Assignment 4

Code Smells

Nishant Kashiv and Saurabh Deotale

1. Tools used for detecting code smells

We used 'Sonarqube' [2] tool for detecting code smells in jEdit and PDFsam. It is an open source tool for inspection of code quality with static code analysis for detecting vulnerabilities, bugs, code smells, duplication, etc. It is simple to use and can be setup very quickly [3]. We used the community edition of the tool (which is free), along with the command line interface for running analysis on the projects.

We also used 'jDeodorant' tool for detecting 'God class' code smells in PDFsam.

2. Code Smells

2.1 PDFsam

Repository – https://www.github.com/sd-26/cs515-001-s20-bughunters-pdfsam

Duplicated Code

Classes - DashboardItem (Interface), ModulesDashboardItem, PreferencesDashboardItem

Package - org.pdfsam.ui.dashboard in pdfsam-gui module

Tool – This code smell can be found in the tool in 'Duplication' section of the 'Measures' tab in the web view.

Description – Duplicated code is a code smell detected whenever same code is used in 'copy paste' fashion. In this case, DashboardItem is an interface which is inherited by classes ModulesDashboardItem and PreferencesDashboardItem. These classes contain identical method signatures as well as the code in these methods is identical. Following three methods are flagged as duplicate:

- id()
- pane()
- graphic()

We do not think that the detected smell is an actual smell. The methods which have similar code work on different class variables, one of which is a constant variable. This constant variable is present in both the classes and has different values in both. These constraints make it difficult to extract these methods.

Implementation – not implemented

Shotgun Surgery

Classes – ShutdownEvent (pdfsam-core:org.pdfsam), PdfLoadController (pdfsam-service:org.pdfsam.pdf), TaskExecutionController (pdfsam-service:org.pdfsam.task), PreferencesRecentWorkspacesService (pdfsam-service:org.pdfsam.ui), PreferencesUsageDataStore (pdfsam-service:org.pdfsam.module), PdfsamApp (pdfsam-gui:org.pdfsam)

Tool – This code smell can be found in the tool in 'Issues' tab under type 'Code Smells' in the web view.

Description – Shotgun surgery is a code smell which result in small changes being made to many different classes. This happens because the implementation of one feature or workflow is split between multiple classes or modules.

In this case whenever the application is shut down, some steps are to be followed before the application can be closed. Following operations are performed before the application is closed.

- Close the executor used for handling pdf documents
- Close the executor used for handling task executions (used mostly for sejda sdk tasks)
- Dump memory used for storing preferences and workspace related data.

The ShutdownEvent class is an empty class, which is supposed to be responsible for these steps. We agree that this is a code smell. All these steps cannot be incorporated in a single class or module as handlers and data storage for these workflows are already handled by different modules/services. Hence the application raises a 'Shutdown' event whenever the application must be closed. Services handling workflows for steps mentioned above are registered as listener for this event. They individually initiate different workflows to accomplish these steps. This results in detection of shotgun surgery code smell.

Resolution – The class ShutdownEvent can be converted to an interface as it is an empty class. This way we can keep the placeholder event for initiating shutdown workflows as well as remove the code smell.

Implementation – not implemented

Bloated Class (Long Class)

Class - RotateParametersBuilder

Package - org.pdfsam.rotate in pdfsam-rotate module

Tool – This code smell was detected using jDeodorant.

Description – Bloated or Long class code smell is detected when any class violates the single responsibility principle. This can be easily detected by identifying a subset of variables and methods that have less correlation with the responsibility of the class.

In this scenario the RotateParametersBuilder class has the responsibility of building parameters into a package for the rotate task performed by sejda sdk on input documents. There exists a nested workflow for compiling all the pages that will have their orientation changed in the output document. There are two methods and two variables that can be encapsulated in another class for compiling all the required pages for changing their orientation.

- Variables - rotation, predefinedRotationType
- Methods - addInput, getPageCombination

We think this is a valid code smell, as the builder class should not handle compiling these pages. This workflow is outside the scope of a builder class, as builder class is only responsible for packaging parameters, not calculating or configuring them.

