50002 - Software Engineering Design - Lecture 11

Oliver Killane

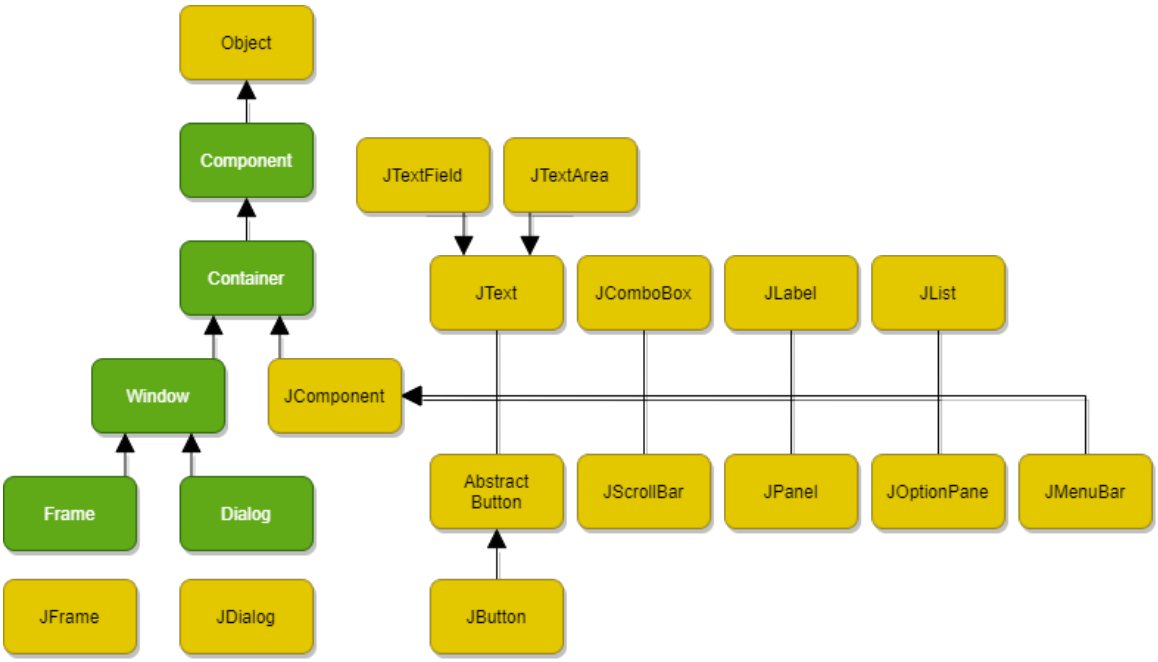
14/11/21

Lecture Recording

Lecture recording is available here

Java Swing

An older object oriented GUI, designed to be cross-platform.




```java
import javax.swing.*;

// For layouts
import java.awt.*;

public class swingApp {
    private static final int HEIGHT = 400, WIDTH = 300;
    public static void main(String[] args) {
        new swingApp().display();
    }

    private void display() {
        // Code to display components goes here

        // Create a new window as a JFrame
        JFrame frame = new JFrame("window title here");

        // Basic window setup
        frame.setSize(WIDTH, HEIGHT);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

```

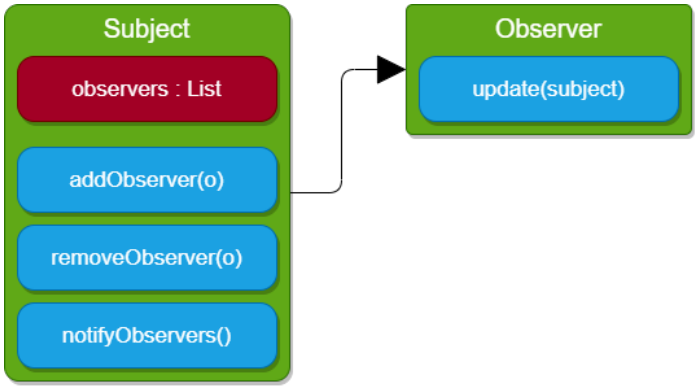
```
23          // Create Components
24          JLabel label = new JLabel("here is the label text");
25          JTextArea text = new JTextArea("This is a really large text box, it can even
                ↪ have scroll bars");
26
27          String words[] = {"this", "word", "that", "word"};
28          JList list = new JList<String>(words);
29
30          // Create Layouts (FlowLayout is the default used by JPanels)
31          FlowLayout sideBySide = new FlowLayout();
32
33          // set the layout of the frame
34          frame.setLayout(sideBySide);
35
36          // add components (layout manages how they appear in the frame)
37          frame.add(label);
38          frame.add(text);
39          frame.add(list);
40      }
41  }
```

Layout Managers

Layout managers inform a **JFrame** or **JPanel** how to position added components.

