Object-Oriented Programming

GUI Programming
Part 3

Data Science
United International College

Outline

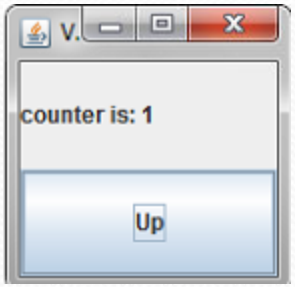
- Problems with GUI programming.
- Design patterns.
- The Model-View-Controller design pattern.
- The Observer design pattern.

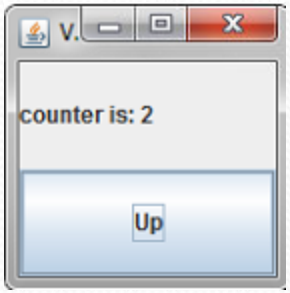
Simple GUI for a counter

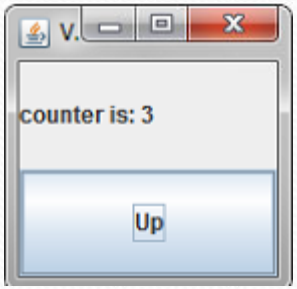
```
import ...
public class ViewUp extends JFrame {
    private int counter = 0;

    public ViewUp() {
        this.setTitle("View Up");
        this.setSize(150, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new GridLayout(2, 1));
        JLabel label = new JLabel("counter is: " + counter); // Label
        this.add(label);
        JButton buttonUp = new JButton("Up");                // Button
        buttonUp.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                counter++;
                label.setText("counter is: " + counter);
            }
        });
        this.add(buttonUp);
        this.setVisible(true);
    }
}
```

Simple GUI for a counter

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







Problems

The data code (the counter) is mixed with the GUI code:

- If there are many GUI components (buttons, labels, etc.) then the counter will be read / written in many different places.
- If there are more pieces of data, each piece might be read / written by each GUI component.
- Result: for **`N`** pieces of data and **`M`** GUI components, you get **`N`** × **`M`** relationships.

  High complexity is bad software engineering!
- We also cannot test the data code separately from the GUI code, so bugs are more likely to happen.

Problems

If we create two **`ViewUp`** objects in the **`main`** method, they cannot share the counter:

- We might want to be able to look at the same data in different ways (number, graph, etc.) using multiple GUI components in different panels / frames.
- Using one GUI component to change the data should automatically update all the other GUI components.
- Can we do this without having all the GUI components become dependent on each other (**`M`** × (**`M-1`**) relationships)?

Problems

To keep the code simple and manageable, we need:

- Separation between the data and GUI components.
- Separation between the GUI components.

- We need separation of concerns:
  - Each piece of code must do one thing and only one thing (and do it well!)
  - Different pieces of code must be as independent from each other as possible.
  - Result is modular software.

All the GUI components must still show the same data!

Solution

These problems have a well-known solution: the Model-View-Controller (MVC) design pattern.

- Design pattern:
  - A "recipe" for solving a specific software design problem.
  - Based on the practical experience of many software engineers over many years.
  - Does not require new technology: what changes is the way you organize your code.
- Dozens of software design patterns exist to solve various problems: Software design pattern (Wikipedia)
- Many books: Design Patterns: Elements of Reusable Object-Oriented Software "Gang of Four" (Wikipedia)

Model-View-Controller

- A well-known design pattern used in all GUI programming, not just in Java's Swing:
  - Also used in web applications.
  - Mobile phone applications.
  - Independent of the programming language.
- Solves both problems at the same time:
  - Separation between the data and GUI components.
  - Separation between the GUI components.

Model-View-Controller

How it works: we split the code into three parts.

- The model contains and manages the data.
- The view shows information about the data to the user.
- The controller transforms user GUI actions into model changes.

In our case:

- The model stores the counter and related methods.
- The view displays the value of the counter and the button.
- The controller transforms button clicks into counter increases.

Model

```
public class Model {
    private int counter;
    private ViewUp view;

    public Model () {
        counter = 0;
    }
    public void addView(ViewUp view) {
        this.view = view;
    }
    public int getCounter() {
        return counter;
    }
    public void setCounter(int counter) {
        this.counter = counter;
        notifyView(); // Counter changed so notify the view.
    }
    private void notifyView() {
        view.update(); // Tell the view that something changed.
    }
}
```

View

```
import ...
public class ViewUp extends JFrame {
    private Model m;
    private ControllerUp c;
    private JLabel label;

    public ViewUp(Model m, ControllerUp c) {
        this.m = m;
        this.c = c;
        // The model knows about the view, because the model needs to
        // notify the view to update itself every time the counter of
        // the model has changed.
        m.addView(this);

        this.setTitle("View Up");
        this.setSize(150, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new GridLayout(2, 1));

        ...
```

View

```
        ...

        label = new JLabel();                                   // Label
        update(); // Initialize the label using the model.
        this.add(label);

        JButton buttonUp = new JButton("Up");                   // Button
        buttonUp.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                c.up(); // Controller decides what the click means.
            }
        });
        this.add(buttonUp);
        this.setVisible(true);
    }
    // When notified of a change by the model, the view gets the new
    // value of the counter from the model, and updates its label.
    public void update() {
        label.setText("counter is: " + m.getCounter());
    }
}
```

