Using Design Patterns in Java Application Development

Princeton Java Users Group
September 19, 2007

Michael P. Redlich
(908) 730-3416
michael.p.redlich@exxonmobil.com

My Background (1)

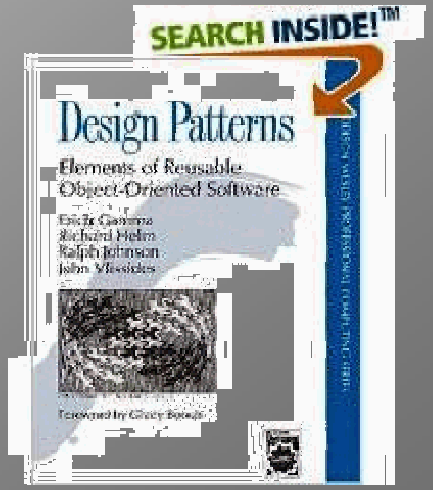
- Degree
  - B.S. in Computer Science
  - Rutgers University (go Scarlet Knights!)
- ExxonMobil Research & Engineering
  - Senior Research Technician (1988-1998, 2004-present)
  - Systems Analyst (1998-2002)
- Ai-Logix, Inc.
  - Technical Support Engineer (2003-2004)
- Amateur Computer Group of New Jersey (ACGNJ)
  - Java Users Group Leader (2001-present)
  - President (2007-present)
  - Secretary (2006)

My Background (2)

- Publications (co-authored with Barry Burd)
  - James: The Java Apache Mail Enterprise Server
  - Avoid Excessive Subclassing with the Decorator Design Pattern
  - Keeping Your Java Objects Informed with the Observer Design Pattern
  - Manufacturing Java Objects with the Factory Method Design Pattern
  - Resistance is Futile – How to Make Your Java Objects Conform with the Adapter Pattern
  - Get to Know Your Java Object's State of Mind with the State Pattern
  - Encapsulating Algorithms with the Template Method Design Pattern

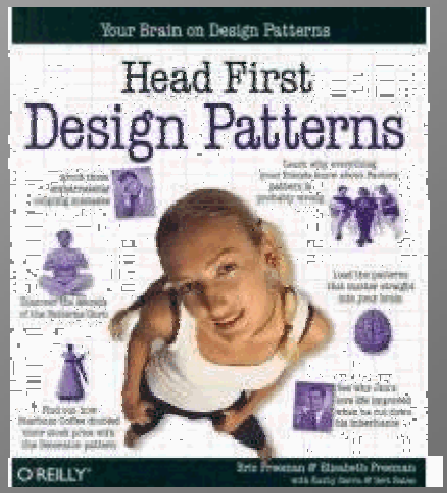
Gang-of-Four (GoF)

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides
- Design Patterns – Elements of Reusable Object-Oriented Software
  - ISBN 0-201-63361-2
  - 1995



Gang-of-Four (GoF) Next Generation?

- Eric Freeman
- Elisabeth Freeman
- Kathy Sierra
- Bert Bates
- Head First Design Patterns
  - ISBN 0-596-00712-4
  - 2004



What Are Design Patterns? (1)

- Recurring solutions to software design problems that are repeatedly found in real-world application development
- All about the design and interaction of objects
- Four essential elements:
  - The pattern name
  - The problem
  - The solution
  - The consequences

What Are Design Patterns? (2)

- A pattern is a solution to a problem in a context
- The context is the situation in which the pattern applies
- The problem refers to the desired goal in the context, but also refers to any constraints that may occur
- The solution is a general design that anyone can apply

"If you find yourself in a context with a problem that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a solution." -- Head First Design Patterns, page 579

How Can Design Patterns Solve Design Problems?

- Help identify less obvious abstractions
- Clients should only know about abstract classes that define an interface
- Reduce implementation dependencies
- Avoid creating objects directly
- Avoid dependencies on specific operations
- Avoid algorithmic dependencies
- Avoid tight coupling

Find appropriate objects

Program to an interface, not an implementation

Design for change

How Are Frameworks & Libraries Related?

- **Frameworks and Libraries:**
  - Provide a specific implementation
  - Most likely use Design Patterns
- **Design Patterns:**
  - Are at a higher level than frameworks and libraries
  - Help identify how to structure objects to solve problems

Thinking in Design Patterns

- Keep it simple
  - Goal should be simplicity
- Design Patterns are not a magic bullet
  - No “plug and play”
- Know when to apply a Design Pattern
  - Ensure that a pattern fits the design
- Consider a Design Pattern when refactoring
  - Goal is to improve structure, not behavior
- Don’t be afraid to remove a Design Pattern
  - Especially if design has become too complex

Design Pattern Categories

- Creational
  - Abstract the instantiation process
  - Dynamically create objects so that they don't have to be instantiated directly
- Structural
  - Compose groups of objects into larger structures
- Behavioral
  - Define communication among objects in a given system
  - Provide better control of flow in a complex application

Creational Patterns (1)

- Abstract Factory
  - Provides an interface for creating related objects without specifying their concrete classes
- Builder
  - Reuses the construction process of a complex object
- Factory Method
  - Lets subclasses decide which class to instantiate from a defined interface
- Prototype
  - Creates new objects by copying a prototype

Creational Patterns (2)

- **Singleton**
  - **Ensures a class has only one instance with a global point of access to it**

Structural Patterns (1)

- Adapter
  - Converts the interface of one class to an interface of another
- Bridge
  - Decouples an abstraction from its implementation
- Composite
  - Composes objects into tree structures to represent hierarchies
- Decorator
  - Attaches responsibilities to an object dynamically
- Façade
  - Provides a unified interface to a set of interfaces

Structural Patterns (2)

- **Flyweight**
  - Supports large numbers of fine-grained objects by sharing
- **Proxy**
  - Provides a surrogate for another object to control access to it

Behavioral Patterns (1)

- Chain of Responsibility
  - Passes a request along a chain of objects until the appropriate one handles it
- Command
  - Encapsulates a request as an object
- Interpreter
  - Defines a representation and an interpreter for a language grammar
- Iterator
  - Provides a way to access elements of an object sequentially without exposing its implementation

Behavioral Patterns (2)

- Mediator
  - Defines an object that encapsulates how a set of objects interact
- Memento
  - Captures an object's internal state so that it can be later restored to that state if necessary
- Observer
  - Defines a one-to-many dependency among objects
- State
  - Allows an object to alter its behavior when its internal state changes

Behavioral Patterns (3)

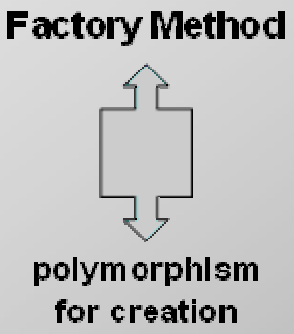
- **Strategy**
  - Encapsulates a set of algorithms individually and makes them interchangeable
- **Template Method**
  - Lets subclasses redefine certain steps of an algorithm
- **Visitor**
  - Defines a new operation without changing the classes on which it operates

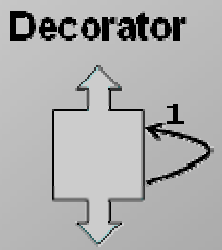
Design Principles

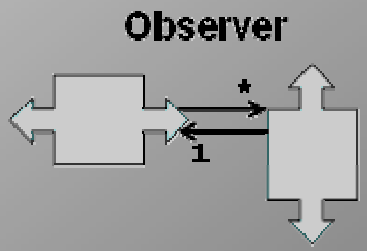
- Guidelines for addressing the issues of managing change in object-oriented applications development
- Understanding the basics of object-oriented programming isn't enough
  - Designs should be flexible, maintainable, and adapt to change
- Examples:
  - Depend upon abstractions, do not depend on concrete classes
  - Classes should be open for extension, but closed for modification
  - Strive for loosely coupled designs among objects that interact

So, Are You Ready...

