Assignment 6: Refactoring Report

Team 18: Teena Thomas, Nazeera Siddiqui, and John Scalley

Due date: 11/10/2019

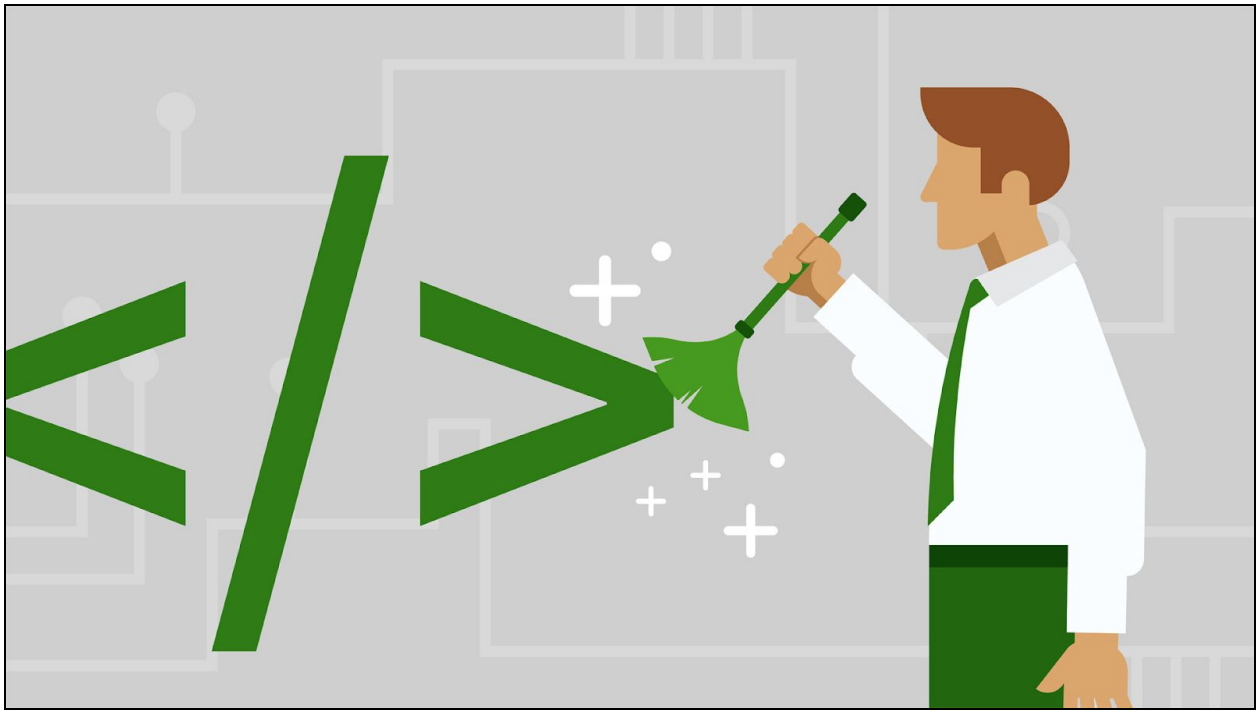

Table of Contents

Task Delegation

PDFsam	
Name	**Assigned Number**
Teena Thomas	#1
	#2
	References
Nazeera Siddiqui	#3

