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

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

Lecture 23b
November 29, 2011
Example: Duck Game

• A startup produces a duck-pond simulation game
• The game shows a large variety of duck species swimming and making quacking sounds
Initial Design

Duck

quack()
swim()
display() = 0
// Other duck-like method

MillardDuck

display() {
  looks like a mallard
}

RedheadDuck

display() {
  looks like a redhat
}

Other types of ducks
Design Change: add fly()

Duck

- quack()
- swim()
- display() = 0
- fly()
- // Other duck-like method

MillardDuck
- display() {
  looks like a mallard
}

RedheadDuck
- display() {
  looks like a redhat
}

Other types of ducks
Problem

• Generalization may lead to unintended behaviors: a rubber duck is flying and quacks

Duck

quack()
swim()
display() = 0
// Other duck-like method

MillardDuck

display() { looks like a mallard }

RubberDuck

display() { looks like a rubber duck }
quack() { // sqeak }
fly() { // cannot fly }
Anticipating Changes

• Identify the aspects of your application that may vary
  – What may change?

• Anticipate that
  – new types of ducks may appear and
  – behaviors (quack, swimming, and flying) may change, even change at run time (swirl flying, circular flying, ...)

Handling Varying Behaviors

• Solution: take what varies and “encapsulate” it
  – Since fly() and quack() vary across ducks, separate these behaviors from the Duck class and create a new set of classes to represent each behavior

super class of all ducks

Varying and run-time changeable behaviors
Design

• Each duck object has a fly behavior

```java
<<interface>>
FlyBehavior

fly()

FlyWithWings

fly() {
    //
}

FlyNoWay

fly() {
    // cannot fly
}
```
Programming to implementation vs. interface/supertype

• Programming to an implementation
  – Dog d = new Dog();
  – d.bark();

• Programming to an interface/supertype
  – Animal a = new Dog();
  – a.makeSound();
Implementation

```
FlyWithWings
fly() {
//}
```

```
FlyNoWay
fly() {
// cannot fly
}
```

```
Duck
quack() 
swim() 
display() = 0
// Other duck-like method
```

```
MallardDuck
display() {
  looks like a mallard
}
```

```
RedheadDuck
display() {
  looks like a redhat
}
```

```
<<interface>>
FlyBehavior
fly()
```

```
Duck
```

```
MallardDuck
```

```
RedheadDuck
```

```
FlyWithWings
```

```
FlyNoWay
```
Exercise

• Add rocket-powered flying?
The Strategy Pattern

- Defines a set of algorithms, encapsulates each one, and makes them interchangeable by defining a common interface
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
Summary: 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
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.
KitchenViewer Example

Wall cabinets

Countertop

Floor cabinets

Modern  Classic  Antique  Arts & Crafts
Selecting *Antique* Style

Modern  Classic  Antique  Arts & Crafts
Aspect of the system that may change/vary?
The Abstract Factory Idea

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

- **ModernKStyle**
  - `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
  - ModernFloorCabinet

- **AntiqueKStyle**
  - getWallCabinet()
  - getFloorCabinet()
  - AntiqueWallCabinet
  - AntiqueFloorCabinet
Abstract Factory Design Pattern

Client
  doOperation( Style myStyle )

Style
  getComponentA()
  getComponentB()

Style1
  getComponentA()
  getComponentB()

Style2
  getComponentA()
  getComponentB()

Collection

ComponentA
  Style1ComponentA
  Style2ComponentA

ComponentB
  Style1ComponentB
  Style2ComponentB
Concrete and Abstract Layers

- KitchenStyle
  - ModernKStyle
  - AntiqueKStyle

- Abstract level
  - Kitchen
  - WallCabinet
  - FloorCabinet

- Concrete level
  - ModernWallCabinet
  - AntiqueWallCabinet
  - ModernFloorCabinet
  - AntiqueFloorCabinet
Abstract Factory Application
Sequence Diagram

Client

myStyle: KitchenStyle

myStyle.
getWallCabinet()

renderKitchen ( myStyle )

-- IF myStyle BELONGS TO ModernKStyle --

myStyle: ModernKStyle

wallCabinet1: ModernWallCabinet

getWallCabinet()

ModernWallCabinet()

-- IF myStyle BELONGS TO AntiqueKStyle --

myStyle: AntiqueKStyle

wallCabinet1: AntiqueWallCabinet

getWallCabinet()

AntiqueWallCabinet()
Potential use of this Design Pattern?

Client

dooOperation( Style myStyle )

Style

gGetComponentA()
gGetComponentB()
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
Attempt 1

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

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 change 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 provides

What we implement

Current Conditions is one of three different displays. The user can also get weather stats and a forecast.
Weather-O-Rama Interface

**WeatherData**

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

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

**Server part**

- **Subject**
  - registerObserver()
  - removeObserver()
  - notifyObservers()

**Client**

- **ConcreteSubject**
  - getState()
  - setState()

- **ConcreteObserver**
  - update()

- For all Observer's o:
  - o.update()

**Client part**

**Observer**

- update()
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:
  
  ```java
  update(Observable o, Object obj);
  ```

  ```java
  Subject
  registerObserver();
  removeObserver();
  notifyObservers();
  ```

  ```java
  Observer
  update();
  ```

  ```java
  ConcreteSubject
  getState();
  setState();
  ```

  ```java
  ConcreteObserver
  update();
  ```

  ```java
  for all Observer's o:
  o.update();
  ```

  ```java
  1..n
  ```

  ```java
  Client
  ```

  ```java
  ```