- ...to review some of these Design Patterns?







Pizza Store Application

- Objectives
  - Design and develop a Pizza Store application that will create pizzas for customers
  - Consider plans for expansion

```
public class PizzaStore {

  public Pizza orderPizza(String type) {

    Pizza pizza = null;

    if(type.equals("cheese"))

      pizza = new CheesePizza();

    else if(type.equals("pepperoni"))

      pizza = new PepperoniPizza();

    pizza.prepare();

    pizza.bake();

    pizza.cut();

    pizza.box();

    return pizza;

  }

}
```

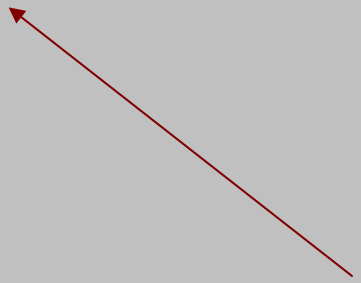
Area expected to change

Area expected to remain unchanged

IS THIS A GOOD APPROACH?

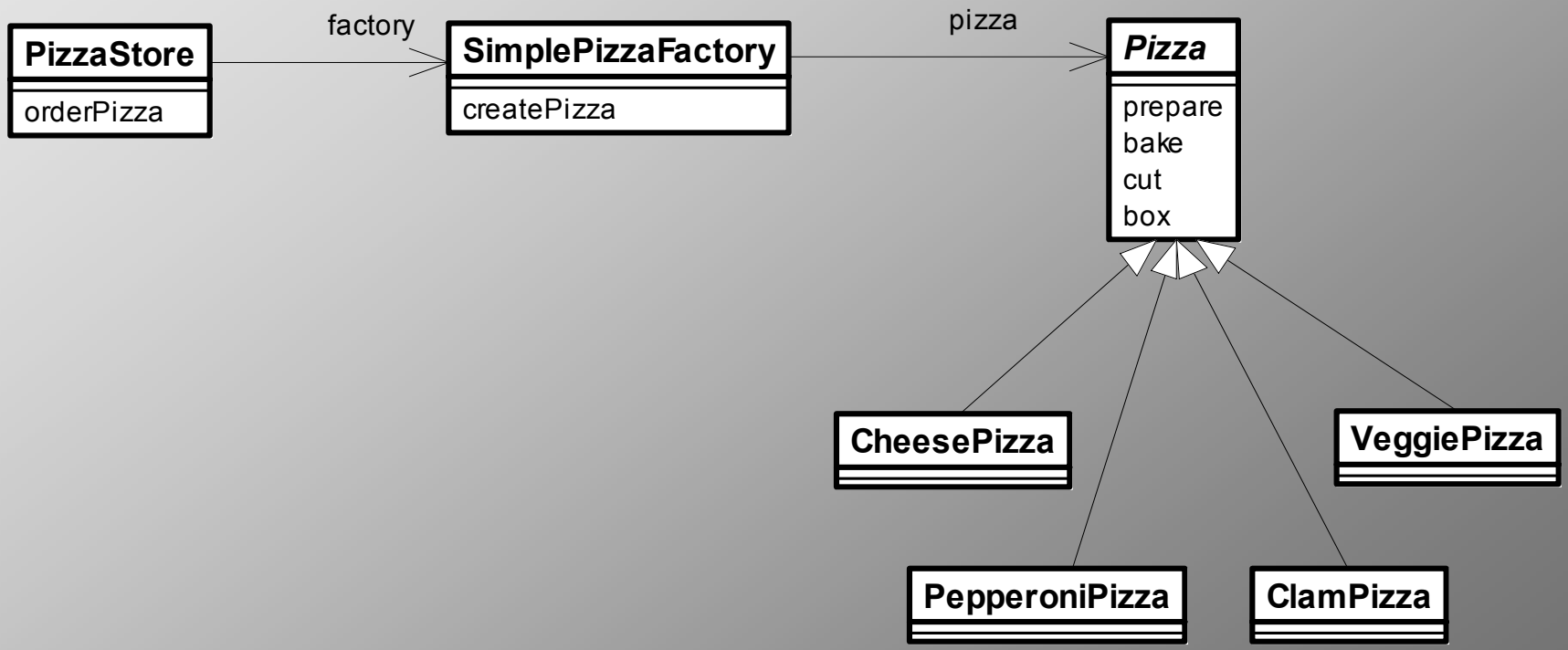
```
public class SimplePizzaFactory {
  public Pizza createPizza(String type) {
    Pizza pizza = null;
    if(type.equals("cheese"))
      pizza = new CheesePizza();
    else if(type.equals("pepperoni"))
      pizza = new PepperoniPizza();
    return pizza;
  }
}
```

```
public class PizzaStore {
  SimplePizzaFactory factory;
  public PizzaStore(SimplePizzaFactory factory) {
    this.factory = factory;
  }
  public Pizza orderPizza(String type) {
    Pizza pizza = factory.createPizza(type);
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

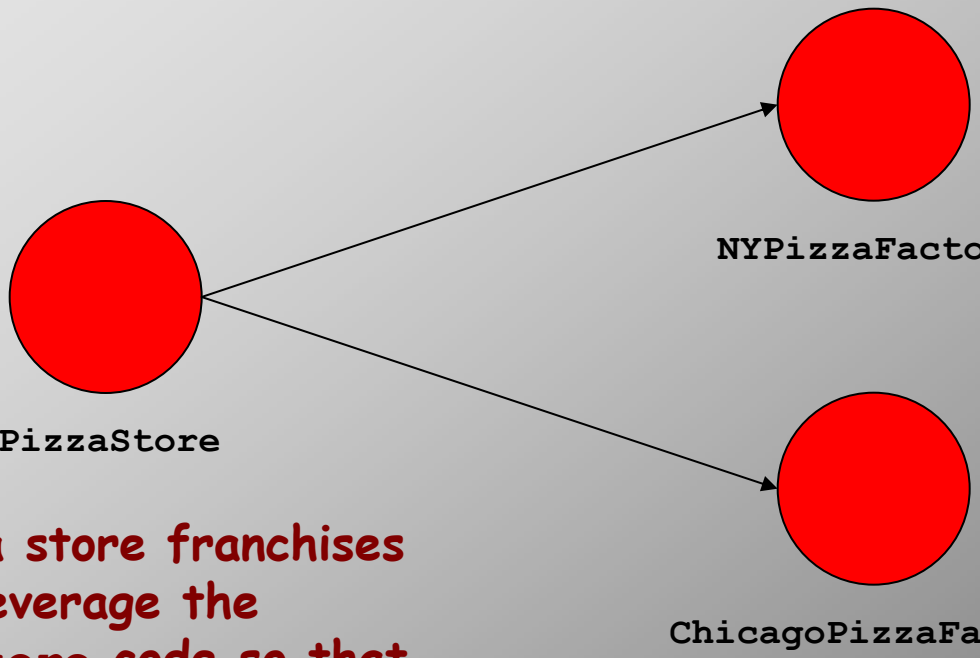


Notice how the `createPizza()` method eliminates the need to use the `new` keyword

A Simple Factory



Expanding the Pizza Store



This franchise likes to make pizza with thin crust, tasty sauce, and just a little cheese.

This franchise likes to make pizza with thick crust, rich sauce and lots of cheese.

All pizza store franchises should leverage the `PizzaStore` code so that pizzas are prepared the same way.