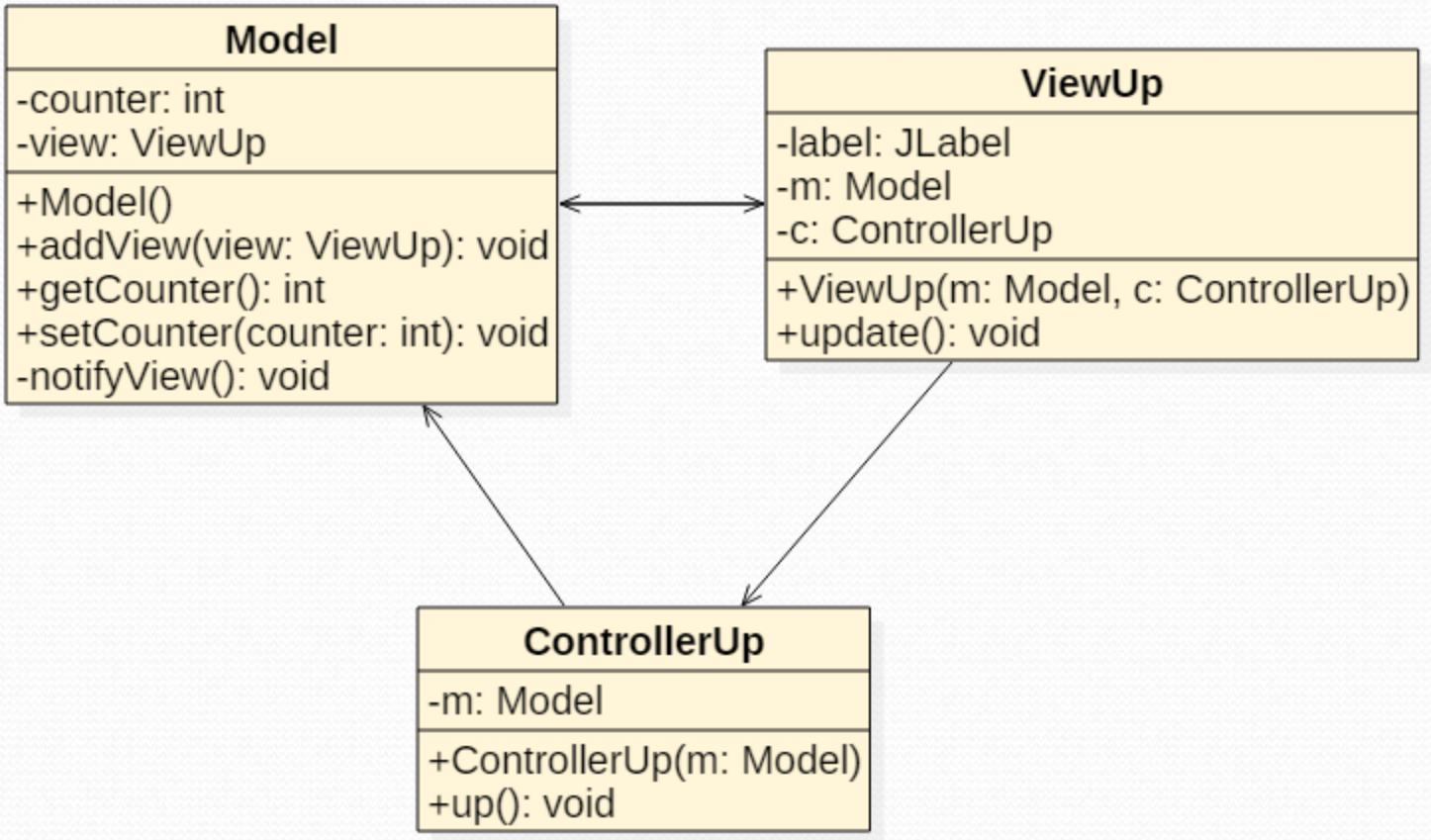
Controller

```
public class ControllerUp {
    private Model m;

    public ControllerUp(Model m) {
        this.m = m;
    }
    // A click on the view's "Up" button means increasing the
    // counter of the model by 1.
    public void up() {
        m.setCounter(m.getCounter() + 1);
    }
}
```

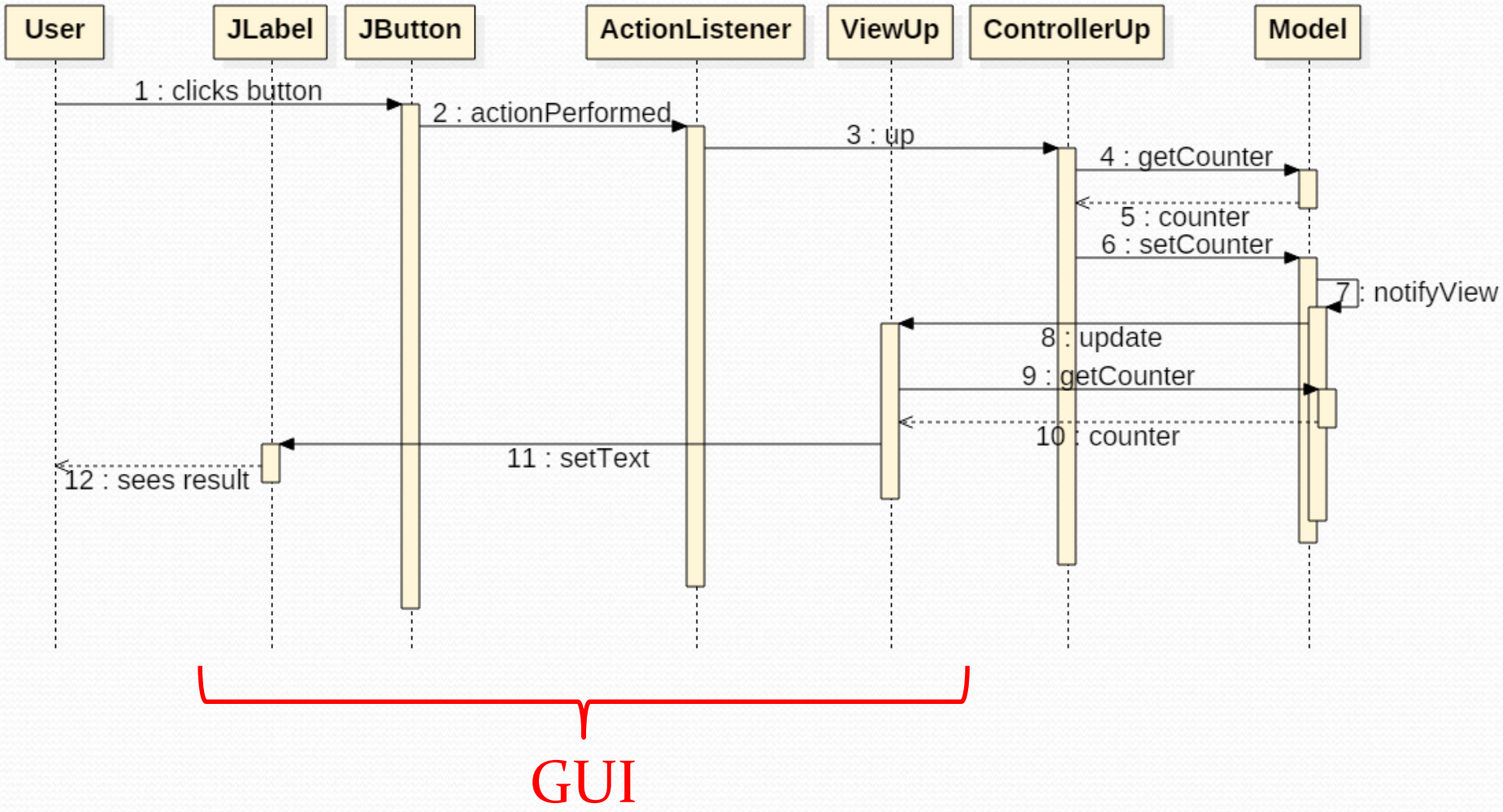
Test

```
public class Test2 {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                Model m = new Model();
                // The controller knows about the model (to increase the counter
                // of the model); the model does not know about the controller.
                ControllerUp c1 = new ControllerUp(m);
                // The view knows about the controller (which implements the
                // meaning of the button) and about the model (because the view
                // needs to display the value of the counter of the model).
                ViewUp v1 = new ViewUp(m , c1);
            }
        });
    }
}
```

