

# Principles of Software Construction: Objects, Design, and Concurrency

Design case study: Java Swing

Josh Bloch

**Charlie Garrod**



# Administrivia

- Reading due today: UML and Patterns 26.1 and 26.4
- Homework 4b due Thursday, March 25<sup>th</sup>

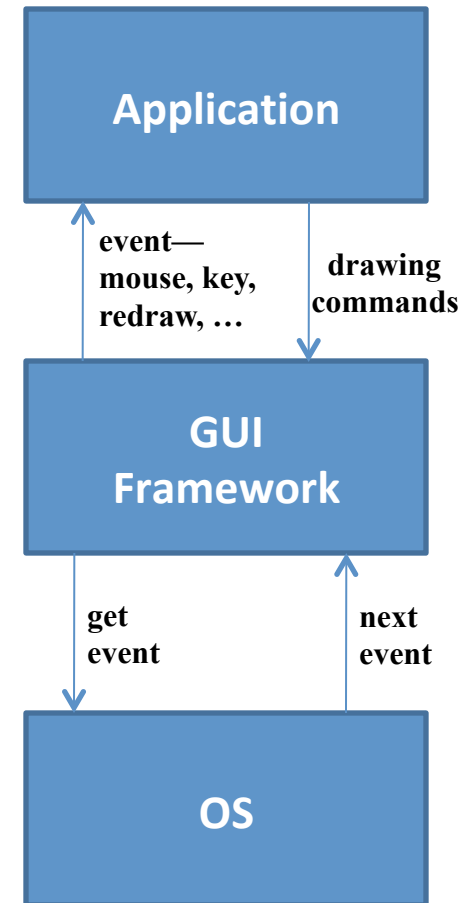


# Key concepts from Thursday

- Observer design pattern
- Introduction to concurrency
  - Not enough synchronization: safety failure
  - Too much synchronization: liveness failure
- Event-based programming
- Introduction to GUIs

# An event-based GUI with a GUI framework

- Setup phase
  - Describe how the GUI window should look
  - Register observers to handle events
- Execution
  - Framework gets events from OS, processes events
    - Your code is mostly just event handlers



See [edu.cmu.cs.cs214.rec06.alarmclock.AlarmWindow...](#)

# GUI programming is inherently multi-threaded

- *Swing Event Dispatch Thread* (EDT) handles all GUI events
  - Mouse events, keyboard events, timer events, etc.
- No other time-consuming activity allowed on the EDT
  - Violating this rule can cause liveness failures

# Today

- Finish introduction to GUIs
- Design case study: GUI potpourri
  - Strategy
  - Template method
  - Observer
  - Composite
  - Decorator
  - Adapter
  - Façade
  - Command
  - Chain of responsibility
- Design discussion: Decoupling your game from your GUI

# Components of a Swing application

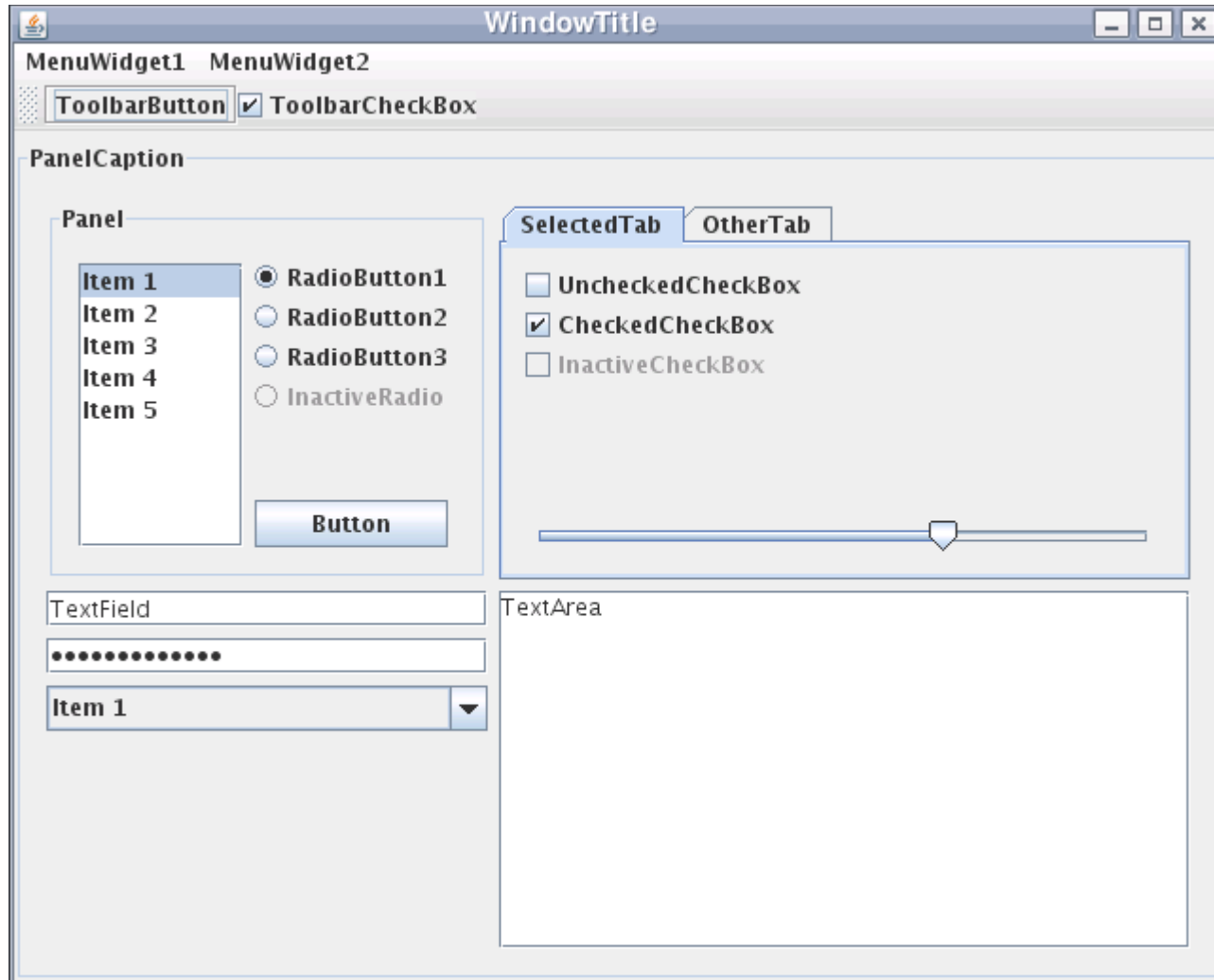
JFrame

JPanel

JButton

JTextField

...



