

# CS1007: Object Oriented Design and Programming in Java

Lecture #14

Mar 9

Shlomo Hershkop  
*shlomo@cs.columbia.edu*

## Outline

- Wrap up patterns
- Examples and code
  
- Reading chapter 5.4 – end of 5

## Announcements

- Almost done, not quite the exams, will post over weekend on courseworks, will return when we meet again
- Will outline the othello work and your part

## Clarification

- Some confusion on AWT vs Swing
- AWT
  - Still very used
  - Very useful
  - Not organized as objects

# Patterns

- We want to identify software patterns
- Want to use them in practical ways
  
- Doing example of layout manager
  - Invisible piece of your code

# Example

- Say you have 2 different classes, A and B
  
- In both A and B you are doing
- ...  
    `calculate1();`  
    `calculate2();`  
    `calculate3();`  
    ...
- A PATTERN!

- What is the best way to take advantage of the patterns?

- Through inheritance
- Through another class
- What is the difference?

## From last class

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

## Generalization of Patterns

1. A class can benefit from different variants for an algorithm
2. Clients sometimes want to replace standard algorithms with custom versions
3. What is the best approach?

## Solution

- Usually you can get away with
- an ***interface*** type that is an abstraction for the algorithm
  - What does this mean?
- 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

## Patterns

- Many different patterns
- We are only covering a subset
- To get a feel for programming design

## Composite pattern

- Idea: have a bunch of little pieces, each one behaves the same way
- Goal: want to simplify the management of all the pieces, by treating them as a single unit
  
- Where have we seen this?

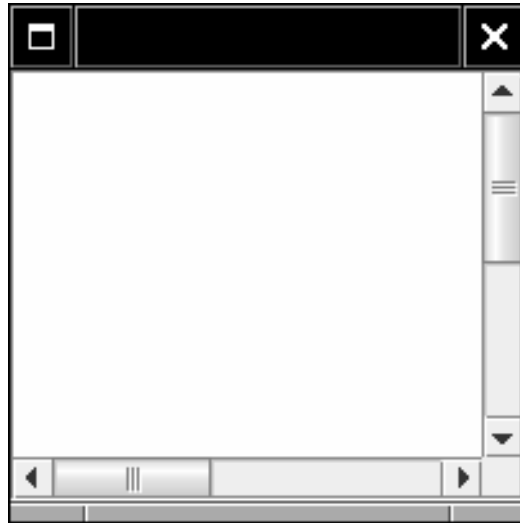
- Container can have tons of little components, to get the size it runs through each and asks them for their preferred size

## Practical Question

- In GUI/Graphics many times have large graphic
- But our screen resolution certain size (lets say smaller)
- Want to display large components
  - Large image on ipod/cellphone
- any ideas on how to approach the problem?



## One solution: the Scroll bar



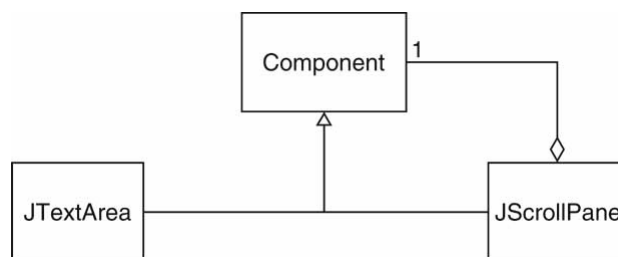
## ScrollBars 2 ideas

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

## How java approaches it

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

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



# Pattern

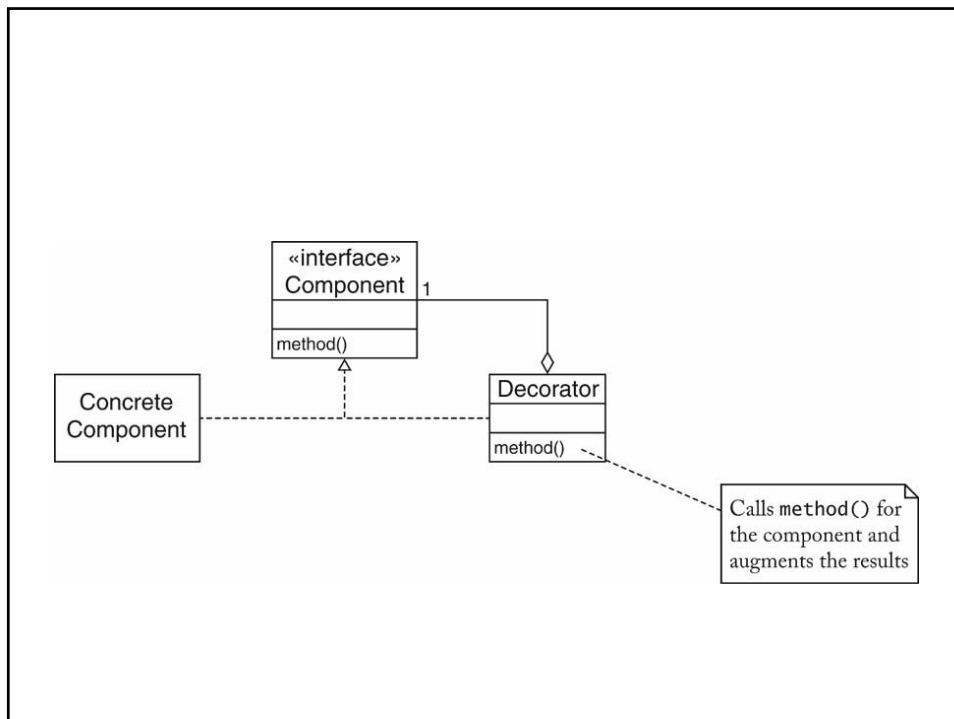
- So what is the pattern here?

