# CMPSCI 220 Programming Methodology

## 06: Observer and Decorator

# Objectives

## Observer Pattern

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

## Decorator Pattern

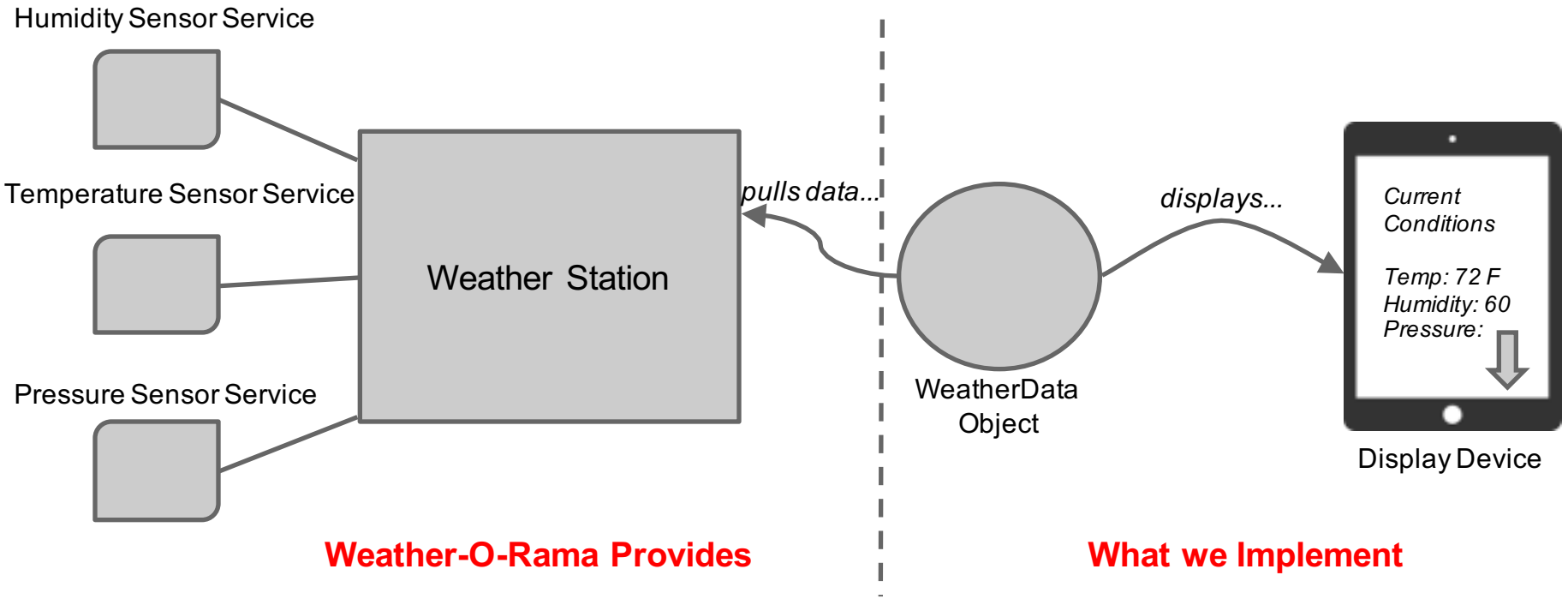
Learn how objects can gain new responsibilities at runtime.
Apply the decorator pattern in Scala.

# i-clicker Question!

Why should you favor composition over inheritance?

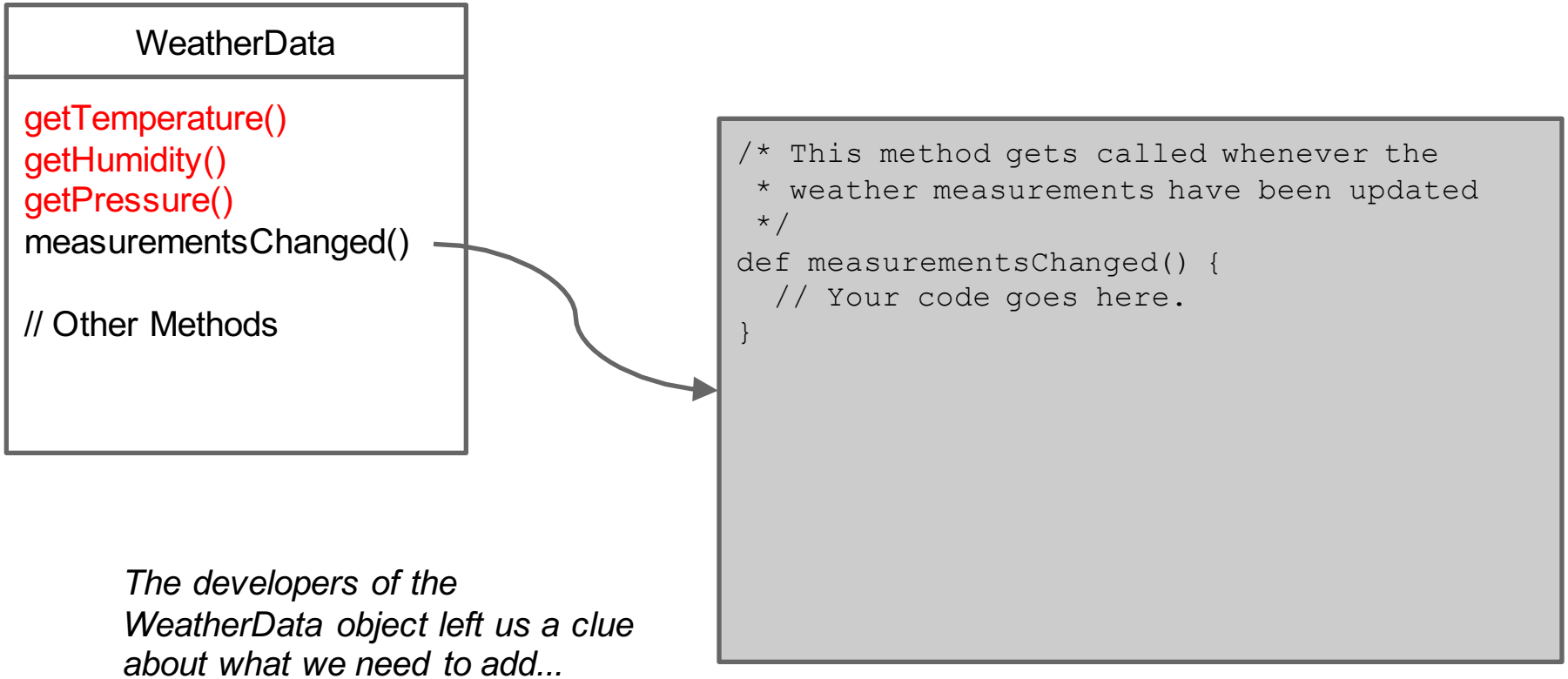
A. It is in fashion
B. It leads to a more flexible class design
C. It allows you to create classes automatically
D. It lets you defer implementation to super classes
E. I am not sure