# Swing has many *widgets*

- JLabel
  - JButton
  - JCheckBox
  - JChoice
  - JRadioButton
  - JTextField
  - JTextArea
  - JList
  - JScrollbar
  - ... and more
- 
- JFrame is the Swing Window
  - JPanel (a.k.a. a pane) is the container to which you add your components (or other containers)



# To create a simple Swing application

- Make a window (a `JFrame`)
- Make a container (a `JPanel`)
  - Put it in the window
- Add components (buttons, boxes, etc.) to the container
  - Use layouts to control positioning
  - Set up observers (a.k.a. listeners) to respond to events
  - Optionally, write custom widgets with application-specific display logic
- Set up the window to display the container
  
- Then wait for events to arrive...

## E.g., creating a button

```
JFrame window = ...

JPanel panel = new JPanel();
window.setContentPane(panel);

JButton button = new JButton("Click me");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
    }
});
panel.add(button);

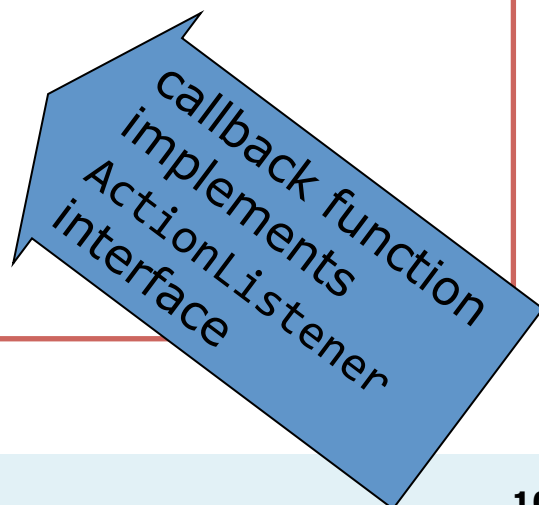
window.setVisible(true);
```



panel to hold  
the button



register callback  
function



callback function  
implements  
ActionListener  
interface

## E.g., creating a button

```
JFrame window = ...
```

```
JPanel panel = new JPanel();  
window.setContentPane(panel);
```

```
JButton button = new JButton("Click me");  
button.addActionListener(  
    (e) -> System.out.println("Button clicked") );  
panel.add(button);
```

```
window.setVisible(true);
```

panel to hold  
the button

callback function  
implements  
ActionListener  
interface

register callback  
function

# The javax.swing.ActionListener

- Listeners are objects with callback functions
  - Can be registered to handle events on widgets
  - All registered widgets are called if event occurs

```
interface ActionListener {  
    void actionPerformed(ActionEvent e);  
}
```

```
class ActionEvent {  
    int when;  
    String actionCommand;  
    int modifiers;  
    Object source();  
    int id;  
    ...  
}
```

# Button design discussion

- Button implementation should be reusable but customizable
  - Different button label, different event-handling
- Must decouple button's action from the button itself
- Listeners are separate independent objects
  - A single button can have multiple listeners
  - Multiple buttons can share the same listener

# Swing has many event listener interfaces

- ActionListener
- AdjustmentListener
- FocusListener
- ItemListener
- KeyListener
- MouseListener
- TreeExpansionListener
- TextListener
- WindowListener
- ...

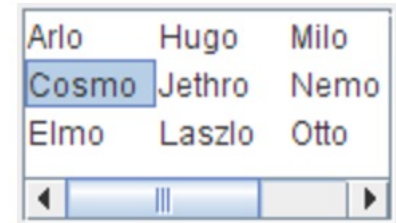
```
class ActionEvent {  
    int when;  
    String actionCommand;  
    int modifiers;  
    Object source();  
    int id;
```

```
interface ActionListener {  
    void actionPerformed(ActionEvent e);  
}
```

# Today

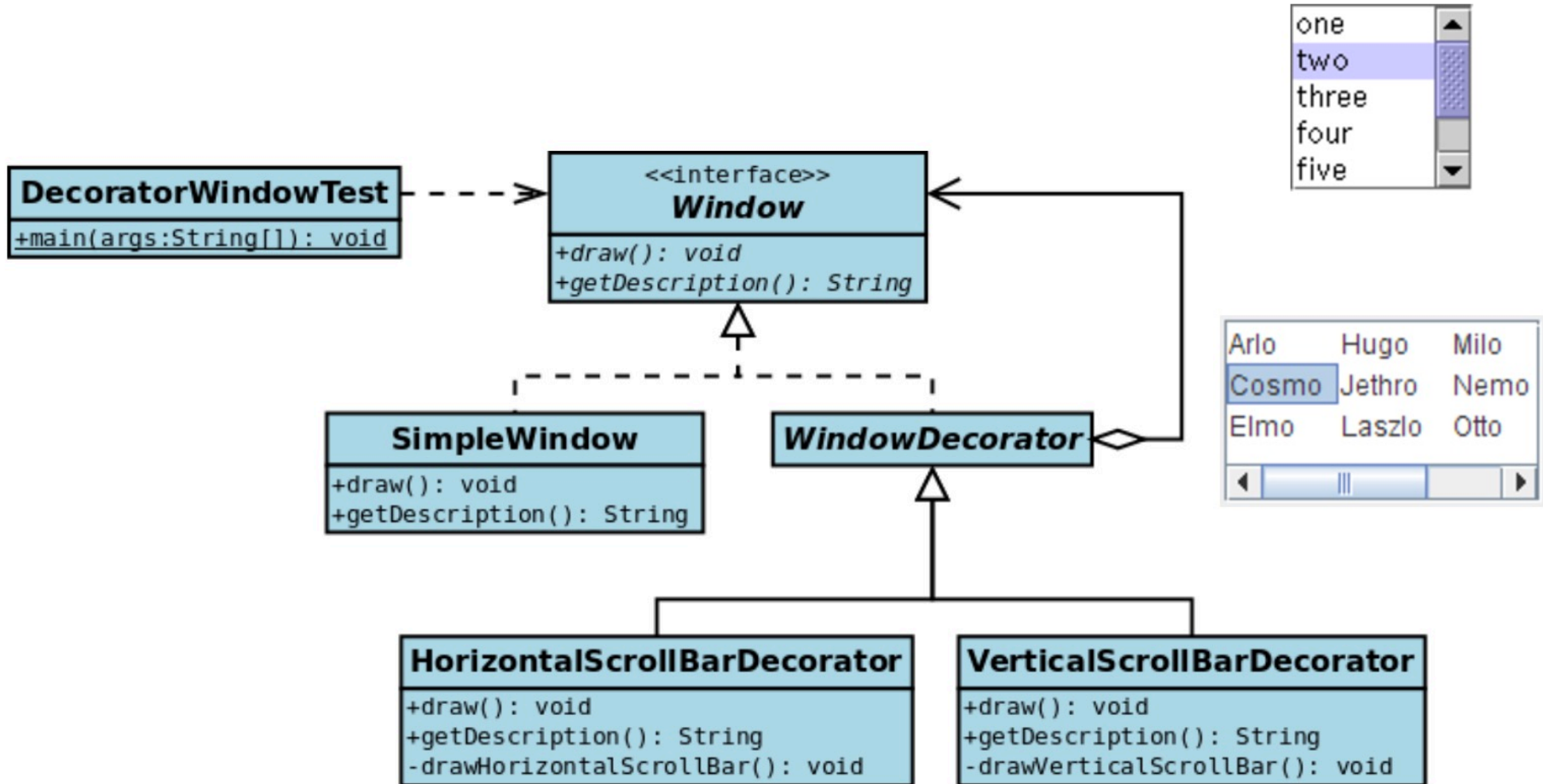
- Finish introduction to GUIs
- Design case study: GUI potpourri
  - Strategy
  - Template method
  - Observer
  - Composite
  - Decorator
  - Adapter
  - Façade
  - Command
  - Chain of responsibility
- Design discussion: Decoupling your game from your GUI

