Principles of Software Construction: Objects, Design, and Concurrency (Part 3: Design Case Studies)

Design Case Study: GUI with Swing (2)

Jonathan Aldrich Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH
15-214

DESIGN OF LIST AND TABLE ENTRIES

JList and JTree

- Lists and trees highly flexible (reusable)
- Can change rendering of cells
- Can change source of data to display

```
// simple use
String [] items = { “a”, “b”, “c” };
JList list = new JList(items);
```

The ListModel

Which design pattern was used here?

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

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

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

The ListModel

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

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

```
interface ListModel<T> {
        int getSize();
        interface ListDataListener extends EventListener {
                void intervalAdded(…);
                void intervalRemoved(…);
                void contentsChanged(…);
        }
}
```

Scenario

- Assume we want to show all buses of the simulation in a list and update the items

```
// design 1
class Simulation implements ListModel<String> {
        List<Bus> items …

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

Scenario

- Assume we want to show all buses of the simulation in a list and update the items

```
// design 2
class World {
        DefaultListModel<Item> items …

        public getListModel() { return items; }
        public Iterable<Item> getItems() {
                return items.elements();

}
```

Scenario

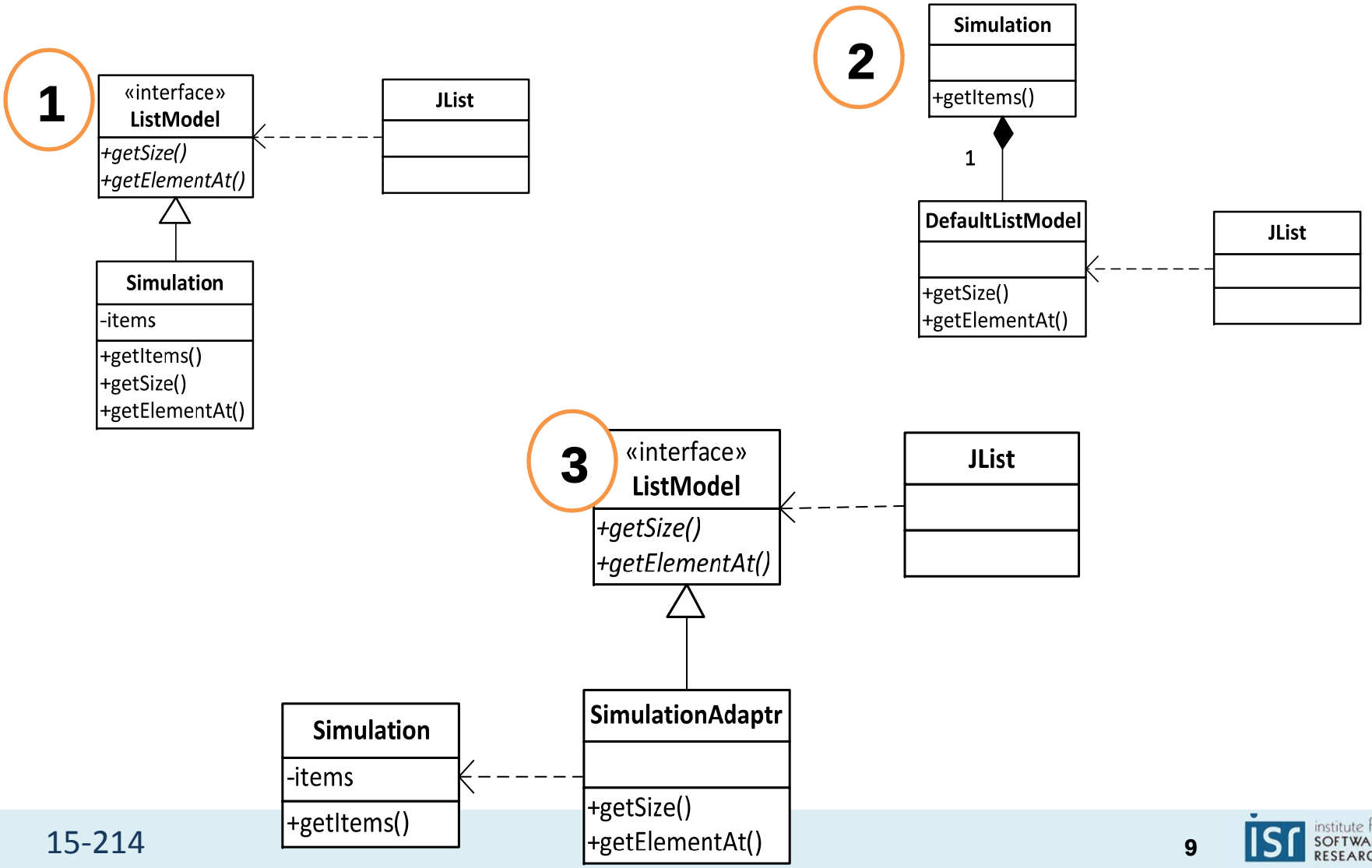
Which design pattern was used here?

- Assume we want to show all buses of the simulation in a list and update the items

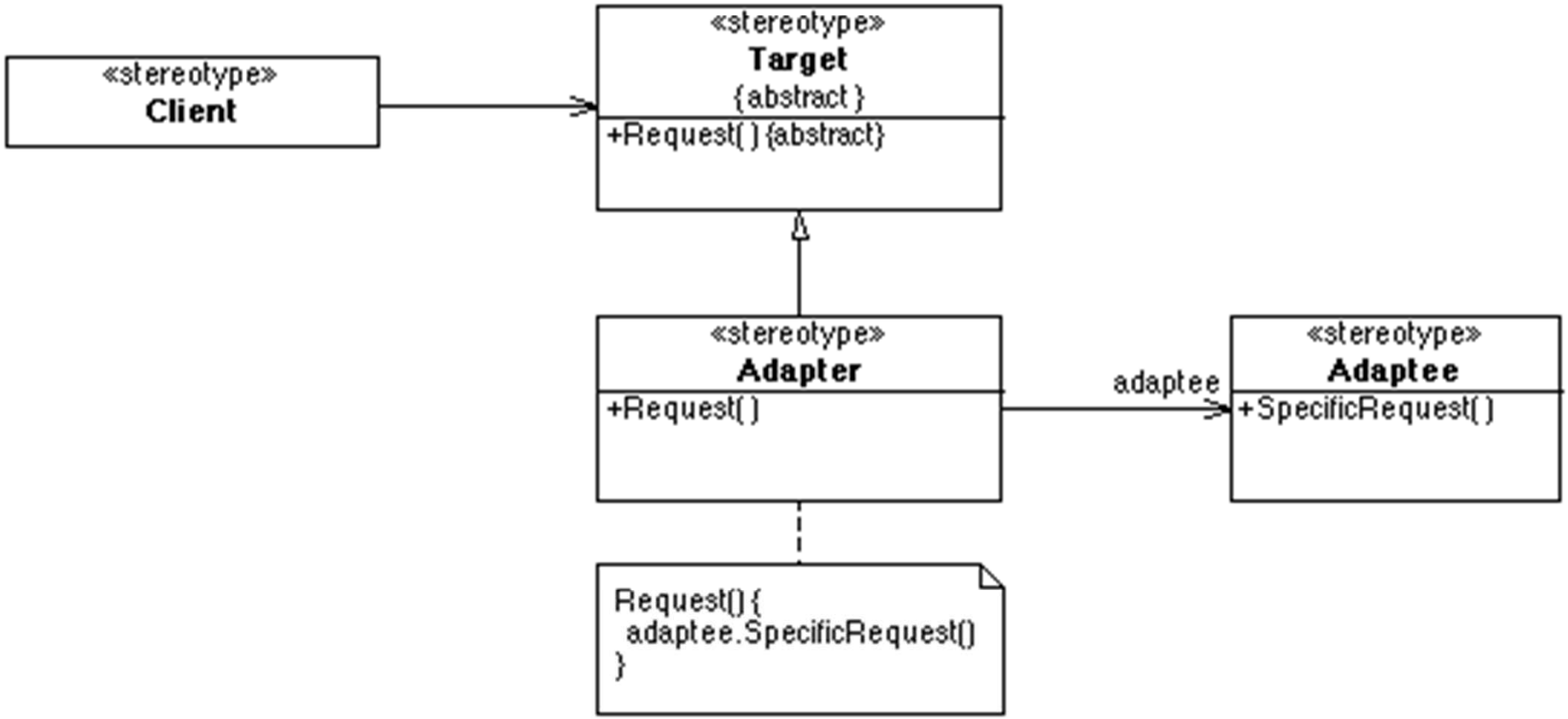
```
// design 3
class SimulationAdapter implements ListModel<String> {
        private final Simulation simulation;
        public SimulationAdapter(Simulation s) {simulation = s;}

