CS 515-Assignment 04-Code Smells and Refactoring

Marylou Nash
Farzaneh Elyaderani

____
_

Note:

The refactored source code for both projects is merged into the A4_refactoring branch.

Detecting and Analysing Code Smell

For this assignment we used different tools to detect and analyze code smells.

- **"IntelliJDeodorant/JDeodorant":**
  IntelliJDeodorant can be run in IntelliJ with some effort. It is not available via IntelliJ's marketplace. For this reason, we downloaded the .jar file directly from [1]. Then we tried to install the plugin from the downloaded .jar file. Installing the intelliJDeodorant plugin from the external jar file was not successful, as this version of plugin was not compatible with our IDE. To fix this issue we modified the plugin.xml file, to make the version compatible with our IDE. JDeodorant is easily loaded into the Eclipse IDE. Unfortunately, the analyzer can only be run on compiled code thus limiting it's ease of use. This tool only detects four major categories of smells - God Class, Type-State Checking, Long Method and Feature Envy.

1. **God class-** As the name suggests it is a name given to a very large and complicated class containing too many components. In other words, a God class is a class that tries to do many tasks, instead of implementing one single responsibility. The plugin usually identifies the set of methods and attributes that can be extracted and moved to a new class, i.e. **Extract Class** refactoring.

2. **Long Method-** Methods that are too long including many local variables and conditional statements detected as long methods. Long methods are not preferable, as they make the code inefficient and hard to understand and maintain. IntelliJDeodorant usually performs **Extract Method** through identifying a block of code that is responsible to implement a specific task and assigning it to new methods.

3. **Feature Envy -** This code smell occurs in method level, when a method uses other classes' methods/ attributes more than ones belonging to its own class. IntelliJDeodorant detects this behavior as a code smell, and provides suggestions to resolve this issue through performing **Move Method** refactoring, i.e.

suggesting the most related class to move the method, in order to reduce coupling.

4. **Type Checking-** This code smell is related to the case when the outcome of a program is determined by a set of complicated conditional statements, such as switch case, if else, etc. The plugin detects such fragments as a code smell and suggests refactoring through **Replacing Conditional with polymorphism.** Replacing conditional with polymorphism provides some benefits, such as:

    a. Remove duplicate code

    b. Satisfy open/ close discipline

    c. Adheres to the Tell-Don't-Ask principle

- **"SonarLint":**

SonarLint is a plugin available on IntelliJ and easily loaded from their marketplace. SonarLint does not require compiled code to run on and even provides feedback on code as you develop it in the editor. It detects a wide variety of code smells, large and small, ranking them into 5 categories with decreasing severity - blocked, critical, major, minor and info. The user can select specific directories or files to run it on and a list of code smells with a short description and ranking will be produced. Clicking on a specific description will pop up a more complete description of the code smell with examples and suggestions on fixes. Double clicking on the code smell will take the user directly to that location of code in the editor.

- **"PMD":**

PMD is a source code analyzer that catches common programming flaws, such as unnecessary object creation, unused variables, etc. The reported violations can be categorized in six different categories, i.e. best practices, code style, design, documentation, error prone and performance. Using this plugin was not really helpful in detecting code smells, as it does not specify that the detected violations are code smells or not.

- **"Code Smell Detector":**

This is another plugin available in intelliJ IDE's marketplace. This code smell detector finds three different smells, including 1-excessive cyclomatic complexity, 2-using conditional instead of polymorphism and 3-return private mutable fields, and suggests proper refactoring to resolve them. We used this tool to refactor type-checking code smell.

1- PDFsam

1.1. Code Smell One - Define a constant instead of repeating fixed values

- **Location:** RotateOptionsPane class **(refactoring is provided for this case)**

1.1.1- Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

This smell was detected by SonarLint. It is located in the RotateOptionsPane class of the Rotate package. There were actually two smells of the same type, so both will be fixed at the same time.

In three different methods of the RotateOptionsPane class, the same hard-coded string value is passed to other methods. The other hard-coded string value also occurs in three different methods.

- *this.rotationType.setId("rotationType");*
  - *this.rotation.setId("rotation");*

1.1.2- Explain why the class/method is flagged as smelly (be specific).

SonarLint is flagging both of these as critical code smells. That is the second highest type of code smell that it detects. The tool is indicating that they should be implemented as constants.

1.1.3- Answer the next question: do you agree that the detected smell is an actual smell? Justify your answer.