# The decorator pattern abounds



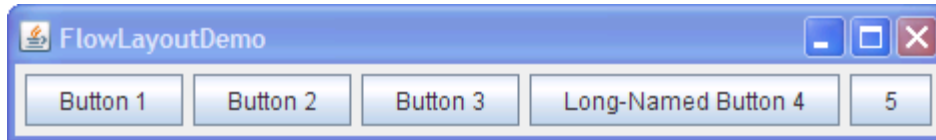


# The decorator pattern abounds

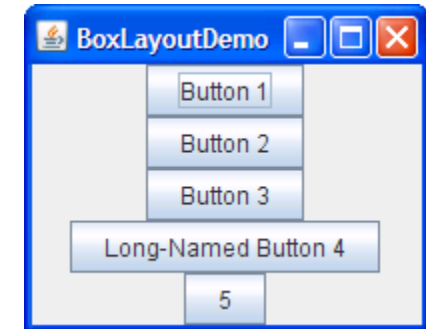


UML from <https://medium.com/@dholnessii/structural-design-patterns-decorator-30f5a8c106a5>

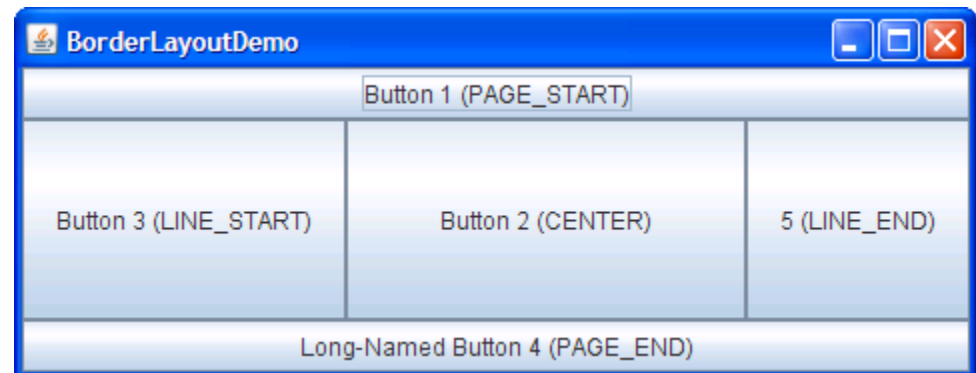
# Swing layouts



The simplest, and default, layout.  
Wraps around when out of space.



Like FlowLayout, but no wrapping



More sophisticated layout managers

see <http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>

# A naïve hard-coded implementation

```
class JPanel {
    protected void doLayout() {
        switch(getLayoutType()) {
            case BOX_LAYOUT: adjustSizeBox(); break;
            case BORDER_LAYOUT: adjustSizeBorder(); break;
            ...
        }
    }
    private adjustSizeBox() { ... }
}
```

- A new layout would require changing or overriding JPanel

# A better solution: delegate the layout responsibilities

- Layout classes, e.g.:

```
contentPane.setLayout(new FlowLayout());
```

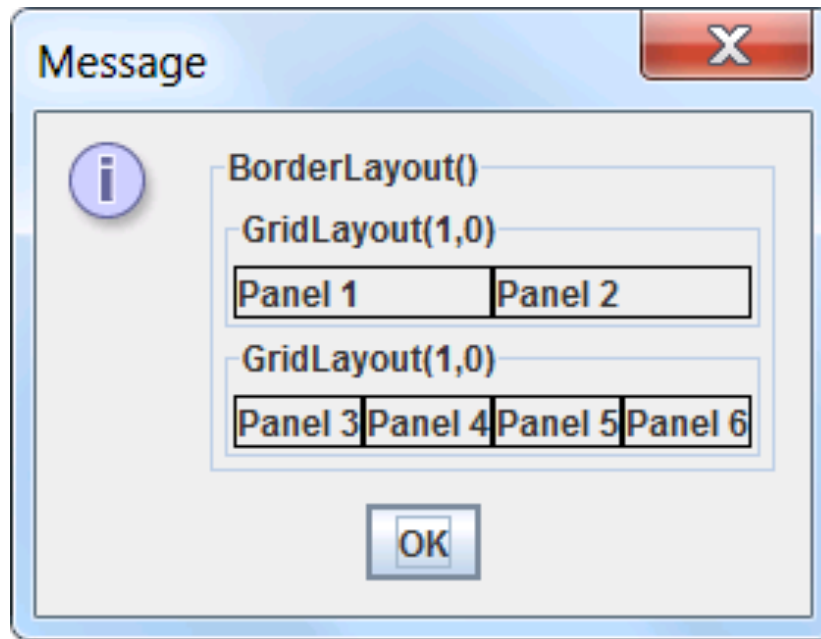
```
contentPane.setLayout(new GridLayout(4,2));
```

- Similarly, there are border classes to draw the borders, e.g.:

```
contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
```

# Another GUI design challenge: nesting containers

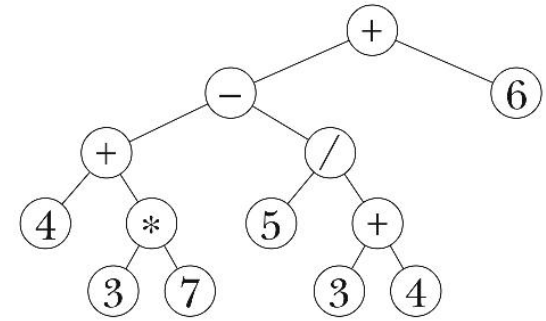
- A JFrame contains a JPanel, which contains a JPanel (and/or other widgets), which contains a JPanel (and/or other widgets), which contains...