        int getSize() { return count(simulation.getBuses()); }
        String getElementAt(int index) {
                find(simulation.getBuses(), index).getRoute();
        }
        void addListDataListener(ListDataListener l) {…}
        …
}
```

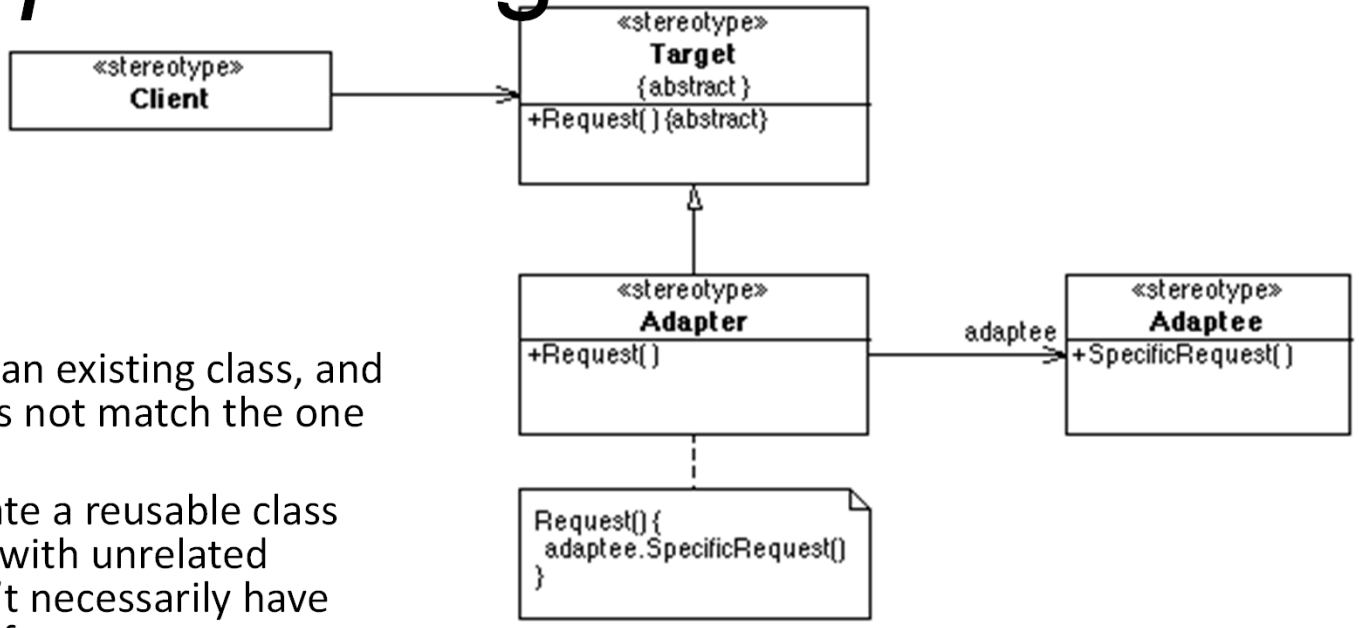
Comparing the three solutions



The *Adapter* Design Pattern



The *Adapter* Design Pattern



- Applicability
  - You want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated classes that don't necessarily have compatible interfaces
  - You need to use several subclasses, but it's impractical to adapt their interface by subclassing each one

•Consequences
- Exposes the functionality of an object in another form
- Unifies the interfaces of multiple incompatible adaptee objects
- Lets a single adapter work with multiple adaptees in a hierarchy
- -> **Low coupling, high cohesion**

Other Scenarios for Adapters

- You have an application that processes data with an Iterator. Methods are:
  - **boolean** hasNext();
  - Object next();
- You need to read that data from a database using JDBC. Methods are:
  - **boolean** next();
  - Object getObject(**int** column);
- You might have to get the information from other sources in the future.

Façade vs. Adapter

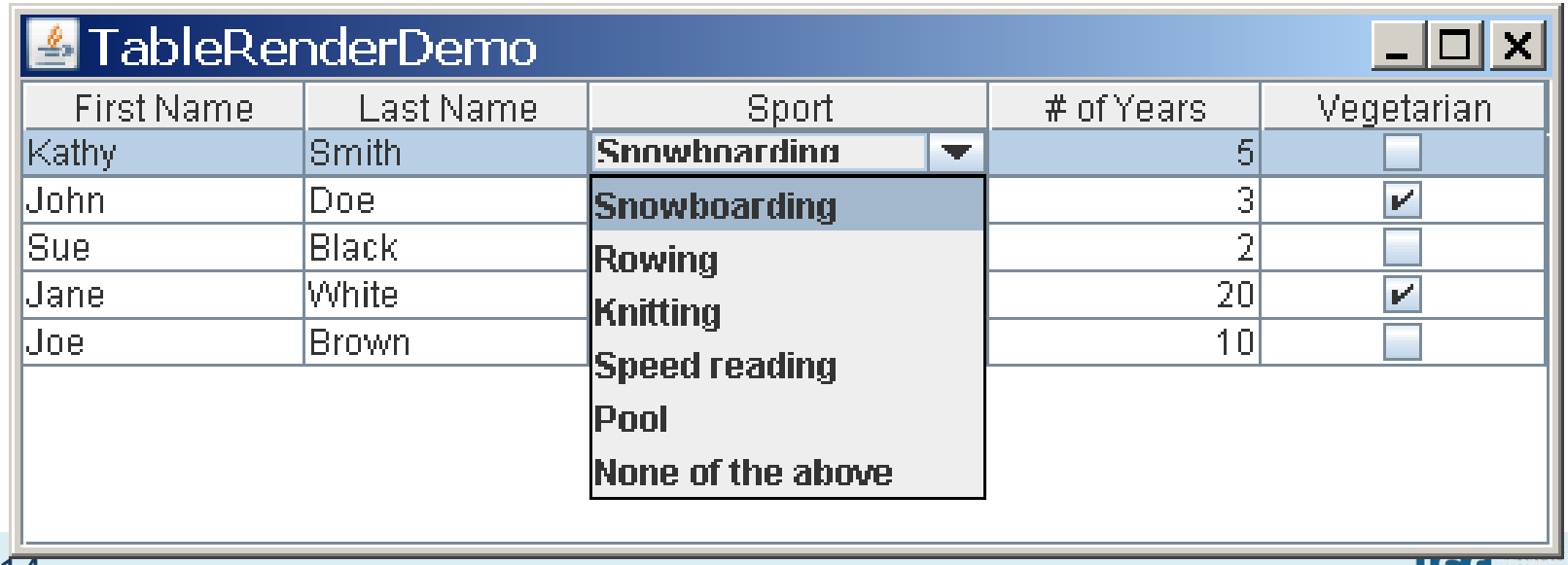
- Motivation
  - Façade: simplify the interface
  - Adapter: match an existing interface
- Adapter: interface is given
  - Not typically true in Façade
- Adapter: polymorphic
  - Dispatch dynamically to multiple implementations
  - Façade: typically choose the implementation statically

Design Goals

- Design to explicit interfaces
  - Façade – a new interface for a library
