Patterns & GUI Programming

Part 2
Creating a Custom Layout Manager

- Layout manager determines how components are arranged/displayed in a container
- Can determine your own layout pattern by implementing the LayoutManager interface
public interface LayoutManager
{
    void layoutContainer(Container parent);
    Dimension minimumLayoutSize(Container parent);
    Dimension preferredLayoutSize(Container parent);
    void addLayoutComponent(String name, Component comp);
    void removeLayoutComponent(Component comp);
}
LayoutManager methods

• minimumLayoutSize() and preferredLayoutSize() determine the minimum and default size of the container when components are laid out.
• Start with preferredLayoutSize - determine the height and width of each component, and use the combined sizes to determine how large window must be.
• Can implement minimumLayoutSize() by just having it call preferredLayoutSize()
LayoutManager methods

• `layoutContainer()` lays out components by setting position and size of each
• Method computes positions of each component then calls `setBounds` to put each component in its correct location
Strategy Pattern

• Context:
  – A class can benefit from different variants for an algorithm
  – Clients sometimes want to replace standard algorithms with custom versions
Strategy Pattern

• Solution
  – Define an interface type that is an abstraction for the algorithm
  – Actual strategy classes realize this interface type.
  – Clients can supply strategy objects
  – Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object
Strategy Pattern
Strategy Pattern - examples

• Pluggable strategy for layout management
  – Layout manager object responsible for executing concrete strategy
  – Generalizes to Strategy Design Pattern

• Other manifestation: Comparators
  Comparator comp = new CountryComparatorByName();
  Collections.sort(countries, comp);
Layout Management - Strategy

• Name in Pattern
  – Context
  – Strategy
  – ConcreteStrategy

  – doWork()

• Actual name
  Container
  LayoutManager
  BorderLayout, FlowLayout, GridLayout, etc.
  layoutContainer()
Sorting - Strategy

• Design Name
  – Context
  – Strategy
  – ConcreteStrategy
  – doWork()

• Actual name
  Collections
  Comparator
  Comparator-implementing class
  compare()
Composite Pattern

• Context
  – Primitive objects can be combined to composite objects
  – Clients treat a composite object as a primitive object
Composite Pattern

• Solution
  – Define an interface type that is an abstraction for the primitive objects
  – Composite object collects primitive objects
  – Composite and primitive classes implement same interface type.
  – When implementing a method from the interface type, the composite class applies the method to its primitive objects and combines the results
Composite Pattern

Calls `method()` for each primitive and combines the results
Containers and Components

• Containers collect GUI components
  – Sometimes, want to add a container to another container
  – Container should be a component

• Composite design pattern
  – Composite methods typically invoke component methods
  – E.g. `Container.getPreferredSize` invokes `getPreferredSize` of components
AWT components - composite

- Design name
  - Primitive
  - Composite
  - Leaf
  - method()

- Actual name
  - Component
  - Container
  - JButton, JPanel, JOptionPane, or other class with no children
  - Component method such as getPreferredSize()
Decorator Pattern

• Context
  – Component objects can be decorated (visually or behaviorally enhanced)
  – The decorated object can be used in the same way as the undecorated object
  – The component class does not want to take on the responsibility of the decoration
  – There may be an open-ended set of possible decorations
Decorator Pattern

• Solution
  – Define an interface type that is an abstraction for the component
  – Concrete component classes realize this interface type.
  – Decorator classes also realize this interface type.
Decorator Pattern

• Solution
  – A decorator object manages the component object that it decorates
  – When implementing a method from the component interface type, the decorator class applies the method to the decorated component and combines the result with the effect of the decoration.
Decorator Pattern

Concrete Component

«interface» Component

1

method()

Decorator

method()

Calls method() for the component and augments the results
Scroll Panes & Scroll Bars

- Scroll bars useful when a component contains more information than can be displayed in available space
- Add functionality to underlying component
- JScrollPane decorates a component and is itself a component
Scroll bars can be attached to components

- Approach #1: Component class can turn on scroll bars
- Approach #2: Scroll bars can surround component
  
  ```java
  JScrollPane pane = new JScrollPane(component);
  ```

- Swing uses approach #2 - JScrollPane is again a component
Decorator Pattern: Scroll Bars

• Design name:
  – Component
  – ConcreteComponent
  – Decorator
  – method()

• Actual name:
  – Component
  – JTextArea
  – JScrollPane
  – paint() or other method of Component
I/O stream filters as decorators

• BufferedReader takes a Reader and adds buffering

• Result is another Reader: Decorator pattern
  ```java
  InputStreamReader r = new InputStreamReader(System.in);
  BufferedReader console = new BufferedReader(r);
  ```

• Many other decorators in stream library, e.g. PrintWriter
Decorator pattern: I/O filters

• Design name:
  – Component
  – ConcreteComponent
  – Decorator
  – method()

• Actual name:
  – Reader
  – InputStreamReader
  – BufferedReader
  – read()
How to Recognize Patterns

• Look at **intent** of pattern: e.g. COMPOSITE has different intent than DECORATOR

• Remember common uses (e.g. STRATEGY for layout managers)

• Use context and solution as "litmus test"
Recognizing Patterns

• Not everything that is strategic is an example of STRATEGY pattern
  – Context must want to use different variants of an algorithm
  – Must be an interface type that is an abstraction of the algorithm; concrete strategy classes must implement the interface
  – Client supplies object of strategy class to context class; context class uses strategy object to invoke algorithm
Litmus test - example

- Can add border to Swing component
  ```java
  Border b = new EtchedBorder();
  component.setBorder(b);
  ```
- Undeniably decorative
- Is it an example of DECORATOR?
Litmus test - example

• Component objects can be decorated (visually or behaviorally enhanced): PASS

• The decorated object can be used in the same way as the undecorated object: PASS

• The component class does not want to take on the responsibility of the decoration: FAIL--the component class has setBorder method