# Observer: Weather Monitoring Application



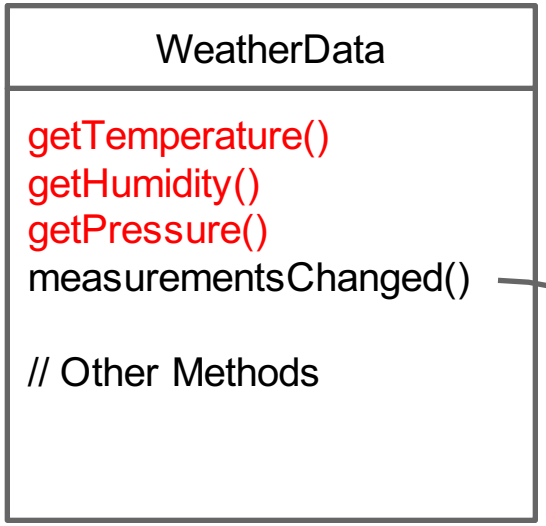

**Our job, if we choose to accept it, is to create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.**

# Weather Monitoring App: First Try...




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

# Weather Monitoring App: First Try...

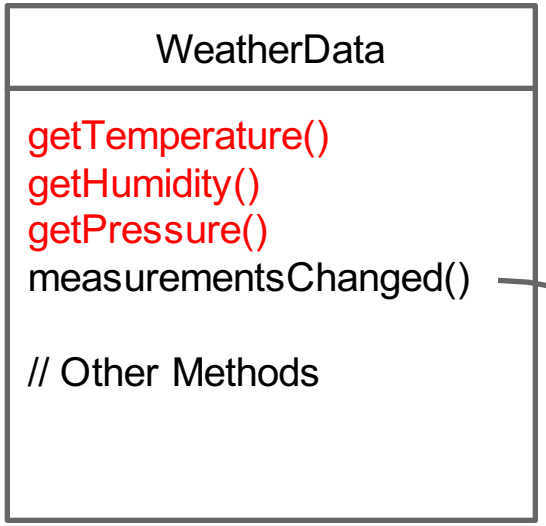



```
/* This method gets called whenever the
 * weather measurements have been updated
 */
def measurementsChanged() {
  val t = getTemperature
  val h = getHumidity
  val p = getPressure

  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...



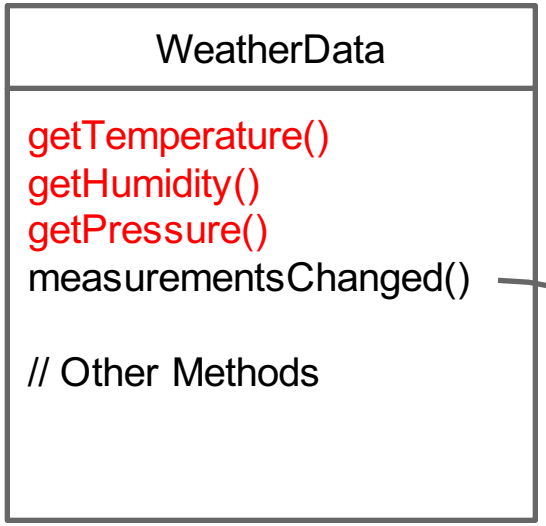

```
/* This method gets called whenever the
 * weather measurements have been updated
 */
def measurementsChanged() {
  val t = getTemperature
  val h = getHumidity
  val p = getPressure