  - Adapter – design application to a common interface, adapt other libraries to that
- Favor composition over inheritance
  - Façade – library is composed within Façade
  - Adapter – adapter object interposed between client and implementation
- Design for change with information hiding
  - Both Façade and Adapter – shields the client from variations in the implementation
- Design for reuse
  - Façade provides a simple reusable interface to subsystem
  - Adapter allows to reuse objects without fitting interface

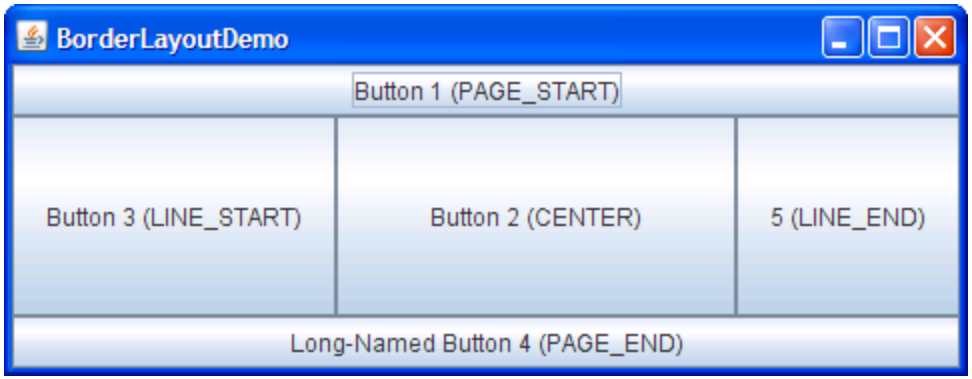
Custom Renderer

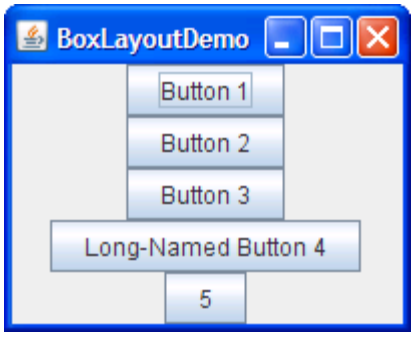
- Rendering of list items and table cells can be customized
- Interfaces TableCellRenderer / ListCellRenderer
- Strategy design pattern (again)
  - DefaultTableCellRenderer: just returns a label
  - But can render as a checkbox, list selection box, etc.

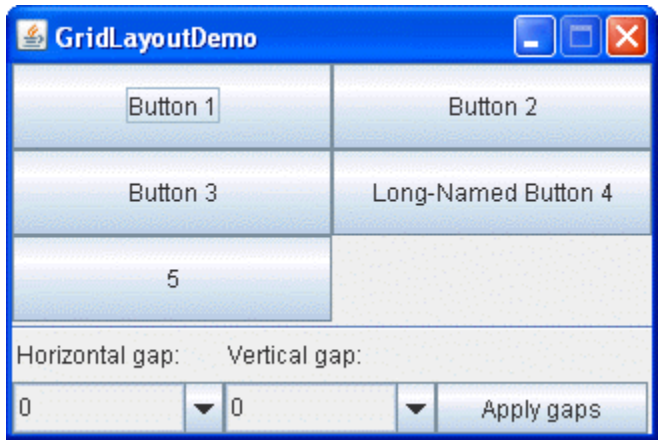


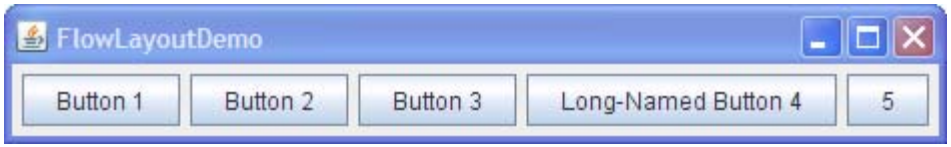
DESIGN OF A LAYOUT MECHANISM

Swing Layout Manager









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

A naïve Implementation

- Hard-code layout algorithms

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

- A new layout requires changing or overriding JPanel

Layout Manager

- A panel has a list of children
- Different layouts possible
  - List of predefined layout strategies
  - Own layouts possible
- Every widget has preferred size

- Delegate specific layout to a separate class implementing an explicit interface
  - Use polymorphism for extensibility

Layout Managers

Which design pattern was used here?

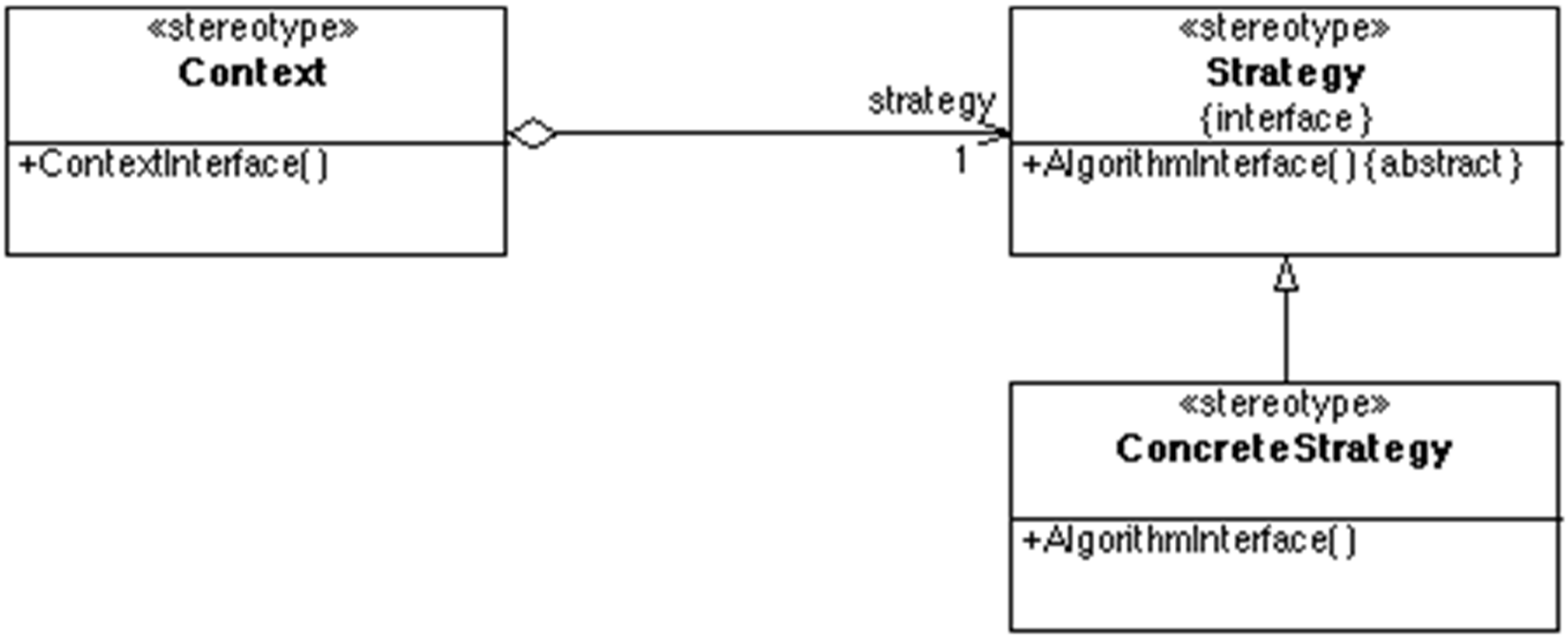
```
panel.setLayout(new BorderLayout(0,0));
```