Implementation – For resolving this code smells, we did 'Extract Class' refactoring. As this code smells points towards violation of single functionality principle, we wanted to maintian single functionality for a class. This

required implementation of another class which would be responsible for compiling the pages which were to be modified. We did the following changes in the source code:

- We extracted the variables and methods mentioned in the description in a new class in the same module - DocumentPageOrientation
- After that we inserted an objected of the new class in RotateParametersBuilder as a public property
- We corrected all the class to previous variables and methods, and changed them to call the DocumentPageOrientation property of the RotateParametersBuilder class
- We had to change variable and method calls in following classes:
    - RotateOptionsPane
    - RotateSelectionPane
- Once all the changes were made, we built the project

Testing – After changing the class structure of RotateParametersBuilder, we were unable to build the project successfully. We had to make changes in the existing unit test cases for the project to be built successfully. We modified following test classes to incorporate the changes as well test the new class.

- RotateOptionsPaneTest
- RotateParametersBuilderTest
- RotateSelectionPaneTest

All the test cases before and after the implemetation for the code smell.

OO Abusers

Class – Services

Package - org.pdfsam in pdfsam-service module

Description – OO Abusers are those code smells that have incorrect or incomplete application of object-oriented principles. We were not able to determine the type of OO abuser code smell as it did not fit any types mentioned in the class lecture. But the nature of the code smell makes it obvious that this is an OO abuser code smell.

In this scenario the class Services contains only one static method. This makes the class a utility class. Any utility class is not meant to be instantiated. Java implicitly creates a constructor for all the classes. The tool detects the code smell because there is not explicit private constructor for the class.

We believe this is a valid code smell as the utility class can be instantiated, whereas the implementation and design of the software has used the Service class as a utility class. It is not instantiated in any of the modules. There is only one usage of the service class during the startup process of the software, and the static method of the class is used directly.

Implementation – For resolving this code smell all we had to do was define a private constructor (‘Insert Method’ refactoring). This prevents the Service class from being instantiated and resolves the code smell.

We only had to insert a private constructor method in Services class.

Testing – Any utility class has deterministic behavior and unit testing is not possible with jUnit for utility classes. The Service class is a true utility class, which means that all its components are static and there are no non-static dependencies. Hence, unit testing is not required for this implementation.

2.2 jEdit

Repository – https://www.github.com/kashivns/cs515-001-s20-bughunters-jEdit

Bloater (Large Method)

These are methods with too many lines of code. As a rule, we should try to split a method if it exceeds cognitive complexity value 15. It is very easy to end up with multiple large methods as the software evolves, because mentally it is easier to add lines to code than creating a new method.

We can identify large method code smell using sonarqube's Cognitive Complexity. According to sonarqube "Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain." [1] Naturally long methods with multiple if then statements and nested structures have high cognitive complexity.

We can observe an instance of a large method code smell in:

Class - org.gjt.sp.jedit.Abbrevs

Method – expandAbbrev

Previous Cognitive Complexity = 23

This method is responsible to expand the abbreviation inside the jEdit buffer when the user presses Ctrl + ; in windows. There are two types of abbreviations that can be defined in jEdit:

1. Positional Abbreviation

   For example, in java a positional abbreviation is defined with F#1#2# which expands into a for loop. Where the positional arguments are used to insert loop initialization and end condition and the incrementation.

2. Ordinary Abbreviation

   We can define an ordinary abbreviation like "gm" which will expand into "good morning".

expandAbbrev was responsible for handling expansions of both types of Abbreviations and also the insertion of the expanded string in the buffer.

To reduce the method length, we have introduced 3 new private methods:

1. handlePositionalAbbrev - for handling expansion of positional abbreviations
2. displayAbbrevDialog – to display Abbreviation Dialogue box when an abbreviation is not defined.
3. insertExpansion – To insert the expanded string in the buffer

These methods are called from the expandAbbrev method. This reduced the Cognitive Complexity of this method from 23 to under the acceptable 15 and distributes the multiple responsibilities which were implemented inside a single method.