# The composite pattern

- Problem: Collection of objects has behavior similar to the individual objects
- Solution: Have collection of objects and individual objects implement the same interface
- Consequences:
  - Client code can treat collection as if it were an individual object
  - Easier to add new object types
  - Design might become too general, interface insufficiently useful

# Another composite pattern example



```
public interface Expression {
    double eval();    // Returns value
}
```

```
public class BinaryOperationExpression implements Expression {
    public BinaryOperationExpression(BinaryOperator operator,
        Expression operand1, Expression operand2);
}
```

```
public class NumberExpression implements Expression {
    public NumberExpression(double number);
}
```

# Recall: Creating a button

```
JFrame window = ...  
  
JPanel panel = new JPanel();  
window.setContentPane(panel);  
  
JButton button = new JButton("Click me");  
button.addActionListener( (e) -> {  
    System.out.println("Button clicked");  
});  
panel.add(button);  
  
window.setVisible(true);
```

callback function  
implements  
ActionListener  
interface

register callback  
function



# An alternative button

```
class MyButton extends JButton {
    public MyButton() { super("Click me"); }
    @Override
    protected void fireActionPerformed(ActionEvent e) {
        super.fireActionPerformed(e);
        System.out.println("Button clicked");
    }
}

//static public void main...
JFrame window = ...
JPanel panel = new JPanel();
window.setContentPane(panel);
panel.add(new MyButton());
window.setVisible(true);
```

# Discussion: Command vs. template method patterns

```
JFrame window = ...
```

```
JPanel panel = new JPanel();  
window.setContentPane(panel);
```

```
JButton button = new JButton("Click me");  
button.addActionListener( (e) -> {  
    System.out.println("Button clicked");
```

```
});
```

```
panel.a
```

```
window.
```

```
class MyButton extends JButton {  
    public MyButton() { super("Click me"); }  
    @Override  
    protected void fireActionPerformed(ActionEvent e) {  
        super.fireActionPerformed(e);  
        System.out.println("Button clicked");  
    }  
} ...
```

# Better use of template method: partial customization

## JComponent:

### paint

```
public void paint(Graphics g)
```

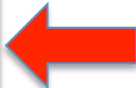
Invoked by Swing to draw components. Applications should not invoke `paint` directly, but should instead use the `repaint` method to schedule the component for redrawing.

This method actually delegates the work of painting to three protected methods: `paintComponent`, `paintBorder`, and `paintChildren`. They're called in the order listed to ensure that children appear on top of component itself. Generally speaking, the component and its children should not paint in the insets area allocated to the border. Subclasses can just override this method, as always. A subclass that just wants to specialize the UI (look and feel) delegate's `paint` method should just override `paintComponent`.

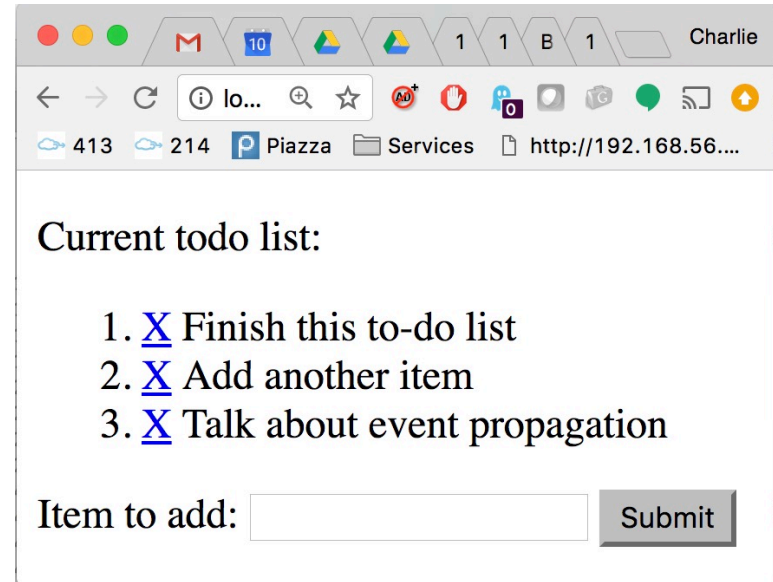
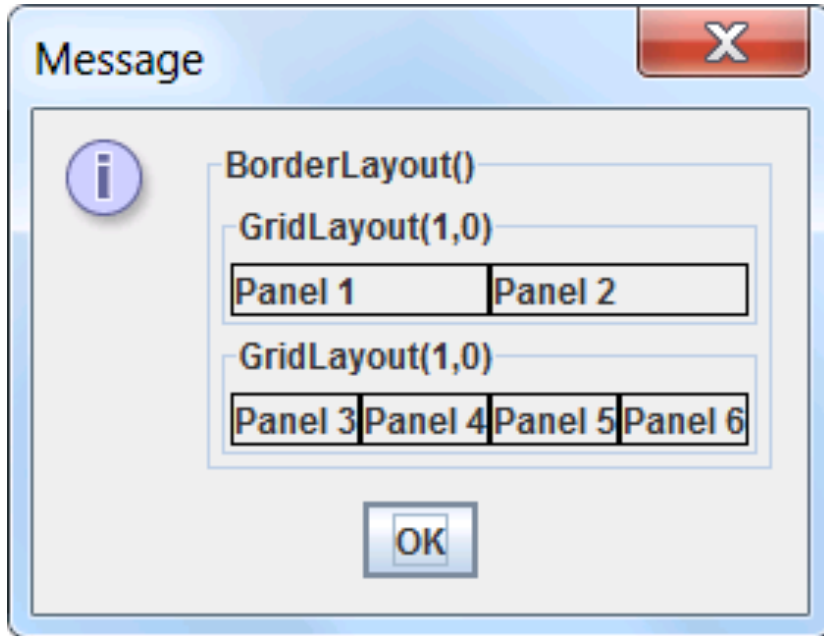
### Overrides:

```
paint in class Container
```