```
abstract class Container { // JPanel is a Container
        private LayoutManager layoutMgr;

        public doLayout() {
                LayoutManager m = this.layoutMgr;
                if (m!=null)
                        m.layoutContainer(this);
        }
        public Component[] getComponents() { … }
}
```

```
interface LayoutManager {
        void layoutContainer(Container c);
        Dimension getMinimumLayoutSize(Container c);
        Dimension getPreferredLayoutSize(Container c);
}
```

Behavioral: Strategy



Remember Design Discussion for Strategy Pattern

- Design to explicit interfaces
  - Strategy: the algorithm interface
- Design for change and information hiding
  - Find what varies and encapsulate it
  - Allows adding alternative variations later
- Design for reuse
  - Strategy class may be reused in different contexts
  - Context class may be reused even if existing strategies don't fit
- Low coupling
  - Decouple context class from strategy implementation internals

Template Method vs Strategy vs Observer

- Template method vs strategy pattern
  - both support variations in larger common context
  - Template method uses inheritance + abstract method
  - Strategy uses interfaces and polymorphism (object composition)
  - strategy objects reusable across multiple classes; multiple strategy objects per class possible
  - Why is Layout in Swing using the Strategy pattern?
- Strategy vs observer pattern
  - both use a callback mechanism
  - Observer pattern supports multiple observers (0..n)
  - Strategy pattern supports exactly one strategy or an optional one if null is acceptable (0..1)
  - Update method in observer triggers update; rarely returns result
  - Strategy method represents a computation; may return a result

Painting Borders

```
// alternative design
class JPanel {
    protected void paintBorder(Graphics g) {
        switch(getBorderType()) {
            case LINE_BORDER: paintLineBorder(g); break;
            case ETCHED_BORDER: paintEtchedBorder(g); break;
            case TITLED_BORDER: paintTitledBorder(g); break;
            ...
        }
    }
}
```

Painting Borders 2

Which design pattern(s) were used here?

- contentPane.setBorder(**new** EmptyBorder(5, 5, 5, 5));
  - Border interface has “paintBorder”, “getBorderInsets” and “isBorderOpague” methods

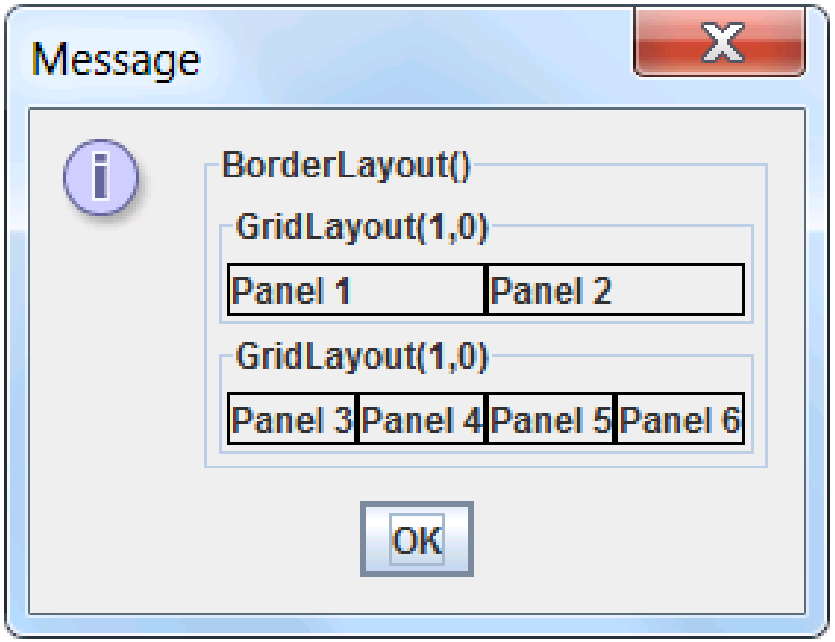
```
// alternative design
class JPanel {
    protected void paintBorder(Graphics g) {
        switch(getBorderType()) {
            case LINE_BORDER: paintLineBorder(g); break;
            case ETCHED_BORDER: paintEtchedBorder(g); break;
            case TITLED_BORDER: paintTitledBorder(g); break;
    }
}
```

```
// actual JComponent implementation
protected void paintBorder(Graphics g) {
        Border border = getBorder();
        if (border != null)
            border.paintBorder(this, g, 0, 0, getWidth(), getHeight());
}
```

Nesting Containers

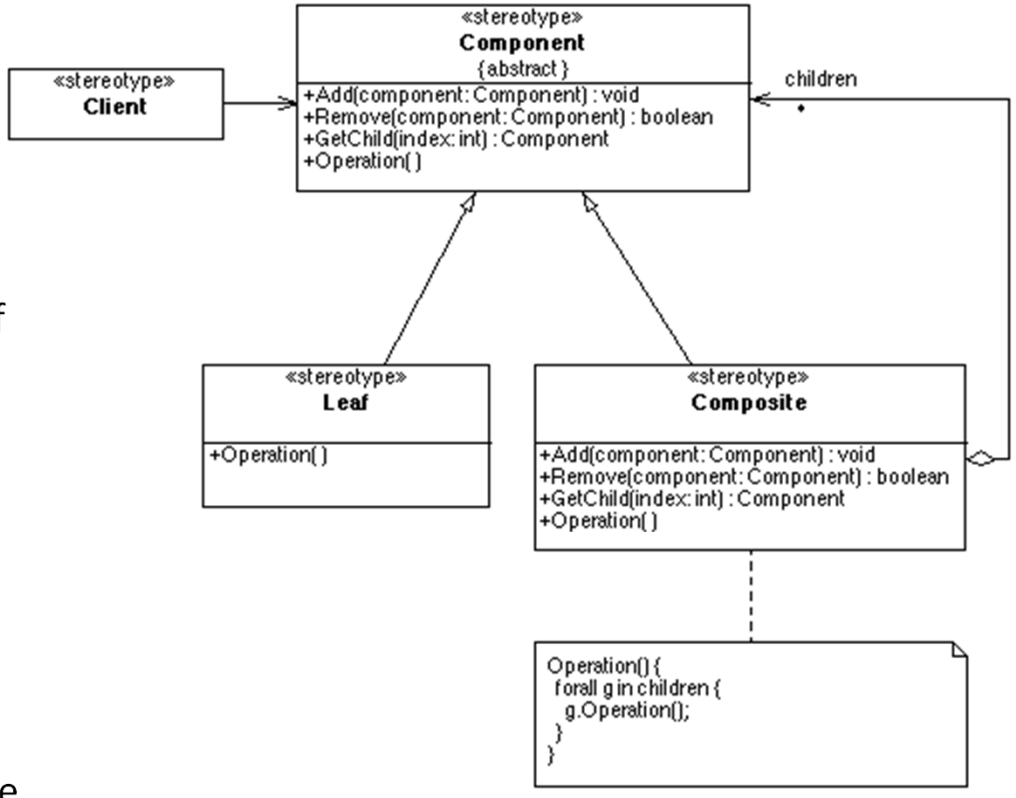
- A `JFrame` contains a `JPanel`, which contains a `JPanel` (and/or other widgets), which contains a `JPanel` (and/or other widgets), which contains…
- Determine the preferred size…

Which design pattern fits this problem?



Composite Design Pattern

- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components



DESIGNING EVENT HANDLING FOR BUTTONS

Swing Button with ActionListener