```
public class PizzaTestDrive {

  // instance variables...

  NYPizzaFactory nyFactory = new NYPizzaFactory();

  PizzaStore nyStore = new PizzaStore(nyFactory);

  nyStore.orderPizza("pepperoni");

  ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();

  PizzaStore chicagoStore = new PizzaStore(chicagoFactory);

  chicagoStore.orderPizza("pepperoni");

}
```

Here we get NY style pizza

Here we get Chicago style pizza

IS THIS A BETTER APPROACH?

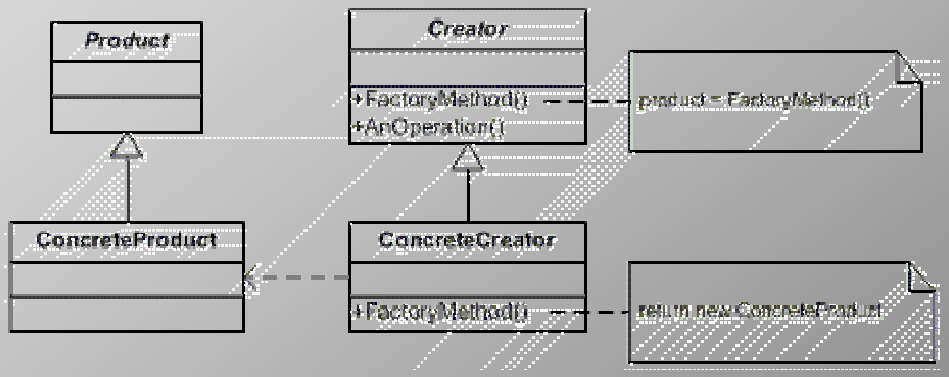
Factory Method (1)

- Intent
  - Defines an interface for creating an object, but lets subclasses decide which class to instantiate
  - Lets a class defer instantiation to subclasses
- Also known as
  - Virtual Constructor
- Motivation
  - To solve the problem of one class knowing *when* to create a class of another type, but not knowing *what kind* of class to create

Factory Method (2)

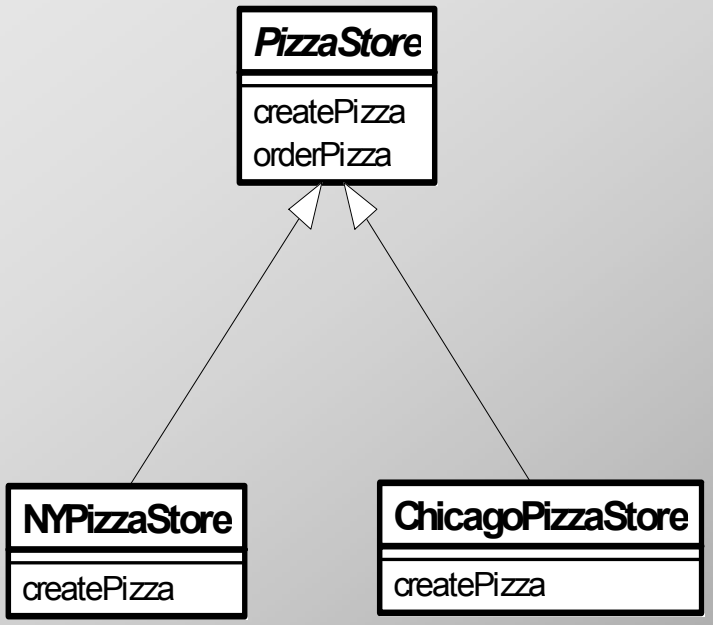
- Design Principle