jEdit	
Name	**Assigned Number**
John Scalley	#1
	#2
Nazeera Siddiqui	#3

*** Each member is to complete all rows associated with their designated number(s) ***

PDFsam

Smelly Classes/Methods

#	Classes/Methods	Explanation
1	**Class:** MergeSelectionPaneTest	**PATH**:MergeSelectionPaneTest.java/MergeSelectionPaneTest **Bad Smell:** God Class **Refactoring Type:** Extract Method **Description:** The God Class, also known as the Large Class bad smell violates the single responsibility principle as it is responsible for controlling a large number of objects implementing different functionalities. The MergeSelectionPaneTest class being flagged by JDerodorent as "smelly" God class because there are a lot of tasks/jobs which are being done by the MergeSelectionPaneTest class. The vast responsibilities assigned to MergeSelectionPaneTest class is indicated by the different methods present (setup(), empty(),emptyByzeroPagesSelected(), emptyPagesSelection(), notEmptyPageSelection(), conersationException(), and Populate()). Although there are a lot of different responsibilities being assigned to the MergeSelectionPaneTest class, I do not think the detected smell is an actual bad smell. This is because by convention the pane class "acts as a base class of all layout panes. Basically, it fulfills the need to expose the children list as public so that users of the subclass can freely add/remove children "(GeeksforGeeks). The ideal solution to this particular God Class bad smell is to extract all the methods and fields which are related to specific functionality and separate into a class but because this was not actually a bad small, I would not make any changes to the current structure of the class.

#1 Refactoring Analysis: Automated

1. List and describe in detail the refactorings (i.e., the code changes) used to remove the smell.

- The MergeSelectionPaneTest class being flagged by JDerodorent as “smelly” God class because there are a lot of tasks/jobs which are being done by the MergeSelectionPaneTest class. The vast responsibilities assigned to MergeSelectionPaneTest class is indicated by the different methods present (setup(), empty(),emptyByzeroPagesSelected(), emptyPagesSelection(), notEmptyPageSelection(), conersationException(), and Populate()). Extract Class was made use of for this refactoring. The “smelliest” component within this class is the MergeOptionsPane() method. Eclipse automated refactoring tool was used to perform this refactoring.
    - Firstly I highlight all portions of code relating to “blankIfOdd” (lines 65 to 69) and right-click. Hovering over on “Refactor”, I click on “Extract Method”.The Extract Method pop-up appeared and I named the method “blankIfOffLayout”. The only parameter present is of type “I18nContext”, named “i18n”. I then click ok, a new method named “blankIfOffLayout(i18n)” is created outside the MergeOptionsPane(i18n) method. The new method is called using “blankIfOffLayout(i18n)”.
    - Secondly, I highlight all portions of code relating to “footer” (lines 71 to 74) and right-click. Hovering over on “Refactor”, I click on “Extract Method”.The Extract Method pop-up appeared and I named the method “footerLayout”. The only parameter present is of type “I18nContext”, named “i18n”. I then click ok, a new method named “footerLayout(i18n)” is created outside the MergeOptionsPane(i18n) method. The new method is called using “footerLayout(i18n)”.
    - Thirdly, I highlight all portions of code relating to “normalize” (lines 76 to 79) and right-click. Hovering over on “Refactor”, I click on “Extract Method”.The Extract Method pop-up appeared and I named the method “normalizeLayout”. The only parameter present is of type “I18nContext”, named “i18n”. I then click ok, a new method named “normalizeLayout(i18n)” is created outside the MergeOptionsPane(i18n) method. The new method is called using “normalizeLayout(i18n)”.
    - Fourthly I highlight all portions of code relating to “normalize” (lines 83 to 92) and right-click. Hovering over on “Refactor”, I click on “Extract Method”.The Extract Method pop-up appeared and I named the method “acroFormsLayout”. The parameters present are of type “I18nContext”named “i18n” and type “GridePane” named “options”. I then click ok, a new method named “normalizeLayout(i18n, options)” is created outside the MergeOptionsPane() method. The new method is called using “acroFormsLayout(i18n, options)”.
    - Fifty, I highlight all portions of code relating to “normalize” (lines 94 to 105) and right-click. Hovering over on “Refactor”, I click on “Extract Method”.The Extract Method pop-up appeared and I named the method “optionsLayout”. The parameters present are of type “I18nContext”named “i18n” and type “GridePane” named “options”. I then click ok, a new method named “outlineLayout(i18n, options)” is created outside the MergeOptionsPane() method. The new method is called using “acroFormsLayout(i18n, options)”.

