jEdit Milestone #3

Name	Student ID
Hamed Kalantari	9411569
Kelvin Iu	4538501
Oualid El Halimi	9385886
Muhammad Farhan Malik	6577792
Youssef Bennis	5570913

Due: March 3rd, 11 marks

Table of Contents

Contents

Table of Contents 2
Summary of Project 3
1-Class Diagram of Actual System 4
1.1 Package “org.gjt.sp.jedit.search" 6
2-Code Smells and System Level Refactorings (SRL) 9
2.1. SearchAndReplace class code smells and SLR 9
2.1.1 God Class 9
2.1.2 Type Checking (InstanceOf) 13
2.1.3 Long Method 15
2.1.4 Implicit dependency of Static methods 18
3-Specific Refactorings that you will implement in Milestone 4 21
4- References 22

Summary of Project

jEdit [1] is a text editor for programmers. It is an open source java project used to handle text and save them into multiple text formats.

The project is well documented and has an active and vibrant open source development team. It has more than ninety (90) registered contributors, most of them are still active within the project. Last stable release version is 5.1 and the last modification of the current build was on 2014-01-02. Development on jEdit started in 1988 by Salava Pestov. This latter left the project after 8 years to the free software community, which is actively preserving and maintaining jEdit up-to-date [2].

jEdit domain is a graphical user interface based on the programmer's text editor which is highly customizable and extensible via plugins. jEdit has many features like multiple selecting, unlimited redo/undo and "Markers" used to provide an enjoyable user experience. Along with these functionalities, a number of important features are coded in jEdit. Having a quick glance at the source code of jEdit, it was visible that the naming of packages are based on the nature of the features. For instance, the search functionality implementation is covered in the package org.gjt.sp.jedit.search.

The jEdit source code includes a unit test module to provide Test-Driven development feature. This allows regular testing of any changes committed by team members to ensure project validation and correctness and to limit the need to invoke a debugger. Ability to test any changes would be a great help in maintaining the code. Their website claim that there are "hundreds (counting the time developing plugins) of person-years of development behind this project".

1-Class Diagram of Actual System

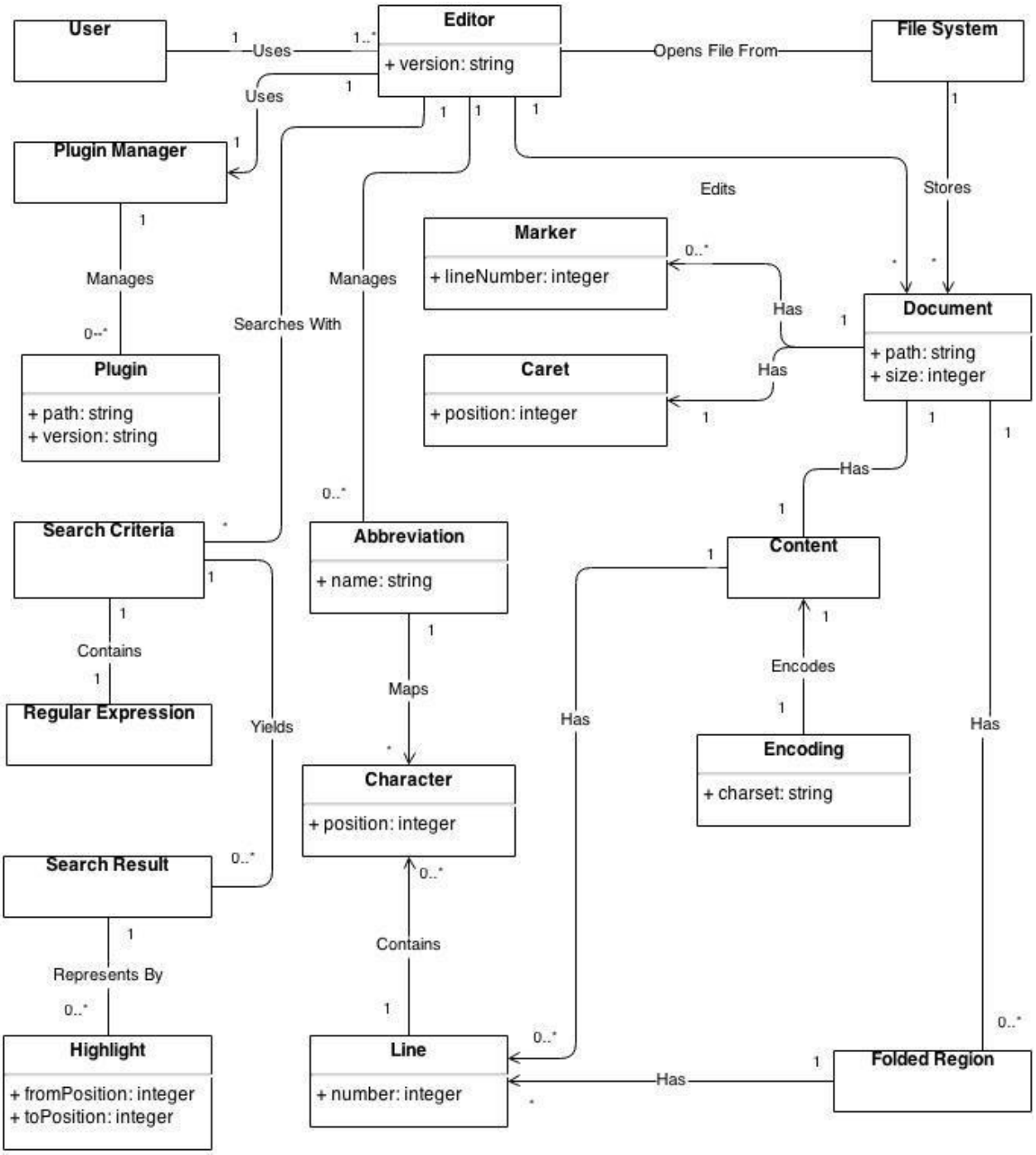

Figure 1: Conceptual Architecture (Domain Model) from jEdit Milestone #2

Relationship between the two (2) classes:

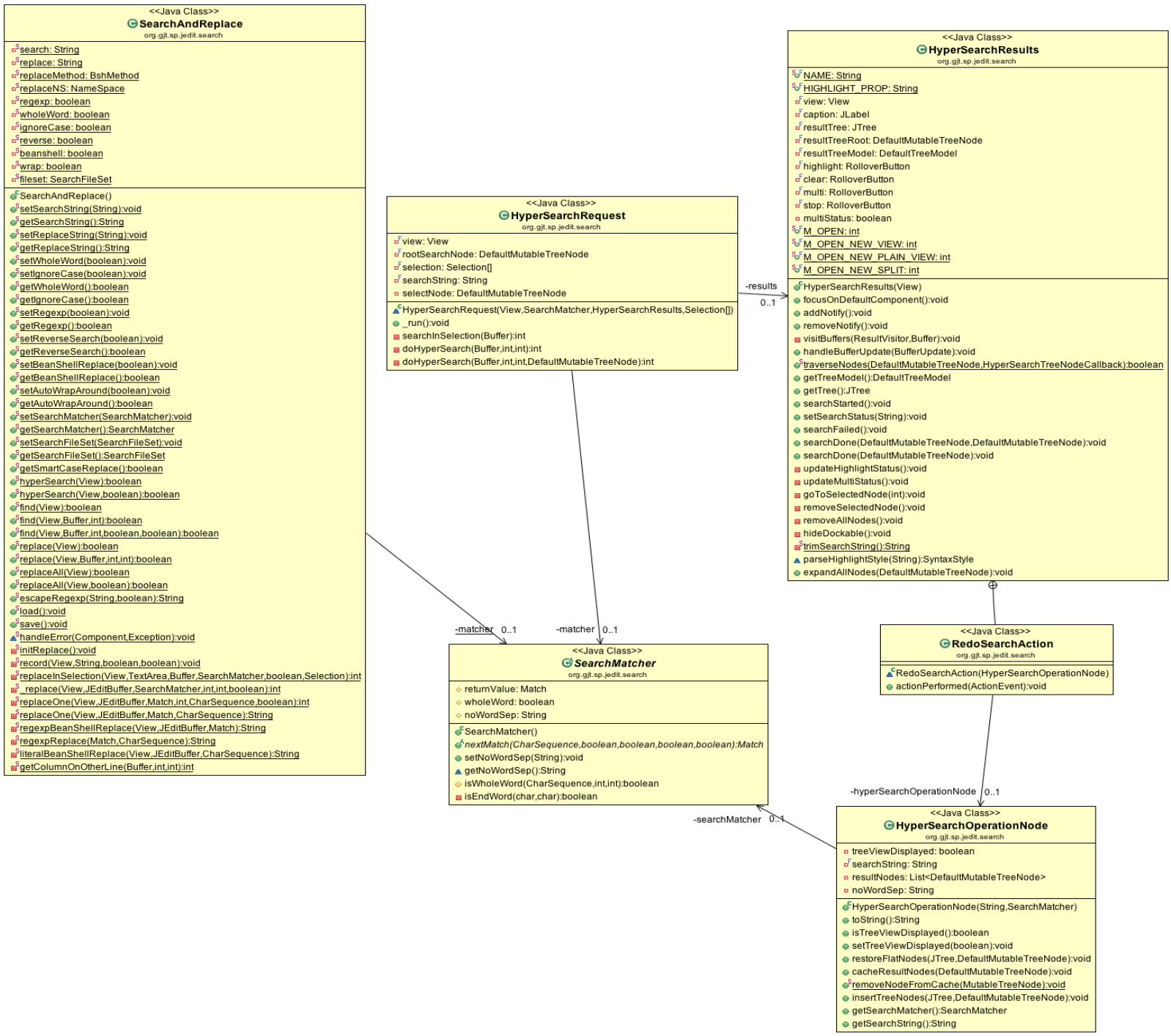

Figure 2: Search and Replace unit's Related Class Diagram

To reverse engineer Jedit source code, out of the box case tools were used to extract the actual class diagram related to our classes of interest (SeachAndReplace and HyperSearchResults). SeachAndReplace main concern is to encapsulate all methods and properties to perform search and replace operations within jEdit Search Pane. This class aggregates SearchMatcher class that is used to return the first match of the specified text within the matcher. The HyperSearchRequest class contains the main functionalities to conduct hyper search within the Hyper Search Results Window. It also aggregates the SearchMatcher when performing a hyper search. HyperSearchRequest is dependant on HyperSerachResults in order to run the hyper search operation via the method: HyperSearchRequest(). The HyperSearchResult has a nested class: RedoSearchAction which is dependant on HyperSearchOperationNode and that is used to set both the search and the matcher strings to perform a search.

1.1 Package "org.gjt.sp.jedit.search"

We used ObjectAid UML plugin for Eclipse to reverse engineer JEdit source code and therefore generate UML Class diagram of the classes of interest before refactoring. After necessary refactorings have been applied, ArgoUML tool is used to create system-under-study UML Diagram. One of the representational limitations of the ArgoUML is not distinguishing method's access modifiers (i.e. private, public in the diagram).

By focusing on the search package, which maps to the search functionality, we can remark that both the conceptual diagram and the class diagram provide a searchResult concept. The searchResult concept links to HyperSearchResult class in the JEdit Class Diagram.