  - Depend upon abstractions; do not depend upon concrete classes
  - The Dependency Inversion Principle
- Use this pattern when:
  - A class can't anticipate the class of objects is must create
  - A class would prefer for its subclasses to specify the objects it creates
  - There is a need for a class to localize one of several helper classes that can be delegated a responsibility

Factory Method (3)

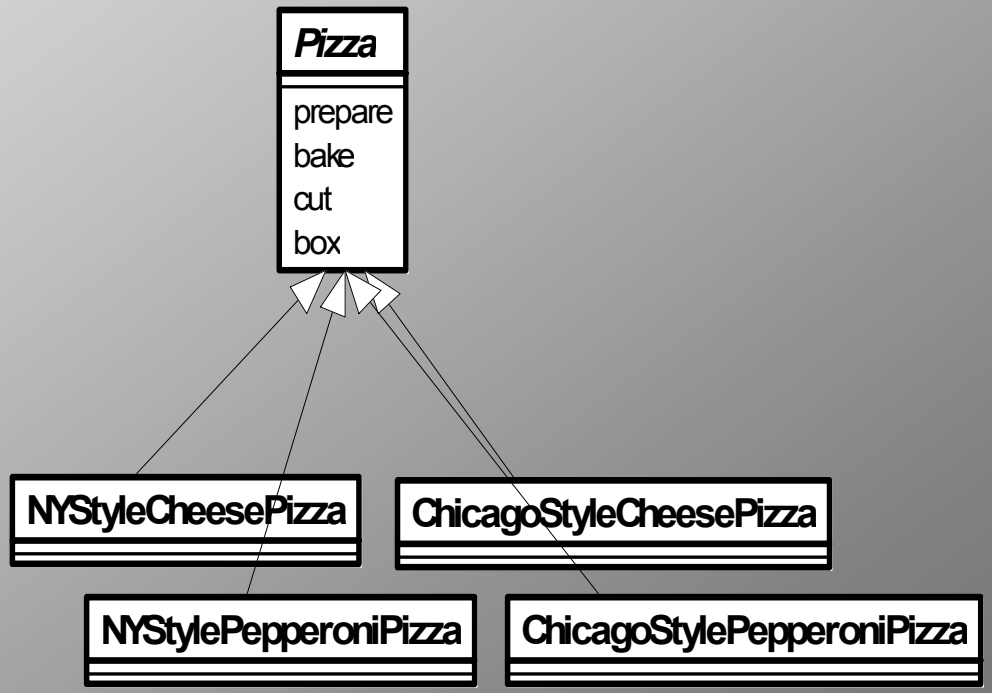


Pizza Store

Creator Classes



Product Classes



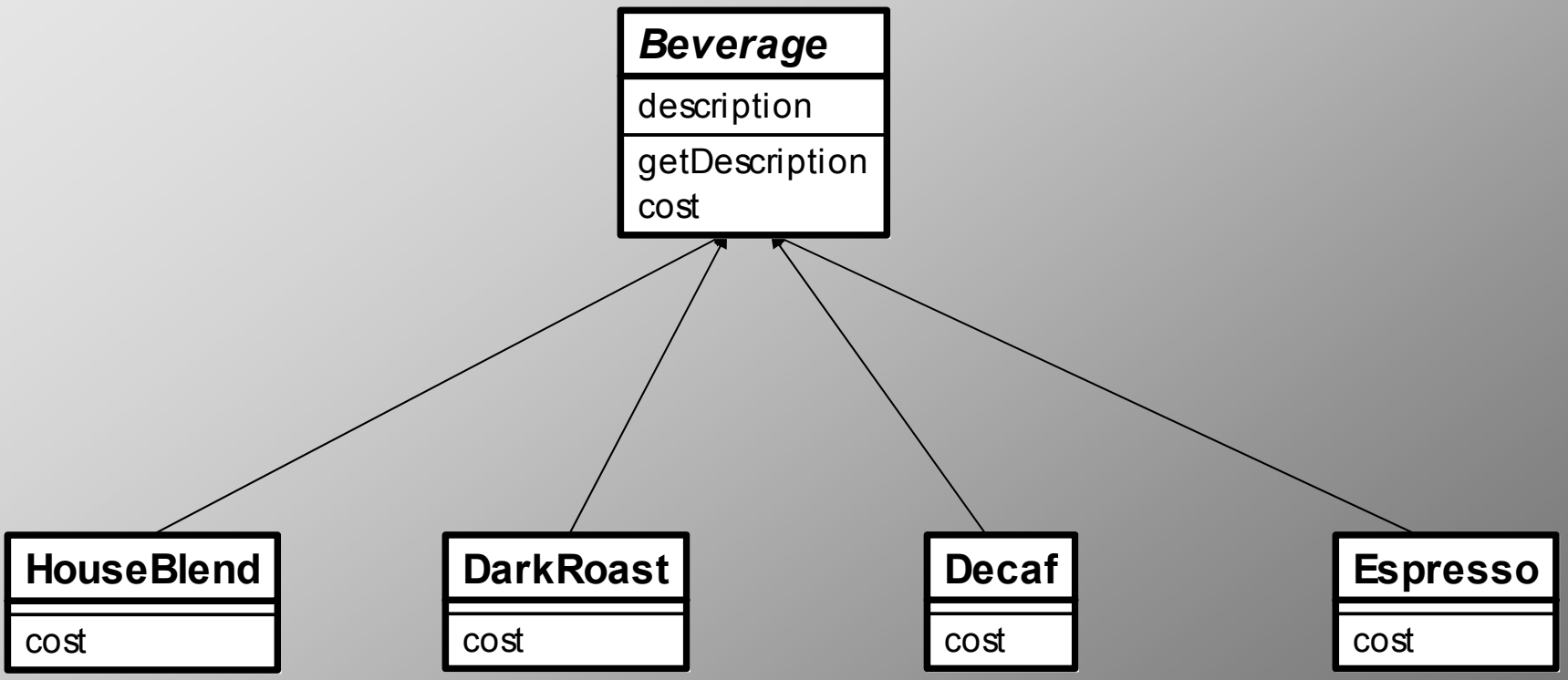
And Now...

- ...for the code review and demonstration of the Factory Method Design Pattern!

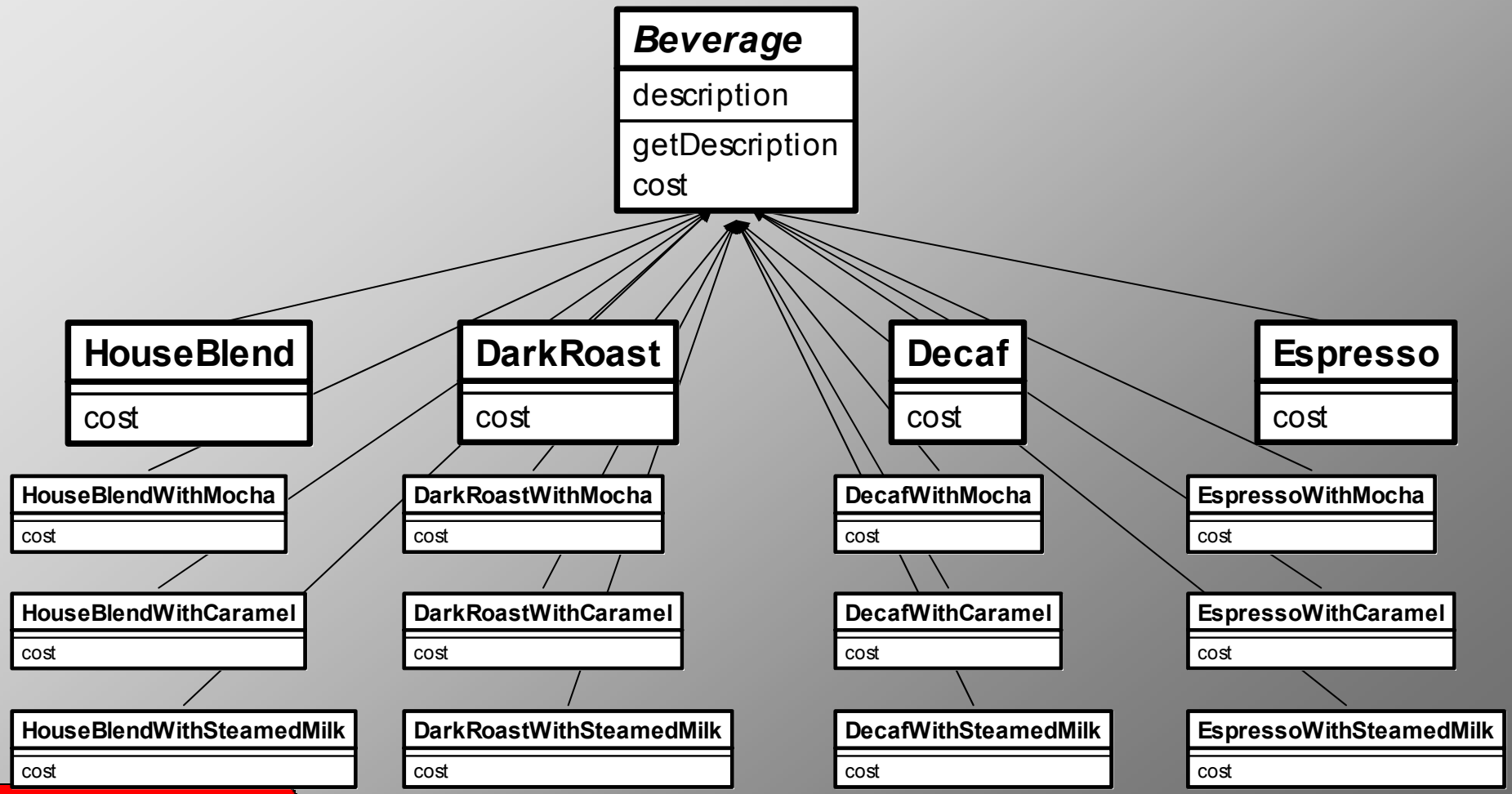
Coffee Shop Application (1)

- Objectives
  - Update an existing coffee shop application due to increase in product offering

Coffee Shop Application (2)

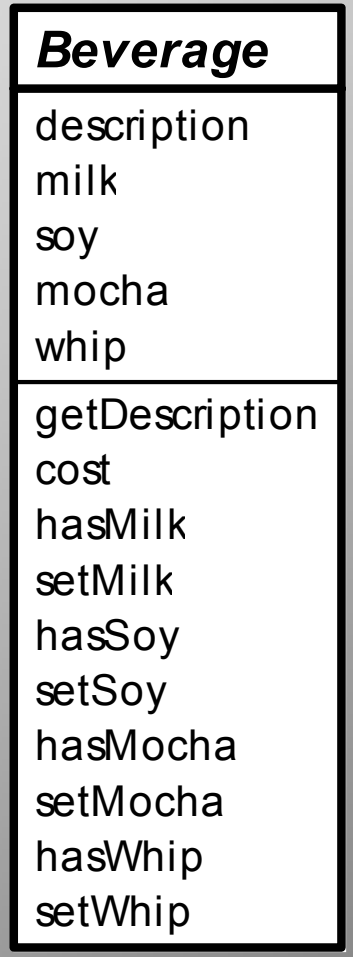


Coffee Shop Application (3)



Improving the Coffee Shop Application

