Assignment 4: Refactoring for jEdit

Name: Martin Hu

1. Automated refactoring

1.1 Smell type: Feature Envy

Source place: org.gjt.sp.jedit.gui.MakerViewer.java
Target class: org.gjt.sp.jedit.view
Method: Move method
Moved method name: goToSelectedMarker

Before: org.gjt.sp.jedit.gui.MakerViewer.java

```java
            clear.setEnabled(false);
        }

    } //}}}

    //{{{ goToSelectedMarker() method
    private void goToSelectedMarker()
    {
        Object value = markerList.getSelectedValue();
        if(!(value instanceof Marker))
            return;

        Marker mark = (Marker)value;
        view.getTextArea().setCaretPosition(mark.getPosition());
        view.toFront();
        view.requestFocus();
        view.getTextArea().requestFocus();
    } //}}}

    //{{{ updateSelection() method
    private void updateSelection()
```

After: org.gjt.sp.jedit.gui.MakerViewer.java

```java
            clear.setEnabled(false);
        }

    } //}}}

    //{{{ goToSelectedMarker() method
    private void goToSelectedMarker()
    {
        view.goToSelectedMarker(markerList);
    } //}}}

    //{{{ updateSelection() method
    private void updateSelection()
    {
        ListModel model = markerList.getModel();
        int currentLine = view.getTextArea().getCaretLine();
        Buffer buffer = view.getBuffer();
        for (int i = 0; i < model.getSize(); i++)
        {
            Object o = model.getElementAt(i);
            if (o instanceof Marker)
```

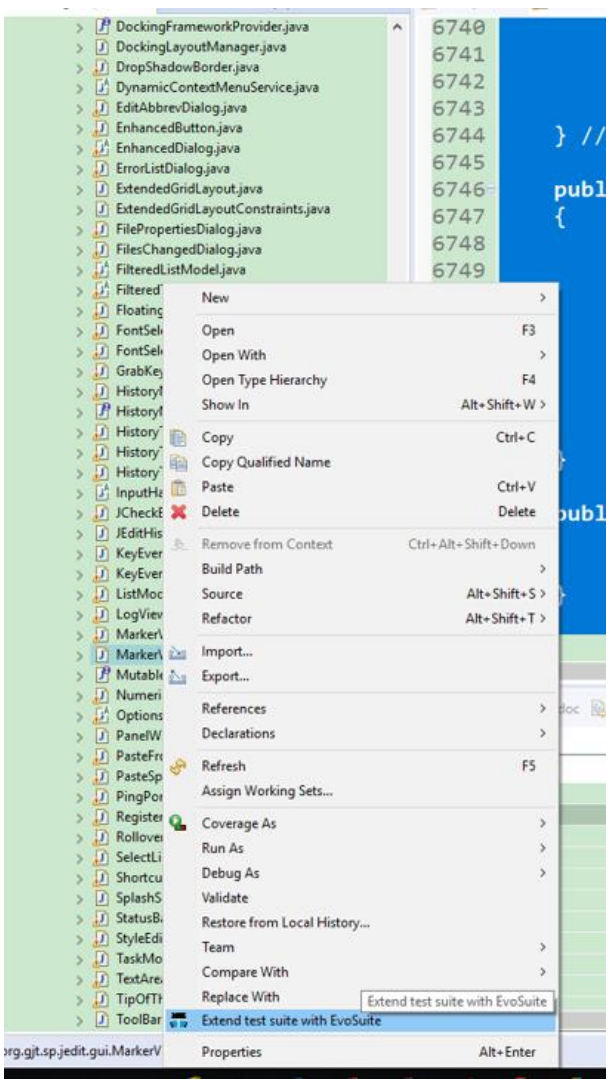
org.gjt.sp.jedit.view

```java
    //}}}

    public void goToSelectedMarker(JList<Marker> markerList) {
        Object value = markerList.getSelectedValue();
        if (!(value instanceof Marker))
            return;
        Marker mark = (Marker) value;
        getTextArea().setCaretPosition(mark.getPosition());
        toFront();
        requestFocus();
        getTextArea().requestFocus();
    }
```

Rationale: Where ever you see a method uses fields of another class extensively to perform some action, consider moving the action's logic into that class itself. Here we moved methods to org.gjt.sp.jedit.view class.

