CMPSCI 220 Programming Methodology

12: Design Patterns Part 1

(Observer and Decorator)

Based on Head First Design Patterns

Objectives

Observer Pattern

- Learn how objects can "observe" other objects.
- Apply the observer pattern in Scala.

Decorator Pattern

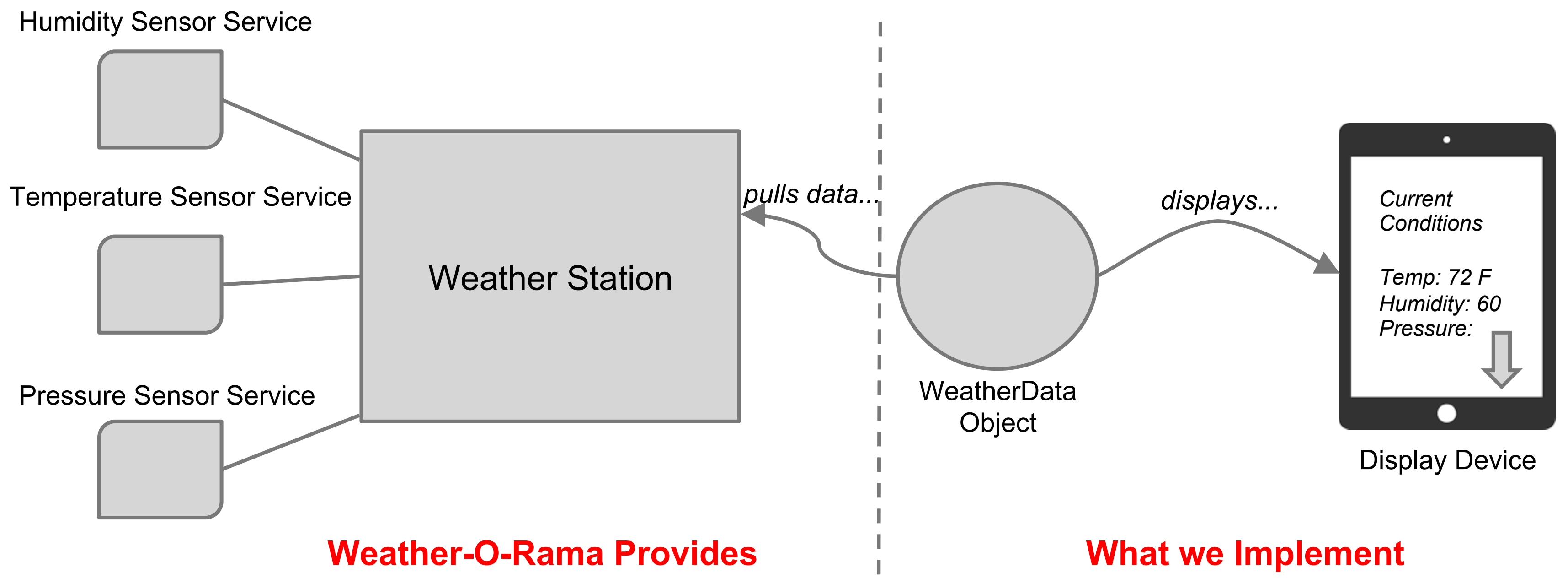
- Learn how objects can gain new responsibilities at runtime.
- Apply the decorator pattern in Scala.

i-clicker Question!

Why should you favor composition over inheritance?

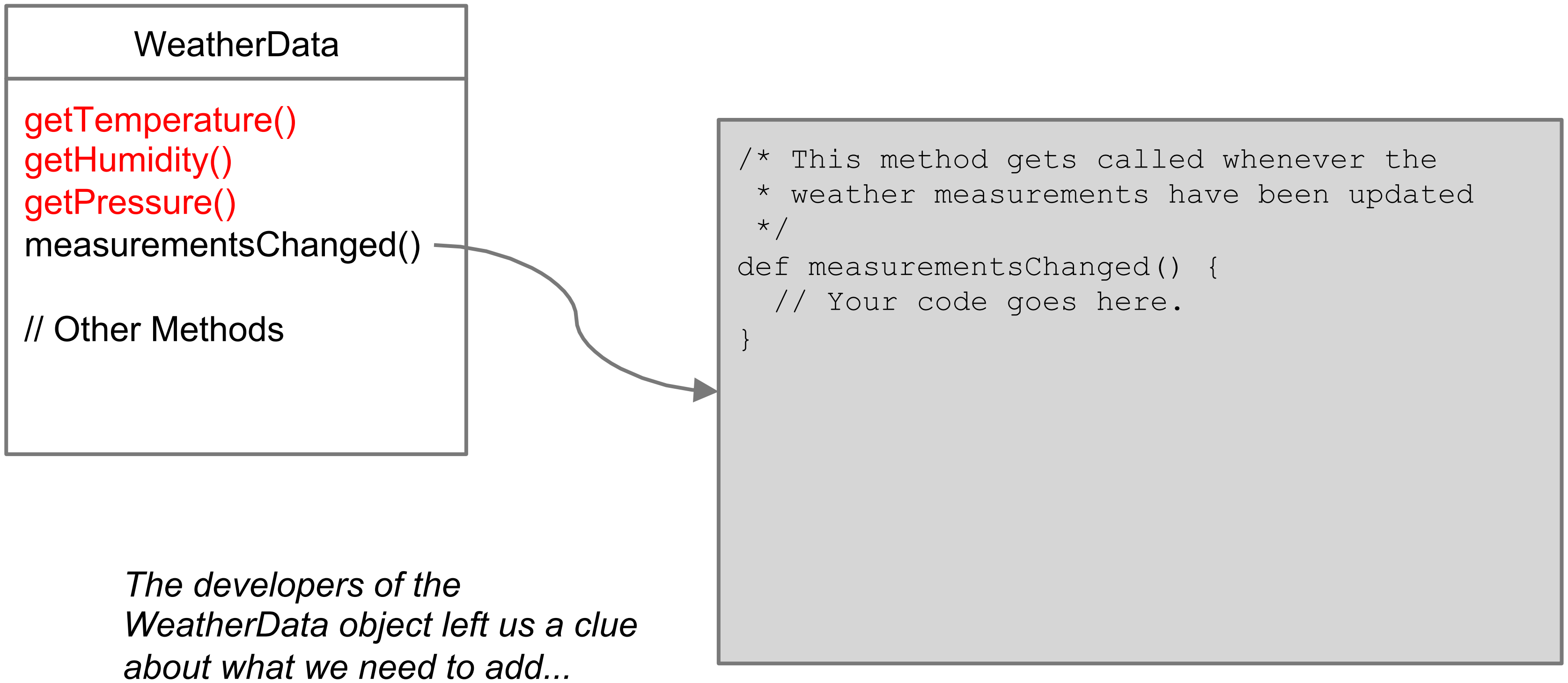
A. It is in fashion
B. It leads to a more flexible class design
C. It allows you to create classes automatically
D. It lets you defer implementation to super classes
E. I am not sure

Observer: Weather Monitoring Application



Our job, if we choose to accept it, is to create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.

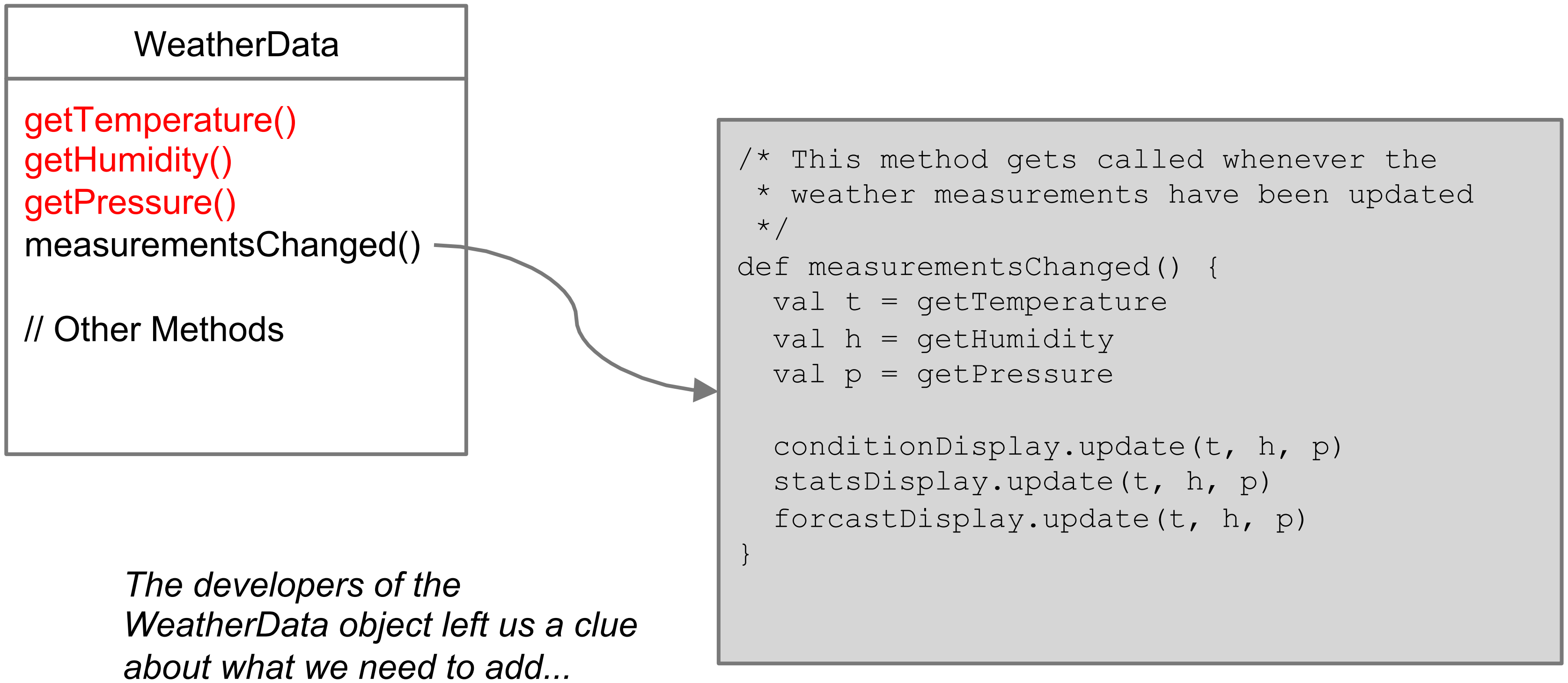
Weather Monitoring App: First Try...



```
/* This method gets called whenever the
 * weather measurements have been updated
 */
def measurementsChanged() {
  // Your code goes here.
}
```

The developers of the WeatherData object left us a clue about what we need to add...