Yes this is a code smell. It is never a good practice to use immutable values inline in the programming statements[5]. If the value is spread throughout the code, it could be considered a "change preventer" type of code smell. This is especially true when the same value is used in multiple places. Should the value ever need to be changed, it can become a maintenance nightmare. If the value is repeated throughout the code, it is difficult for the programmer to ensure she has found and correctly changed every possible location of the value. Imagine trying to do this for a value that is very simple and common such as the number "1". It can be difficult for a programmer to know that two occurrences of a common value are really representing the same thing and both need to be changed.

Additionally, assigning a constant to a well named variable can actually help with readability and understandability of the code and reduce confusion. For example, a value of 100.4 is more quickly understood when it is named FEVER_TEMPERATURE.

1.2. Code Smell Two - Feature Envy

1.2.1- Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

- **Location:** request(pdfFilesListRequest event) method, located in pdfLoadController class

Feature envy is described as a code smell, which occurred at method level. In other words, a method suffers from feature envy code smell, if it uses attributes/ methods of other classes, more than those that belong to its own class [2][3]. The feature envy smell is detected by intelliJDeodorant plugin on request(pdfFilesListRequest event) method, located in pdfLoadController class, one of the classes belong to the pdfsam-service package (org sub package).

- *public void request(PdfFilesListLoadRequest event)*

1.2.2- Explain why the class/method is flagged as smelly (be specific).

IntelliJDeodorant plugin detected this code smell, and flagged the request method as a smelly component that needs to be refactored. The reason that request method is flagged as a smell is because it has coupling with other classes, in particular the PdfFilesListLoadRequest class.

While the request method uses its own class's methods and attributes (i.e. getOwnerModule and list) to implement its functionality, it has a tie coupling with pdfLoadRequestEvent class as well. It seems that request method has more coupling with PdfFilesListLoadRequest class, compared to pdfLoadController class (the original class that request functions is located in) - as pdfLoadRequestEvent and PdfFilesListLoadRequest are both inheriting ModuleOwned class.

Therefore, PdfFilesListLoadRequest class is a better option to locate request class, rather than pdfLoadController class. For this reason, intelliJDeodorant has provided suggestions to perform appropriate refactoring to solve the smell. The details will be explained in the next parts.

1.2.3- Answer the next question: do you agree that the detected smell is an actual smell? Justify your answer.

Yes. Given that the objective of object oriented programming is to encapsulate each concept or section of the program, having as few dependencies on outside classes as possible is desirable. If a class is more dependent on another class than on its own class then the programmer has not done a good job of defining and partitioning the pieces of the project.

In this code smell, the method request() in the PdfLoadController class invokes a method in the PdfFilesListLoadRequest class twice. Thereby, making IntelliJDeodorant think that request() was more strongly attached to the other class.

What IntelliJDeodorant didn't account for sufficiently was that request() also accessed one method in PdfListParser. PdfListParser and PdfLoadController are in the same package while PdfFilesListLoadRequest is in another package. By moving request() to PdfFilesListLoadRequest it introduces a cyclic dependency. (There already was a one-way dependency.) Given that cyclic dependencies are very bad code smells and shouldn't be done, we elected not to fix this code smell.

1.3- Code Smell Three - Long Method

- **Location:** filterEvenOddPages method located in rotateParameteBuilder class **(refactoring is provided for this case)**

1.3.1- Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

As the name suggests, it occured when the method is long (has many lines of codes), and can be divided into smaller methods. Generally, having a method including more than ten lines of code should be taken into account as a candidate to perform refactoring[3]. This is because long methods usually include many variables and conditional statements (which are not preferable), as long procedures are hard to understand and maintain.

CS 515-Assignment 04-Code Smells and Refactoring

Marylou Nash
Farzaneh Elyaderani

The filterEvenOddPages method located in rotateParameteBuilder class is considered as a long method, as it consists of multiple conditional statements to perform multiple tasks.

- protected Set<PageRange> **filterEvenOddPages**(Set<PageRange> pageSelection, PredefinedSetOfPages evenOddAll, Integer lastPage)

1.3.2- Explain why the class/method is flagged as smelly (be specific).

The filterEvenOddPages method is considered as a **critical** code smell by sonarLint plugin, as the cognitive complexity of this method is quite high (by 24), while the maximum allowed range is 15. Cognitive complexity is a measurement indicating the hardness of understanding the control flow of a method. Refactoring could resolve this problem and eliminate the code smell from the list, along with decreasing the complexity.