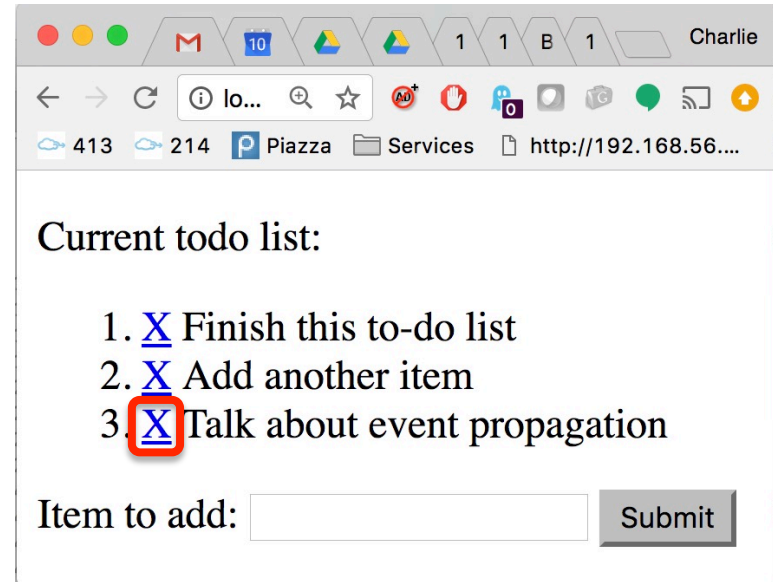
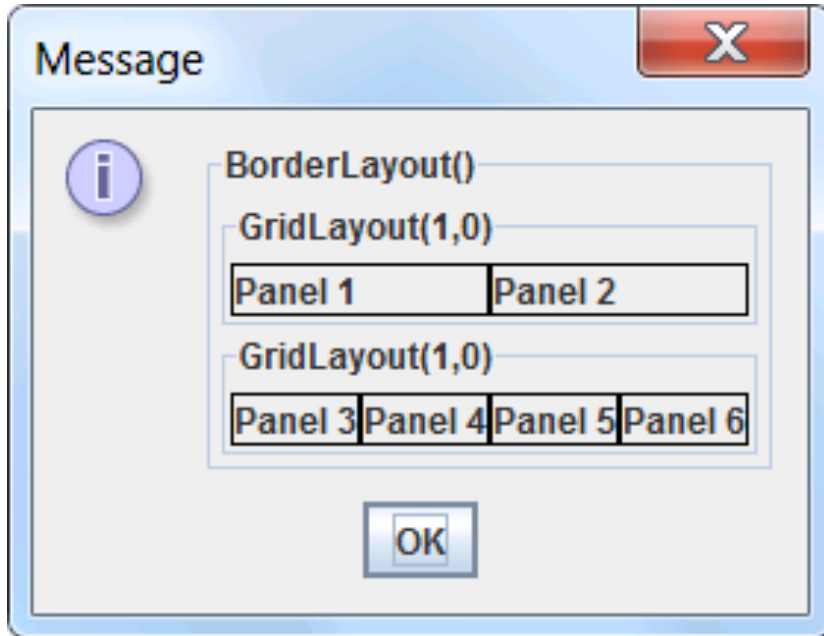
### Parameters:



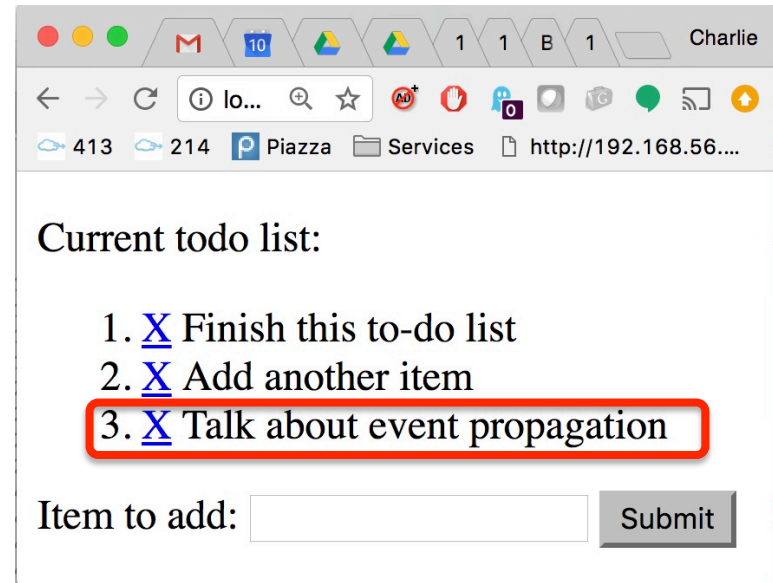
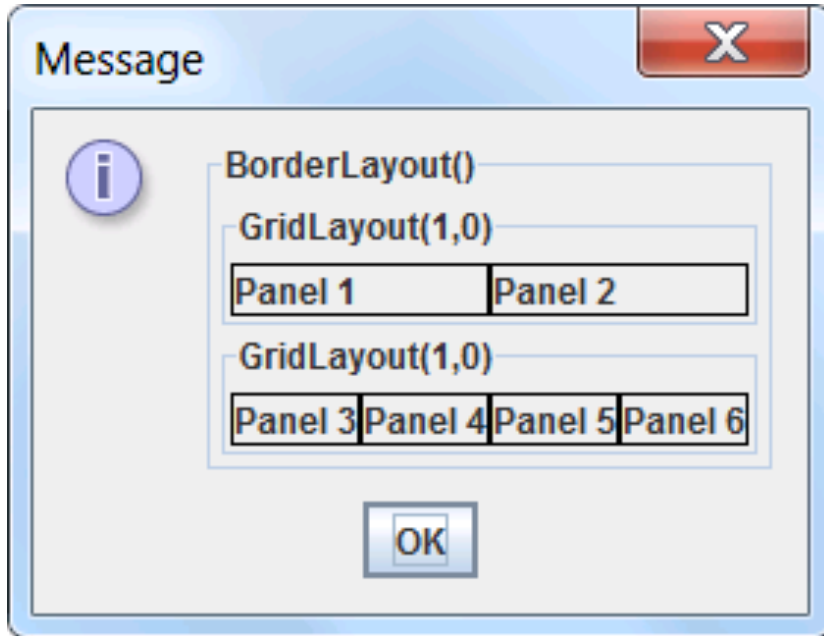
# Event propagation and deep container hierarchies