Weather Monitoring App: First Try...

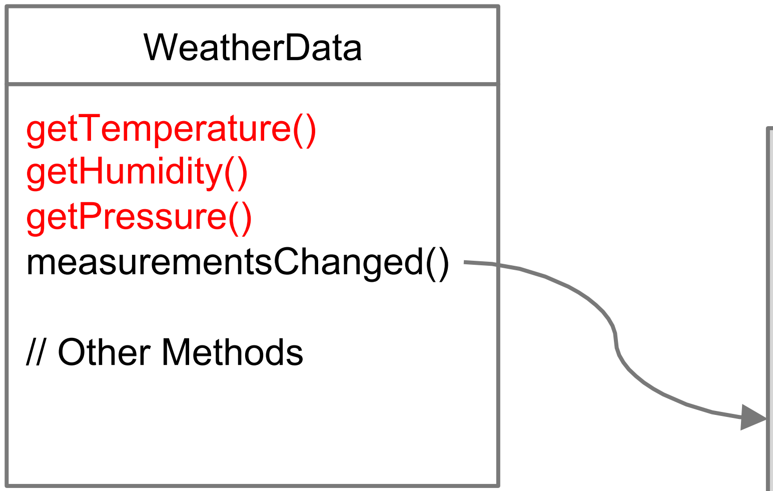


```
/* This method gets called whenever the
 * weather measurements have been updated
 */
def measurementsChanged() {
  val t = getTemperature
  val h = getHumidity
  val p = getPressure

  conditionDisplay.update(t, h, p)
  statsDisplay.update(t, h, p)
  forcastDisplay.update(t, h, p)
}
```

The developers of the WeatherData object left us a clue about what we need to add...

Weather Monitoring App: First Try...



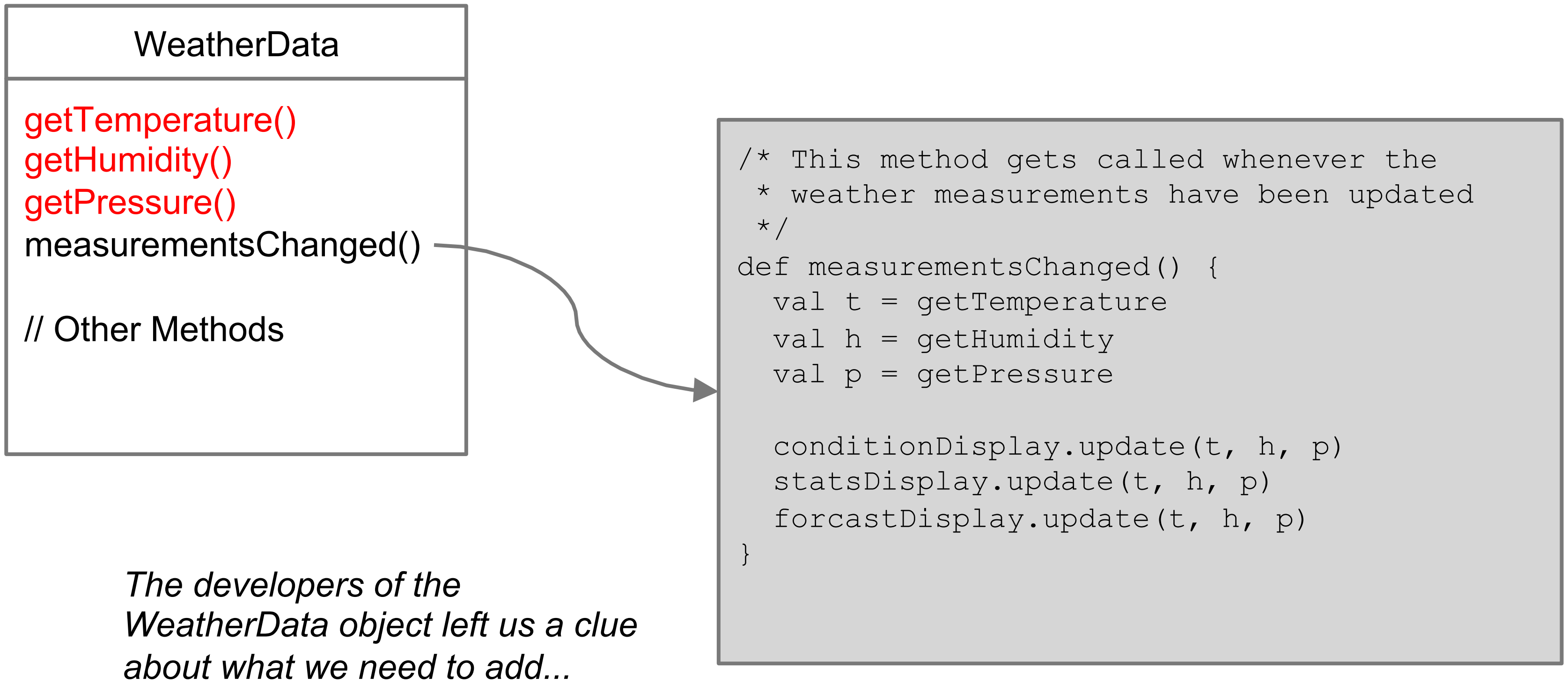
```
/* This method gets called whenever the
 * weather measurements have been updated
 */
def measurementsChanged() {
  val t = getTemperature
  val h = getHumidity
  val p = getPressure

  conditionDisplay.update(t, h, p)
  statsDisplay.update(t, h, p)
  forcastDisplay.update(t, h, p)
}
```

The developers of the WeatherData object left us a clue about what we need to add...

What is wrong with this implementation?

Weather Monitoring App: First Try...



```
/* This method gets called whenever the
 * weather measurements have been updated
 */
def measurementsChanged() {
  val t = getTemperature
  val h = getHumidity
  val p = getPressure

  conditionDisplay.update(t, h, p)
  statsDisplay.update(t, h, p)
  forcastDisplay.update(t, h, p)
}
```

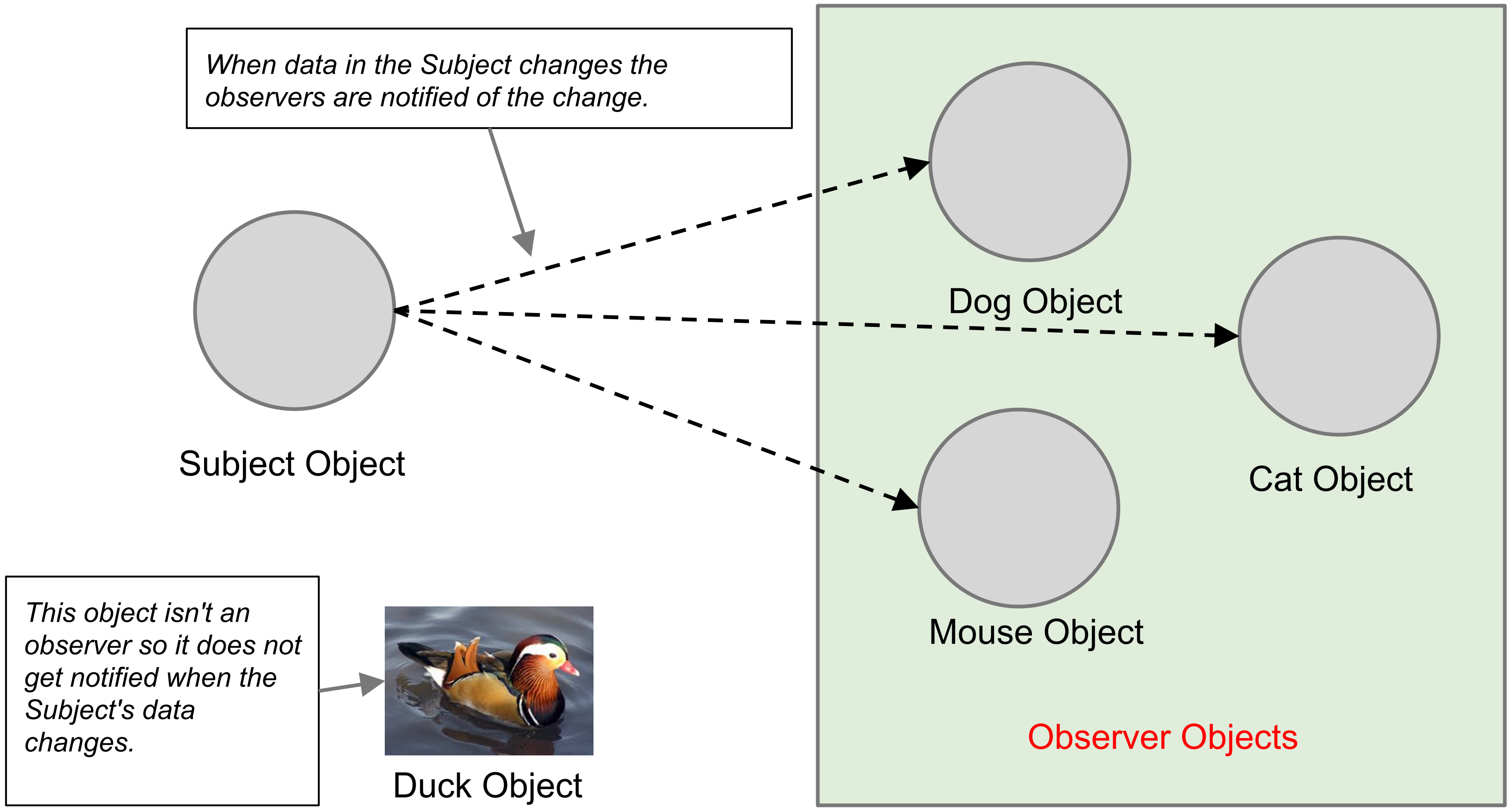
The developers of the WeatherData object left us a clue about what we need to add...

