Lab 12 Requirements

Create a new Eclipse workspace named "**Lab12_1234567890**" on the desktop of your computer (replace **1234567890** with your student ID number). For each question below, create a new project in that workspace. Call each project by its question number: "**Question1**", "**Question2**", etc. If you do not remember how to create a workspace or projects, read the "*Introduction to Eclipse*" document which is on iSpace. Answer all the questions below. At the end of the lab, create a ZIP archive of the whole workspace folder. The resulting ZIP file must be called "**Lab12_1234567890.zip**" (replace **1234567890** with your student ID number). Upload the ZIP file on iSpace.

Question 1

Here is the code from last week's lab, which creates a small GUI that can be used to draw a sequence of red lines:

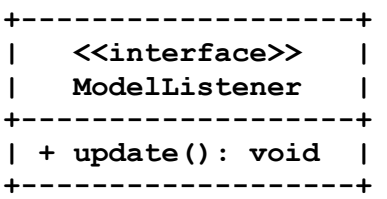
```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.ArrayList;
import javax.swing.JPanel;
public class MyPanel extends JPanel {
   private ArrayList<Point> points;
   public MyPanel() {
      points = new ArrayList<Point>();
      this.addMouseListener(new MouseAdapter() {
         @Override
         public void mouseClicked(MouseEvent e) {
            if(e.getButton() == MouseEvent.BUTTON1) {
               points.add(e.getPoint());
               repaint();
            }
         }
      });
   }
   @Override
   protected void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.setColor(Color.RED);
      if(points.size() == 1) {
         Point p = points.get(0);
         g.drawRect((int)p.getX(), (int)p.getY(), 1, 1);
      } else {
         for(int i = 1; i < points.size(); i++) {
            Point start = points.get(i - 1);
            Point end = points.get(i);
            g.drawLine((int)start.getX(), (int)start.getY(),
                  (int)end.getX(), (int)end.getY());
         }
      }
   }
   public void clearAllPoints() {
      points.clear();
      repaint();
   }
   public void undoPoint() {
      if(points.size() > 0) {
         points.remove(points.size() - 1);
```

```
            repaint();
        }
    }
}

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class MyFrame extends JFrame {
    public MyFrame() {
        this.setTitle("MyFrame Title");
        this.setSize(400, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);
        this.setLayout(new BorderLayout());

        MyPanel centerPanel = new MyPanel();
        this.add(centerPanel, BorderLayout.CENTER);
        JPanel topPanel = new JPanel();
        this.add(topPanel, BorderLayout.PAGE_START);
        topPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
        JButton resetButton = new JButton("Reset");
        resetButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                centerPanel.clearAllPoints();
            }
        });
        topPanel.add(resetButton);
        JButton undoButton = new JButton("Undo");
        undoButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                centerPanel.undoPoint();
            }
        });
        topPanel.add(undoButton);
        this.setVisible(true);
    }
}

public class Start {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new MyFrame();
            }
        });
    }
}
```

Run this code and check that it works correctly.

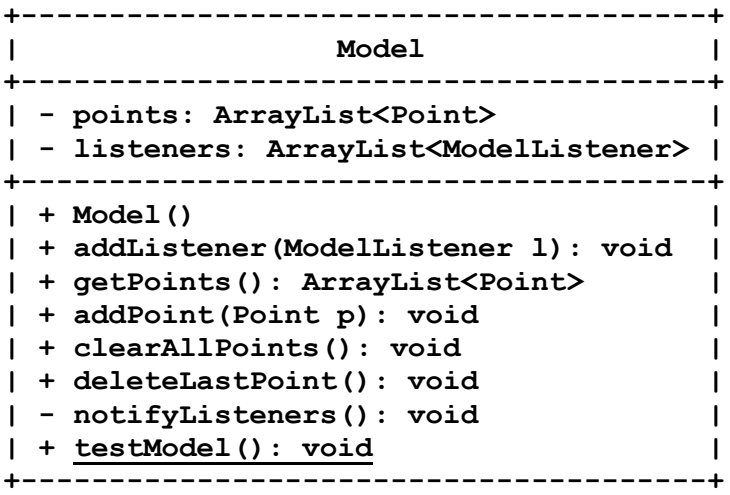
This code is not very good though because it mixes the data code (the arraylist of points) and the GUI code. To solve this problem, we are going to rewrite the code to follow the Model-View-Controller design pattern.

Create an interface **`ModelListener`** with the following UML specification:



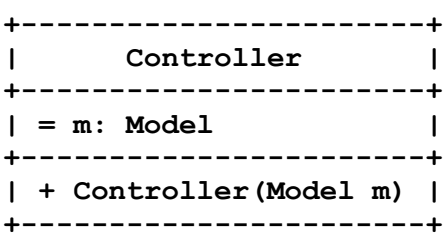
Later, this interface will be implemented by all the views in your software that need to listen to changes in the model.

Then create a class **`Model`** with the following UML specification:



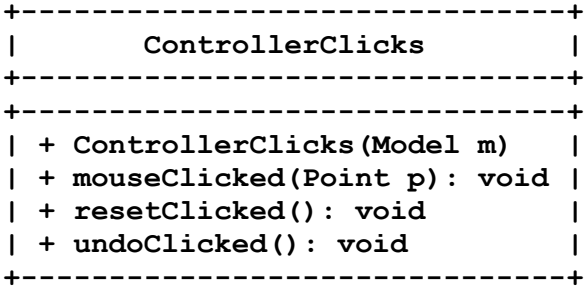
This class stores all the data about the points in your software. The **`clearAllPoints`** method deletes all the points from the arraylist of points. The **`deleteLastPoint`** method deletes the last point in the arraylist of points. Your code must call the **`notifyListeners`** method every time the arraylist of points changes. The **`notifyListeners`** method must then use an enhanced **`for`** loop to notify all the listeners that something has changed in the arraylist of points.