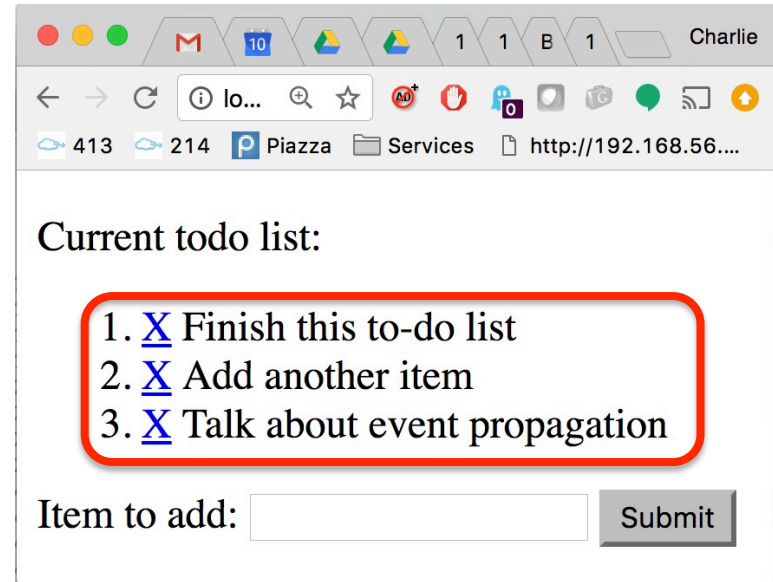
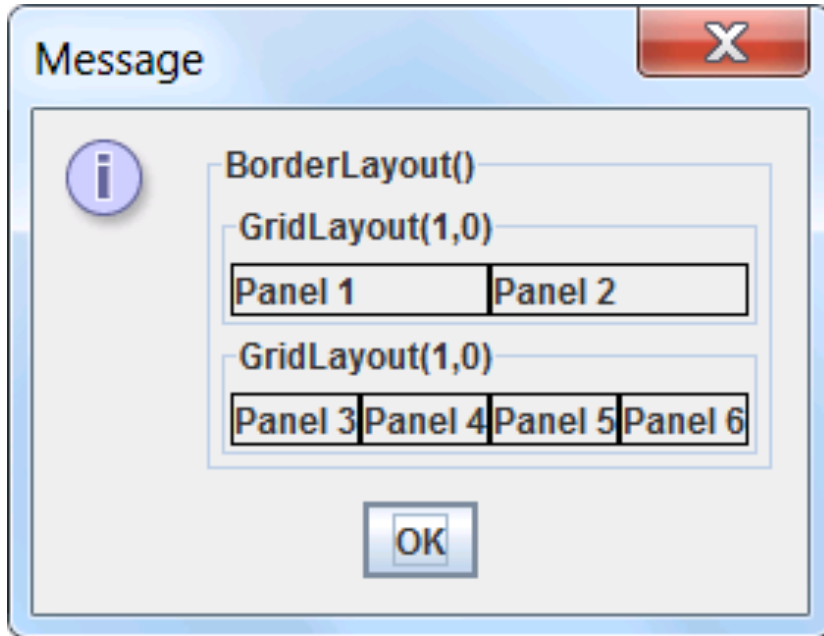
# Event propagation and deep container hierarchies



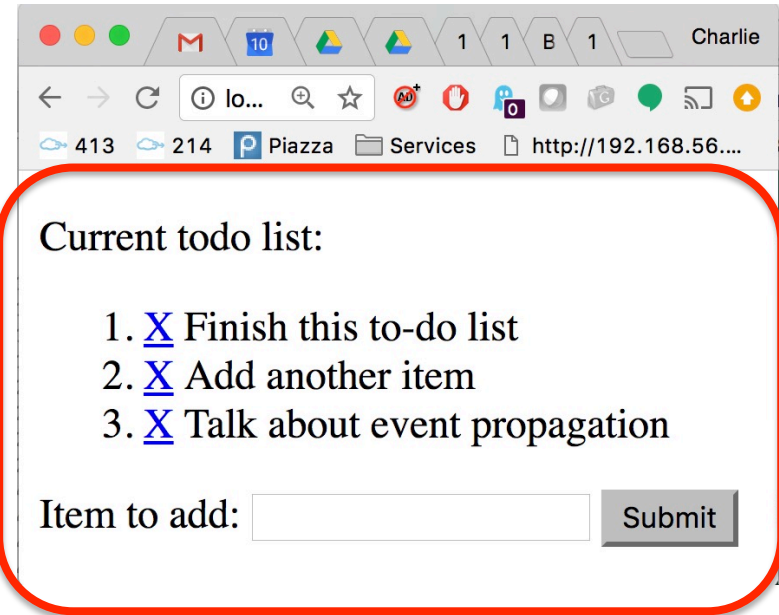
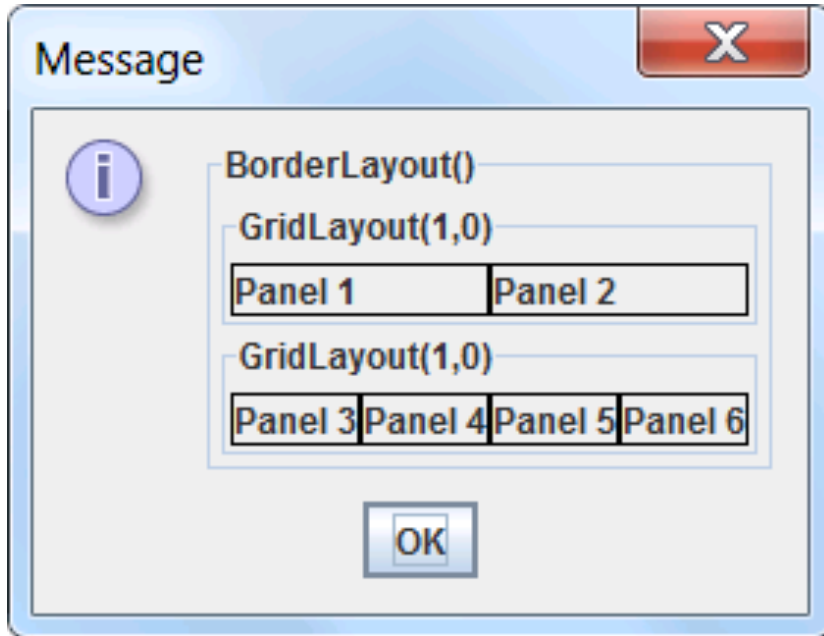
# Event propagation and deep container hierarchies