- Lastly, I highlight all portions of code relating to “normalize” (lines 107 to 117) and right-click. Hovering over on “Refactor”, I click on “Extract Method”.The Extract Method pop-up appeared and I named the method “optionsLayout”. The parameters present are of type “I18nContext”named “i18n” and type “GridePane” named “options”. I then click ok, a new method named “outlineLayout(i18n, options)” is created outside the MergeOptionsPane() method. The new method is called using “acroFormsLayout(i18n, options)”.

2. Give the rationale of the chosen refactoring operations.

- There were alot of components/ tasks being assigned to this method. Each of these tasks seemed to be as code fragment within the MergeOptionsPane() method which could be grouped together. This is because each fragment was responsible for accomplishing a particular task. Without refactoring the method the MergeOptionsPane() method is too long and a bit unorganized. After the extract method refactoring both readability and re-use increased. The new methods were all created as private as, in this particular scenario, they were not being used outside the class.

3. Explain what code changes you had to do manually (if any), in addition to the changes performed with the IDE’s support.

- I did not need to make any manual changes as the IDE did a pretty good job in refactoring the piece of code. All I had to do was remove the unnecessary white-spaces between methods.

4. Compare the manual and automated refactoring processes that you performed. Describe the difficulties, advantages, and disadvantages of using one or the other.

- The automated refactoring is a lot easier than manual refactoring, in my opinion. Regardless of manual and automated refactoring process, which refactoring method to make use of must be through of/decided; this, for me, is the hard part.
  - ***Manual Refactoring Advantage:***
    - Although not prevalent in refactoring I performed, the advantage of manual refactoring is that it allows for a more customized solutions. What I mean by this is that, there are situations where code may be flagged but in reality refactoring may not be needed; in other words there is a purpose to the madness. In such cases, human reasoning is needed.
  - ***Manual Refactoring Disadvantage:***
    - Aside from figuring out how to implement the refactoring method in the class, the second challenge was actually implementing the changes. I am doing the same thing as would an automated tool would have performed so I did not particularly enjoy the manual refactoring, at least not in this case. Manual refactoring is very tedious and time consuming. There are no resources you could make use of other than your brain. You must keep track of which lines of codes you changed, where to add new lines of code, and which lines of code you need to get rid of.
  - ***Automated Refactoring Advantage:***
    - The biggest advantage of using automated refactoring is its simplicity. It is way more convenient to highlight the piece of code you want to change and choose from the

drop-down menu which refactoring method you want rather than performing manual refactoring.The other advantage is that there is no need to keep track of which lines of codes you changed, where to add new lines of code, and which lines of code you need to get rid of because all of this will be taken care of by the tool.

- ***Automated Refactoring Disadvantage:***
  - There are situations where code may be flagged but in reality refactoring may not be needed; in other words there is a purpose to the madness. In such cases, human reasoning is needed.

2	**Class:** RotateParametersBuilderTest	**PATH**:RotateParametersBuilderTest.java/RotateParametersBuilderTest **Bad Smell:** Long Method **Refactoring Type:** Extract Method **Description:** A Long Method bad small is exactly what it sounds line, a method which is too long. Any method which is more than ten lines of code has a risk of being prone to the long method bad smell.The RotateParametersBuilderTest class is being flagged by JDerodorent as "smelly" Long Methods because every method within this class (except validSteps method) is too long. Although this is another bad smell on its own, this class also contains code duplication.I agree with JDerodorent, in that the detected small is actually a bad smell. Thi is because typically builder classes makes use of method chaining."In java, Method Chaining is used to invoke multiple methods on the same object which occurs as a single statement. Method-chaining is implemented by a series of methods that return the reference for a class instance"(GeeksforGeeks). As a result of this method chaining the methods are typically short. In the case of the RotateParametersBuilderTest class the methods: vaildSteps(), onSaveWorkspace(), restoreStateFrom(), and reset() are very long. The ideal solution to this Long Method bad smell is to extract all the methods and fields which are related to specific functionality and separate into classes.