  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

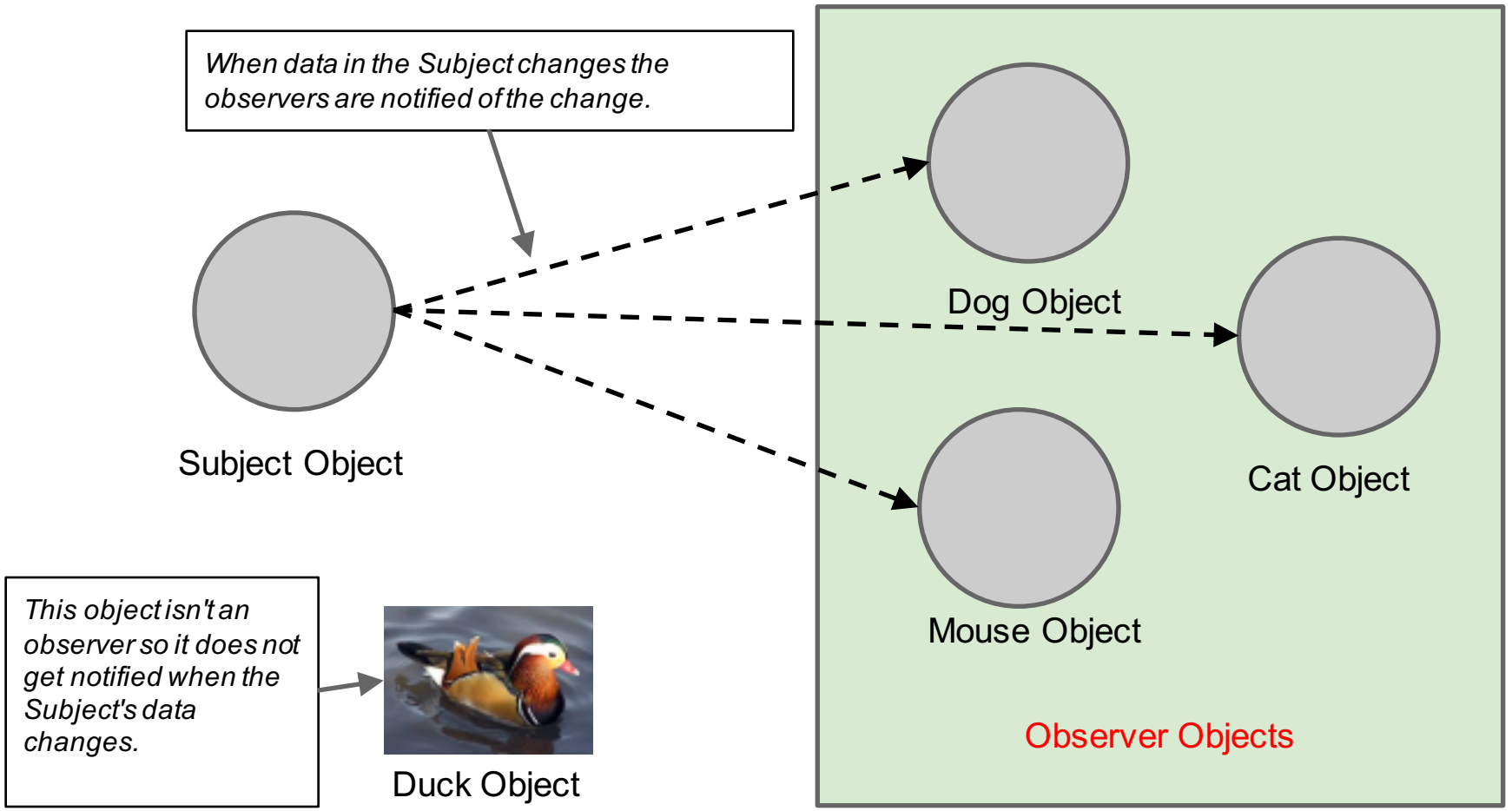
A newspaper publisher goes into business and begins publishing newspapers.

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.

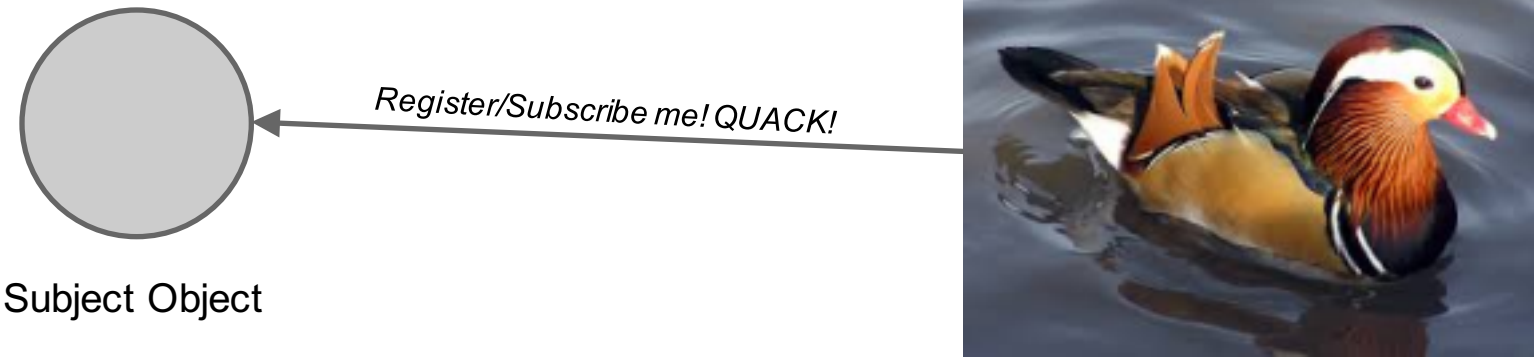
While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.