# Event propagation and deep container hierarchies



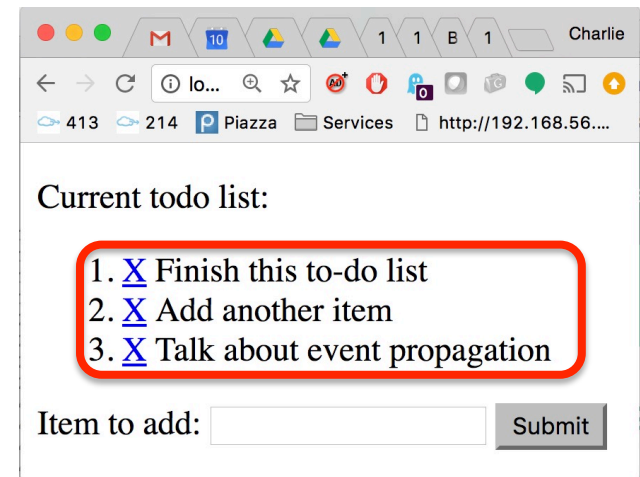
# Event propagation and deep container hierarchies





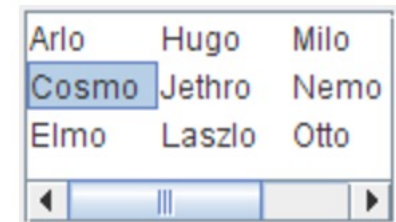
# The chain of responsibility pattern

- Problem: You need to associate functionality within a deep nested or iterative structure, possibly with multiple objects
- Solution: Request for functionality, pass request along chain until some component handles it
- Consequences:
  - Decouples sender from receiver of request
  - Can simplify request-handling by handling requests near root of hierarchy
  - Handling of request not guaranteed

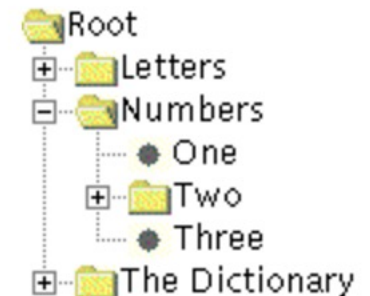


# The design of JList and JTree

- Highly flexible rendering of lists and trees
  - Can change rendering of cells
  - Can change source of data to display



```
// example of simple use
String [] items = { "a", "b", "c" };
JList<String> list = new JList<>(items);
```



# Using JLists with a ListModel

- Allows a list widget (the view) to react to changes in the model

```
// with a ListModel
ListModel<String> model = new DefaultListModel<>();
model.addElement("a");
JList<String> list = new JList<>(model);
```

```
interface ListModel<T> {
    int getSize();
    T getElementAt(int index);
    void addListDataListener(ListDataListener l);
    void removeListDataListener(ListDataListener l);
}
```

# Using JLists with a ListModel

- Allows a list widget (the view) to react to changes in the model

```
// with a ListModel
ListModel<String> model = new DefaultListModel<>();
model.addElement("a");
JList<String> list = new JList<>(model);
```

```
interface ListModel<T> {
    int getSize();
}

interface ListDataListener extends EventListener {
    void intervalAdded(...);
    void intervalRemoved(...);
    void contentsChanged(...);
}
```

# Attaching a data source to a JList

- Assume we have an anagram generator, and we want to update a JList with new anagrams as they are generated

```
// design 1
class AnagramGen implements ListModel<String> {
    List<String> items ...

    int getSize() { return items.size(); }
    String getElementAt(int index) {
        items.get(index).toString();
    }
    void addListDataListener(ListDataListener l) {...}
    ...
}
```

# Attaching a data source to a JList

- Assume we have an anagram generator, and we want to update a JList with new anagrams as they are generated

```
// design 2
class AnagramGen {
    DefaultListModel<String> items ...

    public ListModel<String> getListModel() {
        return items;
    }
    public Iterable<String> getItems() {
        return items.elements();
    }
    ...
}
```

# Attaching a data source to a JList

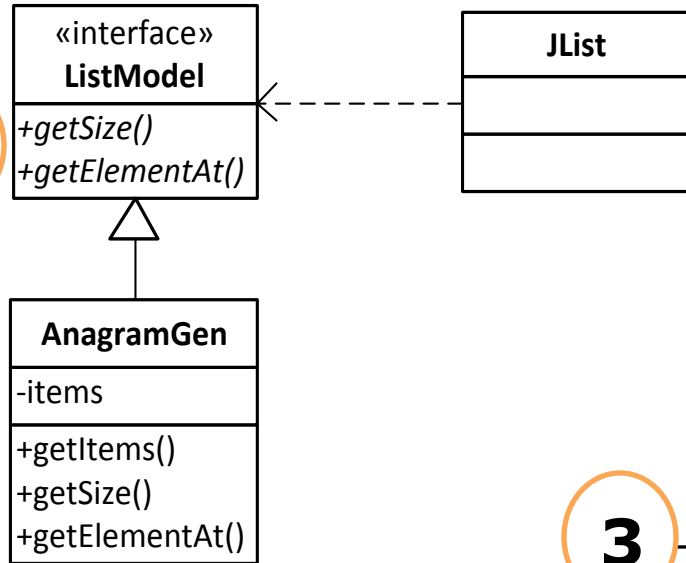
- Assume we have an anagram generator, and we want to update a JList with new anagrams as they are generated

```
// design 3
class AnagramAdapter implements ListModel<String> {
    private final AnagramGen an;
    public AnagramAdapter(AnagramGen s) {an = s;}

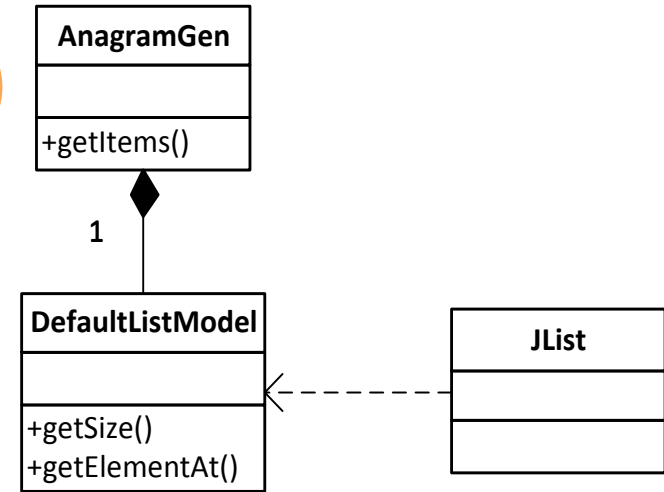
    int getSize() { return count(an.getWords()); }
    String getElementAt(int index) {
        find(an.getWords(), index).toString();
    }
    void addListDataListener(ListDataListener l) {...}
    ...
}
```