Test case: because the component does not have an associated test case, so we choose to use evosuite to automatic generate some test cases.



Case1 the program can successful operate. Before refactoring: passed. After refactoring: passed.

Case 2 actionPerformed has right action when it is clear, add-marker, next-marker, or prev-marker.

Before refactoring: passed. After refactoring: passed.

Furthermore, another tool can be used here. PIT, which is a state of the art mutation testing system, providing gold standard test coverage for Java.

```
org.gjt.sp.jedit.gui.MarkerViewer$KeyHandler (4)
    286: negated conditional
    287: negated conditional
    289: removed call to java/awt/event/KeyEvent::consume
    290: removed call to org/gjt/sp/jedit/gui/MarkerViewer::access$2
org.gjt.sp.jedit.gui.MarkerViewer$MouseHandler (3)
    270: negated conditional
    274: removed call to javax/swing/JList::setSelectedIndex
    276: removed call to org/gjt/sp/jedit/gui/MarkerViewer::access$2
org.gjt.sp.jedit.gui.MarkerViewer$Renderer (7)
    247: negated conditional
    253: negated conditional
    257: changed conditional boundary
    257: negated conditional
    259: Replaced integer addition with subtraction
    259: removed call to org/gjt/sp/jedit/gui/MarkerViewer$Renderer::setText
    261: mutated return of Object value for org/gjt/sp/jedit/gui/MarkerViewer$Renderer::getListCellRendererComponent to ( if (x != null) null else throw new RuntimeException )
```

So we can make sure if we covered all the test situation.

My changes did not affect the component's behavior.

1.2 Smell type: God class
Method: extract class

Source class: org.gjt.sp.jedit.options.GutterOptionPane

Before: org.gjt.sp.jedit.options.GutterOptionPane

```
View.java  TextArea.java  EditPane.java  FilesChange...  HistoryText....  MarkerViewe...  InputHandle...  SelectionMan...  GutterOption...

    //{{{ addFoldStyleChooser() method
    private void addFoldStyleChooser()
    {
        painters = ServiceManager.getServiceNames(JEditTextArea.FOLD_PAINTER_SERVICE);
        foldPainter = new JComboBox<String>();
        String current = JEditTextArea.getFoldPainterName();
        int selected = 0;
        for (int i = 0; i < painters.length; i++)
        {
            String painter = painters[i];
            foldPainter.addItem(jEdit.getProperty(
                "options.gutter.foldStyleNames." + painter, painter));
            if (painter.equals(current))
                selected = i;
        }
        foldPainter.setSelectedIndex(selected);
        addComponent(new JLabel(jEdit.getProperty("options.gutter.foldStyle.label")), foldPain
    } //}}}

    //{{{ isGutterEnabled() method
```

After: org.gjt.sp.jedit.options.GutterOptionPaneProduct

```java
        return foldPainter;
    }

    public String[] getPainters() {
        return painters;
    }

    public void addFoldStyleChooser(GutterOptionPane gutterOptionPane) {
        painters = ServiceManager.getServiceNames(JEditTextArea.FOLD_PAINTER_SERVICE);
        foldPainter = new JComboBox<String>();
        String current = JEditTextArea.getFoldPainterName();
        int selected = 0;
        for (int i = 0; i < painters.length; i++) {
            String painter = painters[i];
            foldPainter.addItem(jEdit.getProperty("options.gutter.foldStyleNames." + painter, p
            if (painter.equals(current))
                selected = i;
        }
        foldPainter.setSelectedIndex(selected);
        gutterOptionPane.addComponent(new JLabel(jEdit.getProperty("options.gutter.foldStyle.la
    }
}
```

Rationale: The basic idea is to extract parts of its functions into other classes.

In order to salve the god class problem in addFoldStyleChooser () method, we extract addFoldStyleChooser() from org.gjt.sp.jedit.options.GutterOptionPane to org.gjt.sp.jedit.options.GutterOptionPaneProduct.

Test case:

Case 1 the program can successful operate. Before refactoring: passed. After refactoring: passed.

