CS1007: Object Oriented Design and Programming in Java

Lecture #12
Nov 1

Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

- Custom Layout manager code
- More patterns
- Recognizing Patterns

- Reading: 5.4.2 - 5.8

Announcements

- Reminder: next Tuesday election day, university holiday (no class).

- Next class (Thursday) will meet in lab
  - Again if you don't have a cs account, please bring a laptop.

Containers and Components

- Containers collect GUI components
  - JPanel holds
    - JButtons
    - JLabels
    - JTextfields
- Sometimes, want to add a container to another container
- Container should act as a component

Important

- Composite design pattern
- Composite method typically invoke component methods

E.g. `Container.getPreferredSize` invokes `getPreferredSize` of components

Composite Pattern

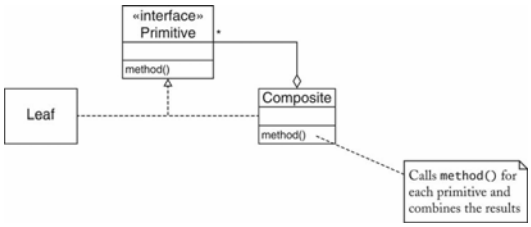
Problem:

- Primitive objects can be combined to composite objects
- Clients treat a composite object as a primitive object

Idea

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

Leaf = low level component



Design layout manager

- All layout managers implement the same interface
- First we need to identify what our layout manager goals are.

Custom Layouts

- Form layout
- Odd-numbered components right aligned
- Even-numbered components left aligned
- Implement `LayoutManager` interface type

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);
}
```

Form Layout

- Ch5/layout/FormLayout.java
- Ch5/layout/FormLayoutTester.java
- Note: Can use GridBagLayout to achieve the same effect

Plan

- Pluggable strategy for layout management
- Layout manager object responsible for executing concrete strategy
- Generalizes to Strategy Design Pattern
- Other manifestation: Comparators

```
Comparator<Country> comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```

Objective

1. A class can benefit from different variants for an algorithm
2. Clients sometimes want to replace standard algorithms with custom versions

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

In short

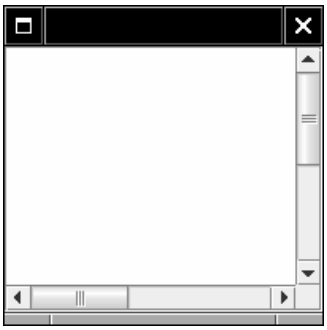
- PLUG AND PRAY

Practical Example

- JPanel/Screen resolution certain size
- Want to display large components …even larger than JPanel

- Ideas?

ScrollBar

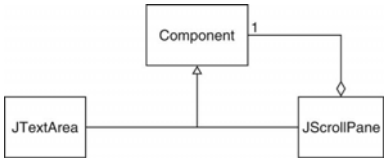


ScrollBars

- Scroll bars can be attached to components
- Approach #1: Component class can turn on scroll bars if too large
- Approach #2: Scroll bars can surround component by user

```
JScrollPane pane = new
  JScrollPane(component);
```

- Swing uses approach #2
- JScrollPane is again a component

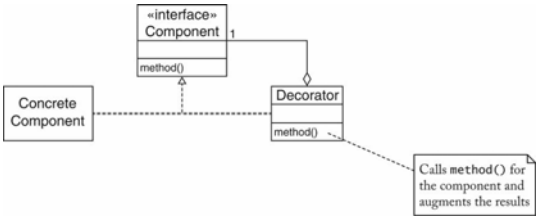


Decorator Pattern

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

Idea

1. Define an interface type that is an abstraction for the component
2. Concrete component classes realize this interface type.
3. Decorator classes also realize this interface type.
4. A decorator object manages the component object that it decorates
5. 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.



Stream Patterns

- `InputStreamReader reader = new InputStreamReader(System.in);`
- `BufferedReader console = new BufferedReader(reader);`

- BufferedReader takes a Reader and adds buffering
- Result is another Reader: Decorator pattern
- Many other decorators in stream library, e.g. PrintWriter

Decorator Pattern: Input Streams

Name in Design Pattern	Actual Name (input streams)
Component	Reader
ConcreteComponent	InputStreamReader
Decorator	BufferedReader
method()	read

How to Recognize Patterns

- Look at the intent of the pattern
- E.g. COMPOSITE has different intent than DECORATOR
- Remember common uses (e.g. STRATEGY for layout managers)
- Not everything that is strategic is an example of STRATEGY pattern
- Use context and solution as "litmus test"

Example

- Can add border to Swing component

Border b = new EtchedBorder()
component.setBorder(b);

- Undeniably decorative
- Is it an example of DECORATOR?