#2 Refactoring Analysis: Manual

1. List and describe in detail the refactorings (i.e., the code changes) used to remove the smell.

1. I created a new private method named "victimLayout". To this private method I copy-pasted lines 55 to 60 (from the setUp method)into, as these lines pertained to "victim". There are no parameters being passed to the method. Now, in place of lines 55 to 60, I call the new private method "victimLayout()".
2. I created a new private method named "assertEqualLayout". To this private method I copy-pasted lines 73 to 82 (from the buildFaultSelection method) into, as these lines pertained to "assertEquals". The parameters for the assertEqualLayout method are "output" (of type FileOrDirectoryTaskOutput), "source" (of typePDFFileSource), and "params"(of type BulkRotateParameters). Now, in place of lines 73 to 82, I call the new private method "assertEqualLayout(output,source,params)".

2. Give the rationale of the chosen refactoring operations.

- The more lines found in a method, the harder it is to figure out what the method does.There were alot of components/ tasks being assigned to both the "setUp" method and "buildFaultSelection" method. As a result, I decided to make use of the Extract Method refactoring. Within each method some lines of code seemed to be related thus I grouped these lines of code together into separate private methods. Now the "setUp" method and "buildFaultSelection" method are shorter, less redundant, more organized, and could be reused, if necessary.

3. Explain what code changes you had to do manually (if any), in addition to the changes performed with the IDE's support.

- There was no IDE support used at all for this refactoring thus all changes were made manually.

4. Compare the manual and automated refactoring processes that you performed. Describe the difficulties, advantages, and disadvantages of using one or the other.

- The automated refactoring is a lot easier than manual refactoring, in my opinion. Regardless of manual and automated refactoring process, which refactoring method to make use of must be through of/decided; this, for me, is the hard part.
    - ***Manual Refactoring Advantage:***
        - Although not prevalent in refactoring I performed, the advantage of manual refactoring is that it allows for a more customized solutions. What I mean by this is that, there are situations where code may be flagged but in reality refactoring may not be needed; in other words there is a purpose to the madness. In such cases, human reasoning is needed.
    - ***Manual Refactoring Disadvantage:***
        - Aside from figuring out how to implement the refactoring method in the class, the second challenge was actually implementing the changes. I am doing the same thing as would an automated tool would have performed so I did not particularly enjoy the manual refactoring, at least not in this case. Manual refactoring is very tedious and time consuming. There are no resources you could make use of other than your brain. You must keep track of which

lines of codes you changed, where to add new lines of code, and which lines of code you need to get rid of.

- ***Automated Refactoring Advantage:***
  - The biggest advantage of using automated refactoring is its simplicity. It is way more convenient to highlight the piece of code you want to change and choose from the drop-down menu which refactoring method you want rather than performing manual refactoring.The other advantage is that there is no need to keep track of which lines of codes you changed, where to add new lines of code, and which lines of code you need to get rid of because all of this will be taken care of by the tool.
- ***Automated Refactoring Disadvantage:***
  - There are situations where code may be flagged but in reality refactoring may not be needed; in other words there is a purpose to the madness. In such cases, human reasoning is needed.

3	**Class:**SplitOptionsPane	**PATH**:SplitOptionsPane.java/SplitOptionsPane **Bad Smell:** God Class **Refactoring Type:** Extract Method **Description:** The God Class, also known as the Large Class bad smell violates the single responsibility principle as it is responsible for controlling a large number of objects implementing different functionalities. The SplitOptionsPane class being flagged by JDerodorent as "smelly" God class because there are a lot of tasks/jobs which are being done by the SplitOptionsPane class. The vast responsibilities assigned to SplitOptionsPane class is indicated by the different methods present (setup(), start(Stage state),apply(), applyError(), saveState(), and restoreState()). Although there are a lot of different responsibilities being assigned to the SplitOptionsPane class, I do not think the detected smell is an actual bad smell. This is because by convention the pane class "acts as a base class of all layout panes. Basically, it fulfills the need to expose the children list as public so that users of the subclass can freely add/remove children "(GeeksforGeeks). The ideal solution to this particular God Class bad smell is to extract all the methods and fields which are related to specific functionality and separate into a class but because this was not actually a bad small, I would not make any changes to the current structure of the class.