Case 2 check if the size of FontProperty is equal to 10. Before refactoring: passed. After refactoring: passed.

Case 3 check if the font of FontProperty is PLAIN. Before refactoring: passed. After refactoring: passed.

My changes did not affect the component's behavior

2 Manual refactoring
2.1 Smell type: duplication
Source: org.gjt.sp.jedit.textarea.TextArea

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

Because we don't use tool in this case, so it is easier for us to find some bad smells like duplication first, there are many methods in org.gjt.sp.jedit.textarea.TextArea so we may find some methods with the similar code. When we opened this class, we indeed found a sequence of source code that occurs more than once (in both toUpperCase() and toLowerCase() methods). Additional, duplication may appear when there are two options for an attribute, and each option has a very similar operation to another one.

2. Explain why the class/method is smelly (be specific).

When we get toUpperCase() and toLowerCase(), those two different methods have almost the same code, so this case can be treated as a duplication bad smell.

```
public void toUpperCase()
{
    if(!buffer.isEditable())
    {
        javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
        return;
    }

    Selection[] selection = getSelection();
    int caret = -1;
    if (selection.length == 0)
    {
        caret = getCaretPosition();
        selectWord();
        selection = getSelection();
    }
    if (selection.length == 0)
    {
        if (caret != -1)
            setCaretPosition(caret);
        javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
        return;
    }

    buffer.beginCompoundEdit();

    for (Selection s : selection)
        setSelectedText(s, getSelectedText(s).toUpperCase());

    buffer.endCompoundEdit();
    if (caret != -1)
        setCaretPosition(caret);
```

```java
public void toLowerCase()
{
    if(!buffer.isEditable())
    {
        javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
        return;
    }

    Selection[] selection = getSelection();
    int caret = -1;
    if (selection.length == 0)
    {
        caret = getCaretPosition();
        selectWord();
        selection = getSelection();
    }
    if (selection.length == 0)
    {
        if (caret != -1)
            setCaretPosition(caret);
        javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
        return;
    }

    buffer.beginCompoundEdit();

    for (Selection s : selection)
        setSelectedText(s, getSelectedText(s).toLowerCase());

    buffer.endCompoundEdit();
    if (caret != -1)
```

In order to combine those two methods together, we can use an if sentence to implement it.

```java
public void toUpperORLowerCase(String toCase)
{
    if(!buffer.isEditable())
    {
        javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
        return;
    }

    Selection[] selection = getSelection();
    int caret = -1;
    if (selection.length == 0)
    {
        caret = getCaretPosition();
        selectWord();
        selection = getSelection();
    }
    if (selection.length == 0)
    {
        if (caret != -1)
            setCaretPosition(caret);
        javax.swing.UIManager.getLookAndFeel().provideErrorFeedback(null);
        return;
    }

    buffer.beginCompoundEdit();


    for (Selection s : selection)
    {
        if (toCase == toUpper)
        setSelectedText(s, getSelectedText(s).toUpperCase());
        if (toCase == toLower)
            setSelectedText(s, getSelectedText(s).toLowerCase());
```

Test Case:

Case 1: the program can successful operate. Before refactoring: passed. After refactoring: passed.

Case 2: check if isCaretBlinkEnabled () return caretBlinks. Before refactoring: passed. After refactoring: passed.

Case 3: check setElectricScroll (). Before refactoring: passed. After refactoring: passed.

JUnit TextAreaTest() can successfully operate.

Before refactoring: passed.

After refactoring: passed.

My changes did not affect the component's behavior

We run JDeodorant and we find that the smell was removed. New methods are not smelly.

2.2 Smell type: type checking
Source: org.gjt.sp.jedit.EditPane

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

In StatusHandler class, when we use switch sentence and there are many constant variables (OVERWRITE_CHANGED, MULTI_SELECT_CHANGED, RECT_SELECT_CHANGED) which are doing some jobs in the program, we are making our code less flexible for no real gain (program should follow object-oriented design principles) while also making it harder to understand and reuse. Most of the time, if we find a lot of if, else, switch, or instanceof in a program, we may consider those situations more if those really make program simple. There are some examples from a research paper online:

Table 1: Type-checking examples where an attribute represents state

a: *switch* statement	b: *if/else if* statement

class Context {
private int **type**;
public void m() {
switch(**type**) {
case *VALUE_0*:
//code for case 0
case *VALUE_1*:
//code for case 1
case …
}
}
}
	class Context {
	private int **type**;
	public void m() {
	if(**type** == *VALUE_0*) {
	//code for case 0
	}
	else if(**type** == *VALUE_1*) {
	//code for case 1
	}
	else if(**type** ==…) {…}
	}
	}

Table 2: Type-checking examples performing RTTI

a: *instanceof*	b: *getClass*	c: The subclass type is polymorphically obtained

class Client {
public void m(SuperType **type**) {
if(**type** *instanceof* Subclass0) {
Subclass0 s = (Subclass0)type;
//code for case 0
} else
if(**type** *instanceof* Subclass1) {
Subclass1 s = (Subclass1)type;
//code for case 1
}
}
}
	class Client {
	public void m(SuperType **type**) {
	if(**type**.*getClass()* == Subclass0.*class*) {
	Subclass0 s = (Subclass0)type;
	//code for case 0
	} else
	if(**type**.*getClass()* == Subclass1.*class*) {
	Subclass1 s = (Subclass1)type;
	//code for case 1
	}
	}
	}
		class Client {
		public void m(SuperType **type**) {
		if(**type**.*getType()* == STATIC_*VALUE_0*) {
		Subclass0 sub = (Subclass0)type;
		//code for case 0
		}
		else if(**type**.*getType()* == *STATIC_VALUE_1*) {
		Subclass1 sub = (Subclass1)type;
		//code for case 1
		}
		}
		}

In this case: If the conditional code fragment is a switch statement, the type field (or an invocation of its getter method) should appear in the switch expression, while the static attributes representing the different values that the type field may obtain should appear in all case expressions (bad type checking smell)

References:

Tsantalis, Nikolaos, et al. “JDeodorant: Identification and Removal of Type-Checking Bad Smells .” JDeodorant: Identification and Removal of Type-Checking Bad Smells - IEEE Conference Publication, ieeexplore.ieee.org/document/4493342.

2. Explain why the class/method is smelly (be specific).

In this class, we used OVERWRITE_CHANGED, MULTI_SELECT_CHANGED, RECT_SELECT_CHANGED three constant variables. Those variables will make other programmers feel hard to understand (because this program is not based on object-oriented design principles), and also reduce the readability and maintainability of the program. The best solution is to take advantage of polymorphism.

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

Before: org.gjt.sp.jedit.EditPane

```
            switch(flag)
            {
            case OVERWRITE_CHANGED:
                status.setMessageAndClear(
                    jEdit.getProperty("view.status.overwrite-changed",
                    new Integer[] { value ? 1 : 0 }));
                break;
            case MULTI_SELECT_CHANGED:
                status.setMessageAndClear(
                    jEdit.getProperty("view.status.multi-changed",
                    new Integer[] { value ? 1 : 0 }));
                break;
            case RECT_SELECT_CHANGED:
                status.setMessageAndClear(
                    jEdit.getProperty("view.status.rect-select-changed",
                    new Integer[] { value ? 1 : 0 }));
                break;
            }
```

After: org.gjt.sp.jedit.EditPane

```
        {
            StatusBar status = view.getStatus();
            if(status == null)
                return;

            getFlagObject(flag).statusChanged(value, status);

            status.updateMiscStatus();
        }
```

```
        private Flag getFlagObject(int flag) {
            switch (flag) {
            case StatusListener.OVERWRITE_CHANGED:
                return new OverwriteChanged();
            case StatusListener.MULTI_SELECT_CHANGED:
                return new MultiSelectChanged();
            case StatusListener.RECT_SELECT_CHANGED:
                return new RectSelectChanged();
            }
            return null;
        }
    } //}}}
```

org.gjt.sp.jedit.RectSelectChanged.java

```java
package org.gjt.sp.jedit;


import org.gjt.sp.jedit.gui.StatusBar;


public class RectSelectChanged extends Flag {
    public void statusChanged(boolean value, StatusBar status) {
        status.setMessageAndClear(
                jEdit.getProperty("view.status.rect-select-changed", new Integer[] { value ? 1
    }
}
```

Here we remove original switch sentence first, and plan to take advantage of polymorphism in the program in order to achieve object-oriented design principle (return). After that, we create a new method statusChanged() in org.gjt.sp.jedit.RectSelectChanged.java to control the relationship between value and status so we can have a single function that accepts multiple types of inputs.