Also, in the concept diagram, the "searchResult" concept has a relationship with the search criteria which is represented by the SearchMatcher class in the class diagram. The class diagram shows clearly the usage of a strategy pattern on the SearchMatcher object that represents the search algorithm, where BoyerMooreSearchMatcher and PatternSearchMatcher objects are the two strategies extending the abstract object SearchMatcher. However SearchMatcher is linked with both classes HyperSearchResults (The SearchResults'view module), and HyperSearchRequest in the class diagram. HyperSearchRequest has no equivalent concept in the conceptual diagram since it's more related to the solution domain. HyperSearchResults (with (s): different from HyperSearchResult) represents the view part related to the specific functionality of the search and has no match in the concept diagram, which represents a gap between the two diagrams. Actually, the "editor" concept is quite general and seems to encompass other related concepts such as Search UI. It's relevant to mention that the class diagram makes the distinction between the search dialog , where the user performs his search request and the search results dialog that displays the results. According to the concept diagram, the "Editor" has the responsibility of performing a search and seems to envelop concepts of search and replace , which are represented by the

searchAndReplace class. The “highlight” concept represents how the “searchResult” is shown to the user, and the possible equivalence in the class diagram could be the combination of both types “Occur ” (start/end , position of line) and “hyperSearchResult” (occurrences on a line) objects.

Classes of interest in this milestone:

Package	Class
org.gjt.sp.jedit.search	SearchAndReplace.java
org.gjt.sp.jedit.textarea	Selection.java Range.java (Inner Class of Selection class) Rect (Inner Class of Selection class)

SearchAndReplace.java

```
    /**
     * Replaces the current selection with the replacement string.
     * @param view The view
     * @return True if the operation was successful, false otherwise
     */
    public static boolean replace(View view)
    {...
    }

    /**
     * Replaces text in the specified range with the replacement string.
     * @param view The view
     * @param buffer The buffer
     * @param start The start offset
     * @param end The end offset
     * @return True if the operation was successful, false otherwise
     */
    public static boolean replace(View view, Buffer buffer, int start, int end)
    {...
    }
    /**
     * Replaces all occurrences of the search string with the replacement
     * string.
     * @param view The view
     * @param dontOpenChangedFiles Whether to open changed files or to autosave
       them quietly
     * @return the number of modified files
     */
    public static boolean replaceAll(View view, boolean dontOpenChangedFiles)
    {...
    }

  private static BshMethod replaceMethod;
  private static NameSpace replaceNS = new NameSpace(BeanShell.getNameSpace(),
```

```
                                                BeanShell.getNameSpace().getClassManager(),
                                                "search and replace");
  private static int replaceInSelection(View view, TextArea textArea,
                                Buffer buffer, SearchMatcher matcher, boolean smartCaseReplace
                                Selection s) throws Exception
{...
 }

 /**
  * Set up BeanShell replace if necessary
 */
 private static void initReplace() throws Exception
 {...
 }

 private static int replaceInSelection(View view, TextArea textArea,
                                Buffer buffer, SearchMatcher matcher, boolean smartCaseReplac
                                Selection s) throws Exception
 {...
 }
 /**
 * Replaces all occurrences of the search string with the replacement
 * string.
 * @param view The view
 * @param buffer The buffer
 * @param start The start offset
 * @param end The end offset
 * @param matcher The search matcher to use
 * @param smartCaseReplace See user's guide
 * @return The number of occurrences replaced
 */
  private static int _replace(View view, JEditBuffer buffer,
                                SearchMatcher matcher, int start, int end,
                                boolean smartCaseReplace) throws Exceptio
 {...
 }

 /**
 * Replace one occurrence of the search string with the
 * replacement string.
 */
 private static int replaceOne(View view, JEditBuffer buffer,
                                SearchMatcher.Match occur,
                                int offset, CharSequence found,
                                boolean smartCaseReplace) throws Exception
 {...
 }

  private static String replaceOne(View view, JEditBuffer buffer,
                                    SearchMatcher.Match occur, CharSequence found)
  throws Exception
  { …
  }

  private static String regexpBeanShellReplace(View view,
                                                JEditBuffer buffer, SearchMatcher.Match occur)
  throws Exception
  { …
  }
```

```
private static String regexpReplace(SearchMatcher.Match occur,CharSequence found)
throws Exception
{ …
}
```

SearchAndReplace.java

```
private static int replaceInSelection(View view, TextArea textArea,
                        Buffer buffer, SearchMatcher matcher, boolean smartCaseReplace,
                        Selection s) throws Exception
{...
}
```

Selection.java

```
public abstract class Selection implements Cloneable
{...
}
```

Range.java

```
public static class Range extends Selection
{...
}
```

Rect.java

```
public static class Rect extends Selection
{...
}
```

2-Code Smells and System Level Refactorings (SRL)

2.1. SearchAndReplace class code smells and SLR

2.1.1 God Class

There is an instance of God Class (Large Class) in “SearchAndReplace” class. The class is large and has several components (methods). This class has two responsibilities: Searching and Replacing content. Hence, there is low cohesion between those distinct functionalities. Therefore, there is no abstraction for the role of the class since it contains different functionalities. There are methods in the “SearchAndReplace” class, that can be extracted to make the class smaller, manageable and also to

increase cohesion .