Model-View-Controller



Model-View-Controller



Model-View-Controller

The organization of the code has changed but from the point of view of the user the result is the same as before! MVC simply organizes the code in a different way.

Disadvantages:

- More classes and more code.
- Interactions between MVC parts is hard to understand for beginners.
- A little slower than before (many method calls).

Model-View-Controller

Advantages:

- The three parts are separate from each other, so each part can be implemented by a different programmer.
- All the data is encapsulated inside the model, so the model can be tested on its own:

```
public static void testModel() { … }
```

Problem

- If we create two **ViewUp** objects in the **main** method, they still cannot share the same model:

```
ControllerUp c2 = new ControllerUp(m);
ViewUp v2 = new ViewUp(m , c2);
```

- Because the model can only know about one view!

```
public class Model {
    ...
    private ViewUp view;
    ...
}
```

- Solution: use an **ArrayList** of views inside the model!

Model

```
import java.util.ArrayList;
public class Model {
    private int counter;
    private ArrayList<ViewUp> views;

    public Model() {
        counter = 0;
        views = new ArrayList<ViewUp>();
    }
    public void addView(ViewUp view) {
        views.add(view);
    }
    public int getCounter() {
        return counter;
    }
    public void setCounter(int counter) {
        this.counter = counter;
        notifyViews(); // Counter changed so notify the views.
    }
    private void notifyViews() {
        for(ViewUp v: views) {
            v.update(); // Tell each view that something changed.
        }
    }
}
```

Model

- `private ArrayList<ViewUp> views;` means that `views` is an arraylist that can only contain objects from the `ViewUp` class.
- The code uses an "enhanced for loop":

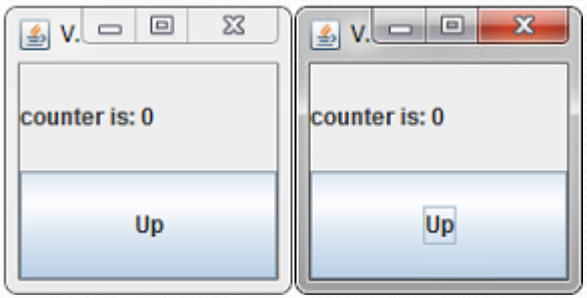
```
for(ViewUp v: views) {
    v.update();
}
```

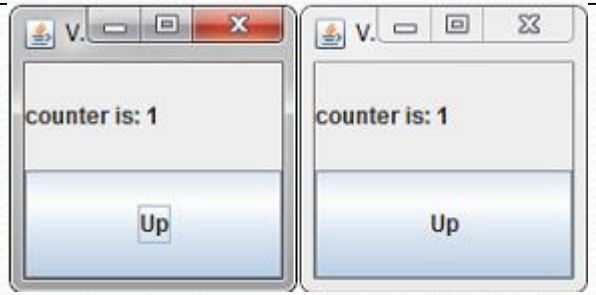
which works the same way as:

```
for(int i = 0; i < views.size(); i++) {
    ViewUp v = views.get(i);
    v.update();
}
```