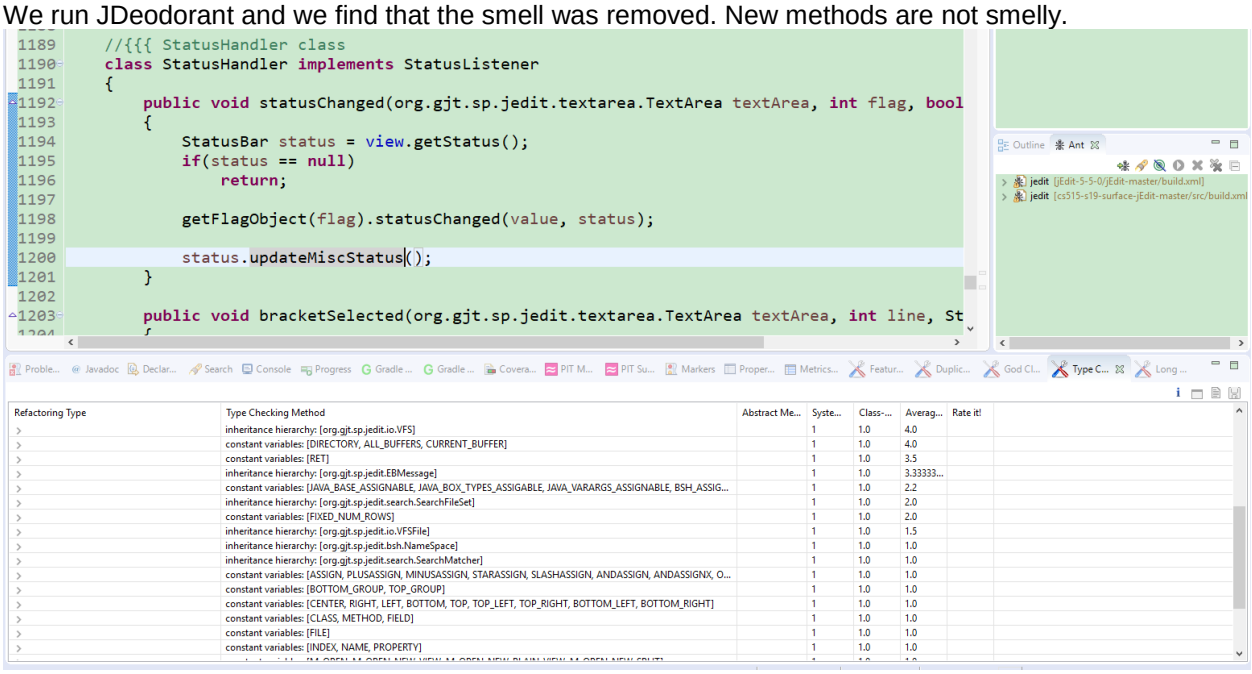
2. Give the rationale of the chosen refactoring operations.
The rationale is to achieve object-oriented design principle and eliminate type-checking conditional statements based on the inheritance and polymorphism.

Test cases:
Case 1: the program can successful operate. Before refactoring: passed. After refactoring: passed.

Case 2: check getTextArea() return textArea. Before refactoring: passed. After refactoring: passed.

Case 2: check bufferRenamed(), if oldPath change to newPath. Before refactoring: passed. After refactoring: passed.

We run JDeodorant and we find that the smell was removed. New methods are not smelly.



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

Automated refactoring gives us a simple and fast way to implement the whole process when we need to spend a lot of time on manual refactoring processes. Also, when we need to refactor a lot of programs at the same time, automated refactoring processes can save our time and provide high efficiency, but in some complex cases, we are more confident with the results from manual refactoring processes because algorithm may be not precise in automated refactoring processes.
Difficulties：it is difficult to find bad smell when we do manual refactoring processes.

Computer sometimes cannot handle some bad smell.