In order to improve the cohesion of the methods in “SearchAndReplace” class and to narrow the class objectives to perform a particular task (i.e. Search or Replace), we will extract all related replace functionalities from ‘SearchAndReplace” class to a separate class “Replace”, which is specialized for replacing content using extract class refactoring. We will keep the original extracted method interfaces in the refactored “SearchAndReplace” class in order to not to break existing call hierarchies. Then, delegation is used to call the actual implementation of the methods in the extracted “Replace” class. The detailed steps of the refactorings are described sequentially in the table below:

ID	Software Artifact	Type	Refactoring\Action Taken	Reason
1	*Replace.java*	Class	*Create class*	To extract all replace functionality from “SearchAndReplace” class to this class
2	*public static boolean replace(View view)*	Method	1-Extract class refactoring to “Replace” class 2- Add delegation design pattern “SearchAndReplace” class	Increase cohesion
3	*public static boolean replace(View view, Buffer buffer, int start, int end)*	Method	1-Extract class refactoring “Replace” class	Increase cohesion
4	*public static boolean replaceAll(View view, boolean dontOpenChangedFiles)*	Method	1-Extract class refactoring “Replace” class	Increase cohesion
5	*private static int replaceInSelection(View view, TextArea textArea,Buffer buffer, SearchMatcher matcher, boolean smartCaseReplace, Selection s)*	Method	1-Extract class refactoring “Replace” class	Increase cohesion,
6	*private static int _replace(View view, JEditBuffer buffer,SearchMatcher matcher, int start, int end, boolean smartCaseReplace)*	Method	1- Extract class refactoring	Increase cohesion,
7	*private static int replaceOne(View view, JEditBuffer buffer,SearchMatcher.Match occur, int offset, CharSequence found, boolean smartCaseReplace)*	Method	1-Extract class refactoring	Increase cohesion,

8	*private static BshMethod replaceMethod*	Attribute	1- Move Method refactoring to "Replace" Class	Increase cohesion,
9	*private static NameSpace replaceNS*	Attribute	1- Move Method refactoring to "Replace " Class	Increase cohesion,
10	*private static String replaceOne(View view, JEditBuffer buffer, SearchMatcher.Match occur, CharSequence found)*	Method	1-Extract class 2- Use Getters and Setters in refactored SearchAndReplace class instead of accessing instance variables directly	Increase cohesion,
11	*public static void initReplace()*	Method	1-Extract class 2- Use Getters and Setters in refactored SearchAndReplace class instead of accessing instance variables directly	Increase cohesion,
12	*private static String regexpBeanShellReplace(View view, JEditBuffer buffer, SearchMatcher.Match occur)*	Method	1-Extract class 2- Use getters and setters in refactored SearchAndReplace class instead of accessing instance variables directly	Increase cohesion,
13	*private static String regexpReplace(SearchMatcher.Match occur,CharSequence found)*	Method	1-Extract class 2- Use getters and setters in refactored SearchAndReplace class instead of accessing instance variables directly	Increase cohesion,
14	*private static String literalBeanShellReplace(View view, JEditBuffer buffer, CharSequence found)*	Method	1-Extract class 2- Use getters and setters in refactored SearchAndReplace class instead of accessing instance variables directly	Increase cohesion,
15	*SearchAndReplace.java*	Class	1-Rename the refactored SearchAndReplace class to Search, in order for the name of class to match its concrete functionality	Better understanding and to increase abstraction of the class

