CS427a: Object-Oriented Programming
Design Patterns for Flexible and Reusable design

Michael J. Fischer
(from slides by Y. Richard Yang)

Lecture 21
November 18, 2010
Example: KitchenViewer Interface

- Wall cabinet
- Counter
- Floor cabinet

Menu

Display area

Styles

Modern | Classic | Antique | Arts & Crafts

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
Selecting *Antique* Style
Aspect of the system that may change/vary?
The Abstract Factory Idea

- **KitchenStyle**
  - getWallCabinet()  
  - getFloorCabinet()

- **ModernKStyle**
  - getWallCabinet()  
  - getFloorCabinet()

- **AntiqueKStyle**
  - getWallCabinet()  
  - getFloorCabinet()

- **WallCabinet**
  - AntiqueWallCabinet

- **FloorCabinet**
  - AntiqueFloorCabinet

```
FloorCabinet getFloorCabinet()
{
    return new ModernFloorCabinet();
}
```

```
FloorCabinet getFloorCabinet()
{
    return new AntiqueFloorCabinet();
}
```
Abstract Factory Design Pattern Applied to KitchenViewer

**Client**
- `renderKitchen(KitchenStyle)`

**KitchenStyle**
- `getWallCabinet()`
- `getFloorCabinet()`

**Kitchen**
- `getWallCabinet()`
- `getFloorCabinet()`

**WallCabinet**

**FloorCabinet**

**ModernKStyle**
- `getWallCabinet()`
- `getFloorCabinet()`

**ModernWallCabinet**

**AntiqueKStyle**
- `getWallCabinet()`
- `getFloorCabinet()`

**AntiqueWallCabinet**

**ModernFloorCabinet**

**AntiqueFloorCabinet**
Abstract Factory Design Pattern

Client
  doOperation( Style myStyle )

Style
  getComponentA()
  getComponentB()

Style1
  getComponentA()
  getComponentB()

Style2
  getComponentA()
  getComponentB()

ComponentA

ComponentB

Style1ComponentA

Style2ComponentA

Style1ComponentB

Style2ComponentB

Collection
Concrete and Abstract Layers

Client

KitchenStyle

ModernKStyle

AntiqueKStyle

Kitchen

Abstract level

ModernWallCabinet

AntiqueWallCabinet

Concrete level

FloorCabinet

ModernFloorCabinet

AntiqueFloorCabinet
Abstract Factory Application
Sequence Diagram

myStyle: KitchenStyle

Client

getWallCabinet()

myStyle.

-- IF myStyle BELONGS TO ModernKStyle --

renderKitchen ( myStyle )

getWallCabinet()

myStyle: ModernKStyle

wallCabinet1: ModernWallCabinet

ModernWallCabinet()

-- IF myStyle BELONGS TO AntiqueKStyle --

myStyle: AntiqueKStyle

wallCabinet1: AntiqueWallCabinet

AntiqueWallCabinet()
Potential use of this Design Pattern?

Client

doOperation( Style myStyle )

Style

getComponentA()
getComponentB()

Collection

ComponentA

ComponentB

Style1

getComponentA()
getComponentB()

Style1ComponentA

Style1ComponentB

Style2

getComponentA()
getComponentB()

Style2ComponentA

Style2ComponentB

Style2ComponentA

Style1ComponentA

Style1ComponentB

Style2ComponentA

Style2ComponentB
References

• Design Patterns

• Headfirst Design Patterns

• Software Design
Example: Starbuzz Coffee

- The coffee shop offers a variety of beverages

```
Beverage

description

getDescription();
cost();
// Other methods

HouseBlend
cost()

DarkRoast
cost()

Decaf
cost()

Espresso
cost()
```
Problem

• A customer may also ask for condiments
  – steamed milk
  – soy
  – mocha (otherwise known as chocolate)
  – whipped milk

• Starbuzz charges a bit for each of these
Beverage

description
milk
soy
mocha
whip

getDESCRIPTION();
cost();

hasMilk(); setMilk();
hasSoy(); setSoy();
hasMocha(); setMocha();
hasWhip(); setWhip();

// Other methods

HouseBlend
cost()

DarkRoast
cost()

Decaf
cost()

Espresso
cost()
Potential Changes

• Potential changes:
  – Price change to condiments
  – New condiments
  – Double moca
  – ...