#3 Refactoring Analysis: Nazeera Automated

1. List and describe in detail the refactorings (i.e., the code changes) used to remove the smell.

-

2. Give the rationale of the chosen refactoring operations.

-

3. Explain what code changes you had to do manually (if any), in addition to the changes performed with the IDE's support.

-

4. Compare the manual and automated refactoring processes that you performed. Describe the difficulties, advantages, and disadvantages of using one or the other.

-

jEdit

Smelly Classes/Methods

#	Classes	Explanation
1	**Class:** TextAreaPainter	**PATH**:jEdit/org/gjt/sp/jedit/textarea/TextAreaPainter.java **Bad Smell:** God Class **Refactoring Type:** Extract Class **Description:** JDeodorant detected that the TextAreaPainter class had several possible God Class code smells. One was regarding the fields selectionColor, multipleSelectionColor, and lineHighlight and the methods setSelectionColor(), setMultipleSelectionColor(), and setLineHighlightEnabled(). JDeodorant suggests extracting a class containing these fields and methods. A God Class code smell is when a class is doing too much. Although JDeodorant detected this as a code smell, I do not agree with the suggestion it has to extract a class for just these fields and methods. Those functions were setter functions for three of TextAreaPainter's variables, and it would not make sense to only move those functions and variables to a new class.

#1 Refactoring Analysis: John Automated

1. List and describe in detail the refactorings (i.e., the code changes) used to remove the smell.

- The TextAreaPainter class was flagged by JDeodorant as a god class, methods setSelectionColor and setMultipleSelectionColor were flagged as methods that could be extracted to a new class to help fix the god class smell. To perform this refactoring, I selected the flagged code smell from JDeodorant and used JDeodorant's built-in "Apply Refactoring" button. When I clicked on this, a new popup appeared where I was prompted to name the new class. I named the new class TextAreaPainterColor to help anyone else looking at the code understand that this class performed actions closely associated with TextAreaPainter and the specific actions of the class related to color. Reran JDeodorant and the smell was still found because there are other refactorings required to completely fix the god class smell but this extract class was no longer recommended.

2. Give the rationale of the chosen refactoring operations.

- The TextAreaPainter class was performing a lot of different actions, by extracting actions pertaining to the color to their own class, it allows the TextAreaPainter class to focus on it's responsibility by delegating other tasks to classes more suited to said task.

3. Explain what code changes you had to do manually (if any), in addition to the changes performed with the IDE's support.

- N/A, IDE automation performed the refactoring and I did not need to do anything more.

4. Compare the manual and automated refactoring processes that you performed. Describe the difficulties,
advantages, and disadvantages of using one or the other.

- When manually performing refactoring, you have to be very thorough, looking through code, finding all references to what you are changing and being sure to change all of them to be consistent so you don't introduce bugs when attempting to refactor. However, with this, when looking through the code you get a better understanding of the code so you can make changes more easily in the future. The disadvantage of this however is the obvious extra time it will take to look through code as opposed to the automated way of doing it from within the IDE. When automatically performing refactoring, the IDE does all the looking for you and makes all easy updates without you having to scour through the code. You lose some understanding of the code but save hours looking for a few references. If making changes to a project still in development that you could benefit from understanding better manual refactoring could benefit, but if you're doing changes to a project sparsely, automated refactoring is better.

2	**Class:**TextArea	**PATH**: jEdit/org/gjt/sp/jedit/textarea/TextArea.java **Bad Smell:** Feature Envy **Refactoring Type:** Move Method **Description:** Feature Envy is when a class uses more data of another object than its own. According to JDeodorant, the method getSelectedText() in the TextArea class should be moved to the Selection class. I would agree that this is actually a code smell. The Selection class has two int fields for the start and end. The getSelectedText method also calls getText in the Selection class. Instead of having getSelectedText in the TextArea class, we can have it in the Selection class since all of the fields needed are already there.