# Publisher + Subscriber = Observer Pattern



# Subscribing...

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



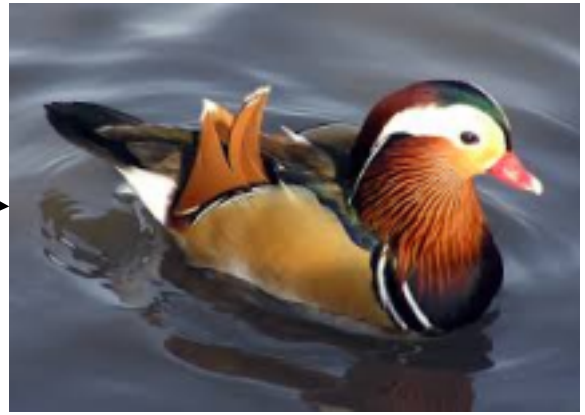

Subject Object

Duck Object

# Subscribing...

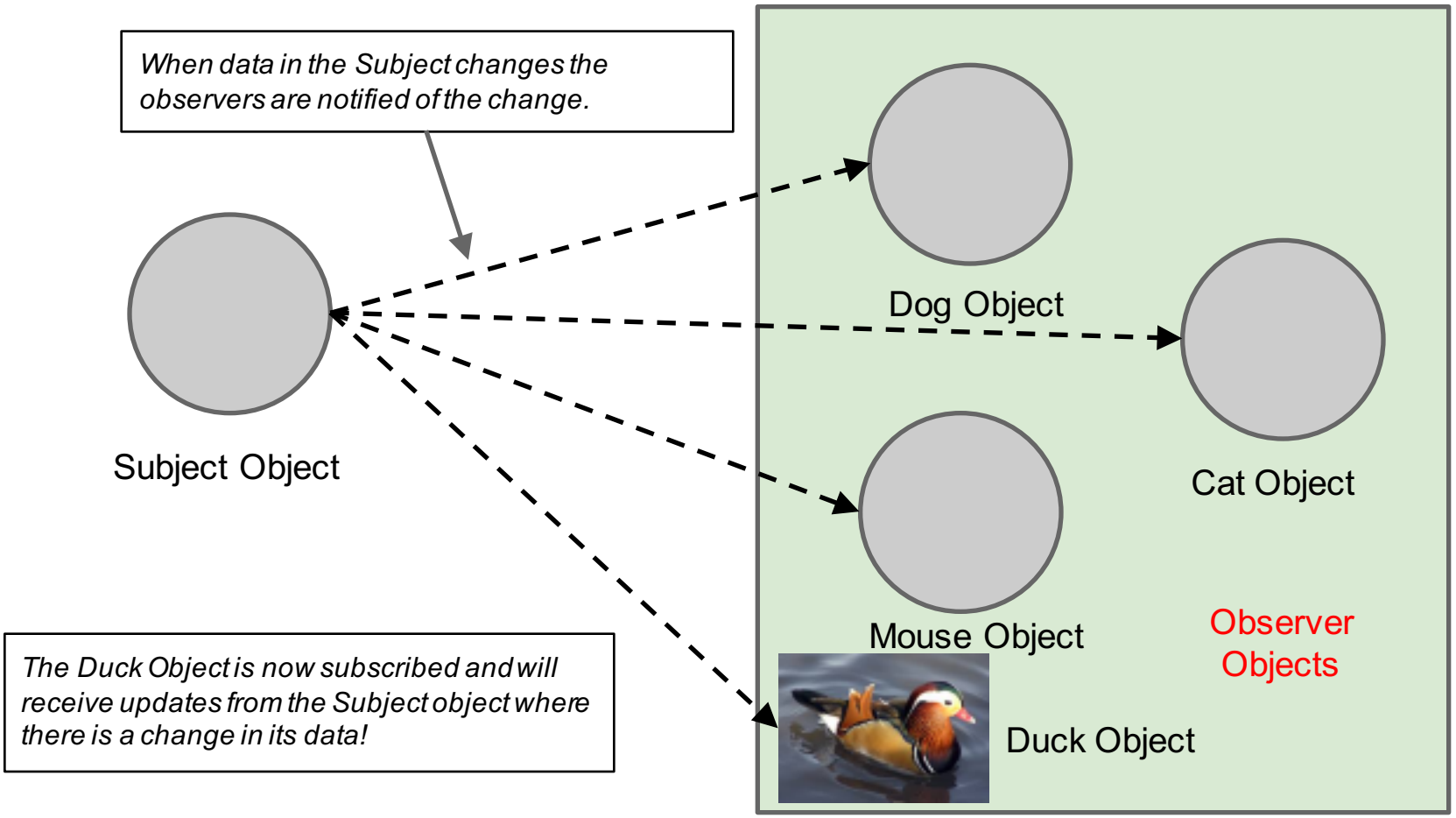
*A duck object comes along and tells the Subject object that it wants to become an observer...*