We have hardcoded the display elements! Yuck!

Observer Pattern Defined

- A newspaper publisher goes into business and begins publishing newspapers.
- You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you - as long as you remain a subscriber.
- You unsubscribe when you don't want the newspaper anymore, and they stop being delivered.
- While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.

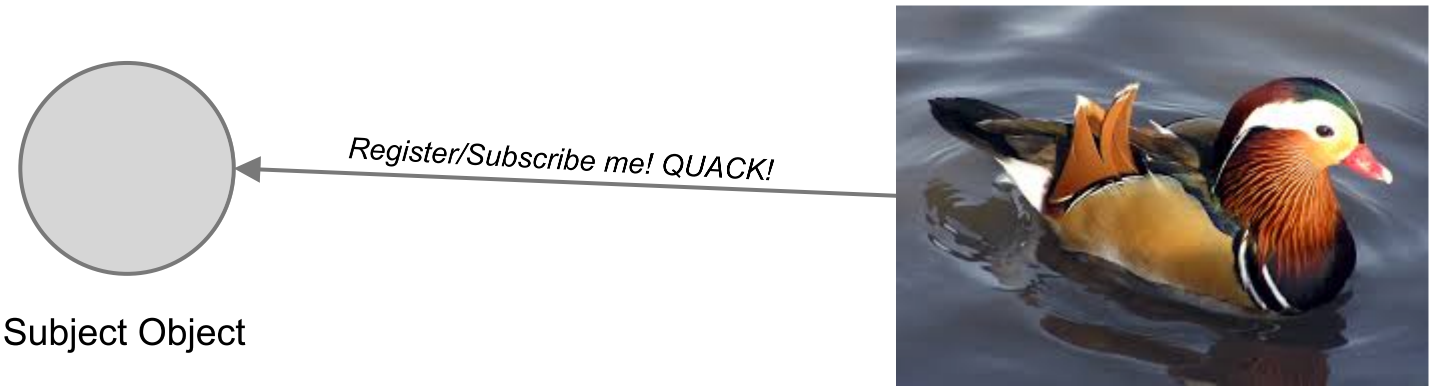
Publisher + Subscriber = Observer Pattern


When data in the Subject changes the observers are notified of the change.

This object isn't an observer so it does not get notified when the Subject's data changes.

Subscribing...

A duck object comes along and tells the Subject object that it wants to become an observer...

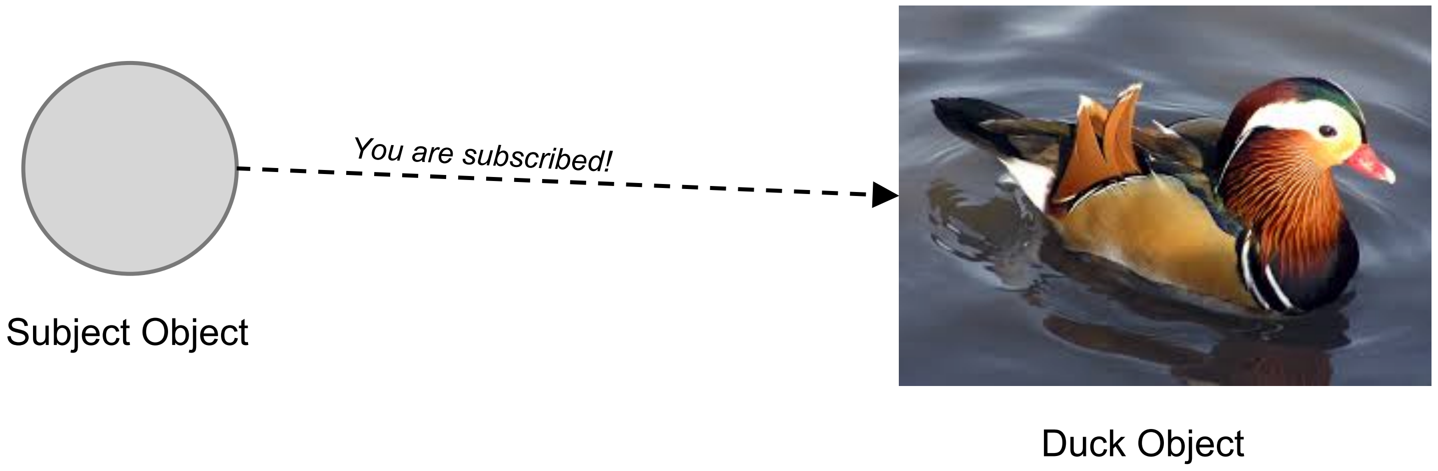


Subject Object

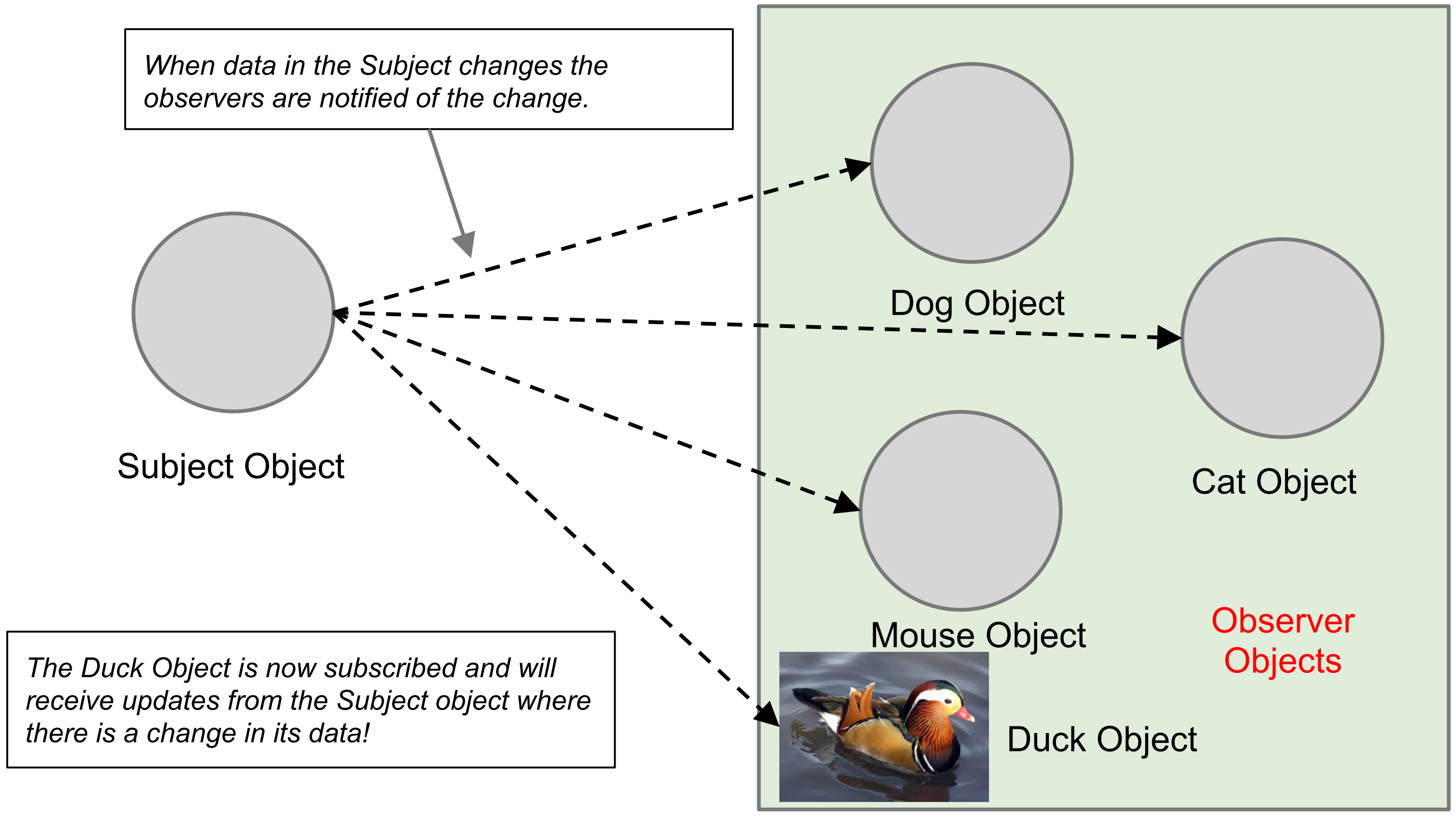
Duck Object

Subscribing...

A duck object comes along and tells the Subject object that it wants to become an observer...