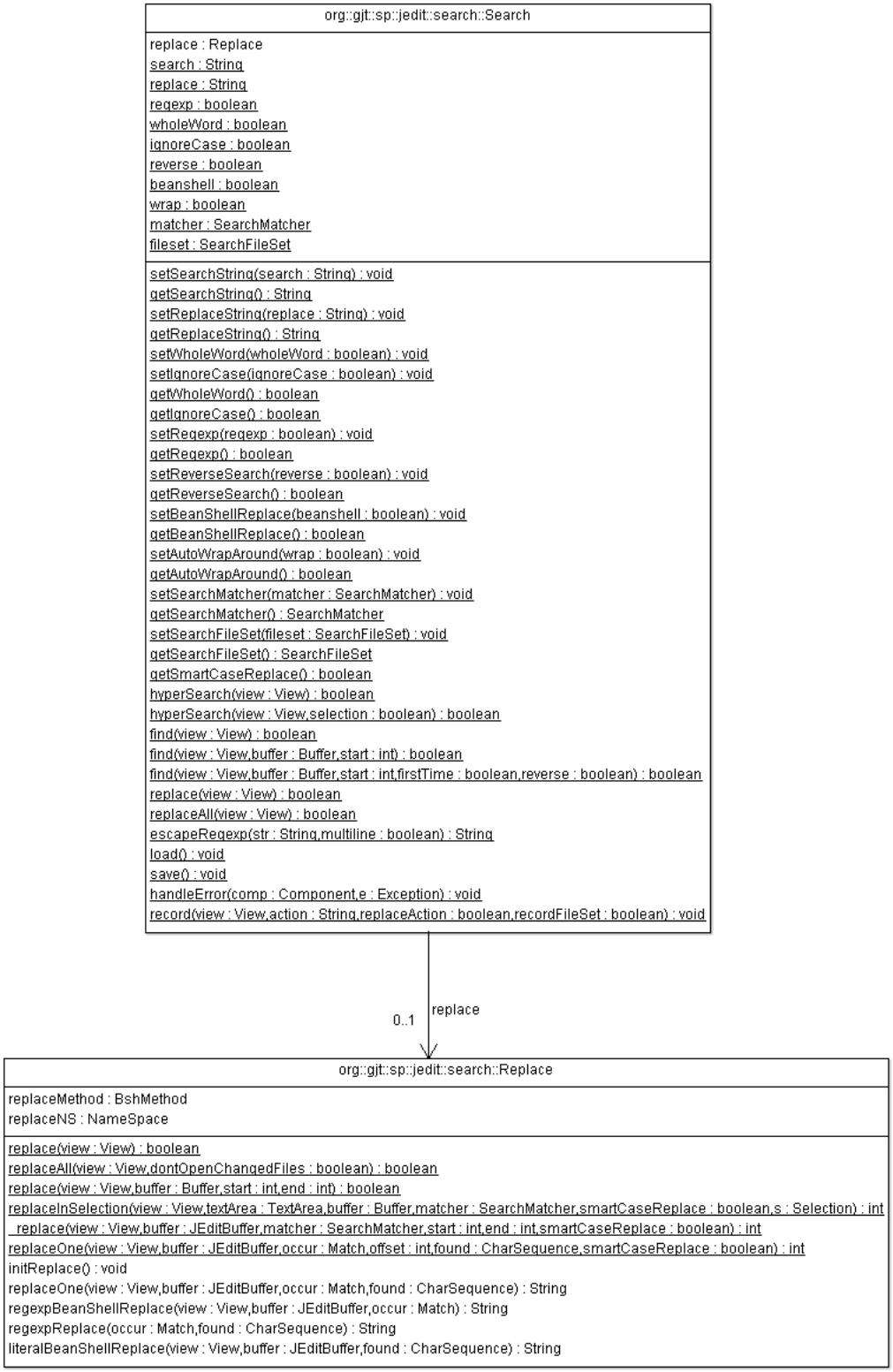
org::gjt::sp::jedit::search::Search
replace : Replace
search : String
replace : String
regexp : boolean
wholeWord : boolean
ignoreCase : boolean
reverse : boolean
beanshell : boolean
wrap : boolean
matcher : SearchMatcher
fileset : SearchFileSet
setSearchString(search : String) : void
getSearchString() : String
setReplaceString(replace : String) : void
getReplaceString() : String
setWholeWord(wholeWord : boolean) : void
setIgnoreCase(ignoreCase : boolean) : void
getWholeWord() : boolean
getIgnoreCase() : boolean
setRegexp(regexp : boolean) : void
getRegexp() : boolean
setReverseSearch(reverse : boolean) : void
getReverseSearch() : boolean
setBeanShellReplace(beanshell : boolean) : void
getBeanShellReplace() : boolean
setAutoWrapAround(wrap : boolean) : void
getAutoWrapAround() : boolean
setSearchMatcher(matcher : SearchMatcher) : void
getSearchMatcher() : SearchMatcher
setSearchFileSet(fileset : SearchFileSet) : void
getSearchFileSet() : SearchFileSet
getSmartCaseReplace() : boolean
hyperSearch(view : View) : boolean
hyperSearch(view : View,selection : boolean) : boolean
find(view : View) : boolean
find(view : View,buffer : Buffer,start : int) : boolean
find(view : View,buffer : Buffer,start : int,firstTime : boolean,reverse : boolean) : boolean
replace(view : View) : boolean
replaceAll(view : View) : boolean
escapeRegexp(str : String,multiline : boolean) : String
load() : void
save() : void
handleError(comp : Component,e : Exception) : void
record(view : View,action : String,replaceAction : boolean,recordFileSet : boolean) : void
0..1
replace
org::gjt::sp::jedit::search::Replace
replaceMethod : BshMethod
replaceNS : NameSpace
replace(view : View) : boolean
replaceAll(view : View,dontOpenChangedFiles : boolean) : boolean
replace(view : View,buffer : Buffer,start : int,end : int) : boolean
replaceInSelection(view : View,textArea : TextArea,buffer : Buffer,matcher : SearchMatcher,smartCaseReplace : boolean,s : Selection) : int
replace(view : View,buffer : JEditBuffer,matcher : SearchMatcher,start : int,end : int,smartCaseReplace : boolean) : int
replaceOne(view : View,buffer : JEditBuffer,occur : Match,offset : int,found : CharSequence,smartCaseReplace : boolean) : int
initReplace() : void
replaceOne(view : View,buffer : JEditBuffer,occur : Match,found : CharSequence) : String
regexpBeanShellReplace(view : View,buffer : JEditBuffer,occur : Match) : String
regexpReplace(occur : Match,found : CharSequence) : String
literalBeanShellReplace(view : View,buffer : JEditBuffer,found : CharSequence) : String

Figure 3: UML Class diagram, after refactoring “God Class”

2.1.2 Type Checking (InstanceOf)

While looking into the God Class code smell (section 2.1.1) in the SearchAndReplace class, another code smell was identified. This code smell fell under the criteria of “Instance Of ” code smell. The method in which “Instance Of” code smell was found is as follows:

```
private static int replaceInSelection(View view, TextArea textArea, Buffer buffer,
                                      SearchMatcher matcher, boolean smartCaseReplace,Selection s)
```

Part of the affected code which represents “Instance Of” code smell which is a specialized type of “Type Checking” code smell is presented:

```
if(s instanceof Selection.Range)
 {
   returnValue = _replace(view,buffer,matcher,
   s.getStart(),s.getEnd(),smartCaseReplace);  textArea.removeFromSelection(s);
   textArea.addToSelection(new Selection.Range(start,s.getEnd()));
 }
else if(s instanceof Selection.Rect)
{
  Selection.Rect rect = (Selection.Rect)s;
  int startCol = rect.getStartColumn(buffer);
  int endCol = rect.getEndColumn(buffer);
  returnValue = 0;
  for(int j = s.getStartLine(); j <= s.getEndLine(); j++)
      {
       returnValue += _replace(view,buffer,matcher,
                      getColumnOnOtherLine(buffer,j,startCol),
                      getColumnOnOtherLine(buffer,j,endCol),
                      smartCaseReplace);
      }
  textArea.addToSelection(new Selection.Rect(start,s.getEnd()));
}
```

This method provides the functionality of replacing the searched string which is selected and therefore the text of the string is highlighted. This method uses the “Selection s” object in an If/ Else condition and checks if the “Section s” object is either an instance of Range Class or Rect Class. Since the “Selection s” object is being placed in an If Else statement, it was clear that these lines of code were a strong case for labeling them as “Instance Of” code smell.