Subject Object

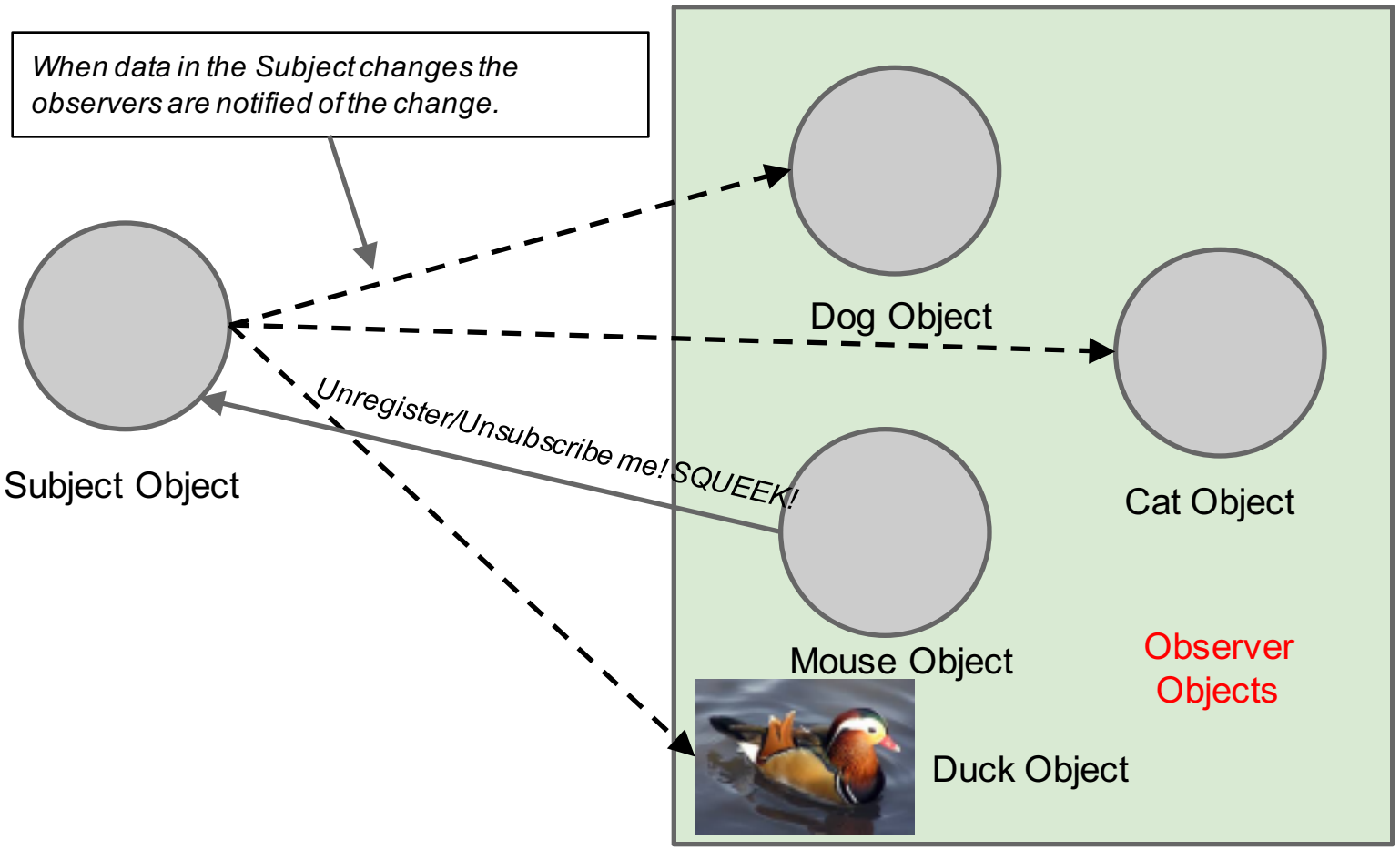


Duck Object

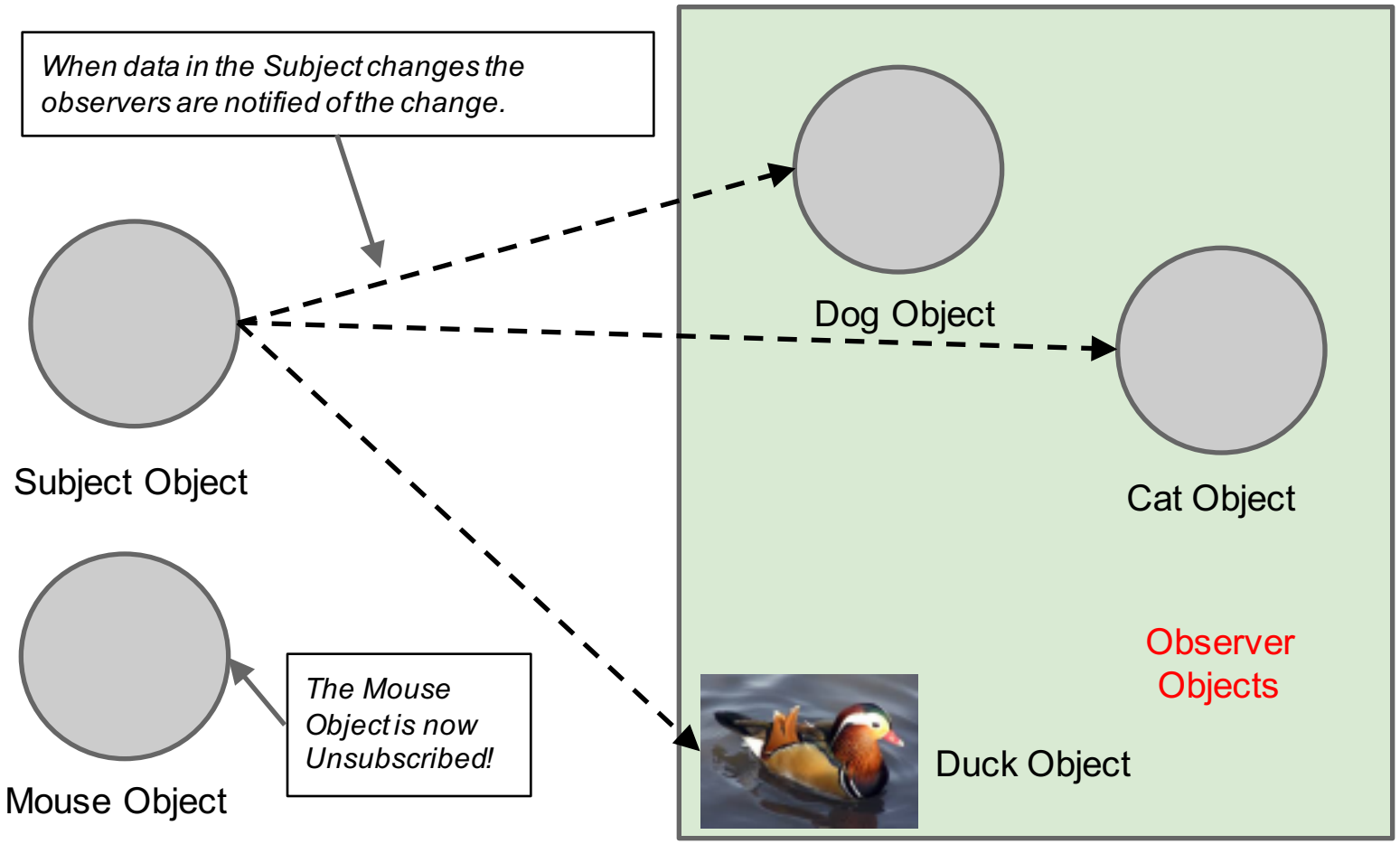
# Publisher + Subscriber = Observer Pattern