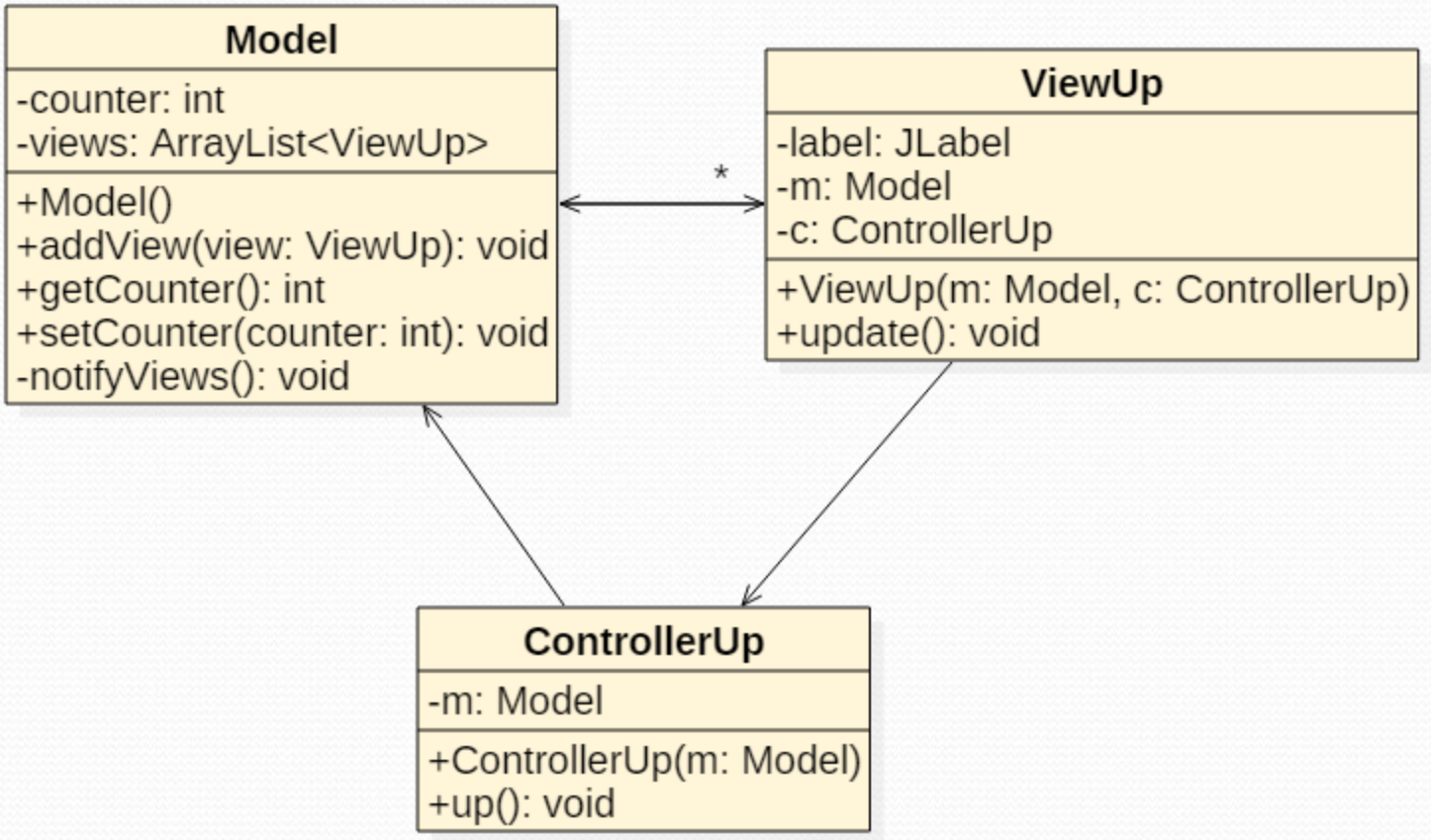
Test

```
public class Test3 {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                Model m = new Model(); // Single shared model.
                ControllerUp c1 = new ControllerUp(m);
                ViewUp v1 = new ViewUp(m , c1);
                ControllerUp c2 = new ControllerUp(m);
                ViewUp v2 = new ViewUp(m , c2);
            }
        });
    }
}
```





Model-View-Controller



Model-View-Controller

Advantages:

- The three parts are separate from each other, so each part can be implemented by a different programmer.
- All the data is encapsulated inside the model, so the model can be tested on its own:

```
public static void testModel() { … }
```

- When the model's data changes, all the views are automatically updated.

Model-View-Controller

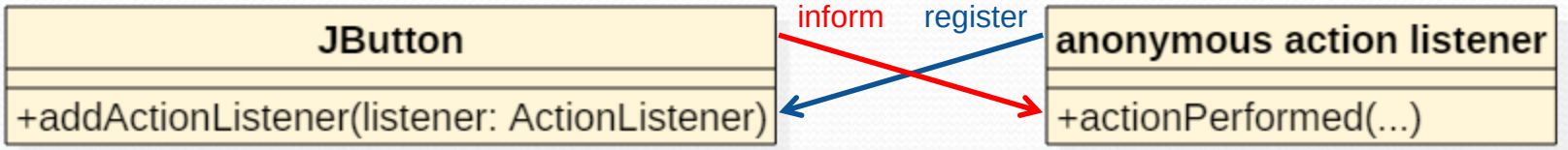
- The model can now have many `ViewUp` views.
- All the views share the same model.
- Each view has its own controller that defines its meaning.
  - In the present case we could share the controller between views too.
  - In practice views are often different from each other and have different controllers.
- Problem: the model can only have views of type `ViewUp`, not any other kind of view!
- Solution: the Observer design pattern.

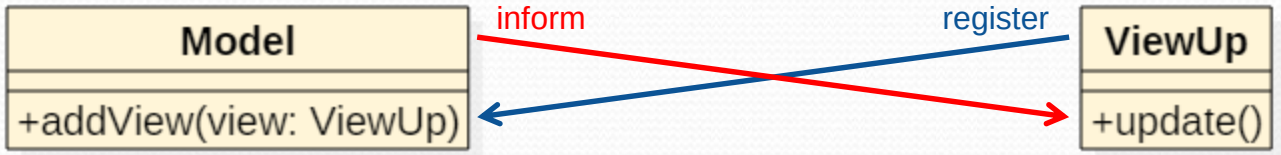
Observer design pattern

Basic idea:

- In Swing, a button / frame / panel / etc. can have many different kinds of listeners.
- In our MVC code, each view acts as a listener too:
  - Each view is registered with the model using the model's **`addView`** method.
    - Just like an action listener is registered with a button using the button's **`addActionListener`** method.
  - When the model changes, the model informs each view by calling the **`update`** method of the view.
    - Just like when a button is clicked, the button informs each action listener by calling the **`actionPerformed`** method of the action listener.