## This is a 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

# pattern

1. Define an interface type that is an abstraction for the component
2. Concrete component classes impliment 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.



- Patterns are not just GUI objects
- Also behavior patterns

## 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?



The image shows a dialog box with two sections. The top section is titled "Style" and contains two checkboxes: "Italic" and "Bold", both of which are unchecked. The bottom section is titled "Size" and contains three radio buttons: "Small", "Medium", and "Large". The "Large" radio button is selected, indicated by a filled circle.

## 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

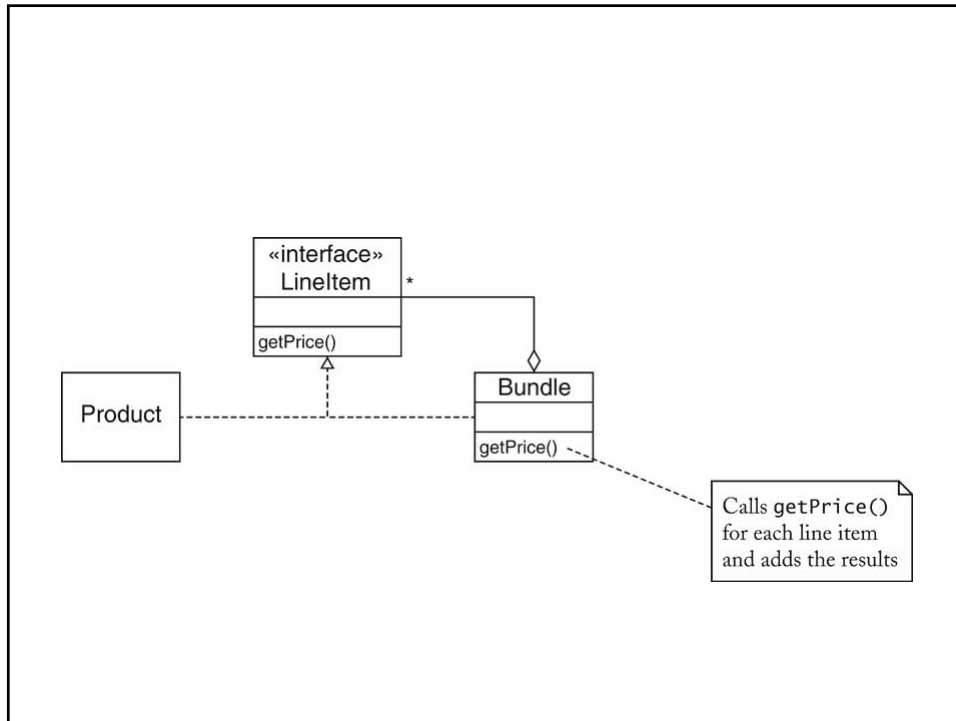
## Using Patterns

- 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

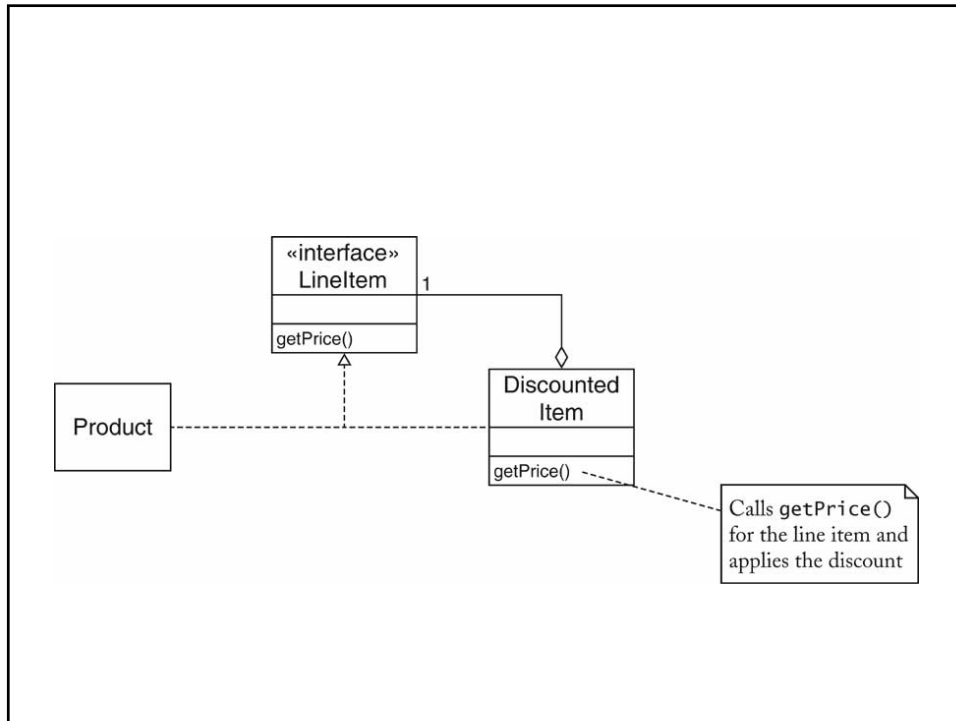
- Bundle = set of related items with description+price
- E.g. stereo system with tuner, amplifier, CD player + speakers
- A bundle has line items
- A bundle is a line item
- COMPOSITE pattern  
Ch5/invoice/Bundle.java (look at getPrice)





## 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 `Lineltem`



## 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;
    }
}
```

## Next Time

- Continue reading
- Start homework