# Comparing the three proposed designs

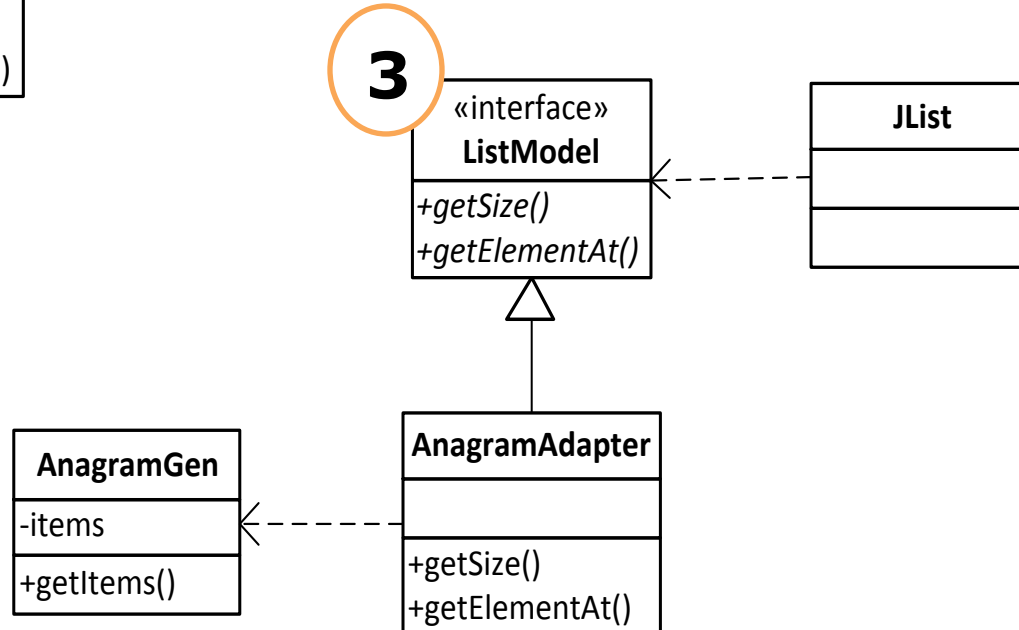
1



2



3





# The adapter pattern

- Problem: You have a client that expects one API for a service provider, and a service provider with a different API
- Solution: Write a class that implements the expected API, converting calls to the service provider's actual API
- Consequences:
  - Easy interoperability of unrelated clients and libraries
    - Client can use unforeseen future libraries
  - Adapter class is coupled to concrete service provider, can make it harder to override service provider behavior

# The adapter pattern, illustrated

Have this



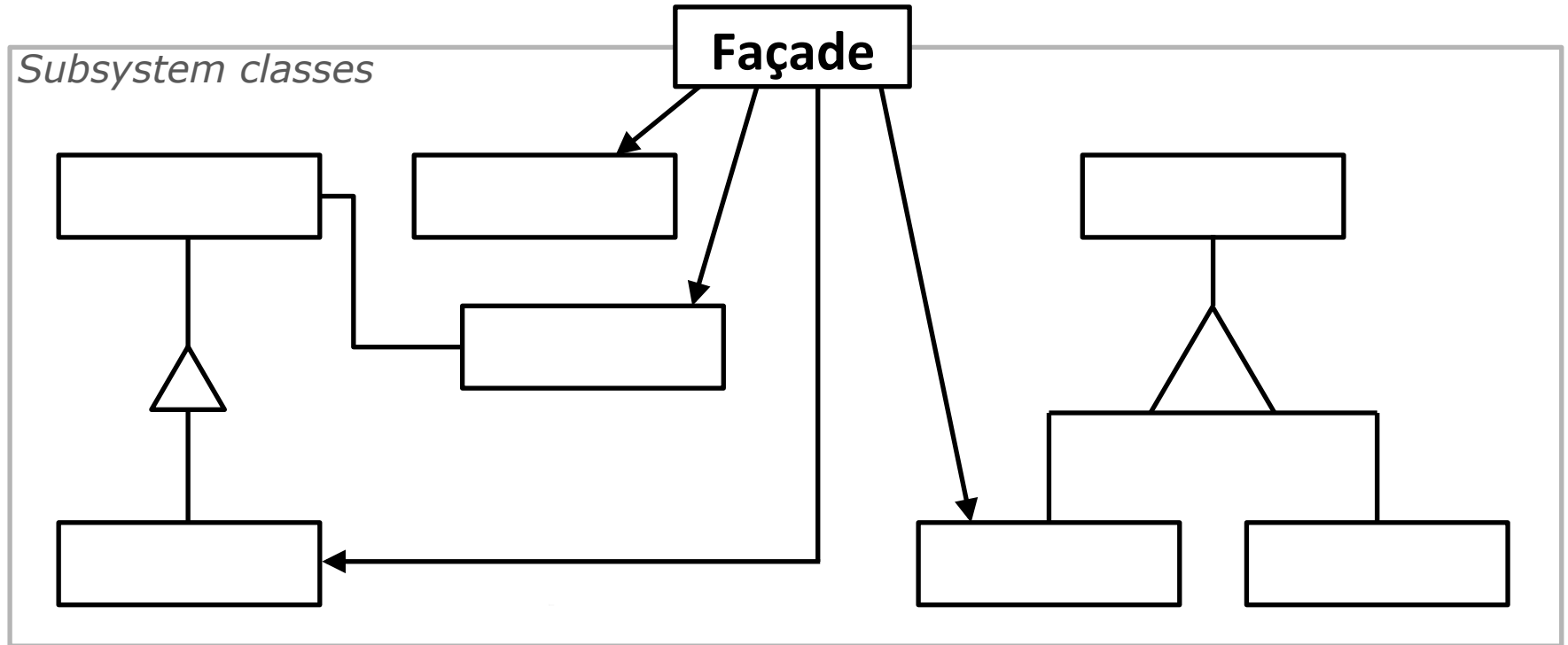
and this?



Use this!



# Aside: The façade pattern



# The façade vs. adapter patterns

- Motivation:
  - Façade: Provide simple interface for a complex API
    - Façade interface is typically *new*
  - Adapter: Match interface expected by an existing client to existing API
    - Adapter interface is defined by the existing client's expectations

# Today

- Finish introduction to GUIs
- Design case study: GUI potpourri
  - Strategy
  - Template method
  - Observer
  - Composite
  - Decorator
  - Adapter
  - Façade
  - Command
  - Chain of responsibility
- Design discussion: Decoupling your game from your GUI

# Design discussion: Decoupling your game from your GUI

# Summary

- GUI programming is inherently multi-threaded
  - Swing calls must be made on the event dispatch thread
  - No other significant work should be done on the EDT
- GUIs are filled with design patterns

Paper slides from lecture are scanned below..