Observer design pattern





Observer design pattern

- This register / inform system is called the Observer design pattern.
- It is one part of the bigger MVC pattern.
- It is also used a lot on its own for all the event listeners (in all GUI programming, not just in Swing).
- It is used every time some objects (called the observers or listeners) need to observe changes in another object (called the subject).
- In our case, the model is the subject, and the views are the listeners.

Observer design pattern

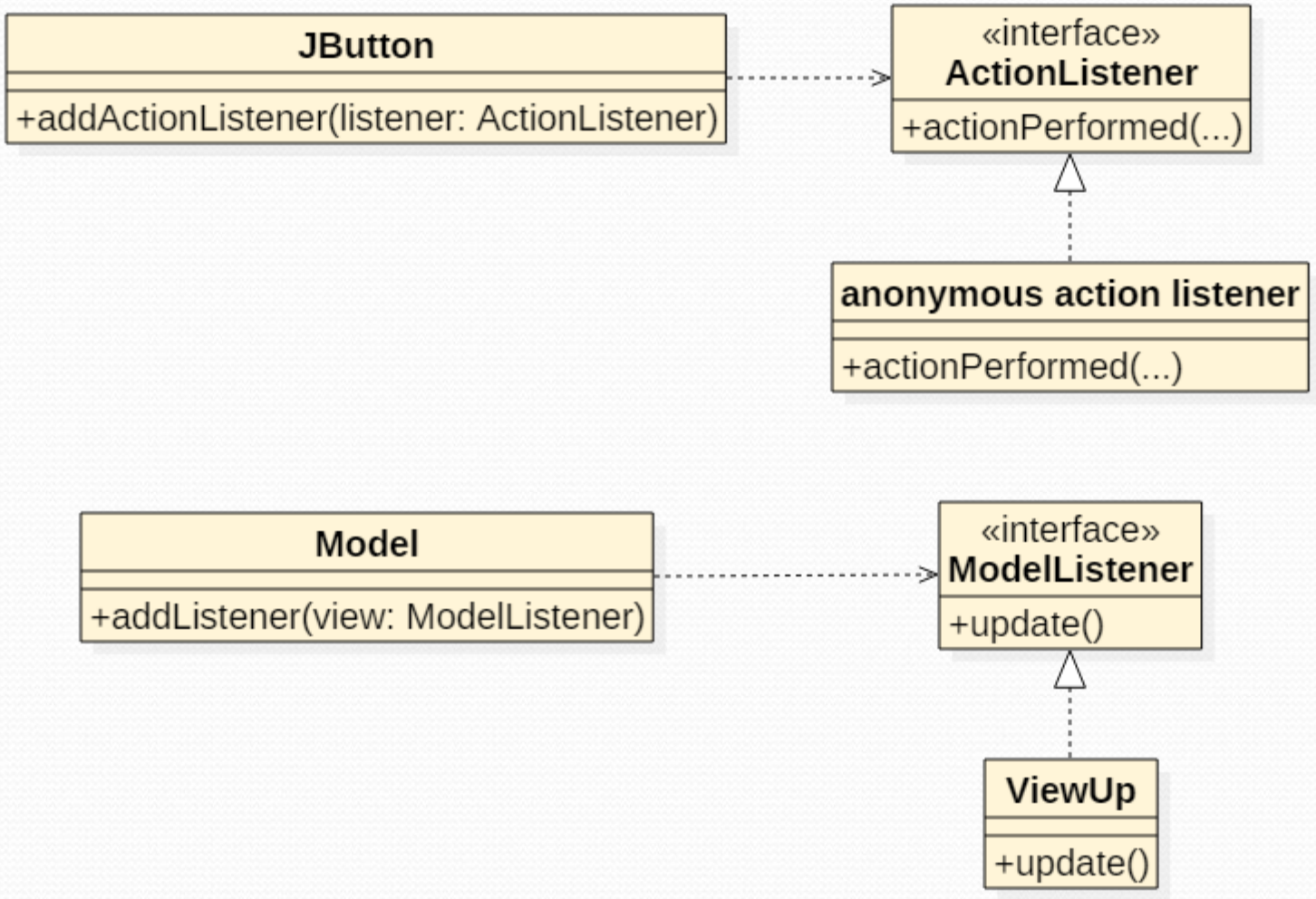
How it works:

1. The listeners must implement an interface which is specific to the subject.
   - Example: `ActionListener` for button click listeners.
2. The listeners can then register themselves with the subject using a method provided by the subject.
   - Example: button click listeners can register themselves with a button using the button's `addActionListener` method.
3. Later the subject informs all its listeners every time something happens.
   - Example: the button calls the `actionPerformed` method of each action listener when there is a button click.

Observer design pattern

- So for each kind of GUI event (button clicks, mouse movements, etc.) Swing provides an interface (`ActionListener`, `MouseMotionListener`, etc.) that listeners must implement before they can register themselves with the corresponding GUI component (button, panel, etc.)
- Similarly, we can create our own interface that views must implement before they can register themselves with the model: the `ModelListener` interface.
- Views become model listeners.

Observer design pattern



Observer design pattern

- Any view that implements the `ModelListener` interface can then register itself with the model.
- Later all these model listeners will be informed by the model when the model changes.