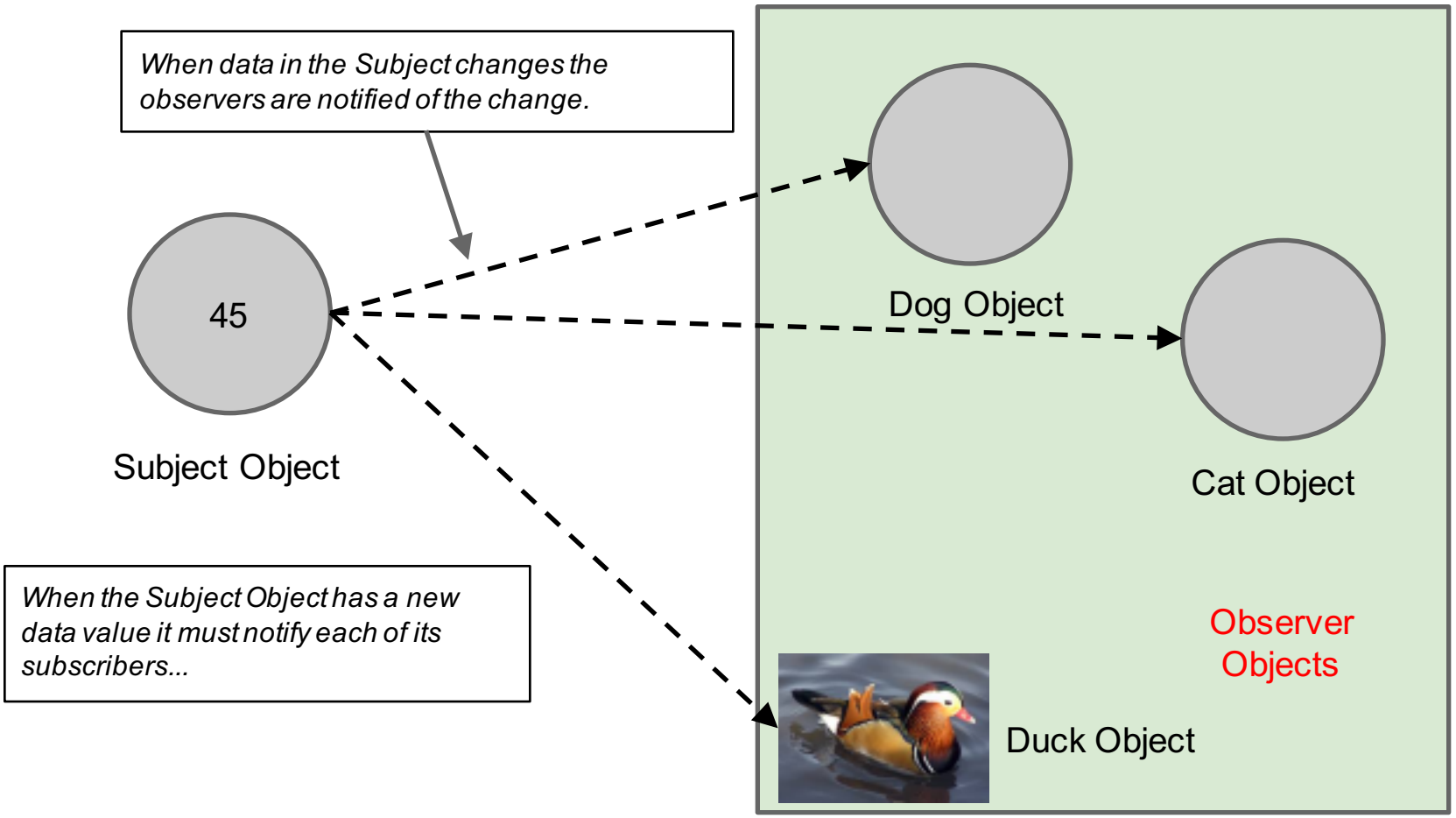
# Publisher + Subscriber = Observer Pattern



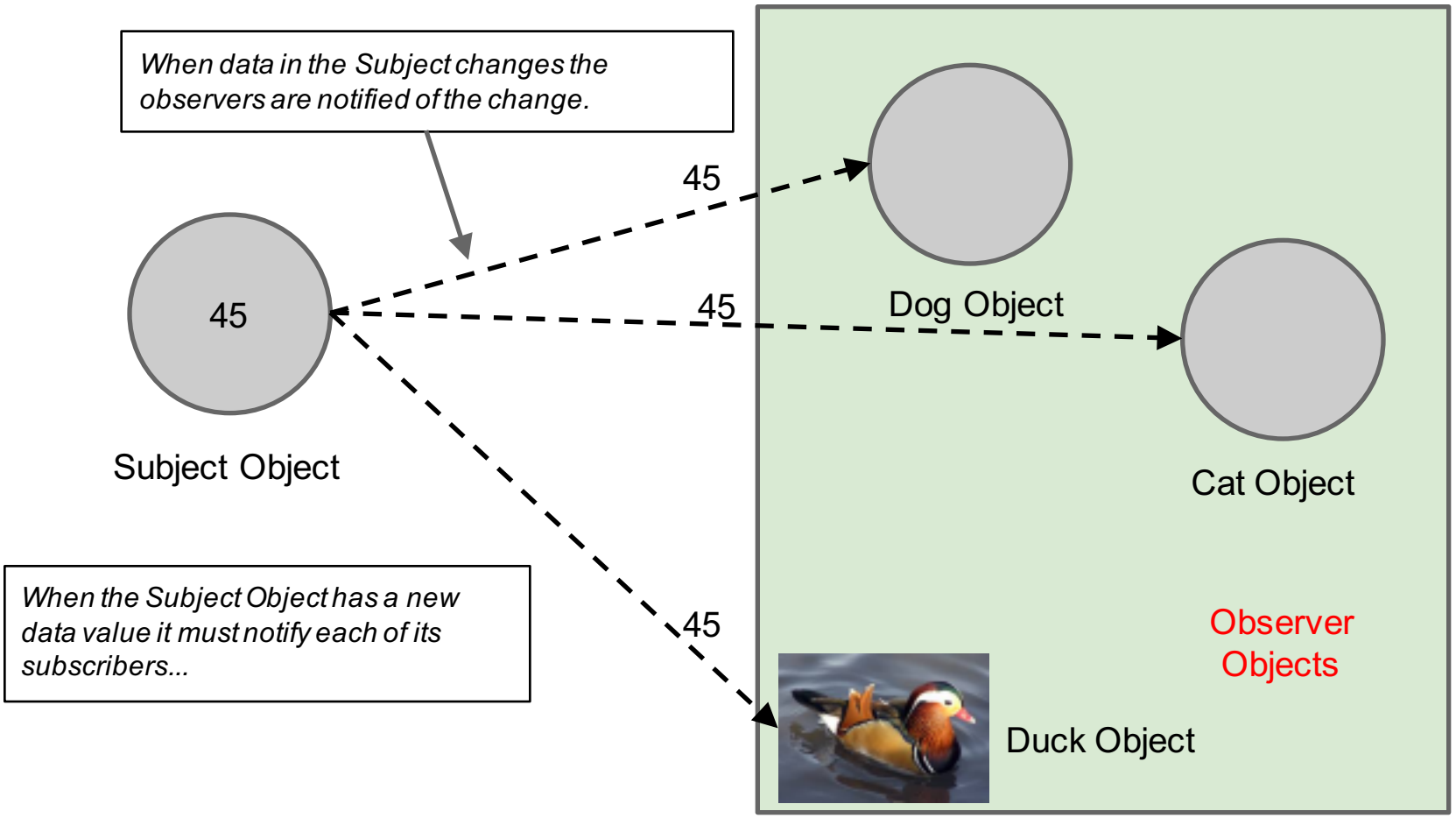
# Publisher + Subscriber = Observer Pattern