```
// static public void main…
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);
```

ActionListeners

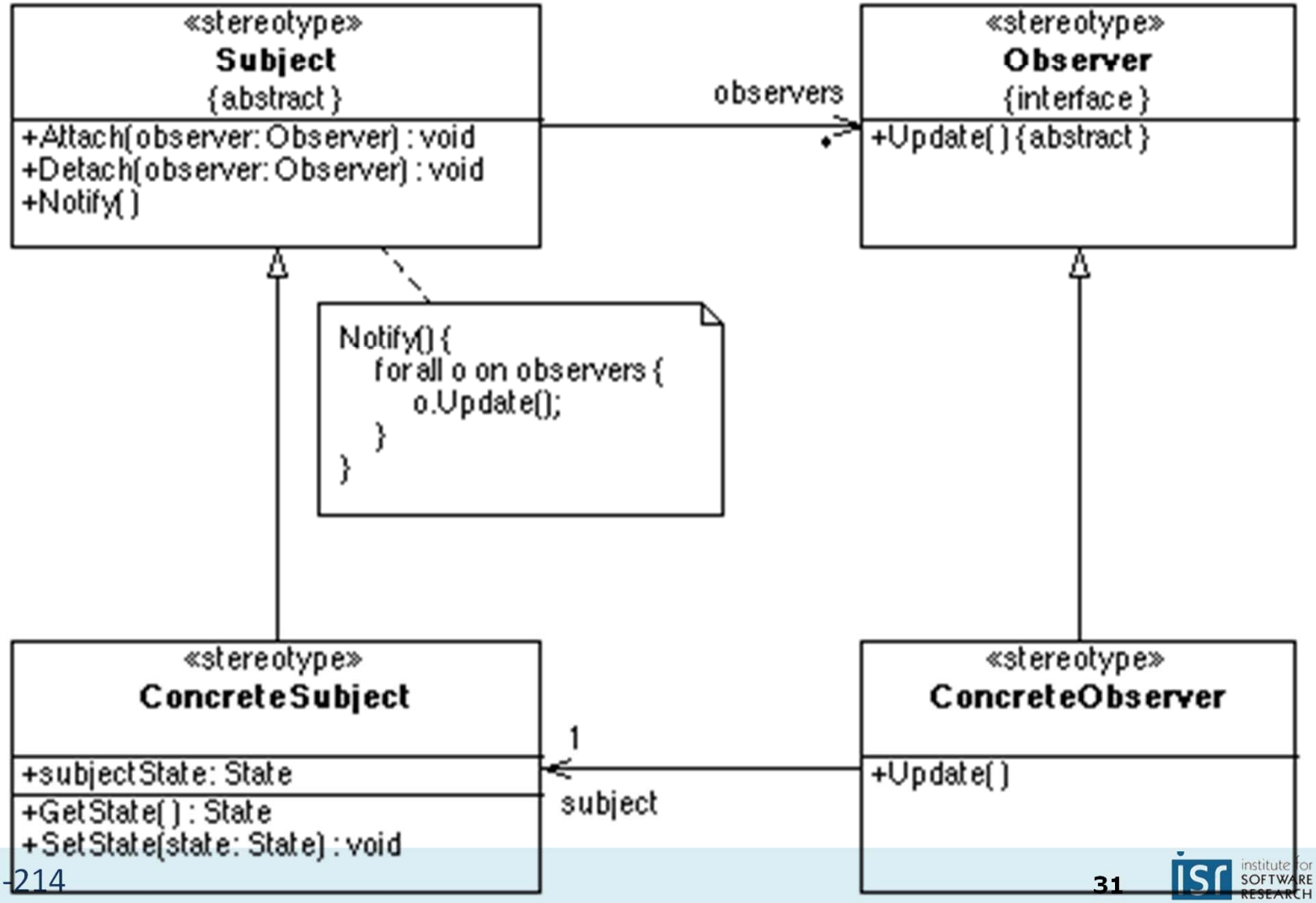
Which design pattern was used here?

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

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

```
class AbstractButton extends JComponent {
        private List<ActionListener> listeners;
        public void addActionListener(ActionListener l) {
                listeners.add(l);
        }
        protected void fireActionPerformed(ActionEvent e) {
                for (ActionListener l: listeners)
                        l.actionPerformed(e);
        }
}
```

The observer design pattern



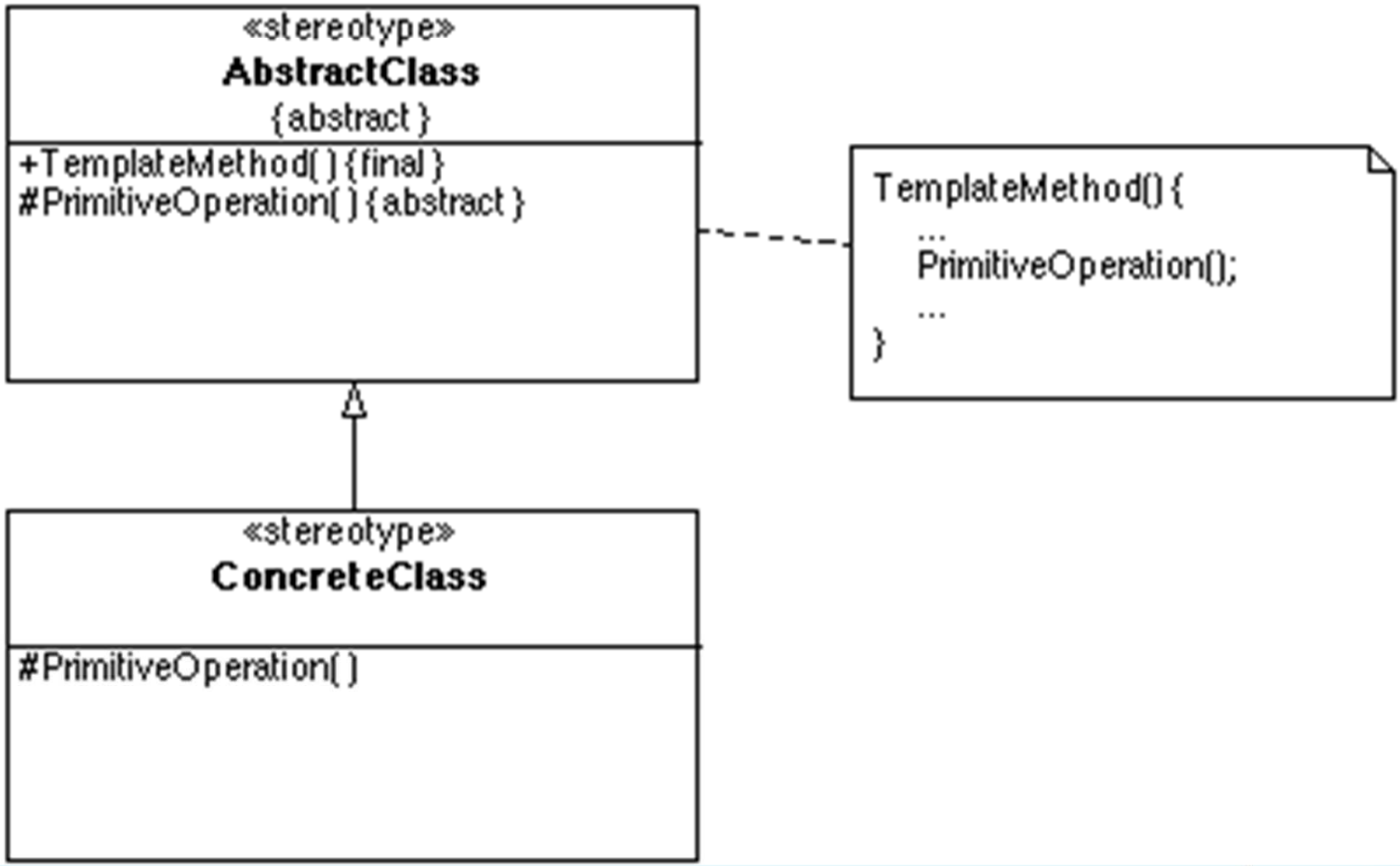
Alternative Button

Which design pattern was used here?

```
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);
```

The Template Method design pattern



Template Method in JButton

- JButton has some behavior of how to handle events
  - eg drawing the button pressed while mouse down
  - The actual template method is not shown in the code two slides ago – the real code implements template method via another instance of the observer pattern
- Some behavior remains undefined until later -> abstract method
  - In this case, default implementation of fireActionEvent already exists
- Template method provides specific extension point within larger shared computation

Design Discussion

- Button implementation should be reusable
  - but differ in button label
  - and differ in event handling
  - multiple independent clients might be interested in observing events
  - basic button cannot know what to do
- Design goal: Decoupling action of button from button implementation itself for reuse
- Template method allows specialized buttons
- Observer pattern separates event handling
  - multiple listeners possible
  - multiple buttons can share same listener

```
JButton btnNewButton = new JButton("New button");

btnNewButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                counter.inc();
        }
});

btnNewButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null,
                        "button clicked", "alert",
                        JOptionPane.ERROR_MESSAGE);
        }
});

panel.add(btnNewButton, BorderLayout.SOUTH);
```

```
class MyButton extends JButton {
        Counter counter;
        public MyButton() {
                ... setTitle...
        }
        protected void fireActionPerformed(ActionEvent e)
                counter.inc();
        }
}