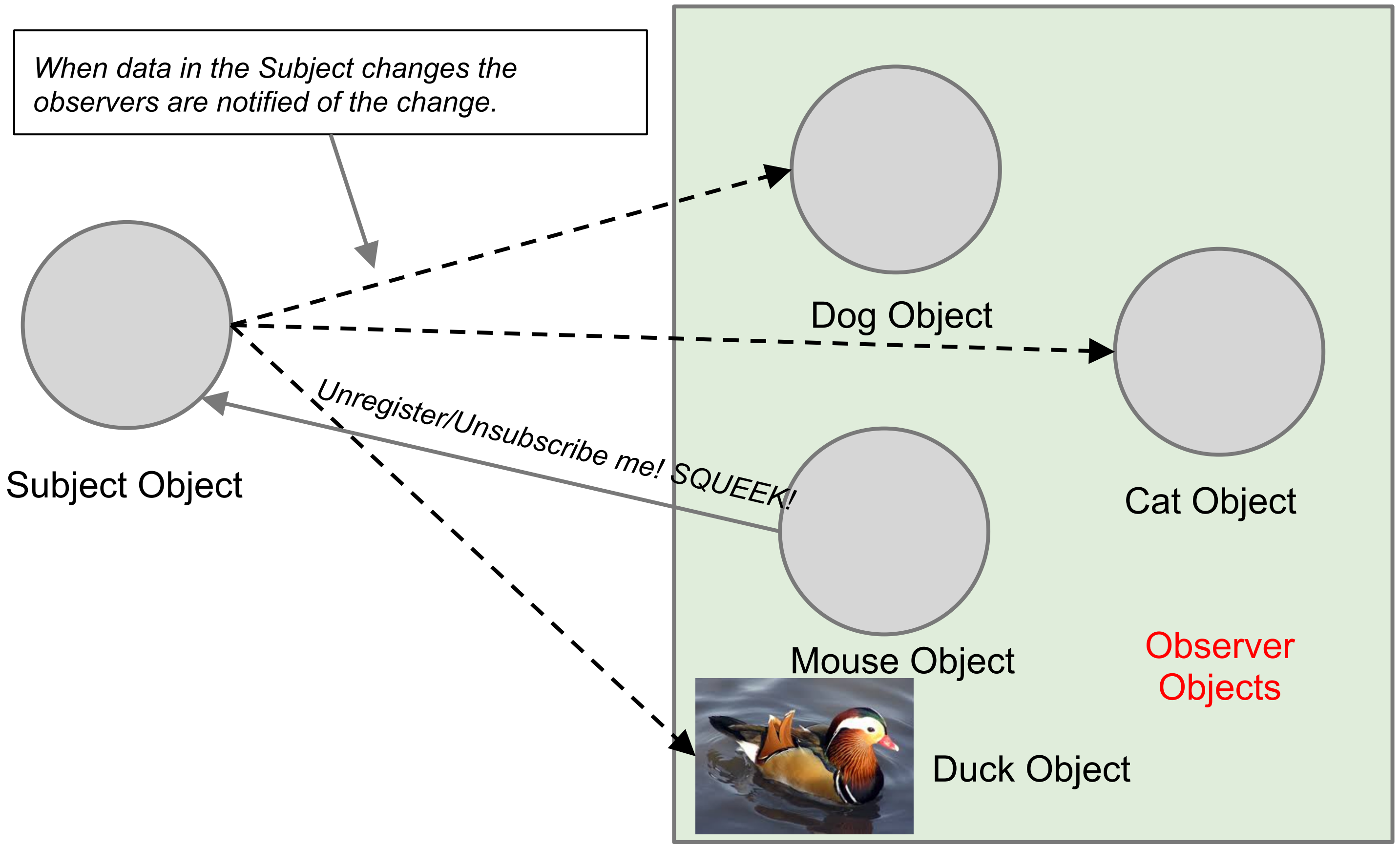


Publisher + Subscriber = Observer Pattern

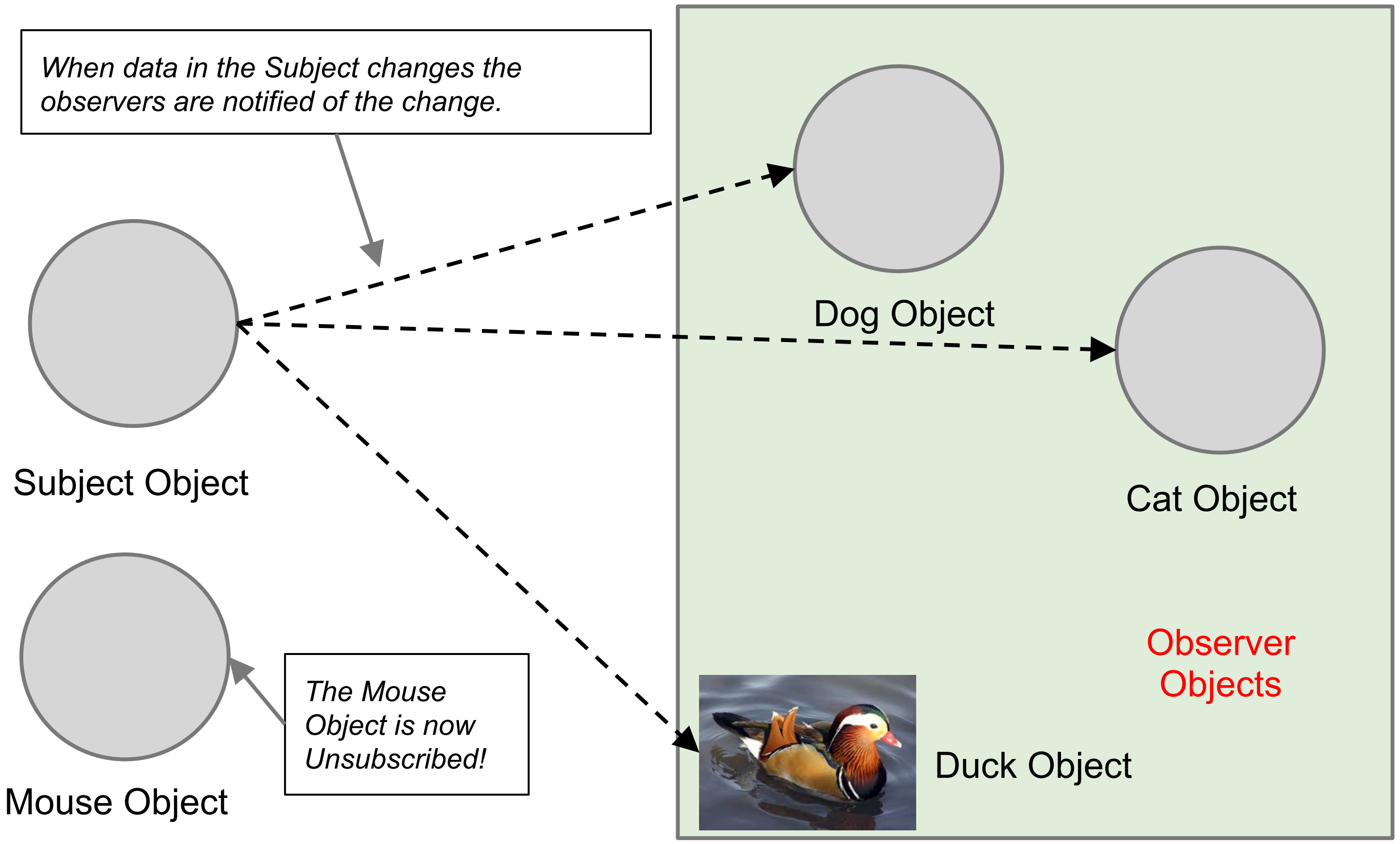

When data in the Subject changes the observers are notified of the change.

The Duck Object is now subscribed and will receive updates from the Subject object where there is a change in its data!

Publisher + Subscriber = Observer Pattern

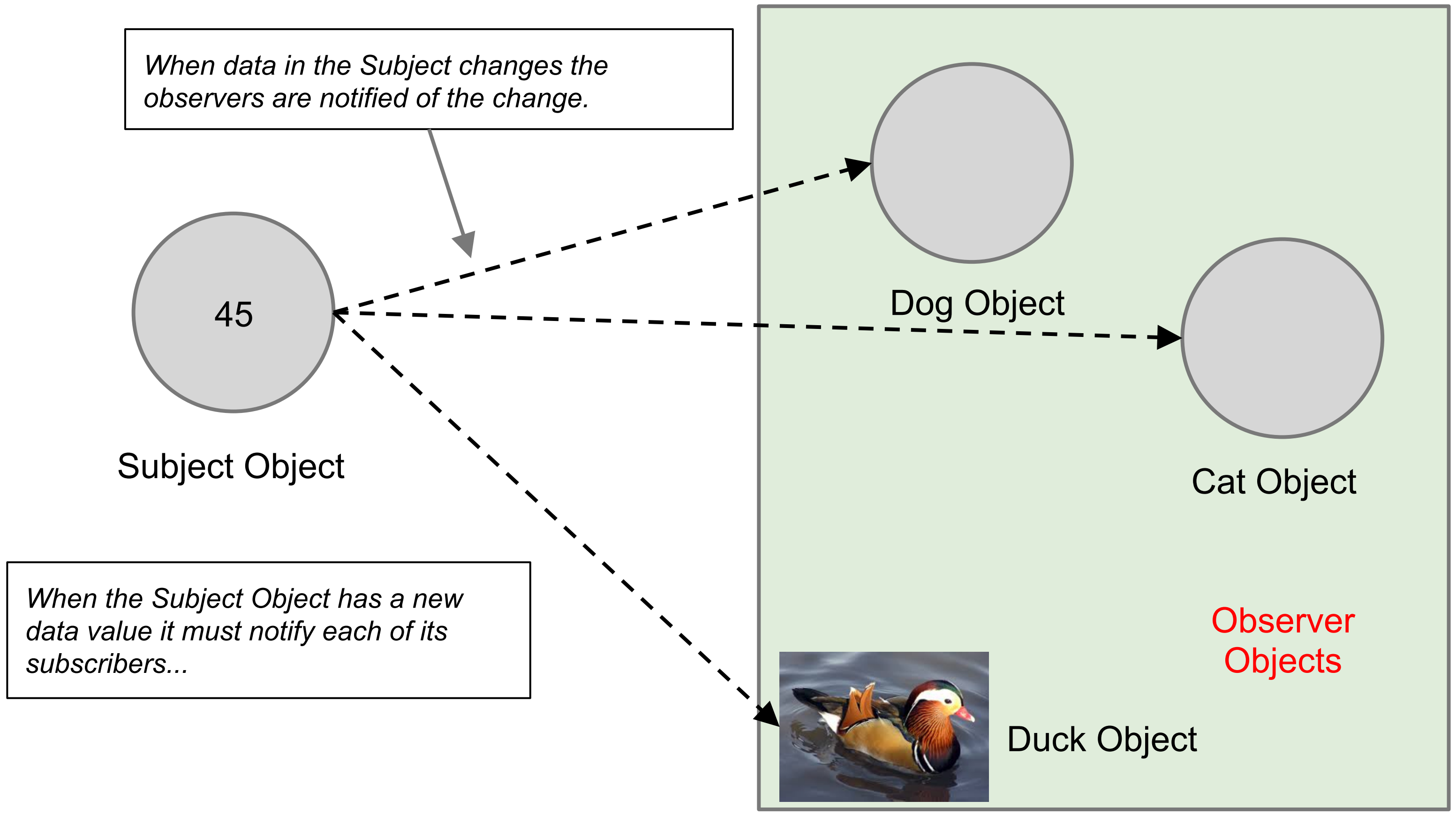

When data in the Subject changes the observers are notified of the change.
Subject Object
Dog Object
Cat Object
Mouse Object
Duck Object
Unregister/Unsubscribe me! SQUEEK!
Observer Objects

Publisher + Subscriber = Observer Pattern



When data in the Subject changes the observers are notified of the change.
Subject Object
Dog Object
Cat Object
Observer Objects
Duck Object
Mouse Object
The Mouse Object is now Unsubscribed!