The classes involved in this code smell are as follows:

1. org.gjt.sp.jedit.search.Replace Class [Created after refactoring God Class (section 2.1.1)]

The Replace class is created after refactoring of God Class (section 2.1.1) it contains the Replace functionality of previously "SearchAndReplace" class

2. org.gjt.sp.jedit.textarea.Selection Class

This is an abstract class that holds data on a region of selected text. Since it is an abstract class, it cannot be used directly, but instead serves as a parent class for two specific types of selection structures.

3. org.gjt.sp.jedit.textarea.Selection.Range (Inner Class of Selection class)

An ordinary range selection is being performed by this Public Static class. It extends the Selection class.

4. org.gjt.sp.jedit.textarea.Selection.Rect (Inner Class of Selection class)

A rectangular selection is being performed by this Public Static Rect class. It extends the Selection class. In the Replace Once feature, the dependence of the feature was based on the search which uses the selection methods to select the required string and then highlight the strings.

So by manual analysis of the classes and its depend classes and their methods, it was evident that the "Instance Of" code smell is present which needs to be refactored, following steps were performed in order to refactor this code smell:

1. public int replaceInSelection(View v…..) method is added in the Selection, Rect and Range classes

2. The implementation of "replaceInSelection(…)" is done in three classes Selection (Base) , Rect(Derived) and Range (Derived) . Each implementation is different as per the class specifications. The implementation of this method is taken from the code block of if and Else If statements in replaceInSelection(…) from the *SearchAndReplace* class (After God class refactoring, in the *Replace* class).

3. In the *Replace* class the if else statements in the replaceInSelection(View v…..) method have been replaced with the following function calls. ReplaceInSelection(view, textArea, buffer, matcher, smartCaseReplace);

4. In the *Replace* class the visibility of _replace(View view, JEditBuffer buffer,SearchMatcher matcher, int start, int end, boolean smartCaseReplace) method Is changed to **public** as it was **private** before, hence it was not accessible in the Selection, Range and Rect Classes.

Benefits of Refactoring / Removing Code Smell (Instance Of) :

The major benefit for removing this code smell is that if, in the future, other forms of selections are to be added (i.e., adding more children to the parent Selection Class), the code in the Replace class would not need to be updated. Due to removal of this code smell by using techniques of polymorphism, the potential for the extension of Selection class is greatly complemented. Adding any new children for the Selection Class would not change the behaviour of the specific method. The other advantage is that by removing this code smell; the code base is now more reusable and the complexity of the code is removed after applying the polymorphic technique on this code smell.

Figure 4: UML Class diagram, after refactoring “Instance Of” code smell (unnecessary methods and attributes are not shown in Selection, Range and Rect classes)

2.1.3 Long Method

The method *regexpReplace* originally located in *SearchAndReplace* class is long (After applying God Class refactoring (section 2.1.1), the method *regexpReplace* is in the *Replace* class). It is also difficult to understand due to its complex flow of execution caused by the compounded for loop, switch, if and while

loop statements. The method can be made shorter by extracting the code within the outer for loop into a separate method with name indicates its purpose: processing each character in the string contains the regular expression used for the replace operation e.g. Add method named *processCharacterInReplaceString*. The logic inside the nested switch and if statements can be further extracted into other smaller methods that handle specific type of character such as capture group, escape characters, etc. within the string containing the regular expression used for the replace operation. e.g. Added methods named *processDollarCharacterInReplaceString* and *processBackSlashCharacterInReplaceString*

Three (3) new private static methods added to the *Replace* class:

```
private static int processCharacterInReplaceString (SearchMatcher.Match occur, CharSequence found, int
index, String stringReplaced);
```

```
private static int processDollarCharacterInReplaceString (SearchMatcher.Match occur, CharSequence found,
int index, String stringReplaced);
```

```
private static int processBackSlashCharacterInReplaceString (int index, String stringReplaced);
```

In order to demonstrate interrelation and sequence of the refactorings, long method refactoring is applied after Type Checking (Instance Of) refactoring (section 2.1.2). Therefore the UML Diagram (figure 4) shows refactoring of Long Method code smell on the changes made to the classes of interest after applying the "Instance Of" refactoring.