JButton btnNewButton = new MyButton();
panel.add(btnNewButton);
```

```
public class ActionListenerPanel extends JPanel
                                  implements ActionListener {
        public ActionListenerPanel() { setup(); }
        private void setup() {
                button1 = new JButton(”a”);
                button1.addActionListener(this);
                button2 = new JButton(“b”);
                button2.addActionListener(this);
                add(button1); add(button2);
        }
        public void actionPerformed(ActionEvent e) {
                if(e.getSource()==button1)
                        display.setText( BUTTON_1 );
                else if(if(e.getSource()==button1) ...
        }
        ...
}
```

```
public class ActionListenerPanel extends JPanel
                                  implements ActionListener {
    public ActionListenerPanel() { setup(); }
    private void setup() {
        button1 = new JButton(”a”);
        button1.addActionListener(this);
        button2 = new JButton(“b”);
        button2.addActionListener(this);
        add(button1); add(button2);
    }
    public void actionPerformed(ActionEvent e) {
}
```

Cohesion?

Class responsibilities include (1) building the display, (2) wiring buttons and listeners, (3) mapping events to proper response, (4) perform proper response

Consider separating out event handling with different listeners

DESIGN OF DRAWING WIDGETS

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:

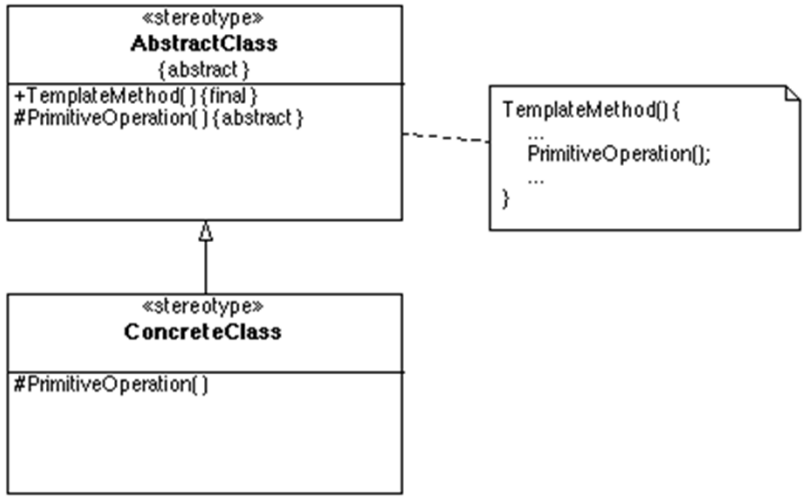
g - the `Graphics` context in which to paint

See Also:

`paintComponent(java.awt.Graphics)`, `paintBorder(java.awt.Graphics)`, `paintChildren(java.awt.Graphics)`, `getComponentGraphics(java.awt.Graphics)`, `repaint(long, int, int, int, int)`

Template Method Design Pattern

- Applicability
  - When an algorithm consists of varying and invariant parts that must be customized
  - When common behavior in subclasses should be factored and localized to avoid code duplication
  - To control subclass extensions to specific operations
- Consequences
  - Code reuse
  - Inverted “Hollywood” control: don’t call us, we’ll call you
  - Ensures the invariant parts of the algorithm are not changed by subclasses



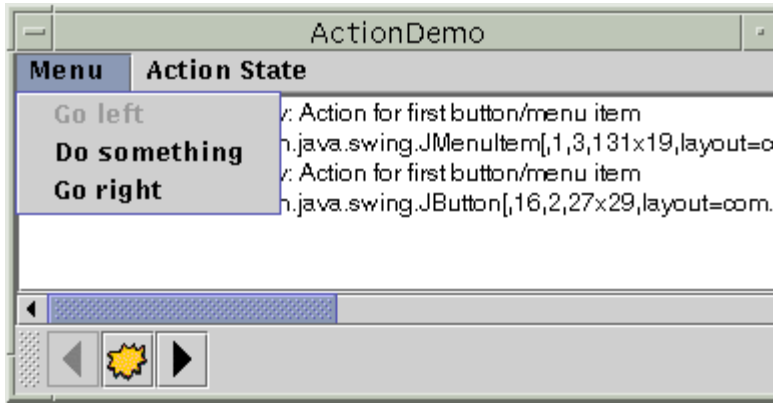
GUI design: Interfaces vs. inheritance

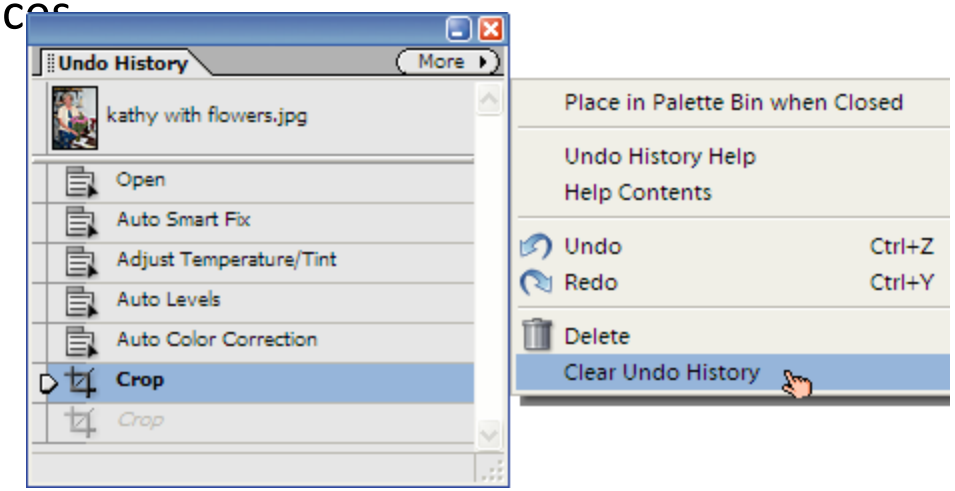
- Inherit from JPanel with custom drawing functionality
  - Subclass “is a” special kind of Panel
  - The subclass interacts closely with the JPanel – e.g. the subclass calls back with super
  - Accesses protected functionality otherwise not exposed
  - The way you draw the subclass doesn’t change as the program executes
- Implement the ActionListener interface, register with button
  - The action to perform isn’t really a special kind of button; it’s just a way of reacting to the button. So it makes sense to be a separate object.
  - The ActionListener is decoupled from the button. Once the listener is invoked, it doesn’t call anything on the Button anymore.
  - We may want to change the action performed on a button press—so once again it makes sense for it to be a separate object

UNDOABLE ACTIONS

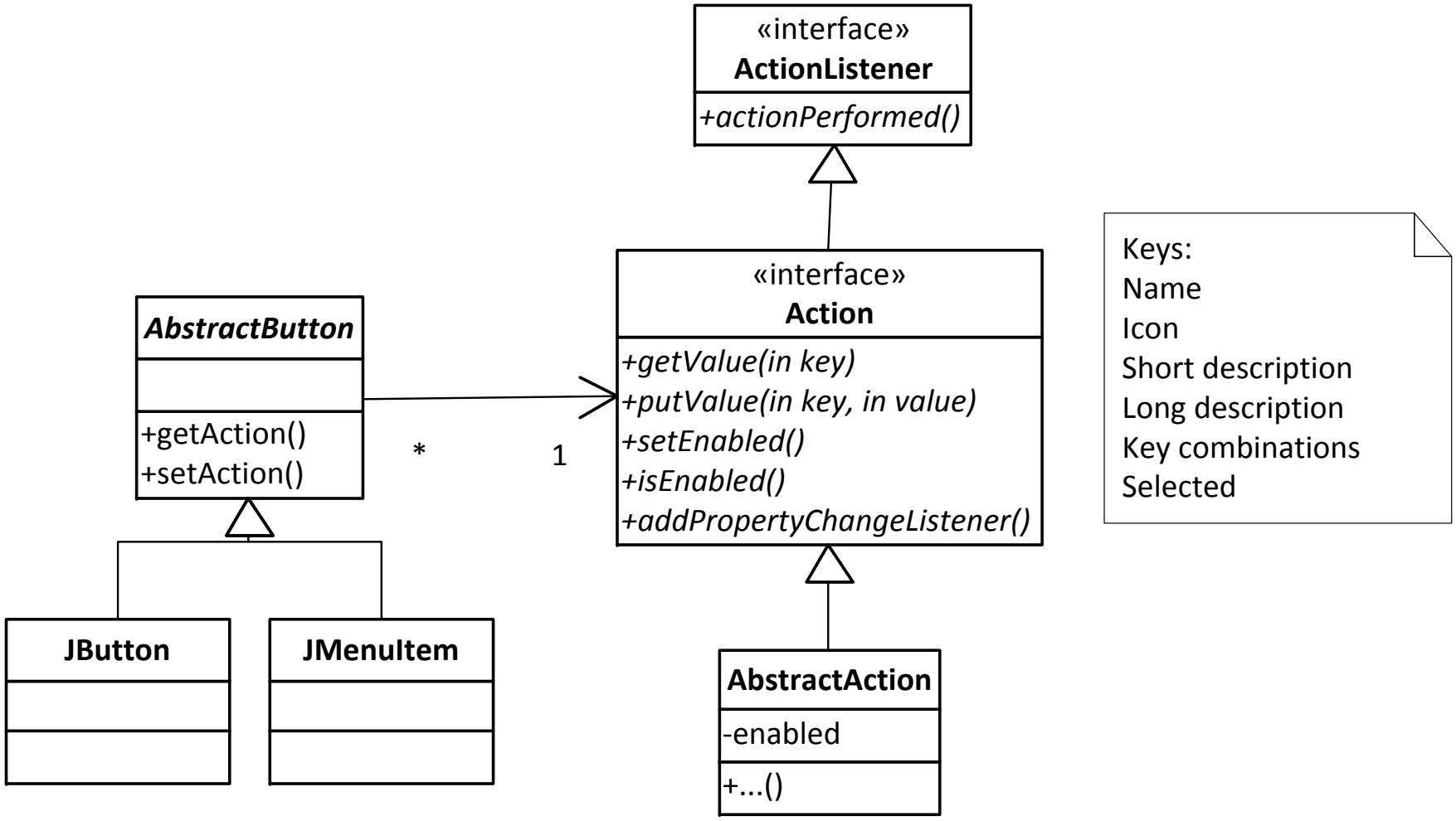
Actions in GUIs

- We want to make actions accessible in multiple places
  - menu, context menu, keyboard shortcut, …
  - When disabling an action, all places should be disabled
- We may want to undo actions
- Separate action execution from presentation
- Delay, queue, or log actions
  - eg macro recording, progress bars, executing remotely, transactions, wizards





Actions in Swing



Action vs. ActionListener

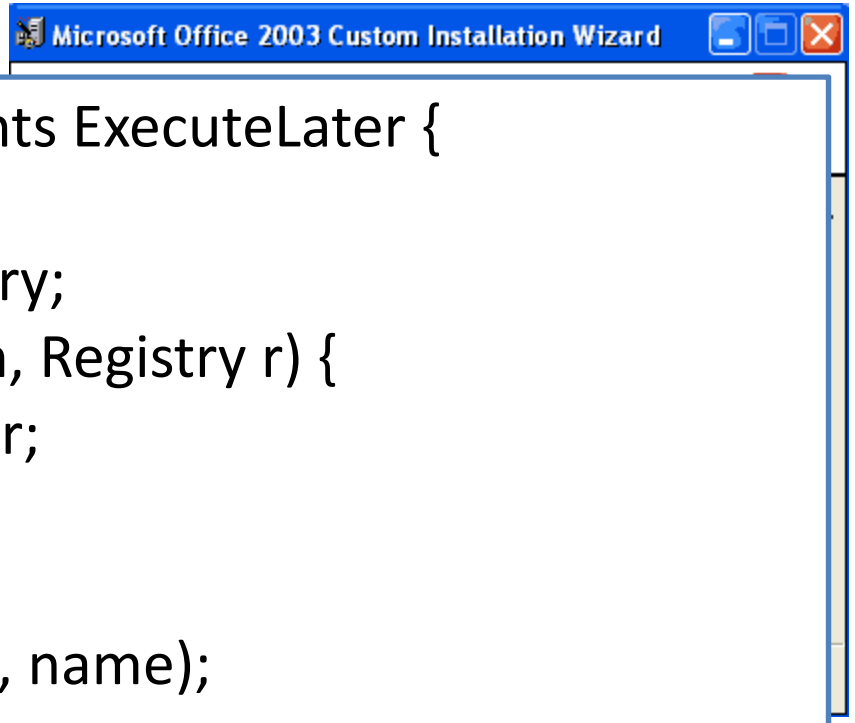
- Action is self-describing (text, shortcut, icon, …)
  - Can be used in many places
  - e.g. log of executed commands, queue, undo list
- Action can have state (enabled, selected, …)
- Actions are synchronized in all cases where they are used
  - with observer pattern: PropertyChangeListener

Implementing a Wizard

- Every step produces some execution
- Execution is delayed until user clicks finish
- Collect objects representing executions in a list
- Execute all at the end
- -> Design for change, division of labor, and reuse; low coupling