Create a class **`Controller`** with the following UML specification:



This class is going to be the superclass for all the controllers in your software. The instance variable **`m`** is **`protected`**.

Create a class **`ControllerClicks`** that extends **`Controller`**, with the following UML specification:



This controller implements the meaning of all the clicks in your software: the **mouseClicked** method will later be called when the user clicks to draw a new red line; the **resetClicked** method will later be called when the user clicks on the "**Reset**" button; the **undoClicked** method will later be called when the user clicks on the "**Undo**" button. The **mouseClicked** method calls the **addPoint** method of the model. The **resetClicked** method calls the **clearAllPoints** method of the model. The **undoClicked** method calls the **deleteLastPoint** method of the model.

Create a class **View** that extends **JFrame** and implements the **ModelListener** interface, with the following UML specification:

```
+------------------+
|       View       |
+------------------+
| = m: Model       |
+------------------+
| + View(Model m)  |
| + update(): void |
+------------------+
```

This class is going to be the superclass for all the views in your software. The instance variable **m** is **protected**. The **update** method is **abstract**.

Modify the **MyFrame** class from the code above to extend **View**, with the following UML specification:

```
+--------------------------------------+
|               MyFrame                |
+--------------------------------------+
| - c: ControllerClicks                |
+--------------------------------------+
| + MyFrame(Model m, ControllerClicks c) |
| + update(): void                     |
+--------------------------------------+
```

In the constructor of **MyFrame**, the **actionPerformed** method of the **ActionListener** of the "**Reset**" button must call the **resetClicked** method of the controller **c**, and the **actionPerformed** method of the **ActionListener** of the "**Undo**" button must call the **undoClicked** method of the controller **c**. The controller **c** will then itself implement the actual meaning of the button clicks.

The **update** method must simply call Swing's **repaint** method: when the model changes (because a new point has been added, for example), the model will notify all its listeners, which will call the **update** method of **MyFrame**, which will ask Swing to repaint everything; Swing will then automatically call the **paintComponent** method of the **MyPanel** (see below) which will then redraw everything (including the new point).

Modify the **MyPanel** class from the code above to extend **JPanel** (as before), with the following UML specification:

```
+--------------------------------------+
|                MyPanel               |
+--------------------------------------+
| - m: Model                           |
| - c: ControllerClicks                |
+--------------------------------------+
| + MyPanel(Model m, ControllerClicks c) |
| = paintComponent(Graphics g): void   |
+--------------------------------------+
```

If you compare with the original code of **MyPanel** above, you will note that the arraylist of points has been removed (it is now in the model). In the constructor of **MyPanel**, the **mouseClicked** method of the **MouseAdapter** of the panel must call the **mouseClicked** method of the controller **c** when the left mouse button is clicked in the panel. The controller **c** will then itself implement the actual meaning of the mouse clicks.

The **paintComponent** method must get all the points from the model and then draw the red lines between the points as before.

In the **run** method of the anonymous class that implements the **Runnable** interface in the **main** method of the **Start** class, create a model object of type **Model**, a controller object of type **ControllerClicks**, and a view object of type **MyFrame**. Do not forget to register the view with the model.

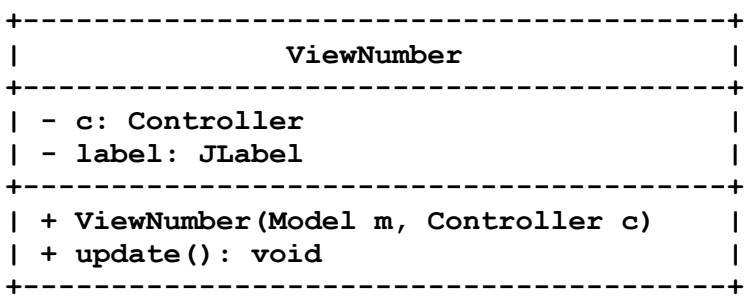
Run your software and check that it still works correctly exactly as before. Also make sure that you clearly understand how each part of your software works and how the different parts interact with each other!

Create a class **Test** with a **main** method that runs the tests of the model. You can then use the **Start** class when you want to run the whole software and you can use the **Test** class when you only want to run the tests (click on the **Start.java** or on the **Test.java** file with the right mouse button and use the **Run As → Java Application** menu to run the file).

Question 2

We now want to add a second view to the software to display the total number of points that are currently being drawn in the GUI. So:

1. Add to the model a new public **numberOfPoints** method that returns as result the number of points that are currently stored in the arraylist of points (note that the **numberOfPoints** method does not change the content of the arraylist so there is no need to call the **notifyListeners** method inside the **numberOfPoints** method).
2. Create a class **ViewNumber** that extends **View**, with the following UML specification:



   In the constructor of **ViewNumber**, create and add the label to the **ViewNumber** (remember that **ViewNumber** extends **View** which extends **JFrame** so **ViewNumber** inherits **JFrame**'s **add** method) and call the **update** method to initialize the label.

   The **update** method calls the **numberOfPoints** method of the model and uses the result to show in the label the total number of points that are currently being drawn in the GUI.

3. The **ViewNumber** does not receive any input from the user, so there is no need for the view's controller to do anything special, so for the controller you can just re-use the existing **Controller** class without any modification.

4. Modify the **`main`** method of the **`Start`** class to create a controller object of type **`Controller`**, and a view object of type **`ViewNumber`**. Do not forget to register the view with the existing model.

Run your software and check that the new view shows the correct number of points every time you add a new point or undo a point or reset all the points in the GUI.