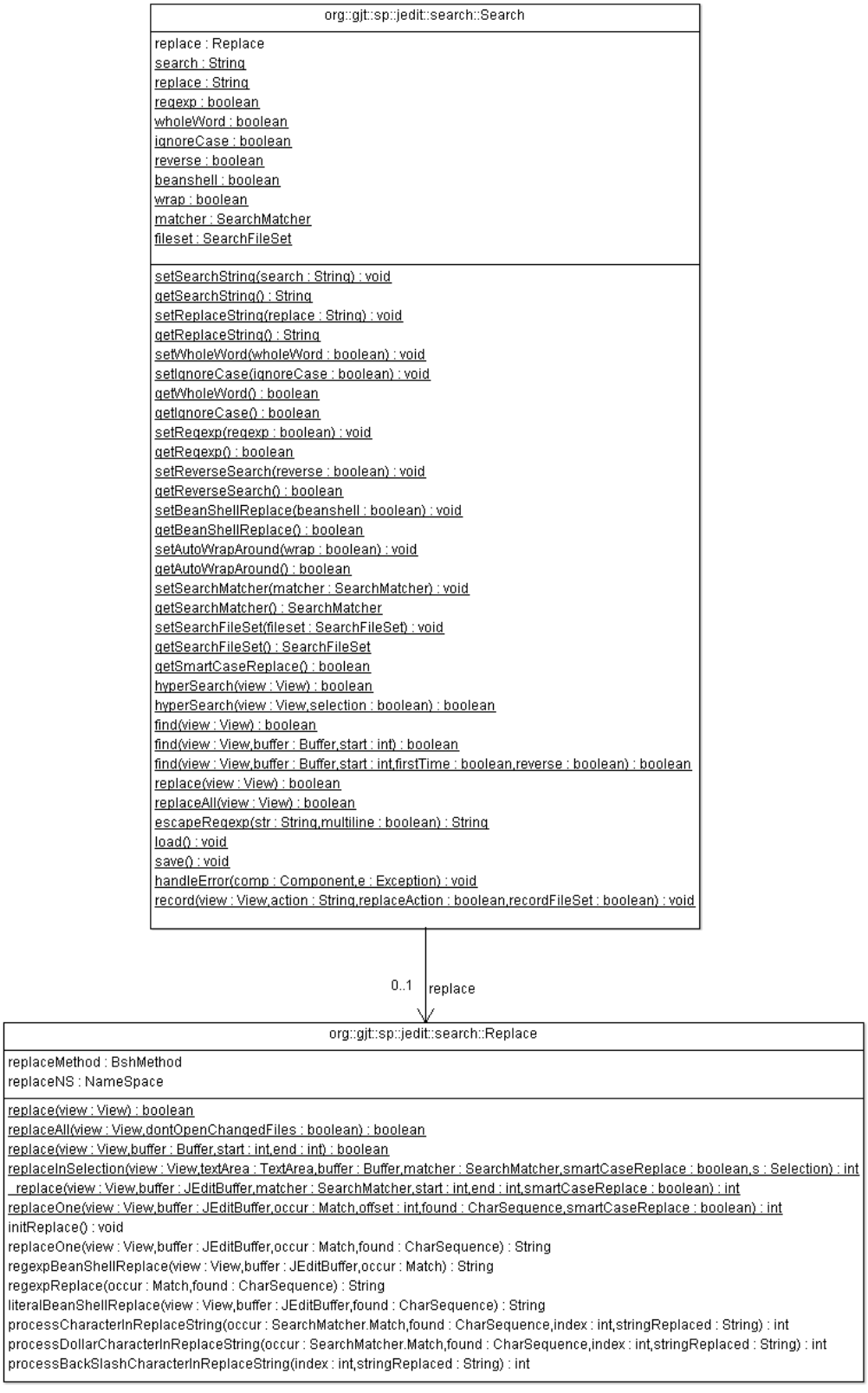

Figure 5: UML Class diagram, after refactoring “Large Method”

2.1.4 Implicit dependency of Static methods

The SearchAndReplace class has no constructor and provides access to a number of utility methods in a static way [3] (class with only static methods). Indeed, these methods should be stateless. In other words, calling a given method with the same attribute should always return the same result. However, these static methods have some internal and hidden dependencies [4]. Such methods are: ”setSearchString(String)”, “setSearchFileSet(SearchFileSet)” and “hyperSearch(View)”. If we perform an analysis on the calls made by the static method hyperSearch(View), we’ll get the conclusion that this later method is always called after the two static methods “setSearchString(String)” and ”setSearchFileSet(SearchFileSet)” are being called. This is due to the dependency to some variables (“Matcher” and “Search”) that the “hypersearch(View)” method relies on. If calling ”hypersearch(View)” without calling “setSearchString(String)” and “setSearchFileSet(SearchFileSet)”, abnormal behavior will occur. Moreover, with the actual code structure, nothing prevents from calling “hyperSearch(View)” without having set “Matcher” and “Search” variables. A quick glance at the “hyperSearch” method of SearchAndReplace class will confirm implicit dependencies to “Matcher” and “Search” variables. In fact, the current architecture of SearchAndReplace class presents a high risk of inconsistency since there is no guarantee all required information are present when search is performed.

To correct this vulnerable structure, two refactoring techniques can be adopted:

- First technique: Change the searchAndReplace class in order to have some constructor that sets the “Matcher” and “Search” variables , turning the hyperSearch method into a non-static method to force the call to this method only through an instance of searchAndReplace class. Then, any call to hypersearch method will be changed from :

```
SearchAndReplace.setSearchString(Search);
SearchAndReplace.setSearchFileSet(SearchFileSet)
SearchAndReplace.hyperSearch(View);
```

to

```
SearchAndReplace search = new SearchAndReplace(Search, SearchFileSet);
search.hyperSearch(View);
```

- Second technique: Turn the hypersearch method to private in order to prevent any call from outside the class. Then offer a new public static method (PerformHyperSearch(Search, SearchFileSet, View)) that takes the 3 parameters “Matcher” ,“Search” and view and change all previous calls from SearchAndReplace.hyperSearch(View) to this new method PerformHyperSearch(Search, SearchFileSet, View). This change will guarantee that all dependencies will be set prior of performing the main operation of hyperSearch.

This change would have this form :

```
SearchAndReplace.PerformHyperSearch(Search, searchFileSet, View);
```

This second method seems to be more secure than the first one and less impacting. Other refactorings proposed earlier on this searchAndReplace class would help to decide which refactoring to select.