Publisher + Subscriber = Observer Pattern



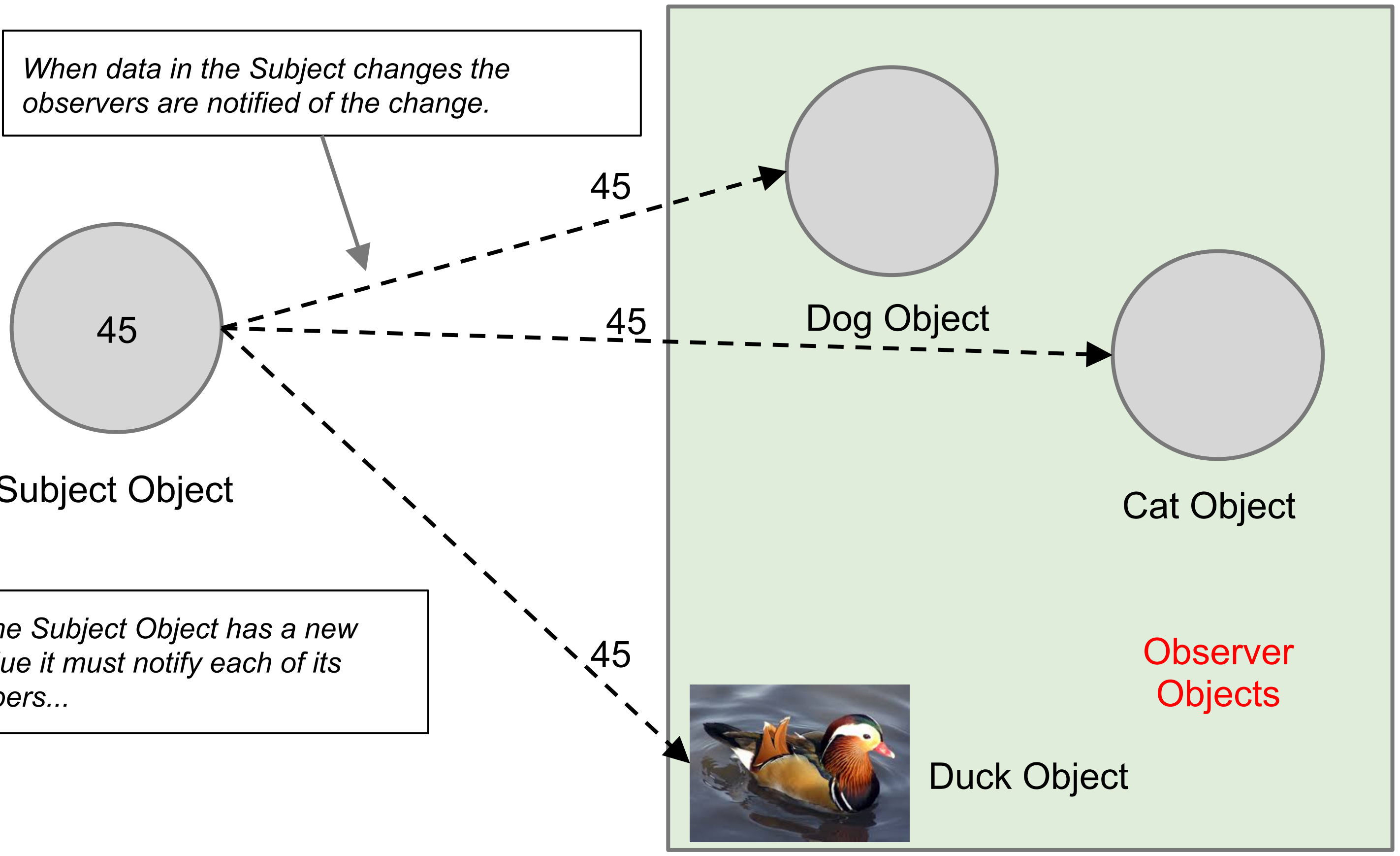
When data in the Subject changes the observers are notified of the change.

When the Subject Object has a new data value it must notify each of its subscribers...

Publisher + Subscriber = Observer Pattern

When data in the Subject changes the observers are notified of the change.

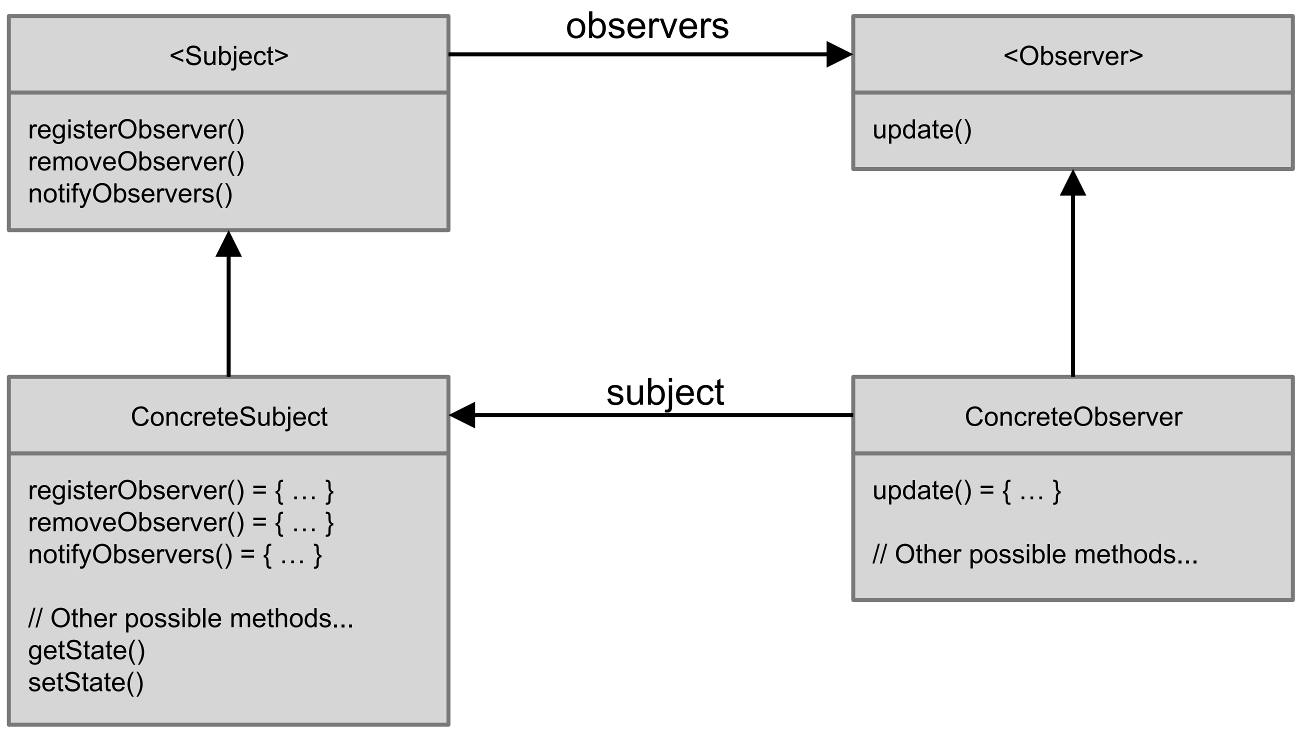
When the Subject Object has a new data value it must notify each of its subscribers...



The Observer Pattern Defined

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Observer Class Diagram



*When two objects are **loosely coupled**, they can interact, but have very little knowledge of each other.*

*The **observer pattern** provides an object design where subjects and observers are loosely coupled.*

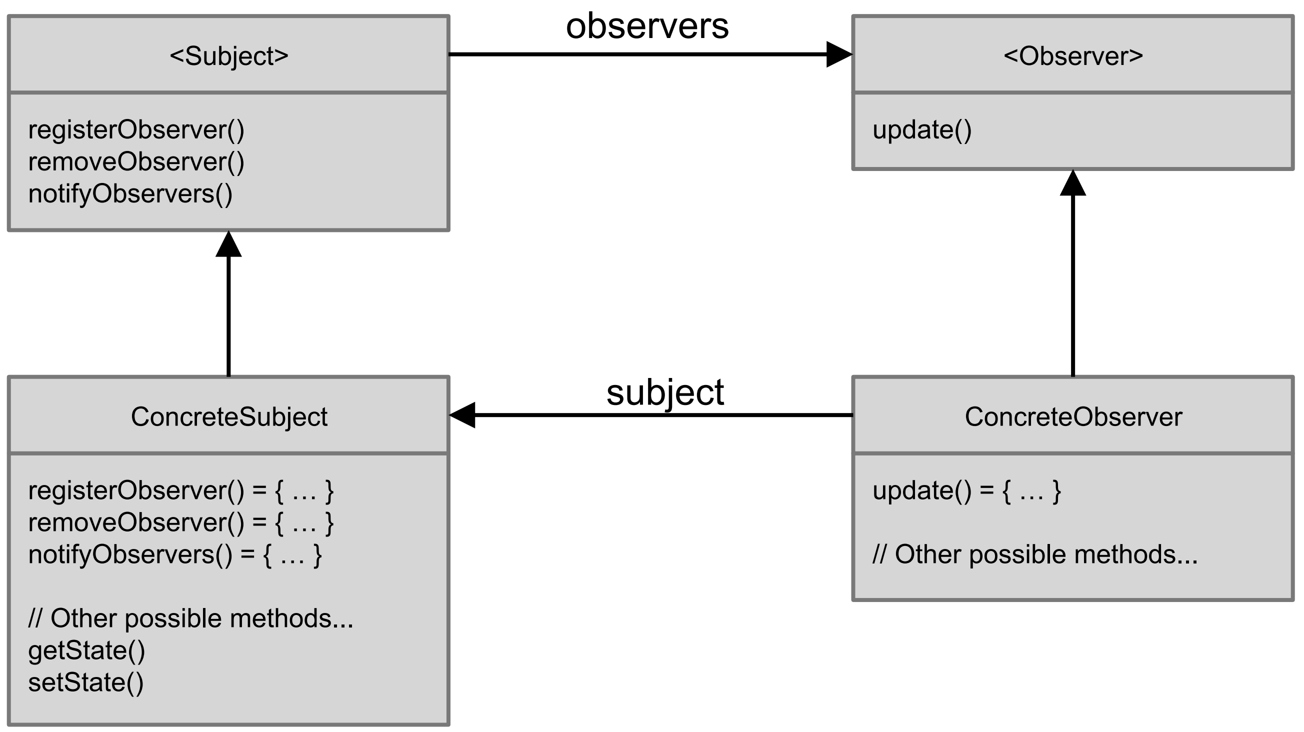
Designing the Weather Station: *Try Sketching it Out!*