# Publisher + Subscriber = Observer Pattern



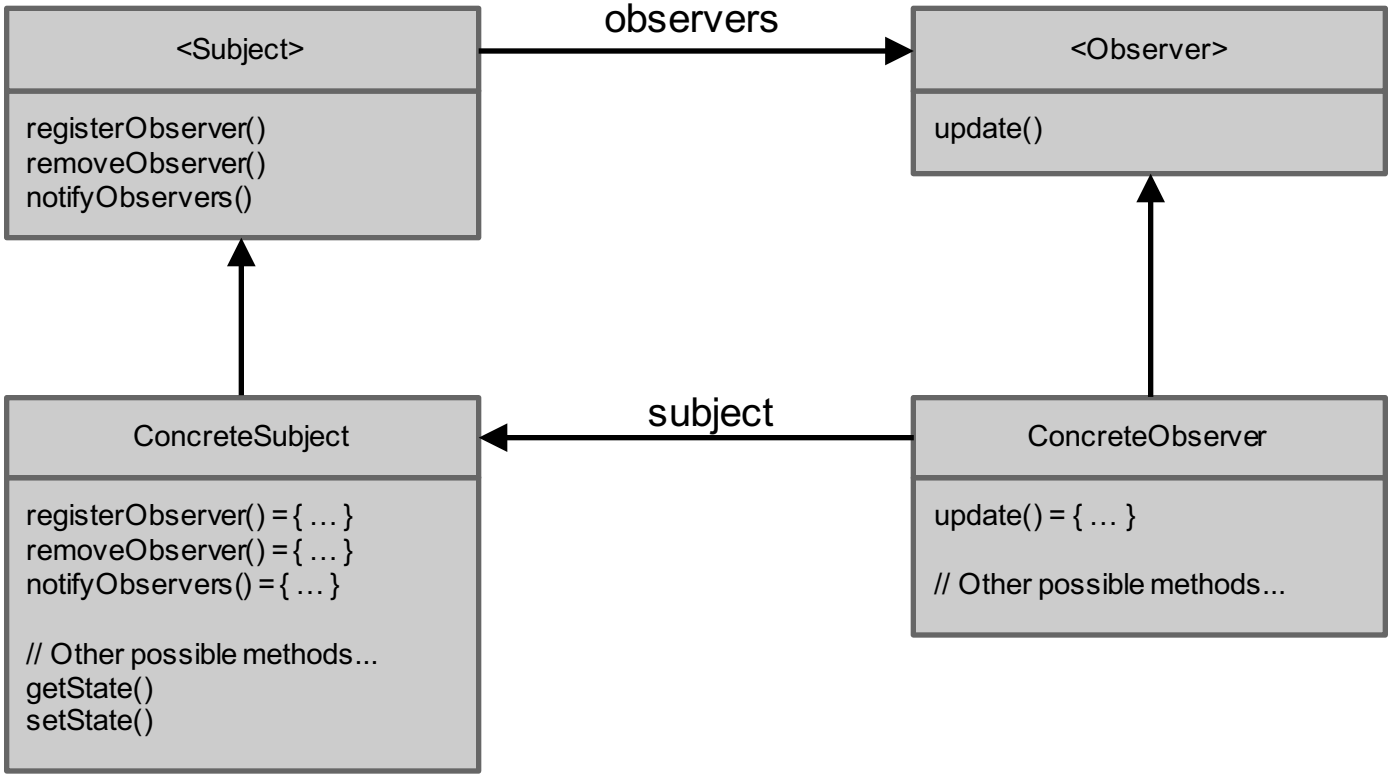
# Publisher + Subscriber = Observer Pattern