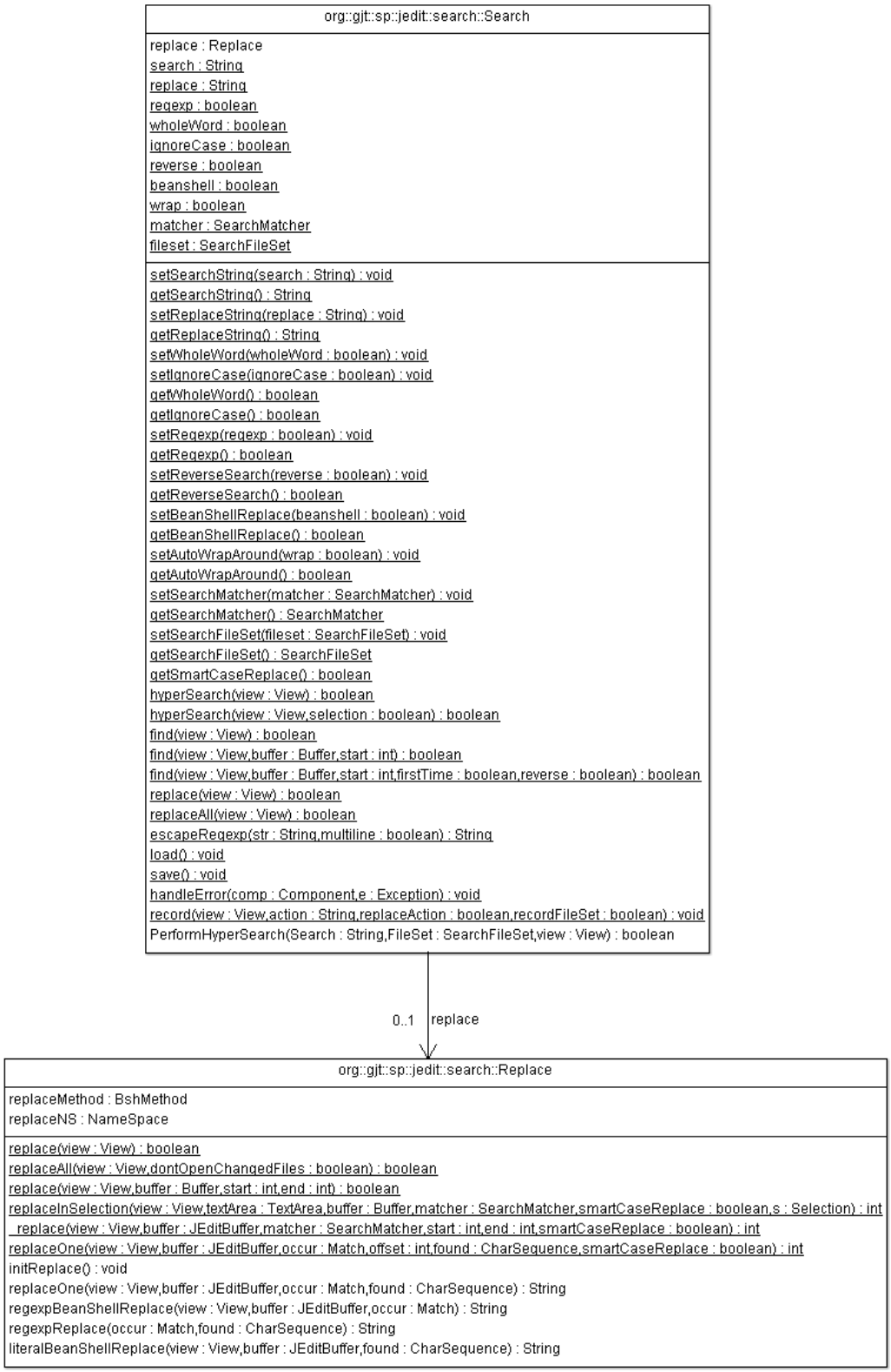

Figure 6: UML Class diagram, after refactoring “Implicit dependency of static methods”

3-Specific Refactorings that you will implement in Milestone 4

The refactorings that will be implemented in Milestone 4 and their source code:

- God class (see section 2.1.1)

- InstanceOf (see Section 2.1.2)

- Long Method (see Section 2.1.3)

In this milestone, we will expose the **instanceOf** refactoring source code, which will also be discussed in details in Milestone #4.

```
private static int replaceInSelection(View view, TextArea textArea,
        Buffer buffer, SearchMatcher matcher, boolean smartCaseReplace,
        Selection s) throws Exception
    {
        /* if an occurence occurs at the
        beginning of the selection, the
        selection start will get moved.
        this sucks, so we hack to avoid it. */
        int start = s.getStart();

        int returnValue;

        if(s instanceof Selection.Range)
        {
            returnValue = _replace(view,buffer,matcher,
                s.getStart(),s.getEnd(),
                smartCaseReplace);

            textArea.removeFromSelection(s);
            textArea.addToSelection(new Selection.Range(
                start,s.getEnd()));
        }
        else if(s instanceof Selection.Rect)
        {
            Selection.Rect rect = (Selection.Rect)s;
            int startCol = rect.getStartColumn(
                buffer);
            int endCol = rect.getEndColumn(
                buffer);

            returnValue = 0;
            for(int j = s.getStartLine(); j <= s.getEndLine(); j++)
            {
                returnValue += _replace(view,buffer,matcher,
                    getColumnOnOtherLine(buffer,j,startCol),
                    getColumnOnOtherLine(buffer,j,endCol),
                    smartCaseReplace);
            }
            textArea.addToSelection(new Selection.Rect(
                start,s.getEnd()));
        }
        else
            throw new RuntimeException("Unsupported: " + s);

        return returnValue;
    }
```

4- References

[1] - Source Forge, (2010). *JEdit*. Available: http://sourceforge.net/projects/jedit/

[2] - jEdit, (2010). Wikipedia, http://en.wikipedia.org/wiki/JEdit

[3] - Ben Day , (2010). *Static methods are a code smell.* Available: http://searchwindevelopment.techtarget.com/tip/Static-methods-are-a-code-smell

[4] - Christian Posta . (2010). *Java static methods can be a code smell.* Available: http://www.christianposta.com/blog/?p=56