What about using additional instance variables to keep track of the condiments?



IS THIS A BETTER DESIGN?

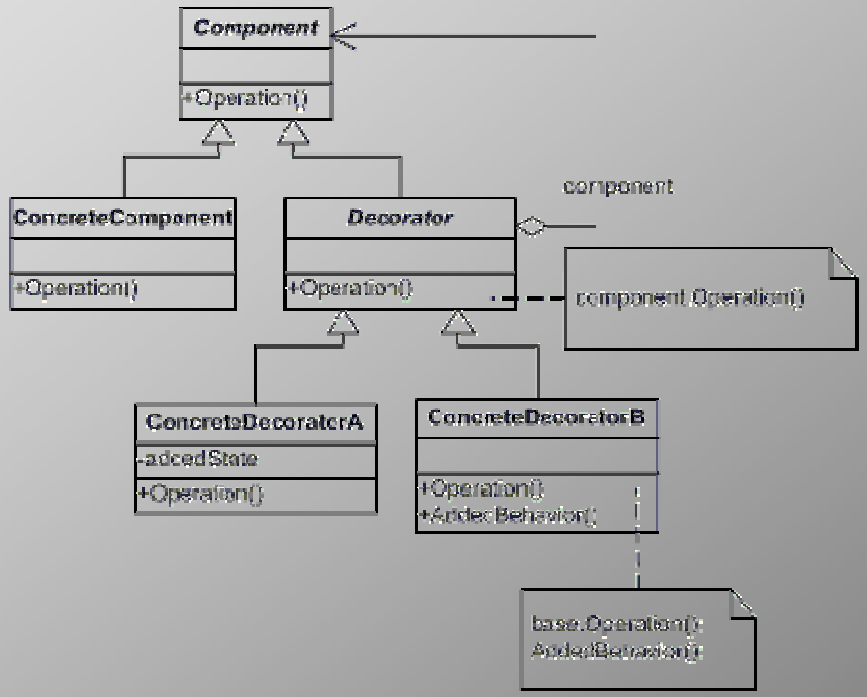
Decorator (1)

- Intent
  - Attaches additional responsibilities to an object dynamically
  - Provides a flexible alternative to subclassing for extending functionality
- Also known as
  - Wrapper
- Motivation
  - Allows classes to be easily extended to incorporate new behavior without modifying existing code
- Design Principle
  - Classes should be open for extension, but closed for modification

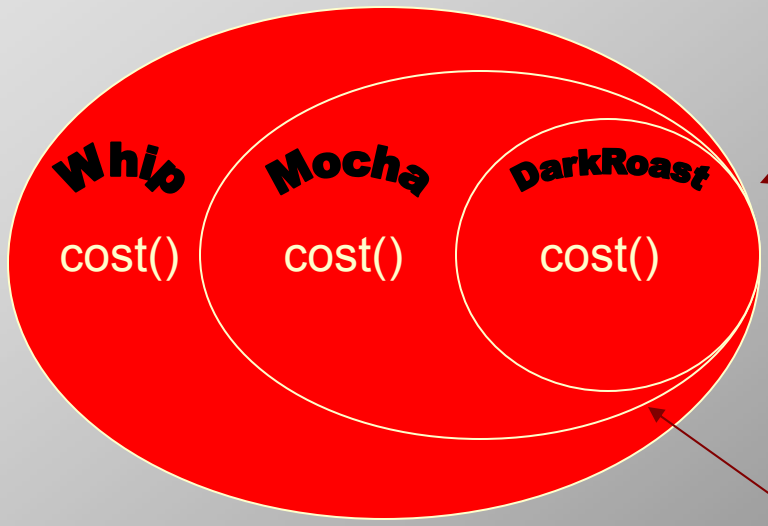
Decorator (2)

- **Use this pattern:**
  - **To add responsibilities to individual objects dynamically and transparently without affecting other objects**
  - **For responsibilities that can be withdrawn**
  - **When extension by subclassing is impractical**

Decorator (3)


Component
+Operation()
component
ConcreteComponent
+Operation()
Decorator
+Operation()
component.Operation()