## 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.*

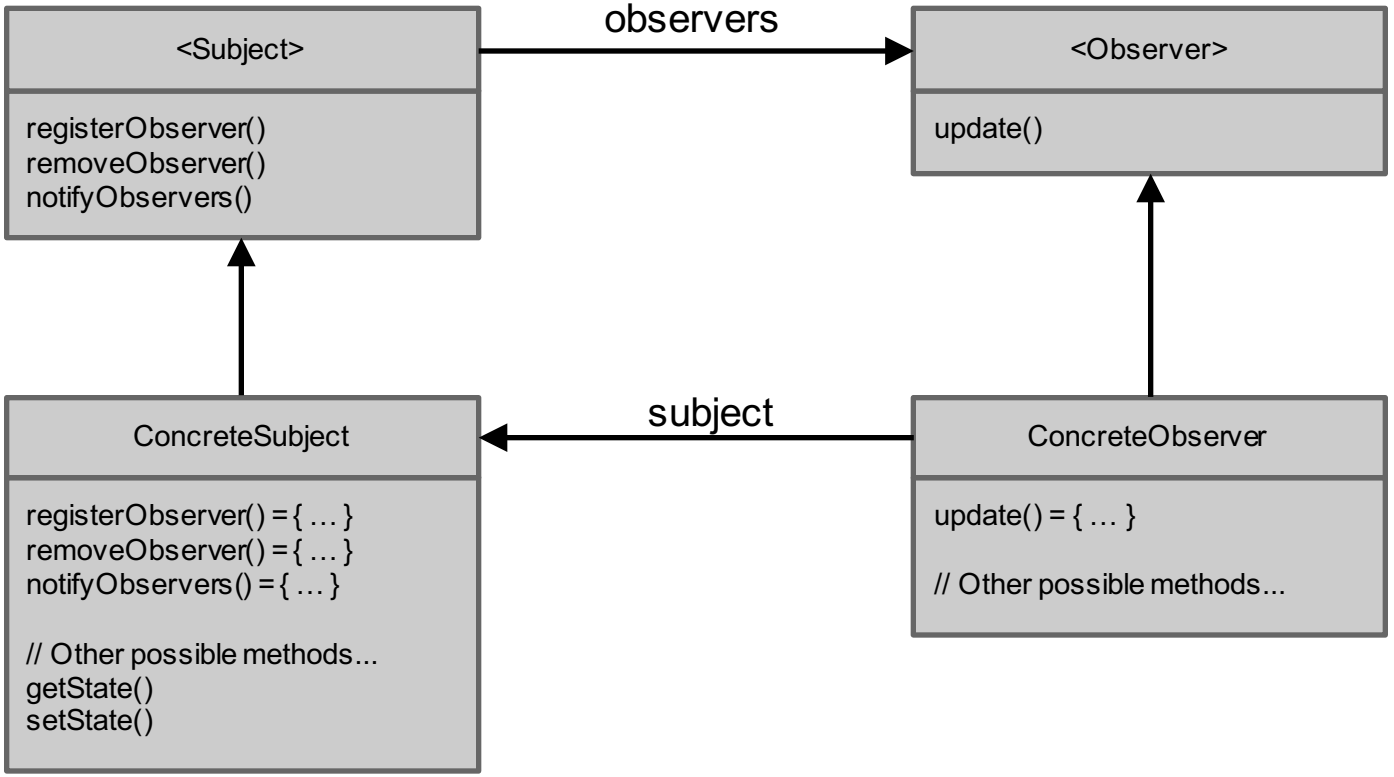
# Designing the Weather Station:
## *Try Sketching it Out!*

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.

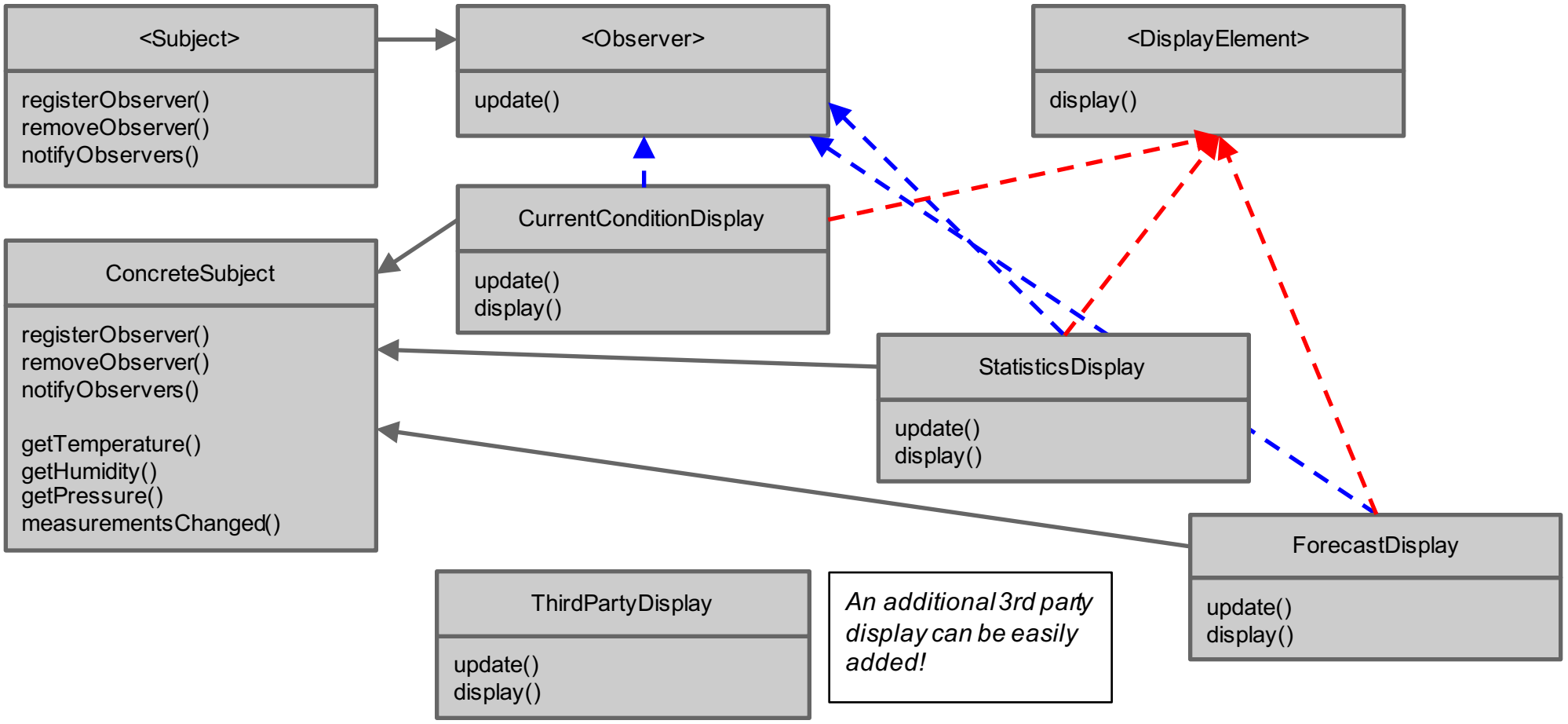
# This Might Help: Observer Class Diagram




*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



## Implementing the Weather Application

src/main/scala/cs220/Observer.scala