Oracle has a guide on this, as does tutorialspoint.

Observer Pattern



For any object we want to observer, we register one or more observers. When an event occurs, the subject informs all observers of what has occurred so that they can react.

In Java we can add **ActionListener** implementing classes to components, this interface defines a single method **actionPerformed**(**ActionEvent e**).

```
1  // sometimes the current class (this)
2  component.addActionListener(actionListeningObj);
```

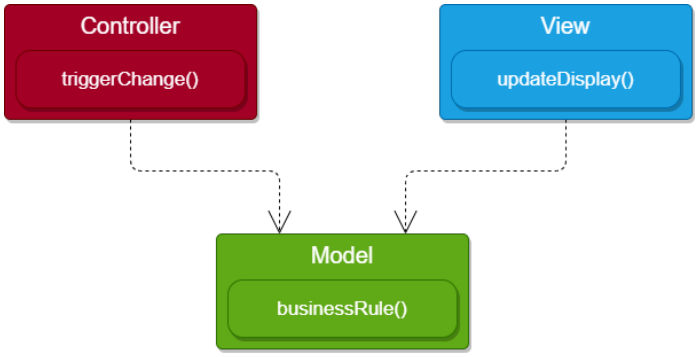
```
component.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // do something
    }
});
```

For example the following program creates a window with a text field, that can be cleared by a button.

```
import javax.swing.*;

// For layouts
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class buttonSwing {
    private static final int HEIGHT = 400, WIDTH = 300;

    public static void main(String[] args) {
        new buttonSwing().display();
    }

    private void display() {
        // Code to display components goes here

        // Create a new window as a JFrame
        JFrame frame = new JFrame("window title here");

        // add button
        JTextField text = new JTextField("Text field");

        JButton add = new JButton("Clear");

        add.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                text.setText("");
            }
        });

        // Create Layouts (FlowLayout is the default used by JPanels)
        BorderLayout layout = new BorderLayout();

        // set the layout of the frame
        frame.setLayout(layout);

        frame.add(text, BorderLayout.CENTER);
        frame.add(add, BorderLayout.PAGE_END);

        // Basic window setup
        frame.setSize(WIDTH, HEIGHT);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    }
}
```

Model View Controller

A common patter for structuring user interface code.



- **Some Event** Controller triggered, calls relevant functions to alter the model, pass information.
- **Model Called By Controller** Updates its internal information, responds to action passed by controller
- **View** Called either by the model of some other (e.g refresh), draws display based on information from model

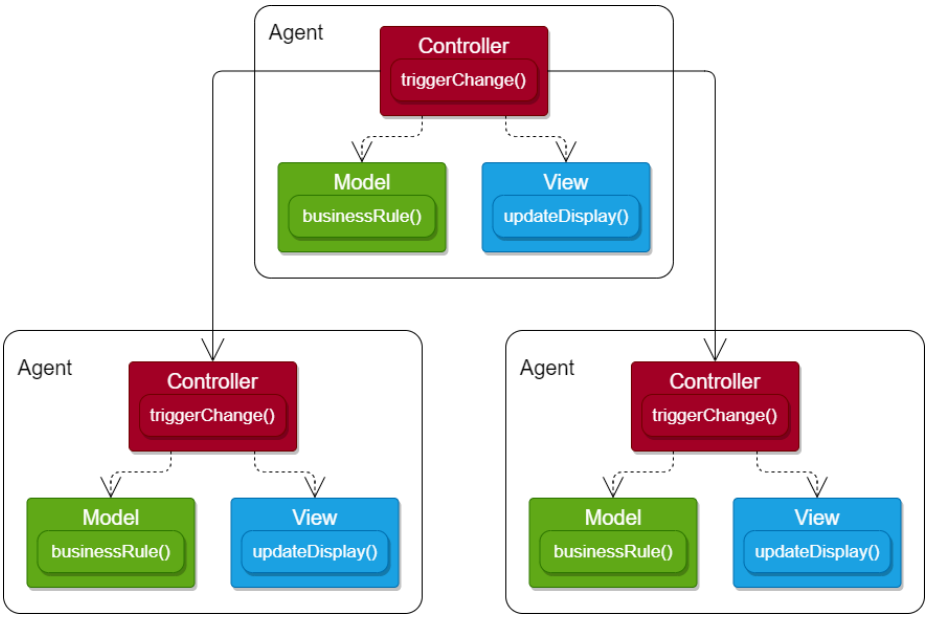
The advanatges of this pattern are:

- Easier to work in teams (can split up development).
- Can have multiple Views, as they interact with model.
- Testing of model is easier (can be isolated for testing).