Litmus Test

1. Component objects can be decorated (visually or behaviorally enhanced)
   PASS
2. The decorated object can be used in the same way as the undecorated object
   PASS
3. The component class does not want to take on the responsibility of the decoration
   FAIL--the component class has setBorder method
4. There may be an open-ended set of possible decorations

Using Patterns

- Invoice contains line items
- Line item has description, price
- Interface type LineItem:

Ch5/invoice/LineItem.java

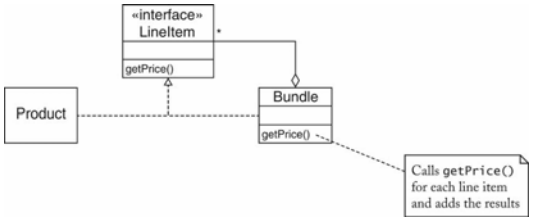
- Product is a concrete class that implements this interface:

Ch5/invoice/Product.java

Bundles

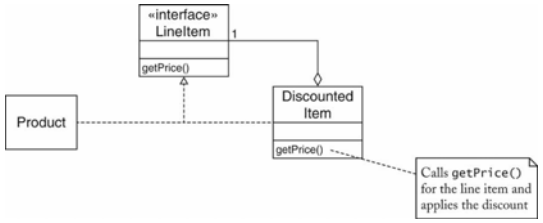
- Bundle = set of related items with description+price
- E.g. stereo system with tuner, amplifier, CD player + speakers
- A bundle has line items
- A bundle is a line item
- COMPOSITE pattern

Ch5/invoice/Bundle.java (look at getPrice)



Discounted Items

- Store may give discount for an item
- Discounted item is again an item
- DECORATOR pattern
- Ch5/invoice/DiscountedItem.java (look at getPrice)
- Alternative design: add discount to LineItem



Model View Separation

- GUI has commands to add items to invoice
- GUI displays invoice
- Decouple input from display
- Display wants to know when invoice is modified
- Display doesn't care which command modified invoice
- OBSERVER pattern

Change Listener

- Use standard ChangeListener interface type

```
public interface ChangeListener
{
   void stateChanged(ChangeEvent event);
}
```

- Invoice collects ArrayList of change listeners
- When the invoice changes, it notifies all listeners:
- ChangeEvent event = new ChangeEvent(this);

```
for (ChangeListener listener : listeners)
   listener.stateChanged(event);
```

Question

- If you run a family tree program and create your family tree in some java class form, how do you keep it saved?

Idea:

- Allow the programmer to take a snapshot of live memory, and save it in a binary form…..
- No need to recreate classes

1. We need to tell java we want to save a certain class
2. Save the class

java.io.Serializable

```
public class student implements Serializable
  {
      private String name;
      private int age;

  …
      public String getName(){
            return name;
  }
  }
```

Save routine

```
public static void main(String args[]) {
    Student one = new Student….

    try{

    FileOutputStream fos = new FileOutputStream("saved.data");

    ObjectOutputStream out = new ObjectOutputStream(fos);

    out.writeObject(one);

    out.close;

    }catch(IOException ioe){ ..  }
```

Load Routine

```
try{
    FileInputStream fis = new FileInputStream("saved.data");
    ObjectInputStream in = new ObjectInputStream(fis);
    Student oldone = (Student)in.readObject();
    }catch(IOException ioe) {…}
```

Important note

- Only objects which extend serializable can be saved
- SO:
  - If your class has field variables which don't implement this….

Two options

1. Mark those non serializable as 'transient' this tells the jvm not to save those variables
2. Implement a custom `writeObject` and `readObject`

   can then choose which fields to save and load, and initialize any others

Next Time

- Meet in lab
- Please see updated reading list on schedule page of web, try to do reading ahead of class.