```
interface ExecuteLater {
        void execute();
}
```

Implementing a Wizard

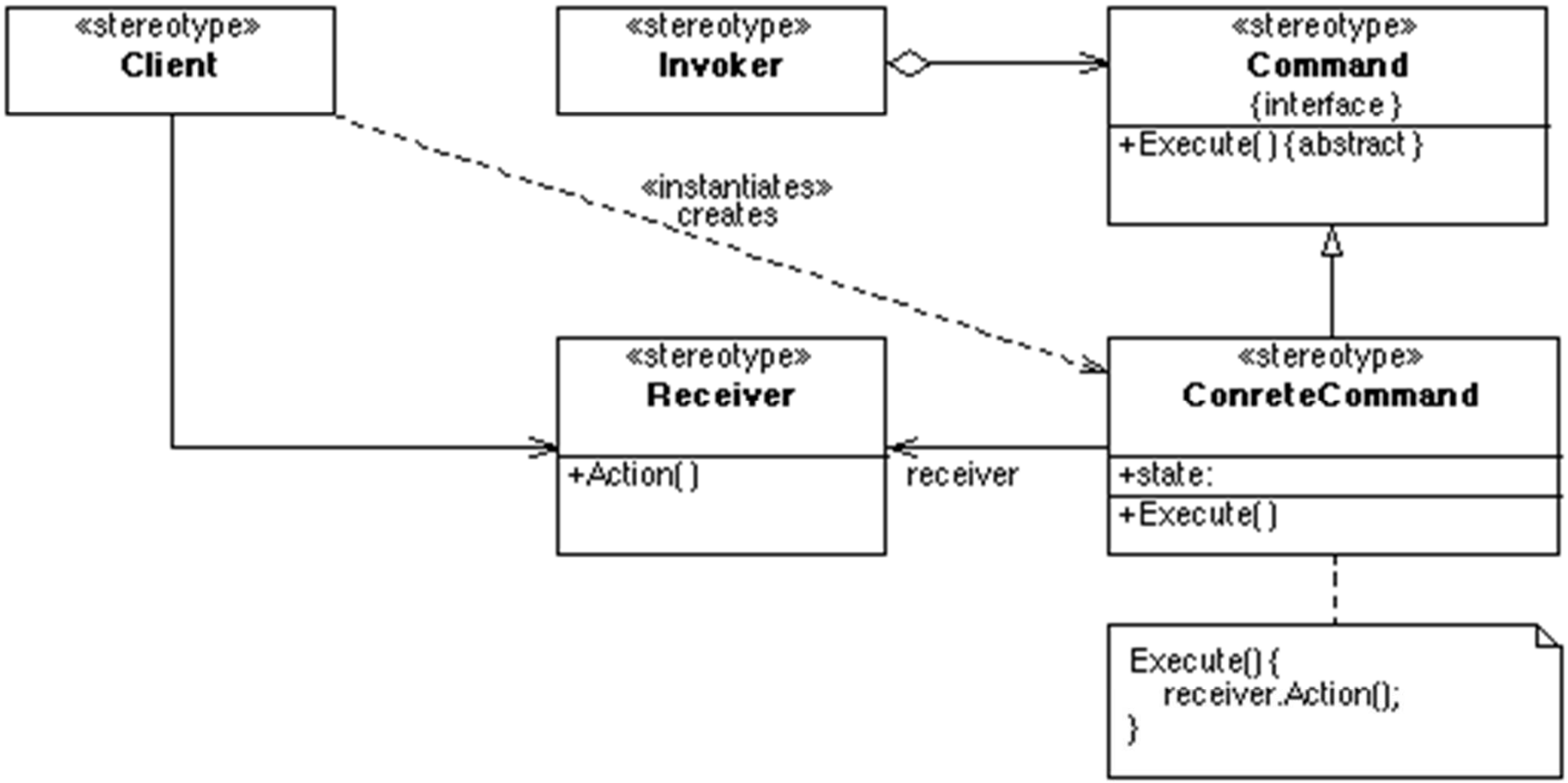



