was a code smell, but since it is a complex system of lambda functions and hooks to the UI layer, our attempts to refactor would have required extracting functions that just pass

lambda functions to the UI layer which we felt didn't solve the problem well. We decided to try to fix a different code smell instead.

### 4.2 Code Smell

In the encrypt (and decrypt) methods in EncryptionUtils.java, there is a magic number, 16, corresponding to the block length of the input to the AES256 encryption algorithm. This is a code smell because it's difficult to know without being familiar with the AES256 algorithm that the number 16 corresponds to the byte length of the 128 bit blocks.

### 4.3 Code Changes

This method has associated tests, so no further tests were written. We extracted a local variable using the AutoRefactor tool's extract local variable with the name BLOCK_SIZE_IN_BYTES to denote a more understandable description. We then noticed that the encrypt and decrypt methods both used this value, so we (manually) moved this local variable to be a class variable and used it in the encrypt and decrypt methods.

### 4.4 Rationale

Having a name for this value allows people who aren't as familiar with the AES256 algorithm's block size to know what the 16 is referring to. Previously, it would have been difficult for a reader to understand this value unless he or she already had the background.

## 5 Manual Refactoring One: jEdit

### 5.1 Code Smell

In jEdit.java's showMemoryDialog method, there are three locations where 1024 is used to convert between bytes and kilobytes, resulting in a magic number code smell. This is a code smell because without knowing how a Runtime object handles memory allocation, it would be difficult to know what the 1024 refers to.

### 5.2 Code Changes

A variable BYTES_PER_KILOBYTES was created that has the value 1024 and is used instead of that magic number in those three locations.

### 5.3 Rationale

Using this variable makes it clear what units the memory is in, as well as showing why the value 1024 is used at all. The addition of this variable improves the readability of the code.

## 6 Manual Refactoring Two: jEdit

### 6.1 Code Smell

The function goToEndOfCode in TextArea.java has a complex conditional relating to determining if a token is part of a comment. Since it uses values from the Token class to determine if it is a comment, it is not clear what the condition of the conditional actually means without further investigation.

### 6.2 Code Changes

We created a function called isComment in the Token class that implements the behavior of the original condition. There is an existing function that checks if a token is a comment or a literal, so this change did create some duplicate code.

### 6.3 Rationale

This function makes it more clear what the conditional is actually doing. Previously it was unclear what Token.COMMENT1 through Token.COMMENT4 referred to, but now they have been shifted so that the reader only needs to see that this deals with determining if something is a comment. The reason the new method was placed in the Token class is that it interacts directly with token behavior and it matches the structure of methods already existing in that class.

# 7 Manual Refactoring Three: PDFsam

## 7.1 Code Smell

The class TooltippedTextFieldTableCell has a constructor that is a complex method. This constructor is smelly because it handles multiple different abstract ideas in the code. One of the main sources of complexity seems to be adding event handlers to the TextField object it holds.

## 7.2 Code Changes

This method has associated tests, so no further tests were written. We extracted the portion of the constructor that added event handlers to the TextField into a new method called addActionEventsToTextField.

## 7.3 Rationale

By extracting that portion of the code, it makes it more clear what each portion of the constructor is doing, abstracting some of the complexity away makes it easier for the reader to understand how to work with this class.

# 8 Manual Refactoring Four: PDFsam

## 8.1 Code Smell

In the toPageRangeSetMergeOverlappingRanges method (that we wrote for assignment two) of ConversionUtils, there is a complex conditional. This is a complex conditional, as it has five separate checks, four of which are similar checks about the bounds of the page ranges. One challenge we ran into in assignment 2 is there is a method that checks for overlapping page ranges, but it doesn't check if one page range is nested within another. We had to add extra conditions to test for this which led to a complex conditional.

## 8.2 Code Changes

This class has associated tests we wrote for assignment two, so no further tests were written. We extracted a method that checked the bounds of the page ranges to combine the four predicates in

the condition check. Ideally this method would go in the PageRange class, but as that is in the sejda module it is not something we are able to change.

### 8.3 Rationale

Rather than having a difficult to interpret block of coding dealing with checking if there is an overlap, there is now a function that abstracts that behavior away so that the toPageRangeSetMergeOverlappingRanges is easier to understand and the new isNested method is easier to interpret with the new name.

## 9 Manual versus Automated Changes

Automated changes were extremely useful for small-scale changes such as extracting methods or variables. However, (especially with jDeodorant) larger scale refactorings could result in behavior that "fixed" the code smell according to the tool used but did not actually result in cleaner code. Additionally, large-scale refactorings seemed to be more likely to introduce errors that are hard to track down due to the addition of many classes/methods (ex. when we tried automatic refactoring to remove type checking smaells in jEdit). When we made manual changes, we didn't have to worry about "fixes" that did not actually address the solution, but manual refactoring requires significantly more time put towards writing code that may not be particularly meaningful (such as method headers). It seems to us that the best method would be to use automated tools for creating the structure of changes while doing the design and intricate implementation of those changes by hand.