We think this code smell detected by sonarqube is an actual code smell because it contained many lines of code and performed multiple responsibilities like handling expansion of positional abbreviation, ordinary abbreviations and inserting the expanded strings into the buffer.

Test:

We tried to create jUnit test cases to test our changes. But could not get them to work. Hence, we are providing some manual tests which test the functionality of expanding ordinary abbreviation and positional abbreviations.

1. Testing expansion of ordinary abbreviation:
    a. Start jEdit
    b. Open an empty buffer by pressing Ctrl + n
    c. Type “gm”
    d. Press Ctrl + ;
    e. A dialog box will appear
    f. Enter the expanded string “good morning”
    g. Click on add global
    h. The string “gm” should be replaced by the string “good morning”
    i. Again, type gm in an new line
    j. Press Ctrl + ;
    k. The abbreviation should be expanded to “good morning” without the dialog box
2. Testing expansion of positional abbreviation:
    a. Start jEdit
    b. Open empty buffer in java mode by pressing Ctrl + Shift + n
    c. Type “F#i#10#”
    d. Press Ctrl + ;
    e. The string should be expanded to the following code block:

```
for(int i = 0; i < 10; i++)
{

}
```

Coupler (Inappropriate intimacy)

This code smell occurs when a class uses internal fields and methods of other class. This causes unnecessary coupling between those classes. This can be detected by observing the commits and noting the group of classes which change together.

To remove this code smell we can try to re-evalute if a method or field belongs to where it is present, or it will be better if the method or field is moved to the class which uses it extensively. This can be done by keeping each classes’ responsibility in mind.

In sonarqube we can observe some instances of inappropriate intimacy by looking at the code smell- "private" methods called only by inner classes should be moved to those classes under the tag confusing. This

smell is detected if a private method is accessed only by its inner class. It can be resolved by moving the method to the inner class instead.

We found an instance of this smell in class: org.gjt.sp.jedit.BufferHistory and its inner class Entry. The private method stringToSelection() of BufferHistory was only called inside the getSelection() method of the inner class Entry.

To remove this code smell we moved the method inside the Entry inner class but then we observed that the getSelection method contained just a call to the outer classes' stringToSelection() method. This prompted us to move all the method contents from stringToSelection() to the getSelection() method.

Test:

Following are the steps to test the changes done to the BufferHistory Class:

1. Open jEdit
2. Open a new buffer using Ctrl + n
3. Input some random text in the buffer.
4. Note the cursor position in the buffer
5. Open another buffer using Ctrl + n
6. Input some random text in this buffer
7. Note the cursor position
8. From the Buffer dropdown list switch the buffer to the previous buffer
9. Check if the cursor is at the same position where it was left last time
10. Close jEdit
11. Open jEdit
12. Select the previously opened buffers
13. Check if the cursor position is at the same position where it was left

OO Abuser (Static fields should not be updated in constructors)

The constructor should not update the static fields of the class. This is a problem because static fields have a common value for the class. If the value of a static field is updated when instantiating an object of the class. The static variable will change for all the instances. This is a clear violation of object-oriented practices. But we could not pinpoint on the type of Object-Oriented Abuse this smell belongs to.

This code smell can be found using sonarqube under the rule: "Static fields should not be updated in constructors"

In jEdit the class org.gjt.sp.jedit.BeanShellFacade sets the static variable iterpForMethods inside its constructor. To rectify this code smell we can make this field as non-static or initialize it statically.

We consider this smell detected by sonarqube as an actual code smell as it is a bad practice to set static variables inside a constructor.

Reference:

[1] https://www.sonarsource.com/docs/CognitiveComplexity.pdf

[2] https://www.sonarqube.org/

[3] https://docs.sonarqube.org/latest/setup/get-started-2-minutes/