Design idea

• Basic idea: extension at run time, not compile time

• Definition: The Decorator pattern attaches additional features to an object dynamically. It provides a flexible alternative to subclassing for extending functionality
Design approach 1

• Each beverage contains a dynamic list of condiments

• Example
  – Take a DarkRoast object
  – Decorate it with a Mocha object
  – Decorate it with a Whip object

UML class model?
Decorator design

• Example
  – Take a DarkRoast object
  – Decorate it with a Mocha object
  – Decorate it with a Whip object
  – Call the cost() method and rely on delegation to add on the condiment cost

• Decorator adds its own behavior before or after calling the decorated object
Decoration Delegation Process

1. First, we call `cost()` on the outermost decorator, Whip.

2. Whip calls `cost()` on Mocha.

3. Mocha calls `cost()` on DarkRoast.

4. DarkRoast returns its cost 99 cents.

5. Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

6. Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.
void doAction()
{
    ...... // do actions special to this decoration
    objDecorated.doAction(); // pass along
}

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
Sequence Diagram for Decorator

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
Decoration Features

• Decorators have the same supertype as the objects they decorate
• You can use one or more decorators to wrap an object
  – Thus, you can pass decorated object in place of original (wrapped) object
• The decorator adds its own behavior either before or after delegating to the object it decorates to
• Objects can be decorated at any time, including run-time, with as many decorators as possible
Exercise

• Suppose we allow different sizes for the beverages
  – Tall (small)
  – Grande (medium)
  – Venti (large)
## Some Common Design Patterns

<table>
<thead>
<tr>
<th>Scope</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Factory Method</td>
<td>Adapter (class)</td>
<td>Interpreter, Template Method</td>
</tr>
<tr>
<td>Object</td>
<td>Abstract Factory</td>
<td>Adapter (object)</td>
<td>Chain of Responsibility</td>
</tr>
<tr>
<td></td>
<td>Builder</td>
<td>Bridge</td>
<td>Command</td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
<td>Composite</td>
<td>Iterator</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Decorator</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flyweight</td>
<td>Memento</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Facade</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Proxy</td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Visitor</td>
</tr>
</tbody>
</table>
Example: Weather-O-Rama

Weath-O-Rama, Inc.
100 Main Street
Tornado Alley, OK 4502

Statement of Work

Congratulations on being selected to build our next generation Internet-based Weather Monitoring Station!

The weather station will be based on our patent pending WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We’d like for you to create an application that initially provides three display elements: current conditions, weather statistics and a simple forecast, all updated in real time as the WeatherData object acquires the most recent measurements.

Further, this is an expandable weather station. Weather-O-Rama wants to release an API so that other developers can write their own weather displays and plug them right in. We’d like for you to supply that API!

Weather-O-Rama thinks we have a great business model: once the customers are hooked, we intend to charge them for each display they use. Now for the best part: we are going to pay you in stock options.

We look forward to seeing your design and alpha application.

Sincerely,

Johnny Hurricane

Johnny Hurricane, CEO

P.S. We are overnighting the WeatherData source files to you.
Weather-O-Rama

Weatherstation provides

- Humidity sensor device
- Temperature sensor device
- Pressure sensor device

WeatherData object pulls data

Display device displays

Current Conditions is one of three different displays. The user can also get weather stats and a forecast.

Weather-O-Rama provides

What we implement
Weather-O-Rama Interface

WeatherData

getTemperature();
getHumidity();
getPressure();
measurementsChanged();
setMeasurements();
// other methods

This method gets called whenever the weather measurements have been updated.
First Implementation

void measurementsChanged() {

    float temp = getTemperature() ;
    float humidity = getHumidity() ;
    float pressure = getPressure() ;

    currentConditionsDisplay->update(temp, humidity, pressure) ;
    statisticsDisplay->update(temp, humidity, pressure) ;
    forecastDisplay->update(temp, humidity, pressure) ;
}

By coding to concrete implementation, we have no way of allowing other displays and plug in.
Observer Pattern

• Design Purpose: defines a run-time, one-to-many dependency between objects so that when one object (the subject) changes state, all of the dependents (observers) are notified.
Observer Design Pattern

Subject:
- registerObserver()
- removeObserver()
- notifyObservers()

ConcreteSubject:
- getState()
- setState()

Observer:
- update()

ConcreteObserver:
- update()

Client part:
- for all Observer's o: o.update();

Server part:
- 1..n

Diagram:
- Client
- Subject
- ConcreteSubject
- Observer
- ConcreteObserver

Client part

Server part
How does Observer apply these design principles?

• Identify the aspects of your application that vary and separate them from what stay the same

• Program to an interface not implementation

• Favor composition over inheritance
Discussion

- Java Observation design:
  update(Observable o, Object obj);

- ConcreteSubject
  getState();
  setState();

- ConcreteObserver
  update();

- Observer
  update();

- Subject
  registerObserver();
  removeObserver();
  notifyObservers();

- Client

- 1..n

- for all Observer’s o:
  o.update();