ModelListener

```
public interface ModelListener {
    public void update();
}
```

Model

```
import java.util.ArrayList;
public class Model {
    private int counter;
    private ArrayList<ModelListener> listeners;

    public Model() {
        counter = 0;
        listeners = new ArrayList<ModelListener>();
    }
    public void addListener(ModelListener l) {
        listeners.add(l);
    }
    public int getCounter() {
        return counter;
    }
    public void setCounter(int counter) {
        this.counter = counter;
        notifyListeners(); // Counter changed so notify the listeners.
    }
    private void notifyListeners() {
        for(ModelListener l: listeners) {
            l.update(); // Tell the listener that something changed.
        }
    }
}
```

ViewUp

```
import ...
public class ViewUp extends JFrame implements ModelListener {
    private Model m;
    private ControllerUp c;
    private JLabel label;

    public ViewUp(Model m, ControllerUp c) {
        this.m = m;
        this.c = c;
        m.addListener(this);

        this.setTitle("View Up");
        this.setSize(150, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new GridLayout(2, 1));

        label = new JLabel();                                    // Label
        update(); // Initialize the label using the model.
        this.add(label);

        ...
```

ViewUp

```
        ...

        JButton buttonUp = new JButton("Up");                    // Button
        buttonUp.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                c.up(); // Controller decides what the click means.
            }
        });
        this.add(buttonUp);
        this.setVisible(true);
    }
    // When notified of a change by the model, the view gets the new
    // value of the counter from the model, and updates its label.
    @Override
    public void update() {
        label.setText("counter is: " + m.getCounter());
    }
}
```

ControllerUp

Same as before:

```
public class ControllerUp {
    private Model m;

    public ControllerUp(Model m) {
        this.m = m;
    }
    // A click on the view's Up button means increasing the
    // counter of the model by 1.
    public void up() {
        m.setCounter(m.getCounter() + 1);
    }
}
```

ViewReset

```
import ...
public class ViewReset extends JFrame implements ModelListener {
    private Model m;
    private ControllerReset c;
    private JLabel label;

    public ViewReset(Model m, ControllerReset c) {
        this.m = m;
        this.c = c;
        m.addListener(this);

        this.setTitle("View Reset");
        this.setSize(150, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new GridLayout(2, 1));

        label = new JLabel();                                  // Label
        update(); // Initialize the label using the model.
        this.add(label);

        ...
```

ViewReset

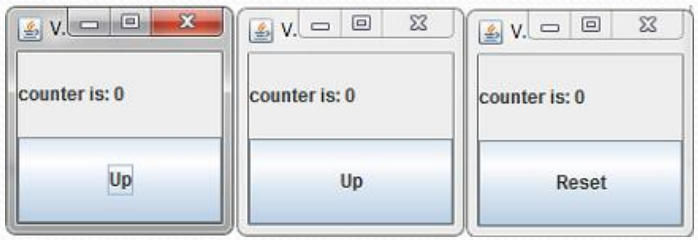
```
        ...

        JButton buttonReset = new JButton("Reset");          // Button
        buttonReset.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                c.reset(); // Controller decides what the click means.
            }
        });
        this.add(buttonReset);
        this.setVisible(true);
    }
    // When notified of a change by the model, the view gets the new
    // value of the counter from the model, and updates its label.
    @Override
    public void update() {
        label.setText("counter is: " + m.getCounter());
    }
}
```

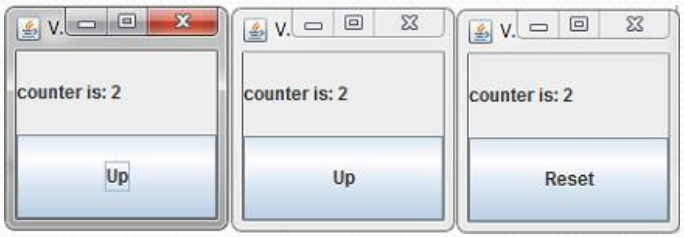
ControllerReset

```
public class ControllerReset {
    private Model m;

    public ControllerReset(Model m) {
        this.m = m;
    }
    // A click on the view's Reset button means the
    // counter of the model goes back to zero.
    public void reset() {
        m.setCounter(0);
    }
}
```