This method is responsible for finding the page range provided by users, extract the ultimate pages according to the desired filter(i.e. all pages, even pages, or odd pages) and rotate them. The method implementation is quite complicated, and contains lots of conditional statements (if else). For this reason, filterOddEven method is detected as a long metod with high cognitive complexity, and code smell detector highlighted it to consider for refactoring.

Generally, having long methods in code implementation reduces the quality of software, understandability and maintainability of the code. For this reason, refactoring long methods and extracting shorter methods is highly suggested/ recommended by smell detector tools to increase the code quality and maintainability.

To solve this smell, plugin suggested extracting a block of code - that performs a specific task- and assigning that to a new method. Then remove the extracted block of code and call the new function instead(explain in detecting and refactoring section).

1.3.3- Answer the next question: do you agree that the detected smell is an actual smell? Justify your answer.

CS 515-Assignment 04-Code Smells and Refactoring

Marylou Nash
Farzaneh Elyaderani

Yes, Long Method is detected as one of the most significant and worst types of code smell by several tools, such as IntelliJDeodorant, PMD and sonarLint, Code smell Detector, etc. and they highly emphasize on removing/refactoring them.

In general, an actual code smell is an alarming sign of having deeper problems. This can be extended to the fact that long methods obviously increase the size of code, which makes a code less readable/ understandable. On the other hand, they significantly increase the complexity of code and decrease code reusability. These all can make the code's maintainability more complicated and decrease the quality of software. For this reason, most code smell detector tools are sensitive to this smell, and detect it as a critical smell that strongly needs to be refactored.

2- jEdit

2.1- Code Smell One- Type Checking (Case one)

2.1.1- Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

- **Location:** smartEnd method located in JeditTextArea class **(Refactoring is provided for this case)**.
    - *public void smartEnd(boolean select)*

Code smell detector plugin embedded in intelliJ detected this method as a component, which suffers from type checking smell. Type checking smell is related to the case when the outcome of a program is determined by a set of complicated conditional statements, such as switch case, if else, etc.

The reason that this method is reported as a smelly method is due to using conditional statements (switch case) to implement the method, instead of using polymorphism. Using switch statement usually implements a poor object oriented design, thereby most smell detector tools suggest to replace that with polymorphism mechanism. In our case, the smartEnd method is implemented with switch cases, depending on the last action count achieved by getInputHandler. More details are provided in the next section in detecting and refactoring part.

2.1.2- Explain why the class/method is flagged as smelly (be specific).

____
_

Code smell detector tool detects this smell and flagged the smartEnd method as a smelly component that needs to be refactored. The reason that smartEnd method (located in JeditTextArea class) is reported as a smell is due to determining the outcome of the smartHome method using switch case (conditional) statement. In fact, depending on the integer value returned by getLastActionCount() via View object, the method decided to call three different methods, i.e. *goToStartOfWhiteSpace*, *goToStartOfLine* and *goToFirstVisibleLine* methods as follow:

- *public void goToFirstVisibleLine(boolean select)*
- *public void goToStartOfLine(boolean select)*
- *public void goToStartOfWhiteSpace(boolean select)*

Calling these methods is implemented with poor design, i.e. switch case statement. To perform refactoring, we replaced the conditional implementation with a polymorphism mechanism (details provided in detect and refactoring section).

2.1.3- Answer the next question: do you agree that the detected smell is an actual smell? Justify your answer.

Yes, type checking is one of the most significant code smells that most smell detector tools (including IntelliJ/JDeodorant and code smell detector plugins, etc.) strongly emphasize on resolving that through refactoring. The proper refactoring suggested for this smell is replacing conditional with polymorphism.

The main reason that this smell is considered as an actual code smell and tools strongly suggested to provide refactoring is because the switch statement provides very poor design for object oriented design, which is in contrast with object oriented design principles.

Moreover, performing refactoring through replacing conditional statements with polymorphism provides some benefits, such as removing duplicate codes, satisfying open/ close discipline and the last but not least, adhering to the Tell-Don't-Ask principle. However, it should be noted that the condition (switch if) statement needs to have at least two cases to be considered as a valid candidate for checking smell, otherwise, the single case is not highlighted as a sign for potential future problems.

Type Checking Detected- Case two:

2.1.1- Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

CS 515-Assignment 04-Code Smells and Refactoring

Marylou Nash
Farzaneh Elyaderani

- **Location:** getViewConfig method located in View.java class.