Before moving on, try sketching out the classes/traits you'll need to implement the Weather Station, including the WeatherData class and its display elements. Make sure your diagram shows how all the pieces fit together and also how another developer might implement her own display element.

You can do this with the people around you…

We will collect a paper from each of you at the end of class.

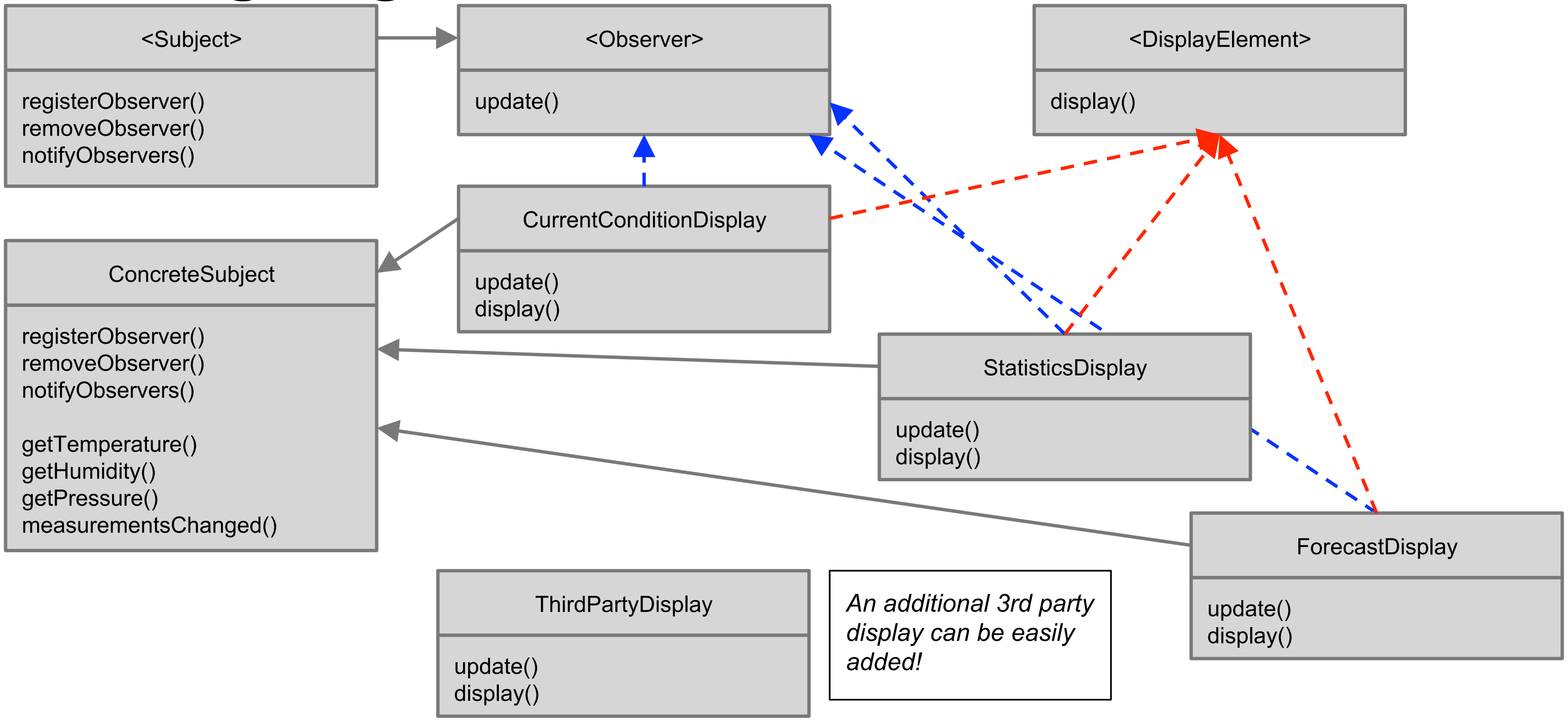
This Might Help: Observer Class Diagram



*When two objects are **loosely coupled**, they can interact, but have very little knowledge of each other.*

*The **observer pattern** provides an object design where subjects and observers are loosely coupled.*

Designing the Weather Station



An additional 3rd party display can be easily added!

Implementing the Weather Application

src/main/scala/cs220/Observer.scala

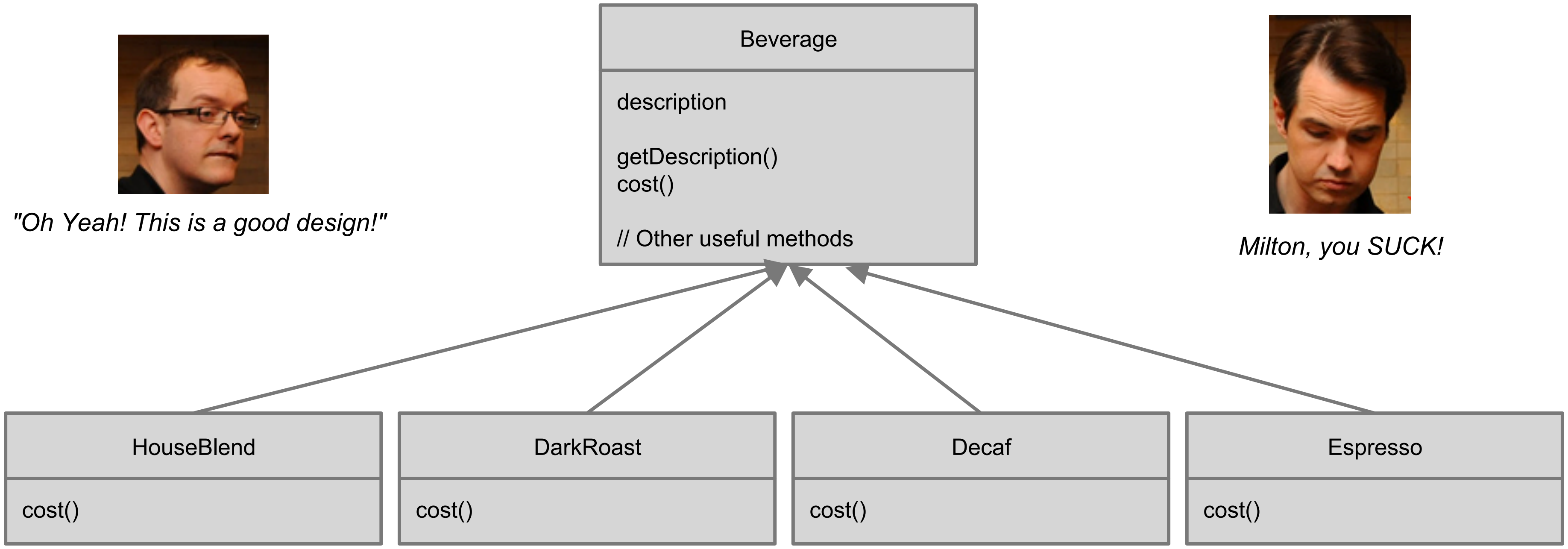
Welcome to Starbuzz Coffee!

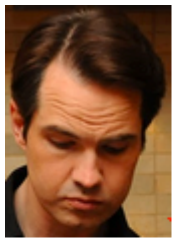
*This is **Milton Waddams**. He didn't take CMPSCI 220 at UMass Amherst. He thinks he is awesome, but he is not. He is the mastermind behind the poorly designed ordering system at Starbuzz. He can't code himself out of a paper bag. He really likes staplers.*

*This is **Peter Gibbons (aka Jimmy)**. He is a sad man because he is upset with the developer of Starbuzz's ordering system. Every time there is a new beverage it takes **forever** for the beverage to become available in their system. Not to mention the system is sooo slow!*

Starbuzz Ordering System Design

"Oh Yeah! This is a good design!"