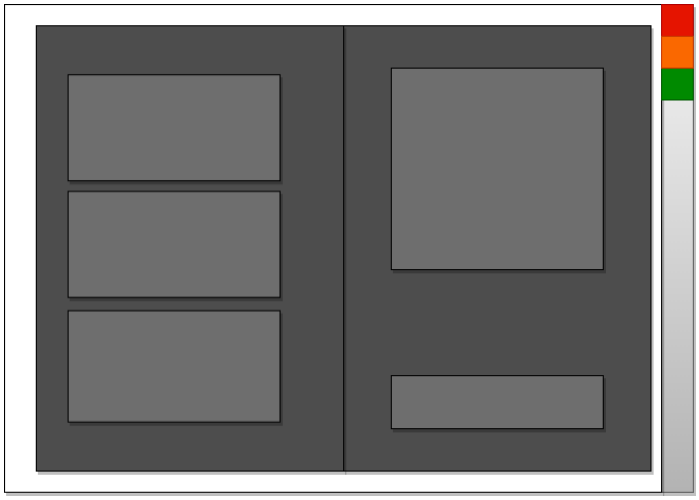
Ideally the main function creates, and connects the model, view and controllers.

Presentation Abstraction Control

PAC is typically most useful in a hierarchical/nested structure.



We can therefore allow nested components to manage themselves, having many nested components instead of very large classes. Sometimes propagating received events up the hierarchy.

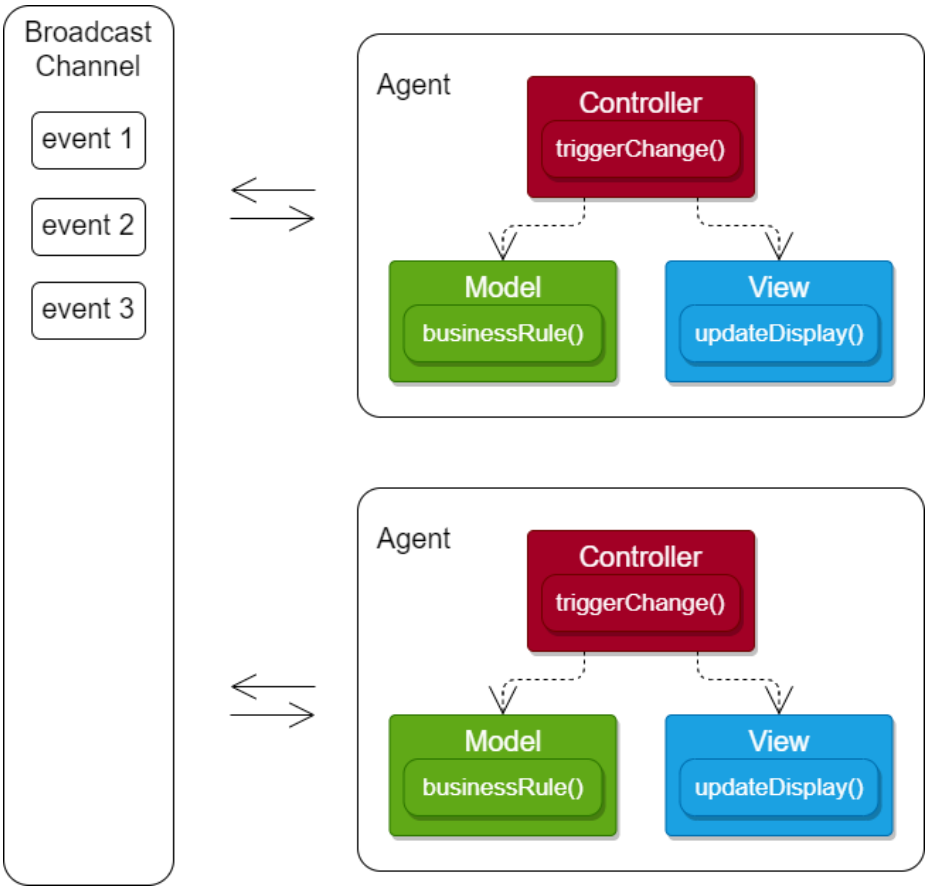


This allows for further benefit in splitting the code and testing.

However we can run into issues when communicating between objects (e.g child to an uncle), as this requires a tree traversal. Refactoring this code can be difficult.

Event Bus

We can create broadcast channels, with agents able to view events from, and send events to a given channel.


Broadcast Channel
event 1
event 2
event 3
Agent
Controller
triggerChange()
Model
businessRule()
View
updateDisplay()
Agent
Controller
triggerChange()
Model
businessRule()
View
updateDisplay()