The method getViewConfig() located in the view class is detected as a method, which is suffering from type checking smell. Looking into this method we see that the frame configuration (for four different cases, i.e. vertical, horizontal, both vertical and horizontal, and default) is implemented by conditional (switch case) statement. Code smell detector detects such cases as a type checking smell, and suggests to perform refactoring by replacing the conditional statement by polymorphism.

- *public ViewConfig getViewConfig()*

2.1.2- Explain why the class/method is flagged as smelly (be specific).

Code smell detector detects this smell and flagged the getViewConfig method as a smelly component that needs to be refactored. The reason that this class is reported as a smell is due to implementing frame configuration using conditional statements (switch statement along with multiple case types). For this reason, code smell detector highlighted it as a smell and provide suggestions to resolve this issue by replacing the conditional statement with polymorphism

2.1.3- Answer the next question: do you agree that the detected smell is an actual smell? Justify your answer.

Yes, type checking is one of the most important code smells that IntelliJ/JDeodorant and code smell detector plugins both strongly emphasize on resolving via refactoring. The proper refactoring suggested for this smell is replacing conditional with polymorphism.

The main reason that this smell is considered as an actual code smell and tools strongly suggested to provide refactoring originated from the fact that switch statement provides very poor design for object oriented design, which is in contrast with object oriented design principles.

Moreover, performing refactoring through replacing conditional statements with polymorphism provides some benefits, such as removing duplicate codes, satisfying open/ close discipline and the last but not least, adhering to the Tell-Don't-Ask principle. However, it should be noted that the condition (switch if) statement needs to have at least two cases to be considered as a valid candidate for checking smell, otherwise, the single case is not highlighted as a sign for potential future problems.

CS 515-Assignment 04-Code Smells and Refactoring

Marylou Nash
Farzaneh Elyaderani

2.2- Code Smell Two- Eliminate use of break with label

2.2.1- Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

- **Location:** markTokens method located in TokenMarker class **(Refactoring is provided for this case)**.
    - *public synchronized LineContext markTokens(LineContext prevContext, TokenHandler tokenHandler, Segment line)*

2.2.2- Explain why the class/method is flagged as smelly (be specific).

The use of a break and label statement in the code is being flagged as a code smell. This smell was detected by the SonarLint tool. It was categorized as a major code smell which is the middle level of severity among five different levels. Labels are flagged as a code smell because they are not commonly used in Java. Since they are used infrequently in Java, some programmers may not be very familiar with how they work. This lack of understanding may lead to mistakes when altering or adding code and could make it prone to bugs and more difficult to maintain.

A break statement without a label is used to jump out of the current iteration of a loop it is contained in - usually a switch, while or for loop. A label and break statement can be used together to allow the program to exit from an inner loop to wherever the label appears in a series of nested loops. When the code is very long and complicated this can reduce the code's readability and understandability.

2.2.3- Answer the next question: do you agree that the detected smell is an actual smell? Justify your answer.

We do agree that this is a code smell. Just because one can write the code using a break or continue statement with a label doesn't mean it is the only way to write the program. One should take time to carefully consider other options. This is especially true when there are multiple labels or long complicated methods.

When the label is located farther away from the break, it makes the control and flow of the program harder to understand. When there are multiple breaks and labels, the code begins to resemble "spaghetti" because of the myriad of ways the program can branch. It can be a very tangled mess. This makes it hard to understand.

Another consequence of this confusing control is that it makes the code hard to alter. If the code becomes bloated over time and has a control structure with break and label statements it can become very hard to detangle and break apart later.

One might argue that break/label combinations that are simple and small are not code smells. But given the propensity of code to become bloated over time, why not change the structure while it is simpler and use a more standard and understandable manner to code the program?

2.3 - Code Smell Three- Duplicate code

- **Location:** countWords and getWordOffset method located in JEditBuffer class **(refactoring is provided for this code smell).**

2.3.1- Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.

Duplicate code is a fragment of code that looks almost identical. This smell occurs when a piece of code is repeating in different places within a program (e.g. different methods in the same class). It is also very likely to have duplicated code, when multiple programmers work on different parts of a same program, thus they are not aware that similar code/task is already implemented with other programmers [6].

We can have duplicate codes while they look different but they are performing exactly the same tasks. This kind of duplication is usually harder to find and fix.

It is highly suggested to remove/refactor duplicated code, as they increase the number of lines of code and complexity, which both can impact the understandability and maintainability of the code.