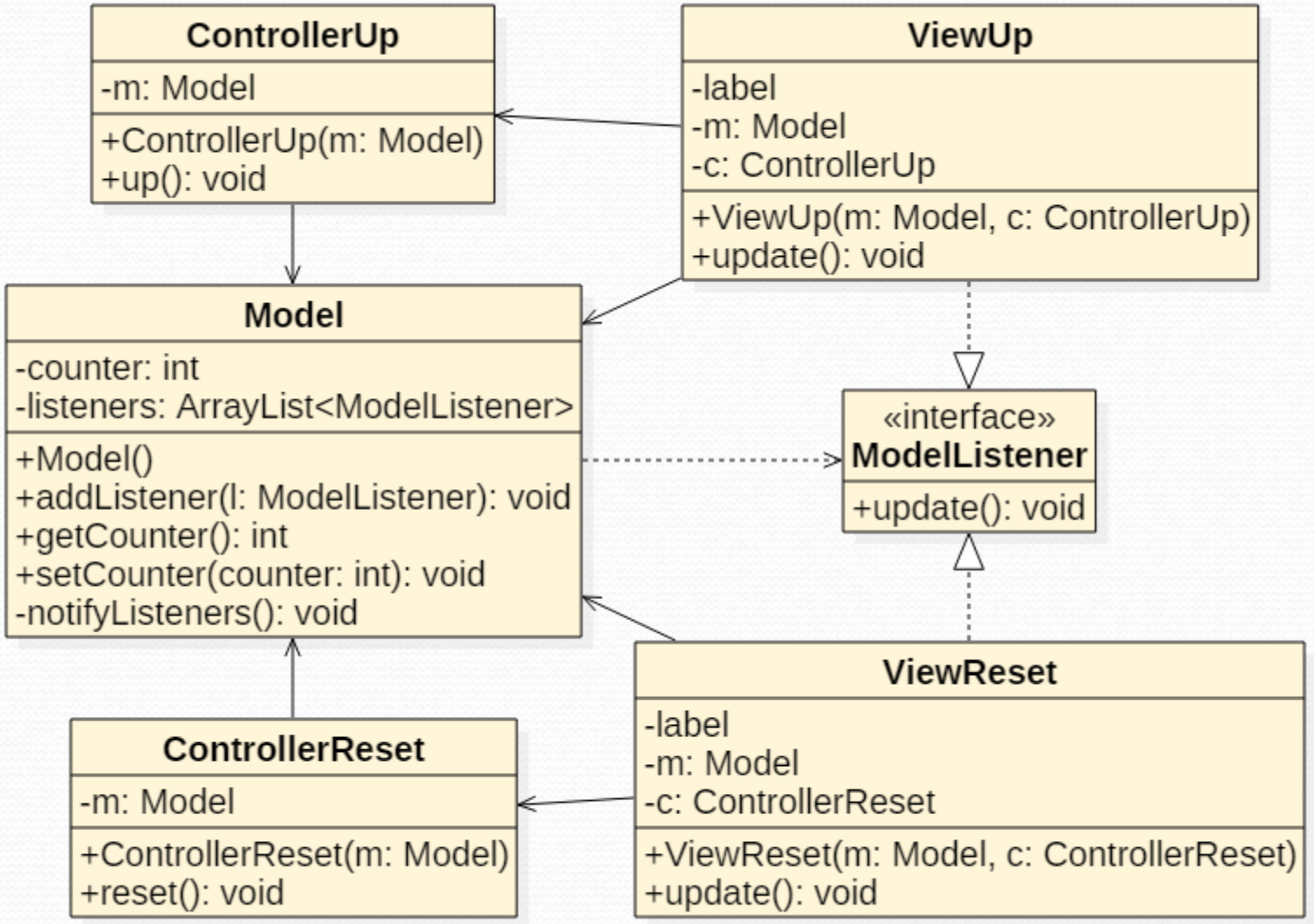
Test

```
public class Test4 {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                Model m = new Model(); // Single shared model.

                ControllerUp c1 = new ControllerUp(m);
                ViewUp v1 = new ViewUp(m , c1);

                ControllerUp c2 = new ControllerUp(m);
                ViewUp v2 = new ViewUp(m , c2);

                ControllerReset c3 = new ControllerReset(m);
                ViewReset v3 = new ViewReset(m, c3);
            }
        });
    }
}
```





Model-View-Controller



Model-View-Controller

Advantages:

- The three parts are separate from each other, so each part can be implemented by a different programmer.
- All the data is encapsulated inside the model, so the model can be tested on its own:

```
public static void testModel() { … }
```

- When the model's data changes, all the views are automatically updated.
- We can now use any view we want with the model, as long as each view implements `ModelListener`!

Observer design pattern

- The Observer design pattern is so common that Java provides:
  - An **`Observable`** class that can be extended by any subject.
  - An **`Observer`** interface that can be implemented by any observer / listener.
- In our case, we could make **`Model`** a subclass of **`Observable`** and make **`ViewUp`** and **`ViewReset`** implement **`Observer`** (instead of **`ModelListener`**).
- Note: this is only useful if the **`Model`** class does not need to extend some other class.
- In practice using our own **`ModelListener`** interface instead of using **`Observer`** works just fine.

Model-View-Controller

- To finish, every view:
  - Extends **`Jframe`** .
  - Implements **`ModelListener`** .
  - Has private instance variables for the model and the controller.
  - Registers itself with the model.
  - Calls **`setDefaultCloseOperation`**.
- Every controller:
  - Has a private instance variable for the model.
- To make life a little bit easier, we can then have an **`abstract`** superclass **`View`** for all views, and a superclass **`Controller`** for all controllers, with **`protected`** instance variables for the other parts.

Model-View-Controller

- Problem: every view uses a controller of a different type:
  - `ViewUp` uses the type `ControllerUp`;
  - `ViewReset` uses the type `ControllerReset`.
- So which type do we use if we want to store all the different types of controllers into a single instance variable of the superclass `View`?
- Solution: use an abstract generic `View` superclass where the type of the controller is a type variable!
  - We then restrict the `View`'s type variable for controllers to be a bounded type variable: it can be any type that extends `Controller`!

Model-View-Controller

- Then adding a new graphical interface to our software becomes very simple:
  - Create a new view class that extends `View<...>`
  - Create a new controller class that extends `Controller`.
  - Create objects for the new view and controller.
- Everything else remains the same.

ModelListener

```
public interface ModelListener {
    public void update();
}
```

Model

```
import java.util.ArrayList;
public class Model {
    private int counter;
    private ArrayList<ModelListener> listeners;

    public Model() {
        counter = 0;
        listeners = new ArrayList<ModelListener>();
    }
    public void addListener(ModelListener l) {
        listeners.add(l);
    }
    public int getCounter() {
        return counter;
    }
    public void setCounter(int counter) {
        this.counter = counter;
        notifyListeners(); // Counter changed so notify the listeners.
    }
    private void notifyListeners() {
        for(ModelListener l: listeners) {
            l.update(); // Tell the listener that something changed.
        }
    }
}
```

Views

```
import javax.swing.JFrame;
public abstract class View<T extends Controller> extends JFrame
implements ModelListener {
    protected Model m;
    protected T c;

    public View(Model m, T c) {
        this.m = m;
        this.c = c;
        m.addListener(this);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    @Override
    public abstract void update();
}
```

Views

```
public class ViewUp extends View<ControllerUp> {
    private JLabel label; // No model or controller anymore.
    public ViewUp(Model m, ControllerUp c) {
        super(m, c);
        this.setTitle("View Up");
        this.setSize(150, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new GridLayout(2, 1));
        label = new JLabel();
        update(); // Initialize the label using the model.
        this.add(label);
        JButton buttonUp = new JButton("Up");           // Button
        buttonUp.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                c.up(); // Controller decides what the click means.
            }
        });
        this.add(buttonUp);
        this.setVisible(true);
    }
    @Override
    public void update() {
        label.setText("counter is: " + m.getCounter());
    }
}
```