```
class SetCompanyName implements ExecuteLater {
        private final String name;
        private final Registry registry;
        SetCompanyName(String n, Registry r) {
                name=n; registry = r;
        }
        void execute() {
                registry.writeKey(…, name);
        }
}
```

```
        void execute();
}
```

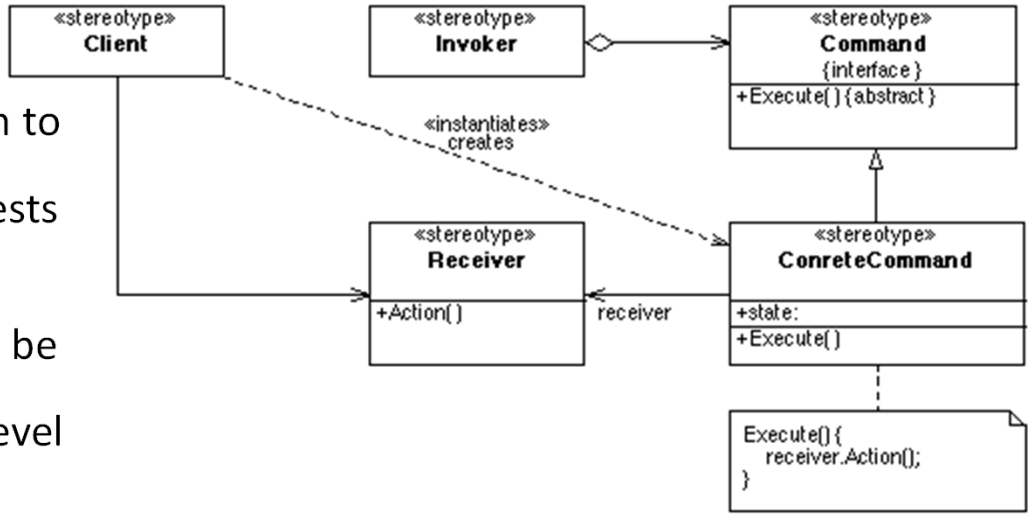
division of labor, and reuse; low coupling

The *Command* Design Pattern



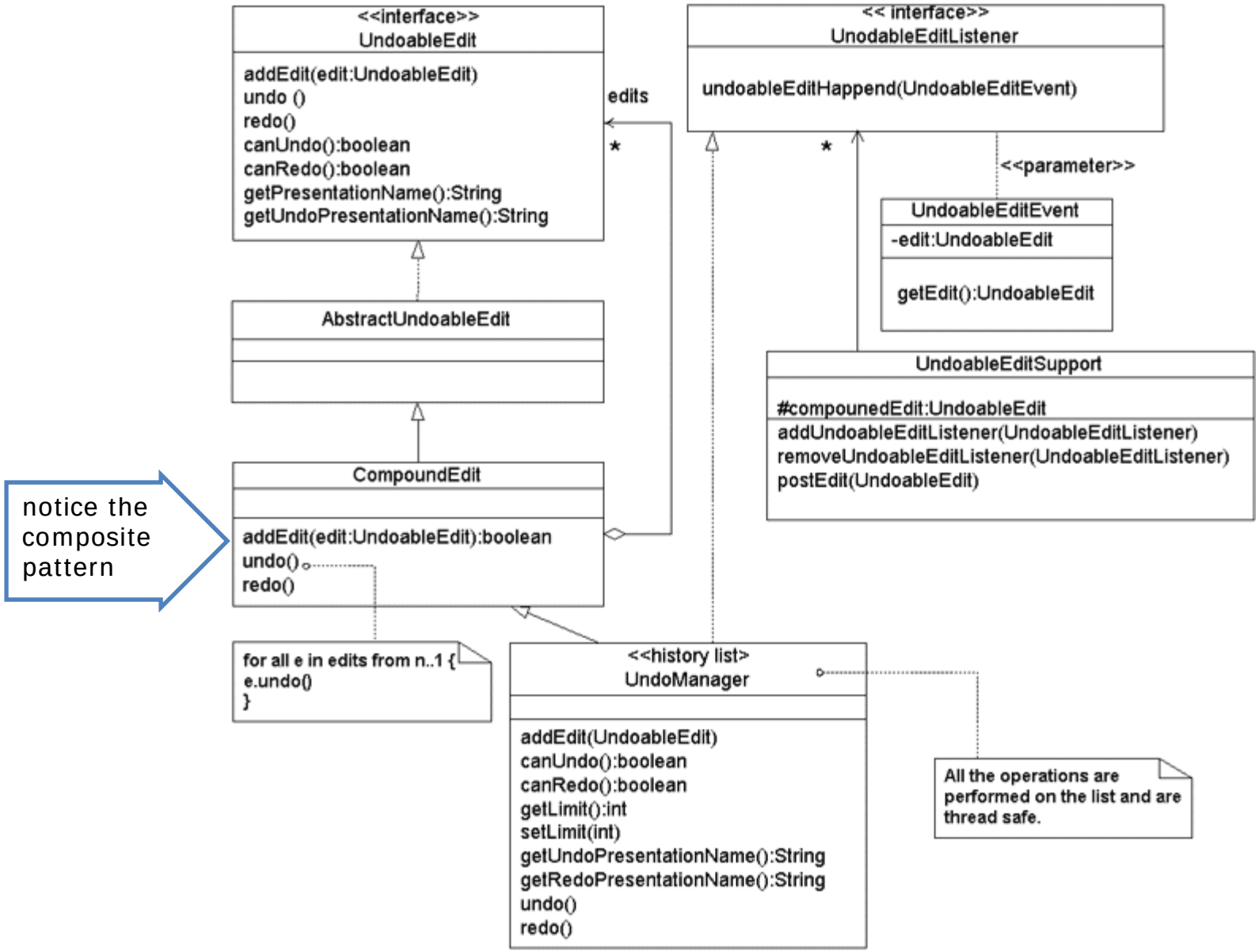
The *Command* Design Pattern

- Applicability
  - Parameterize objects by an action to perform
  - Specify, queue and execute requests at different times
  - Support undo
  - Support logging changes that can be reapplied after a crash
  - Structure a system around high-level operations built out of primitives
- Consequences
  - Decouples the object that invokes the operation from the one that performs it
  - Since commands are objects they can be explicitly manipulated
  - Can group commands into composite commands
  - Easy to add new commands without changing existing code



Common Commands in Java

- javax.swing.Action
  - see above
- java.lang.Runnable
  - Used as an explicit operation that can be passed to threads, workers, timers and other objects or delayed or remote execution
  - see FutureTask



Source: http://www.javaworld.com/article/2076698/core-java/add-an-undo-redo-function-to-your-java-apps-with-swing.html

Summary

- GUIs are full of design patterns
  - Strategy pattern
  - Template Method pattern
  - Composite pattern
  - Observer pattern
  - Decorator pattern
  - Façade pattern
  - Adapter pattern
  - Command pattern
  - Model-View-Controller
- Separation of GUI and Core