2.3.2- Explain why the class/method is flagged as smelly (be specific).

The code smell detector tool installed in intelliJ detects this smell within the jEditBuffer class and suggests removing/ refactoring that. In fact, there are two methods called countWords and getWordOffset, which are implemented to count the number of words and to find the caret words' offset, respectively, in a file.

- *Public int countWords()*
- *Public int getWordOffset(int position)*

There is an identical fragment repeated in both methods, which is responsible for calculating the value of the variable space. The plugin distinguished these two fragments as a repeated/ duplicated code. The tool suggested extracting/creating a new method called getSpace() and placing calls for the new method in both classes (instead of repeated fragments).

- *Private int getSpace(String text)*

2.3.3- Answer the next question: do you agree that the detected smell is an actual smell? Justify your answer.

Yes, we definitely do agree that duplicated code is an actual code smell, known as a bad behaviour that needs to be refactored/ removed. The first reason to justify our statement is because having duplicated codes increases the number of lines of code, which effectively result in less readable code. In addition, having a large number of lines of code can increase other software quality metrics, such as cognitive complexity and maintainability, etc.

On the other hand, removing duplication can isolate the independent fragment of code, meaning that the errors are less likely to occur. This fact can highly increase code reusability [7]. For the reasons mentioned above, we can strongly claim that duplicated code is one of the major code smells that need to be refactored, if we want to increase our code quality.

CS 515-Assignment 04-Code Smells and Refactoring

Marylou Nash
Farzaneh Elyaderani

____
_

Removing Code Smells via Refactoring

1- PDFsam

1.1- Refactoring 1 - Define a constant instead of repeating fixed values

- **Location:** RotateOptionsPane class

1.1.1 Create at least one test case for the code component using JUnit. The test cases must pass at this point.

There were already several tests (onSaveWorkspace, restoreStateFrom, reset) in RotationOptionsPaneTest() that used the hard coded values to access data in the RotationOptionsPane object. These tests were just refactored to utilize the new methods that were created when the class and its attributes were refactored.

1.1.2 Perform the necessary refactoring operations to remove the smell on the code component.

The refactoring required creating two new static attributes.

- *private static final String ANGLE_ID = “rotation”;*
- *private static final String PAGES_ID = “rotationType”;*

Since these were kept as private attributes, "getter" routines were added to the class so tests or other classes could access the values. The "getter" methods were made static so the values could be accessed without creating an object.

1.2- Refactoring 2 - Extract method for Long Method

- **Location:** filterEvenOddPages method located in rotateParameteBuilder class

1.2.1 Create at least one test case for the code component using JUnit. The test cases must pass at this point.

Few test cases were already provided for rotate action over odd pages in rotateParameteBuilderTest class. We added two more new test cases in this class to make sure rotation action works properly for **"all pages"** and **"even pages"** options selected by users. The tests are pushed to the A4_refactoring branch.

1.2.2 Perform the necessary refactoring operations to remove the smell on the code component.

The refactoring is done by extracting two different blocks of code within the filterEvenOddPages method and replacing them with new methods. The first block is responsible to filter page range, based on the chosen filter by user, i.e. All pages, even pages or odd pages. We replace this part of code by filteredPaged method (#1). The second block of code was responsible to extract the pages to be rotated. We replaced this part of code with the extractPageRange method(#2). The new code is merged into the A4_refactoring branch.

#1- *private Set<PageRange> filteredPaged(Set<PageRange> pageSelection, PredefinedSetOfPages evenOddAll, Integer lastPage)*

#2- *private void extractPageRange(Integer lastPage, boolean even, Set<PageRange> filteredPages, Iterator<PageRange> it)*

2- Jedit

2.1 Refactoring 1 - Replacing Conditional with polymorphism to resolve type checking smell.

- **Location:** smartEnd method located in JeditTextArea class

CS 515-Assignment 04-Code Smells and Refactoring

Marylou Nash
Farzaneh Elyaderani

____
_

2.2.1 Create at least one test case for the code component using JUnit. The test cases must pass at this point.

The test is written in the jEditTextAreaTest.java class. We faced some issues running the tests. First of all, lots of objects that we needed to set up for this test belong to abstract classes, thereby we could not set up/ instantiate our objects properly. In addition, we got a nullPointerException error, which originated from not being able to set up our objects before testing. For this reason, we could not get our test fully passing. However, we manually tested the jEdit editor and observed that refactoring did not impact its functionality.