ConcreteDecoratorA
-addedState
+Operation()
ConcreteDecoratorB
+Operation()
+AddedBehavior()
base.Operation();
AddedBehavior()

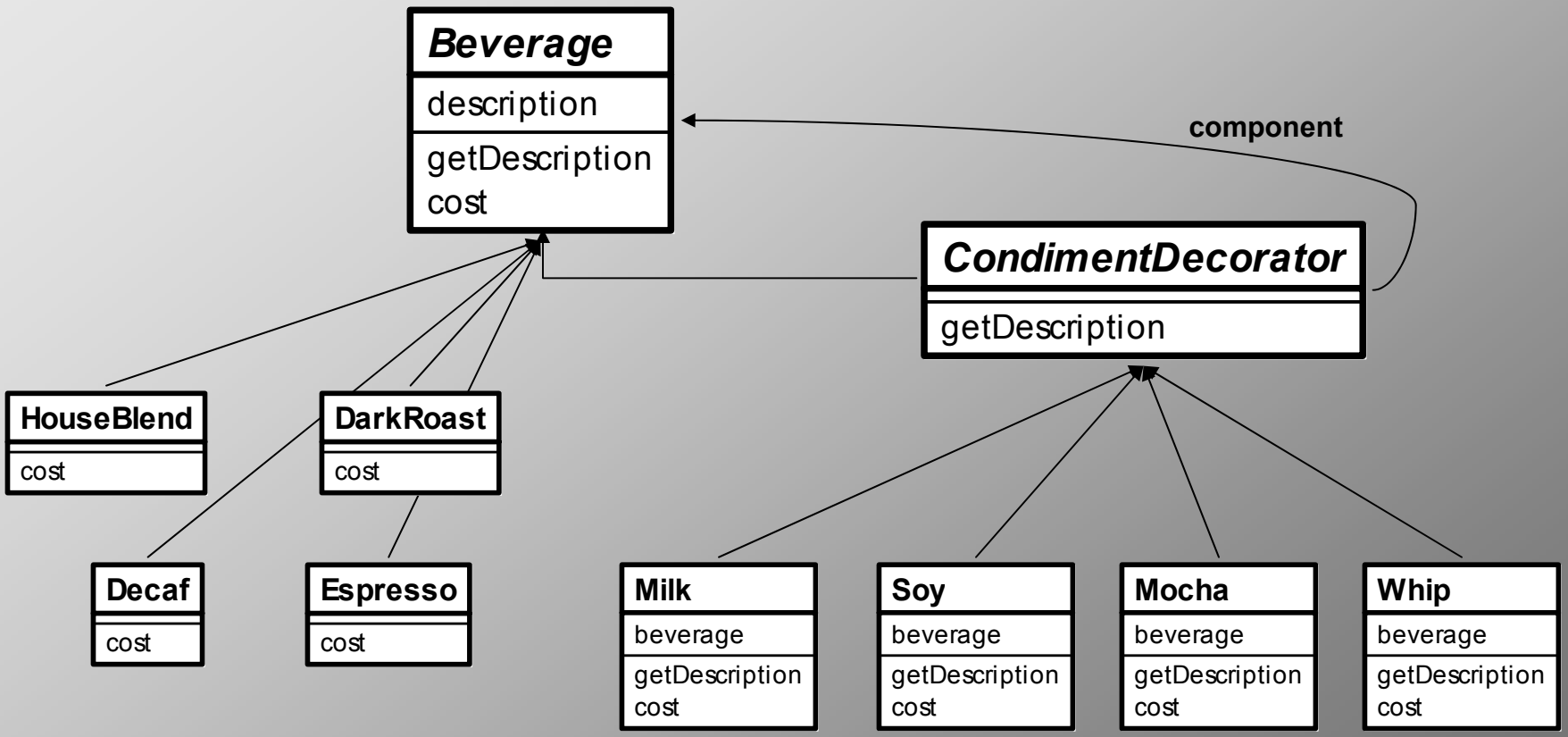
Constructing A Drink Order With Decorators



`DarkRoast` inherits from `Beverage` and has a `cost()` method that calculates the cost of the drink.

`Mocha` is a decorator that mirrors the object it is decorating, in this case, `Beverage`.

Coffee Shop Application (4)



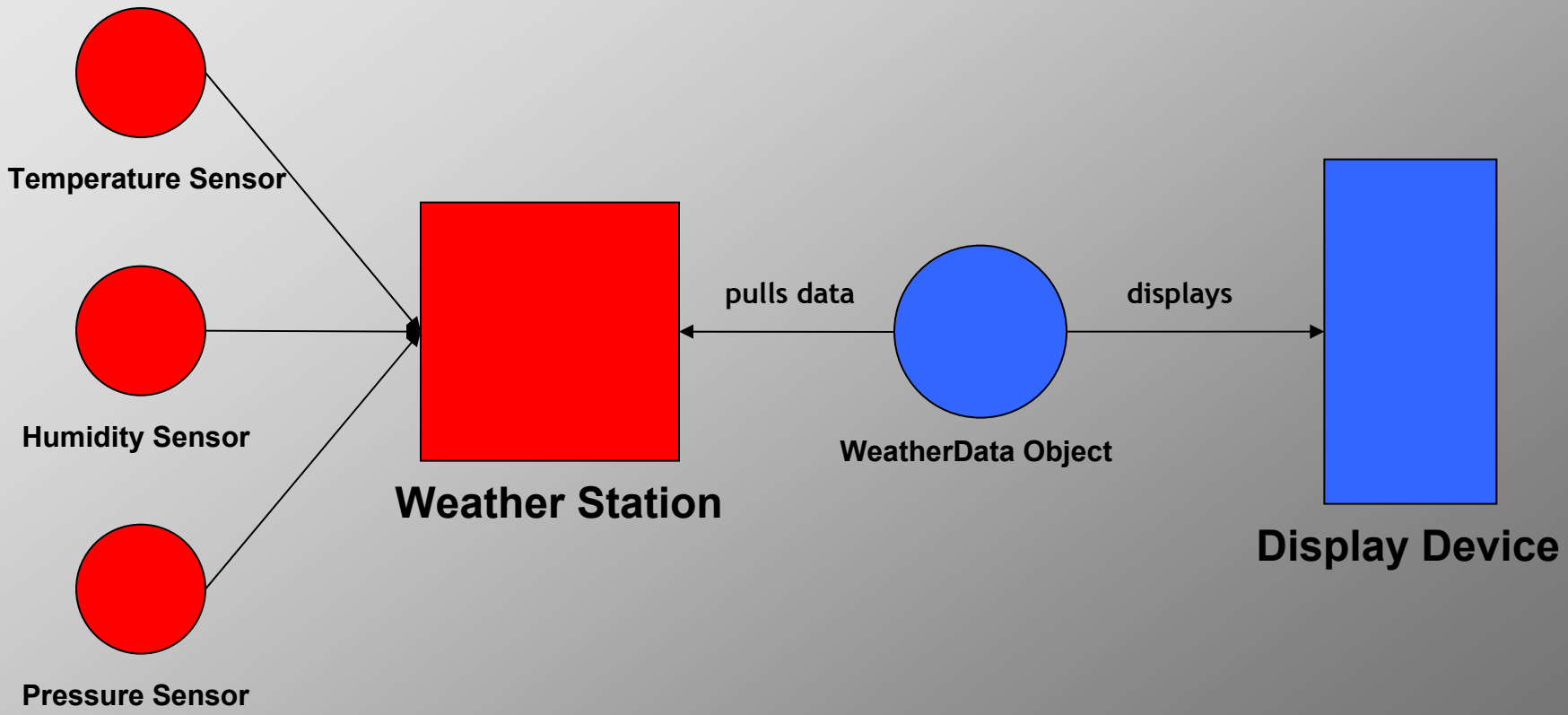
And Now...

- ...for the code review and demonstration of the Decorator Design Pattern!

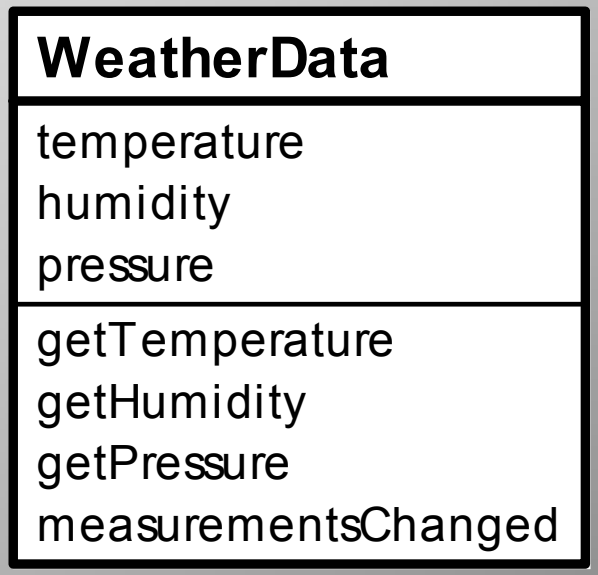
Weather Station Monitoring Application (1)

- Objectives
  - Design and develop a weather station monitoring application that pulls weather-related data from a weather station and displays it onto a device
  - Temperature, humidity, and barometric pressure will be monitored

Weather Station Monitoring Application (2)



WeatherData Object