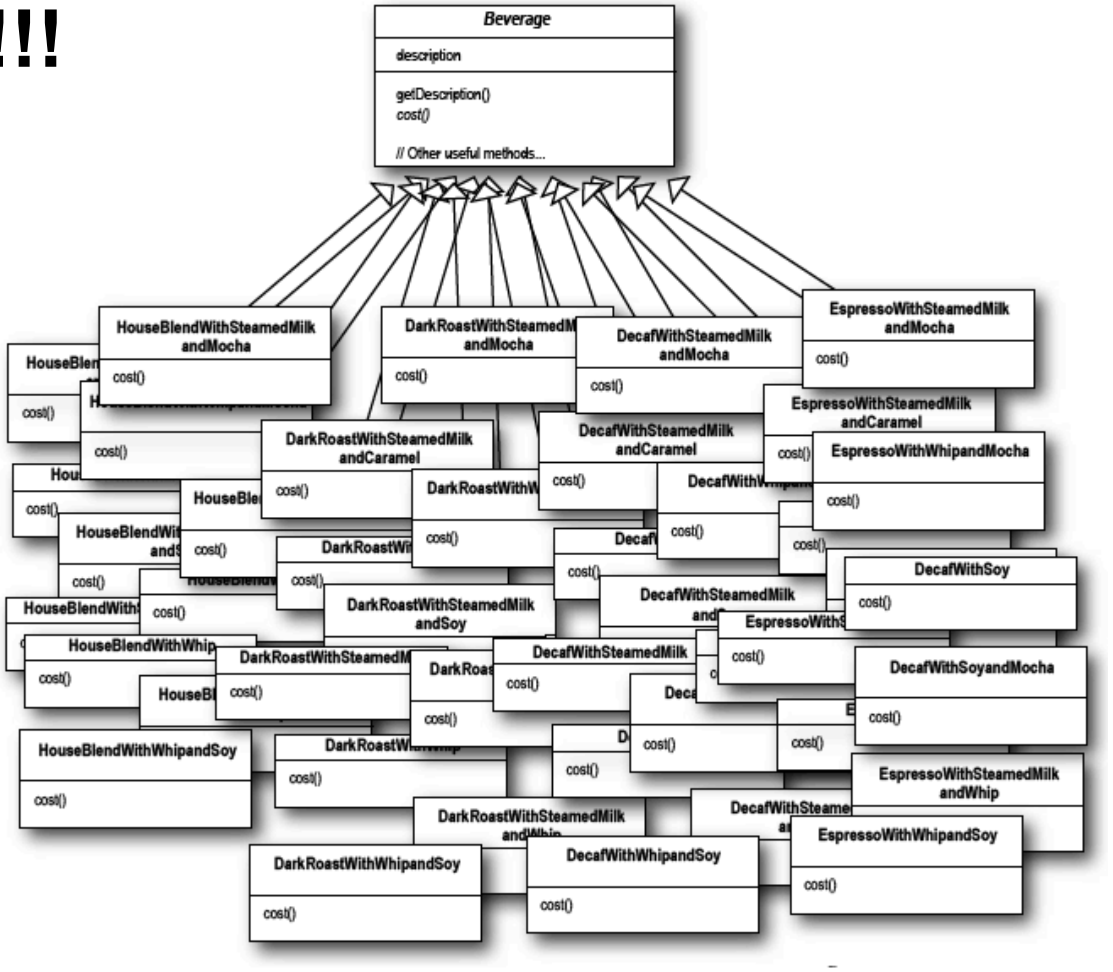

Milton, you SUCK!

In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha, and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them build into their ordering system.

How might this be done in this design?

Class Explosion!!!

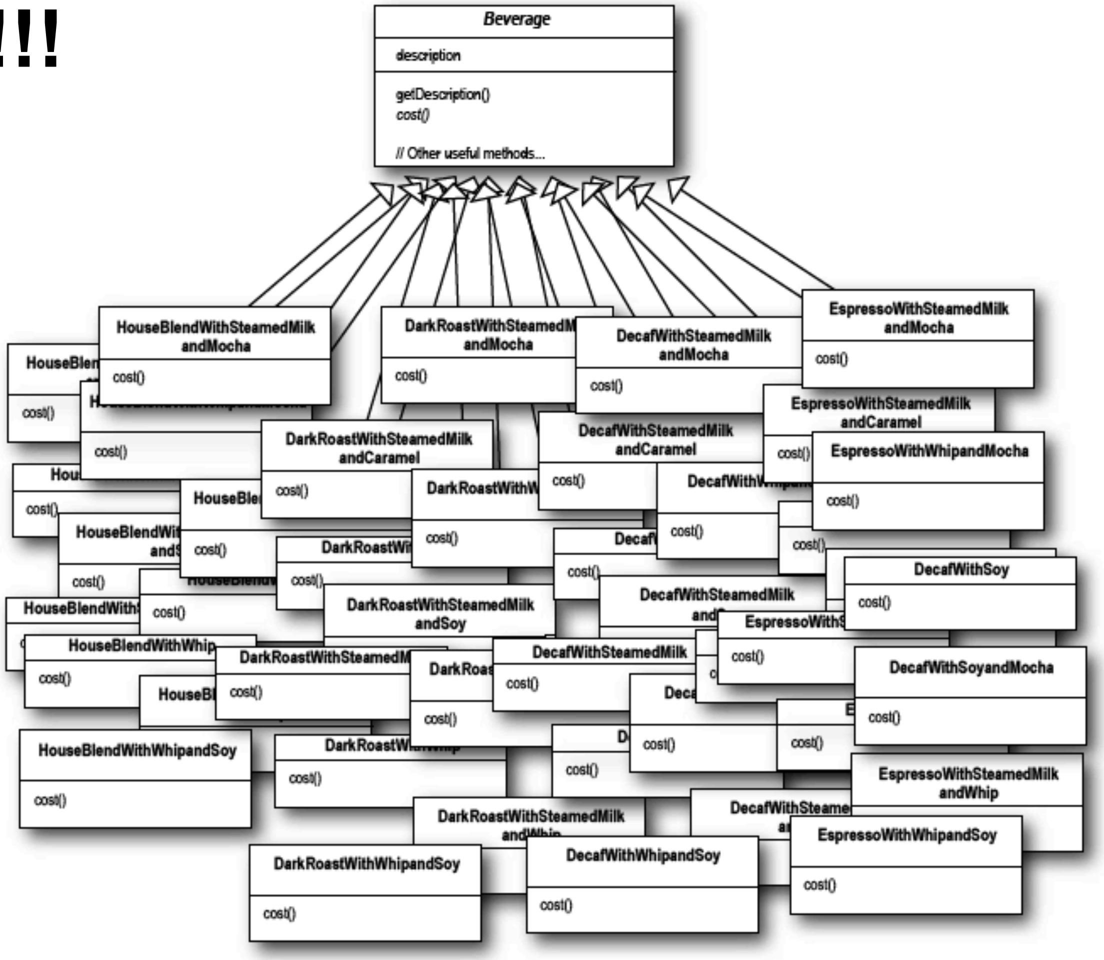
"So, what is your point..."


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
DarkRoastWithSteamedMilk andMocha
DecafWithSteamedMilk andMocha
EspressoWithSteamedMilk andMocha
EspressoWithSteamedMilk andCaramel
DarkRoastWithSteamedMilk andCaramel
DecafWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DarkRoastWithSteamedMilk andSoy
DecafWithSoy
DecafWithSteamedMilk
DecafWithSoyandMocha
HouseBlendWithWhipandSoy
EspressoWithSteamedMilk andWhip
EspressoWithWhipandSoy
DarkRoastWithSteamedMilk andWhip
DarkRoastWithWhipandSoy
DecafWithWhipandSoy
cost()

Class Explosion!!!

"You are FIRED! And leave the stapler!"

How can we fix this?

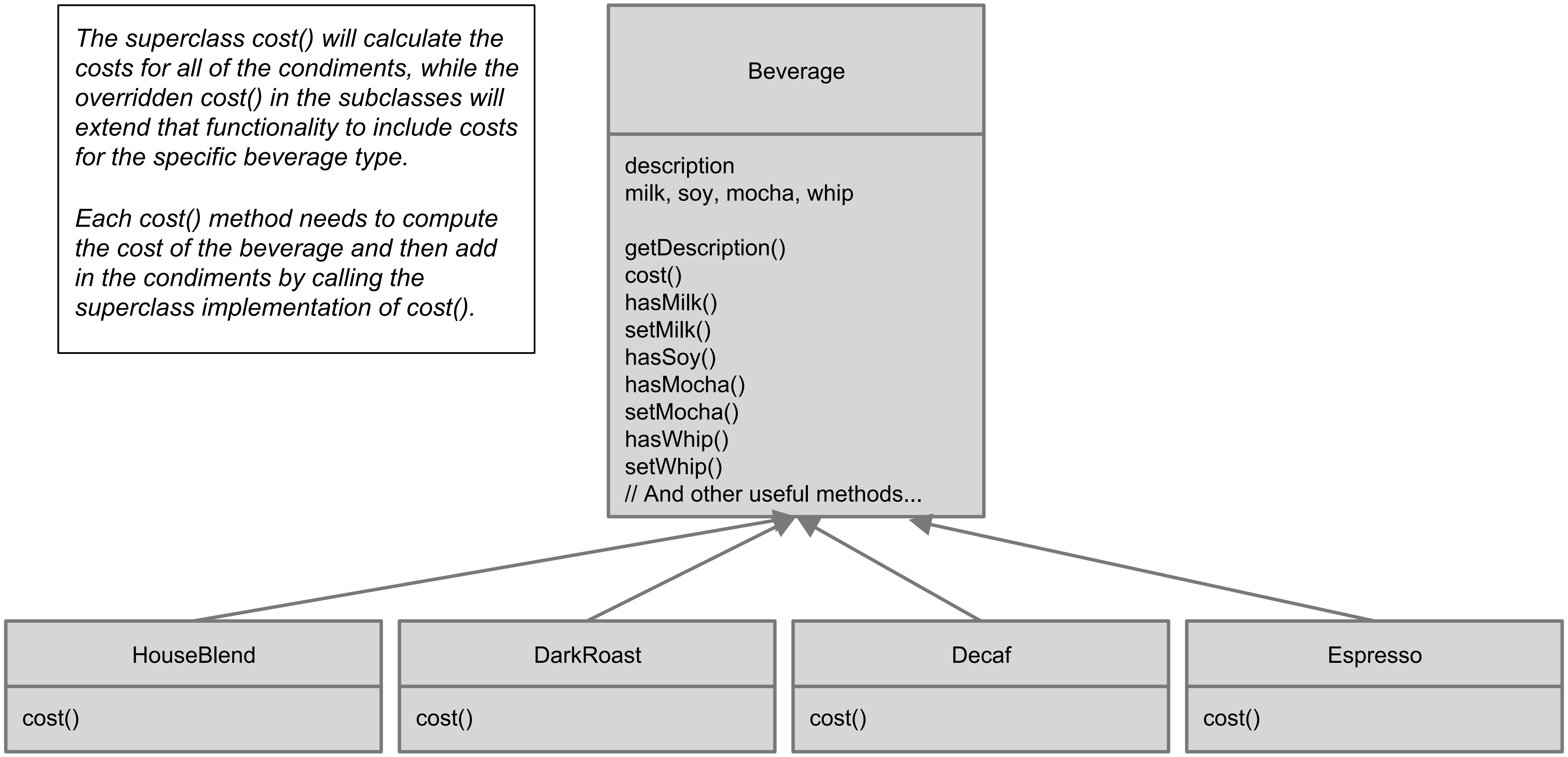


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
DarkRoastWithSteamedMilk andMocha
DecafWithSteamedMilk andMocha
EspressoWithSteamedMilk andMocha
EspressoWithSteamedMilk andCaramel
DarkRoastWithSteamedMilk andCaramel
DecafWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DecafWithSoy
DarkRoastWithSteamedMilk andSoy
DecafWithSoyandMocha
EspressoWithSteamedMilk andWhip
HouseBlendWithWhipandSoy
DarkRoastWithSteamedMilk andWhip
DecafWithWhipandSoy
EspressoWithWhipandSoy
DarkRoastWithWhipandSoy
cost()

Starbuzz Ordering System Design Take 2

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for the specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



Your Turn!