#2 Refactoring Analysis: John Manual

1. List and describe in detail the refactorings (i.e., the code changes) used to remove the smell.

- The getSelectedText(Selection s) method was flagged by JDeodorant for feature envy. To start doing this change, I first copied and pasted the method and its relevant comments into the destination class, Selection.java. From there, I noticed the method used a variable from the TextArea class that it no longer has access to, buffer. Since there were no changes being made to buffer, I passed it as a parameter to the moved method. I also noticed that there was a selection object being passed as a parameter, I instead changed each Selection parameter used to "this" so when the method is called the Selection object calling will do the work on its own data. From there I looked through TextArea.java to all the locations the previous method was called and updated the method calls to be of the Selection object that was a parameter, removed it as a parameter and added buffer as a parameter. Reran JDeodorant and the smell was no longer found.

2. Give the rationale of the chosen refactoring operations.

- The TextArea class was doing functions that fall under the realm of actions performed by Selections, thus the TextArea class was envious of the Selections class. To fix this, the method, getSelectedText(Selection s) was moved to the Selections class so the TextArea class would no longer be performing actions better done by Selections and could ask it's local Selection objects to do the work instead.

3. Explain what code changes you had to do manually (if any), in addition to the changes performed with the IDE's support.

- I did all changes manually.

4. Compare the manual and automated refactoring processes that you performed. Describe the difficulties,
advantages, and disadvantages of using one or the other.

- When manually performing refactoring, you have to be very thorough, looking through code, finding all references to what you are changing and being sure to change all of them to be consistent so you don't introduce bugs when attempting to refactor. However, with this, when looking through the code you get a better understanding of the code so you can make changes more easily in the future. The disadvantage of this however is the obvious extra time it will take to look through code as opposed to the automated way of doing it from within the IDE. When automatically performing refactoring, the IDE does all the looking for you and makes all easy updates without you having to scour through the code. You lose some understanding of the code but save hours looking for a few references. If making changes to a project still in development that you could benefit from understanding better manual refactoring could benefit, but if you're doing changes to a project sparsely, automated refactoring is better.

3	**Class:** TextArea	**PATH**:jEdit/org/gjt/sp/jedit/textarea/TextArea.java Method int addExplicitFold(int,int,int,int) **Bad Smell:** Feature Envy **Refactoring Type:** Move Method **Description:** This bad smell affects the addExplicitFold method, that is a member of the TextArea class, in this specific flag. This method was flagged for having the feature envy bad smell because it uses a lot of data from the buffer object. I agree that this method contains the bad smell and should be moved, a majority of the data used in it is directly or indirectly from the buffer object. Moving the method to the buffer class would allow the references within the method to be removed in exchange for a buffer object reference to call the method.

#3 Refactoring Analysis: Nazeera Manual

1. List and describe in detail the refactorings (i.e., the code changes) used to remove the smell.

-

2. Give the rationale of the chosen refactoring operations.

-

3. Explain what code changes you had to do manually (if any), in addition to the changes performed with the IDE's support.

-

4. Compare the manual and automated refactoring processes that you performed. Describe the difficulties,
advantages, and disadvantages of using one or the other.

-

Reference

Author: coderhs, and Author: “God Class: Breaking Single Responsibility Principle.” *GOD CLASS: BREAKING SINGLE RESPONSIBILITY PRINCIPLE*, Red Panthers, 26 July 2016, redpanthers.co/god-class-breaking-single-responsibility-principle/.

gaurav miglaniCheck out this Author's contributed articles., et al. “Builder Pattern in Java.” *Builder Pattern in Java*, GeeksforGeeks, 8 Aug. 2018, www.geeksforgeeks.org/builder-pattern-in-java/.

“Long Method.” *Long Method*, Refactoring Guru, refactoring.guru/smells/long-method.