```
public class WeatherData {
  // instance variables...
  public void measurementsChanged() {
    float temperature = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();
    currentConditions.update(temperature,humidity,pressure);
    statisticsDisplay.update(temperature,humidity,pressure);
    forecastDisplay.update(temperature,humidity,pressure);
  }
  // other methods here...
}
```

Area expected to change

Use of a common interface

Coding to concrete implementations

IS THIS A GOOD APPROACH?

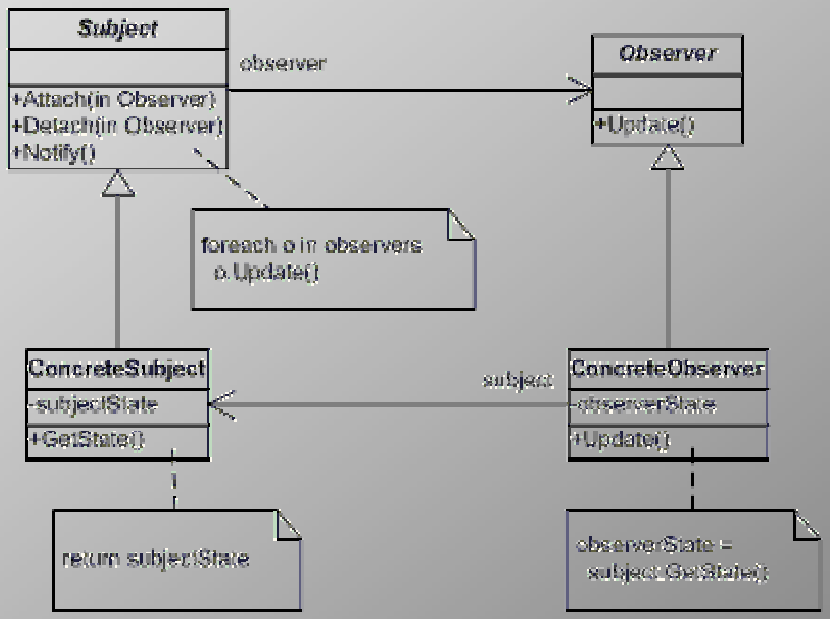
Observer (1)

- Intent
  - Defines a one-to-many dependency among objects so that when one object changes state, all its dependents are notified and updated automatically
  - A way of notifying change to a number of classes
- Also known as
  - Dependents
  - Publish-Subscribe
- Motivation
  - To avoid making classes tightly coupled that would reduce their reusability

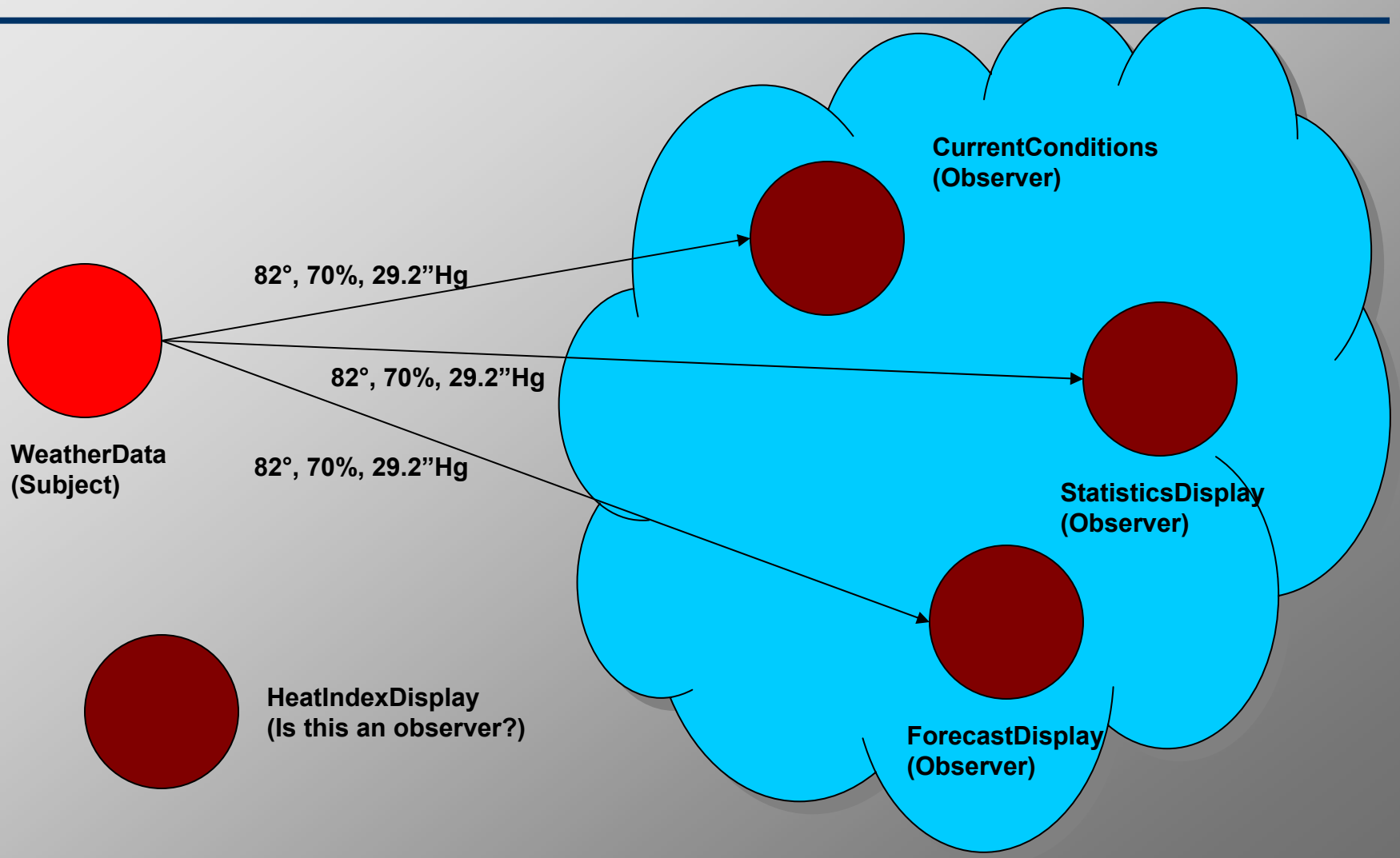
Observer (2)

- Design Principle
  - Strive for loosely coupled designs among objects that interact
- Use this pattern when:
  - A change to one object requires changing others, and the number of objects to be changed is unknown
  - An object should be able to notify other objects without making assumptions about who these objects are
    - Avoids having these objects tightly coupled

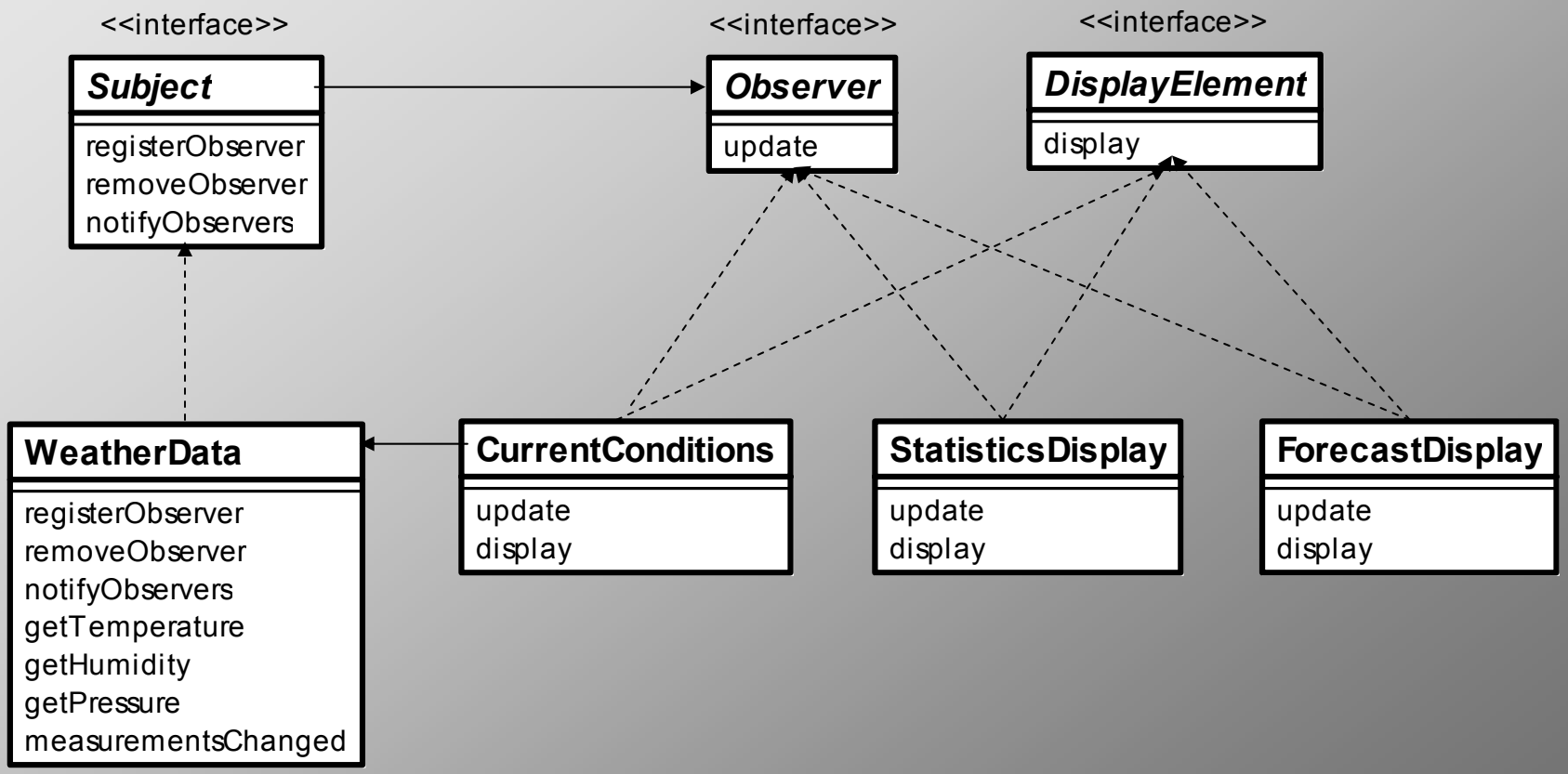
Observer (3)