Write the cost() methods for the two classes below (pseudo-Scala is okay)

```
class Beverage {
  def cost: Double = {






  }
}
```

```
class DarkRoast extends Beverage {
  val description = "Most Excellent Dark Roast"
  def cost: Double = {






  }
}
```

What Might Impact This Design?

- Price changes for condiments will force us to alter existing code.
- New condiments will force us to add new methods and alter the cost method of the superclass.
- We may have new beverages. For some of these beverages (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like hasWhip().
- What if a customer wants a double mocha or espresso?

The Open-Closed Principle

Design Principle:
Classes should be open for extension, but closed for modification.

How does this relate to our existing ordering system design?

Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code.
What do we get if we accomplish this?
Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

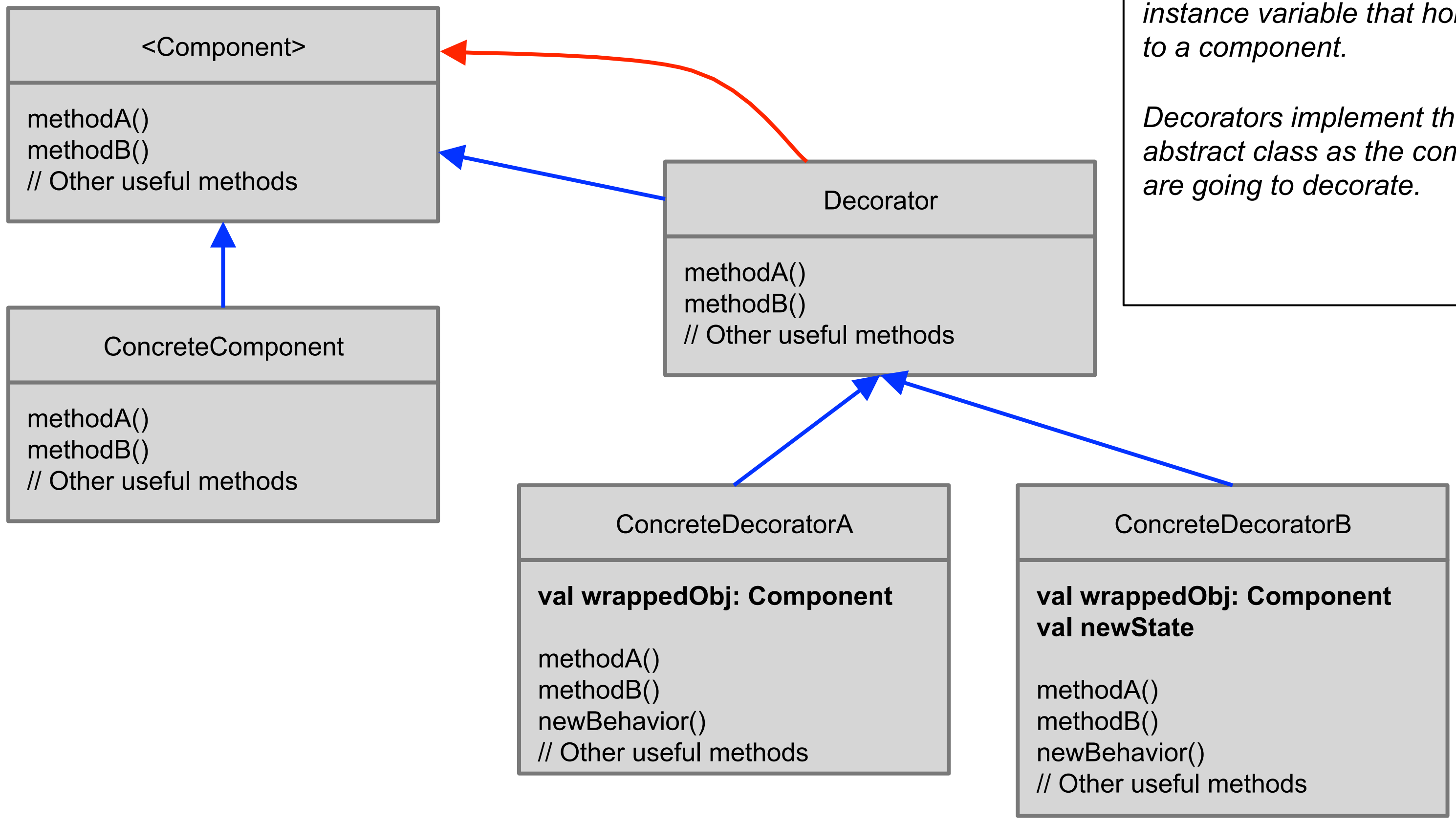
Meet The Decorator Pattern

1. Take a DarkRoast object.
2. Decorate it with a Mocha object.
3. Decorate it with a Whip object.
4. Call the cost() method and rely on *delegation* to add on the condiment costs.

The Decorator Pattern Defined

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

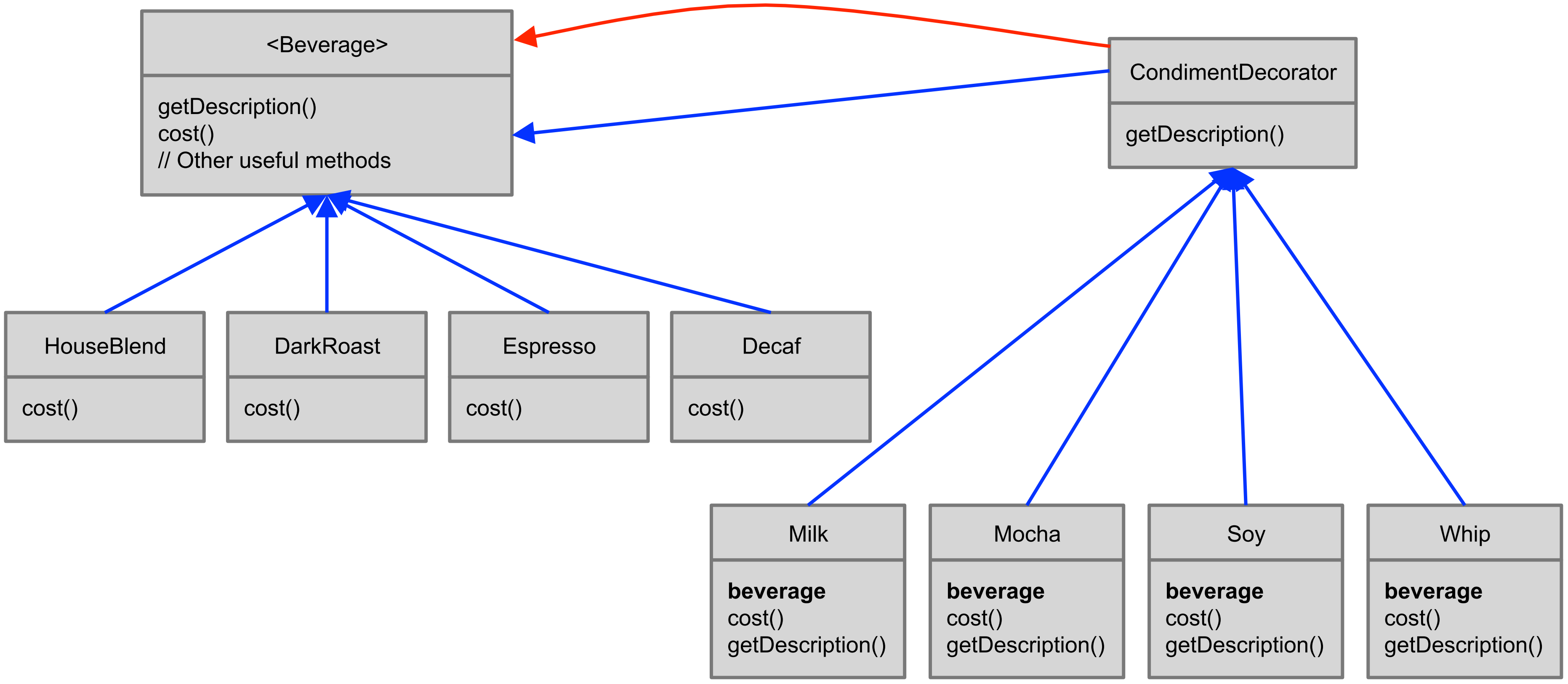
Decorator Class Diagram



Each decorator HAS-A component, which means that the decorator has an instance variable that holds a reference to a component.

Decorators implement the same trait or abstract class as the component they are going to decorate.

Starbuzz's New Design



<Beverage>
getDescription()
cost()
// Other useful methods
CondimentDecorator
getDescription()
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
Milk
beverage
cost()
getDescription()
Mocha
beverage
cost()
getDescription()
Soy
beverage
cost()
getDescription()
Whip
beverage
cost()
getDescription()