2.2.2 Perform the necessary refactoring operations to remove the smell on the code component.

To refactor this code smell and replace the conditional statement with polymorphism, we create a method named createJEditTextArea as a factory method that matches the case types of switch statements.

- *public static JEditTextArea createJEditTextArea (View view)*

Then we create three subclasses that will be called within the factory method depending on the case types corresponding to the switch statement (switch(view.getInputHandler().getLAstActionCount()). We Called the three subclasses as follow:

1. JEditTextArea1 for case 1;
2. JEditTextArea2 for case 2;
3. JEditTextAreaDefault for case 3 and default case;

Finally we create a shared method (abstract method named checkRecorder) that will be overridden in each subclass. We override this method in each subclass and move the code from the corresponding type case to it.

- *Protected abstract void checkRecorder (boolean select, Macros.Recorder recorder)*

Note: after creating the factory method, we replaced switch case with if else statement, since switch case statements in factory method still detected as a smell, and code smell detector suggests to replace that with if else statement as a better option.

CS 515-Assignment 04-Code Smells and Refactoring

Marylou Nash
Farzaneh Elyaderani

____
_

2.2 Refactoring 2 - Eliminate use of break with label

- **Location:** markTokens() method located in TokenMarker class

2.2.1 Create at least one test case for the code component using JUnit. The test cases must pass at this point.

The TokenMarker class has the responsibility of parsing the text in the editor and mapping it to tokens. No tests existed for any of its code. The markTokens() method, which contained the label that was removed, was a very long, complicated method. Initially, it contained about 170 lines of code with dependencies on several other classes. It had a cognitive complexity of 54 and would have been a nightmare to write a JUnit test for. So, the refactoring process was started by extracting the sequence of code containing the label and placing it unchanged in a new method.

Extracting the code sequence which performed an “unwinding” of the text lines and placing it in the method unwind(), allowed us to reduce the dependencies to just a few classes - LineContext, ParserRule and ParserRuleSet. This helped simplify the task of writing JUnit tests.

JUnit tests were written to cover each possible conditional branch through the code. This required 6 tests to cover all possible conditional branches. With these tests passing, we then performed the refactoring of the code in unwind() and removed the label and break statement.

Given that this class may also be used to parse the text in the buffer to find the location of any markers the user may have set, we performed some manual testing to ensure that markers still worked. The following options on the Marker menu were tested - Add/Remove Marker, Add Marker with Shortcut, GoTo Marker, Go to Previous Marker, Go to Next Marker.

2.2.2 Perform the necessary refactoring operations to remove the smell on the code component.

To facilitate testing, this code was first extracted into its own method without removing the label. This was done using the refactoring capabilities in the IntelliJ editor. The editor quickly and easily extracted the code and determined what parameters needed to be passed to the new method. The refactoring failed to realize that the code was operating on an object that needed to be passed back to the calling routine, so this was modified manually.

Once all the JUnit tests were in place and passing on the new method to protect the code, the label was removed manually.

The refactoring changes created a new method in order to remove the label and break statement thus eliminating the code smell. A side effect of the changes was to reduce the cognitive complexity of the original method (markTokens) from 54 to 47.

2.3 Refactoring 3 - Remove Duplicate Code

- **Location:** countWords and getWordOffset method located in JEditBuffer class

2.3.1 Create at least one test case for the code component using JUnit. The test cases must pass at this point.

The test case provided for this refactoring (extract method) is done manually. We manually test jEdit editor buffer and its functionality to ensure it is working properly (i.e. the functionality after refactoring is similar to the functionality before performing refactoring).

2.3.2 Perform the necessary refactoring operations to remove the smell on the code component.

We facilitate refactoring by taking advantage of the code smell detector tool. The plugin distinguished the duplicated fragment in both methods (*countWords* and *getWordOffset*) and extracted that to a new method called getSpace(). Then we placed calls for the new method in both classes. The code is merged to the A4_refactoring branch.

Reference:

[1] https://plugins.jetbrains.com/plugin/14016-intellijdeodorant
[2] https://waog.wordpress.com/2014/08/25/code-smell-feature-envy/
[3]
https://github.com/farikdk/cs515-001-s20-Fari-Marylou-pdfsam/tree/A4_refactoring_fari
[4] https://refactoring.guru/smells/long-method

[5] https://www.integer-net.com/test-smell-hard-coded-values/
[6] https://sourcemaking.com/refactoring/smells/duplicate-code
[7] https://refactoring.guru/extract-method