Subject
observer
Observer
+Attach(in Observer)
+Detach(in Observer)
+Notify()
+Update()
foreach o in observers
o.Update()
ConcreteSubject
-subjectState
+GetState()
subject
ConcreteObserver
-observerState
+Update()
return subjectState
observerState =
subject.GetState()

Publisher + Subscriber = Observer



Weather Station Monitoring Application (3)



And Now...

- ...for the code review and demonstration of the Observer Design Pattern!

Resources (1)

- Princeton Java Users Group
  - Facilitated by Yakov Fain
  - http://www.myflex.org/princetonjug/
- ACGNJ Java Users Group
  - Facilitated by Mike Redlich
  - http://www.javasig.org/
- Capital District Java Users Group
  - Facilitated by John Andrew
  - http://www.cdjdn.com/

Resources (2)

- NYJavaSIG
  - Facilitated by Frank Greco
  - http://www.javasig.com/
- NYSIA Java Users Group
  - Facilitated by Ajanta Phatak
  - http://www.nysia.org/events/SIGpgs/sigvil.cfm?sid=33

Further Reading (1)

- Design Patterns – Elements of Reusable Object-Oriented Software
  - Erich Gamma, et. al
  - ISBN 0-201-63361-2
- Head First Design Patterns
  - Eric & Elisabeth Freeman (with Kathy Sierra & Bert Bates)
  - ISBN 0-596-00712-4
- Java Design Patterns
  - James W. Cooper
  - ISBN 0-201-48539-7

Further Reading (2)

- UML Distilled
  - Martin Fowler (with Kendall Scott)
  - ISBN 0-201-32563-2
- Data & Object Factory
  - http://www.dofactory.com/