Views

```
public class ViewReset extends View<ControllerReset> {
    private JLabel label; // No model or controller anymore.
    public ViewReset(Model m, ControllerReset c) {
        super(m, c);
        this.setTitle("View Reset");
        this.setSize(150, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new GridLayout(2, 1));
        label = new JLabel();
        update(); // Initialize the label using the model.
        this.add(label);
        JButton buttonReset = new JButton("Reset");          // Button
        buttonReset.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                c.reset(); // Controller decides what the click means.
            }
        });
        this.add(buttonReset);
        this.setVisible(true);
    }
    @Override
    public void update() {
        label.setText("counter is: " + m.getCounter());
    }
}
```

Controllers

```
public class Controller {
    protected Model m;

    public Controller(Model m) {
        this.m = m;
    }
}
```

Controllers

```
public class ControllerUp extends Controller {
    public ControllerUp(Model m) {
        super(m);
    }
    public void up() {
        m.setCounter(m.getCounter() + 1);
    }
}
```

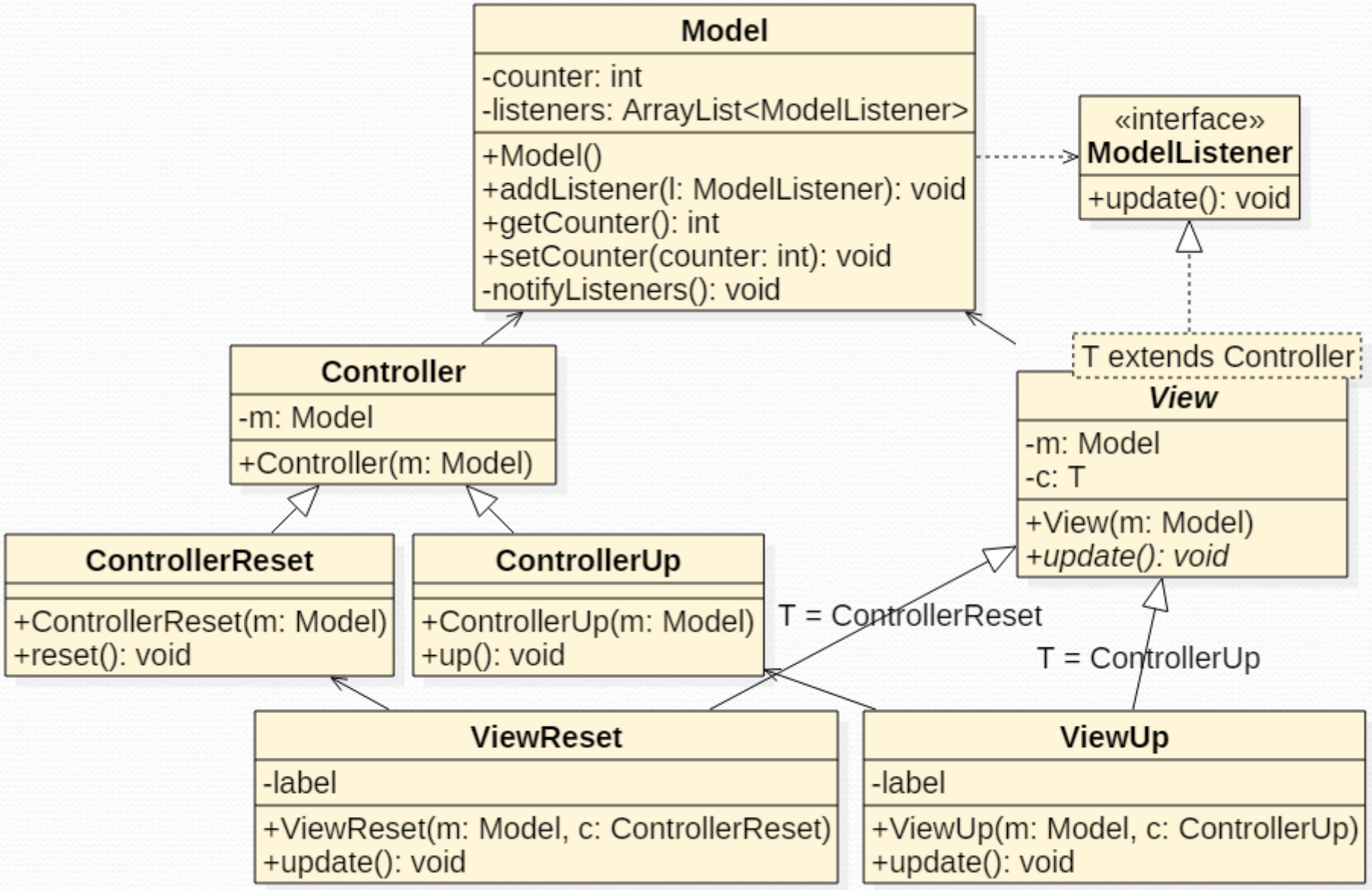
Controllers

```
public class ControllerReset extends Controller {
    public ControllerReset(Model m) {
        super(m);
    }
    public void reset() {
        m.setCounter(0);
    }
}
```

Test

```
public class Test5 {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                Model m = new Model(); // Single shared model.

                ControllerUp c1 = new ControllerUp(m);
                ViewUp v1 = new ViewUp(m , c1);

                ControllerUp c2 = new ControllerUp(m);
                ViewUp v2 = new ViewUp(m , c2);

                ControllerReset c3 = new ControllerReset(m);
                ViewReset v3 = new ViewReset(m, c3);
            }
        });
    }
}
```

Model-View-Controller



Summary

- Problems with GUI programming.
- Design patterns.
- The Model-View-Controller design pattern.
- The Observer design pattern.