Design Patterns

Design Pattern

- Someone has already solved your problems.
- You need to learn why (and how) you can exploit the wisdom and lessons learned by other developers who’ve been down the same design problem road and survived the trip.

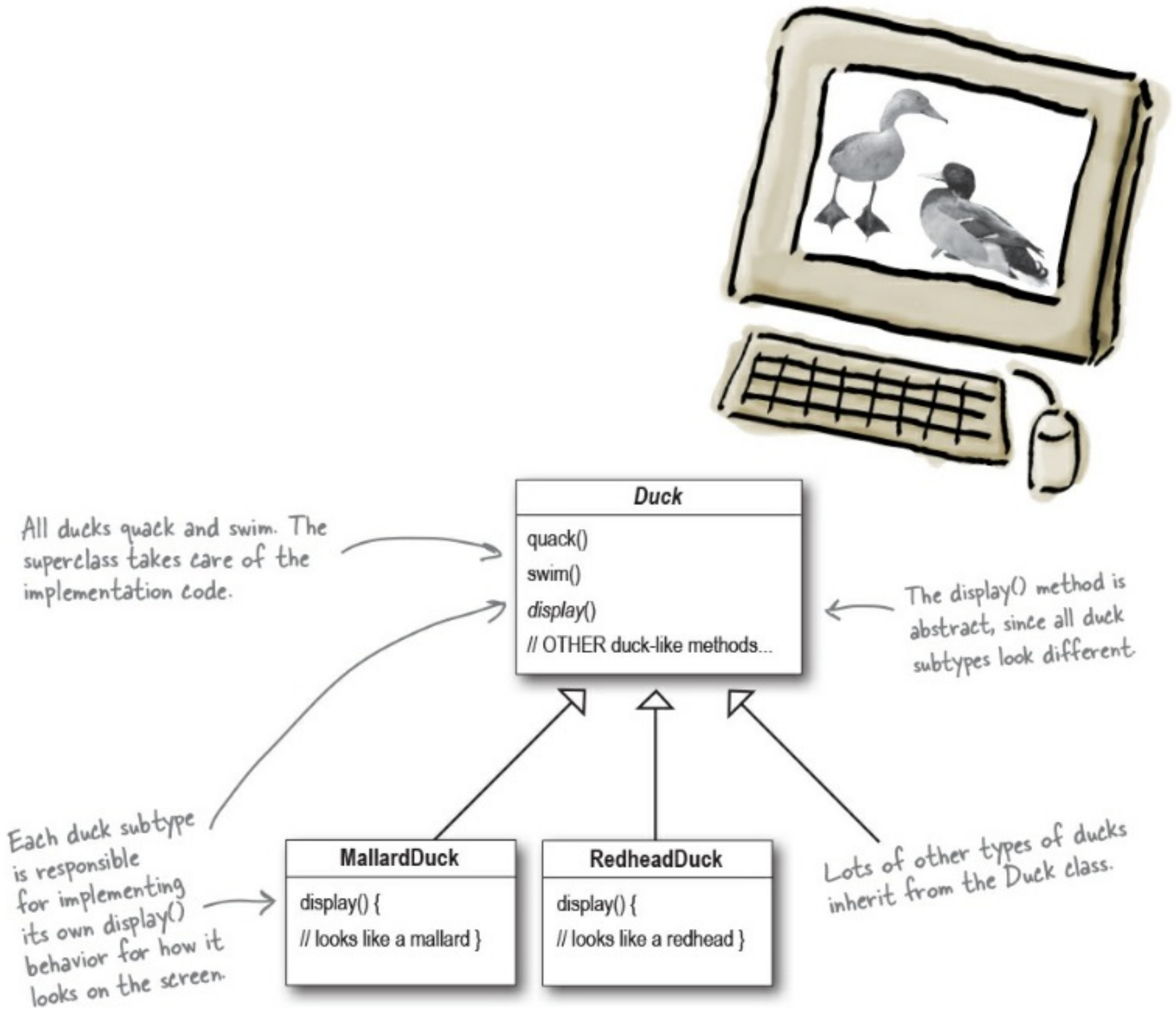
A Simple problem

- Create a simulated duck pond filled with large varieties of duck species swimming and making quacking sounds

- Additional Requirement
  - Let ducks fly

- Solution
  - Add a fly method

- Problem
  - There were some rubber ducks



Duck

quack()
swim()
display()
// OTHER duck-like methods...

MallardDuck

display() {
// looks like a mallard }

RedheadDuck

display() {
// looks like a redhead }

All ducks quack and swim. The superclass takes care of the implementation code.

The display() method is abstract, since all duck subtypes look different.

Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.

Lots of other types of ducks inherit from the Duck class.

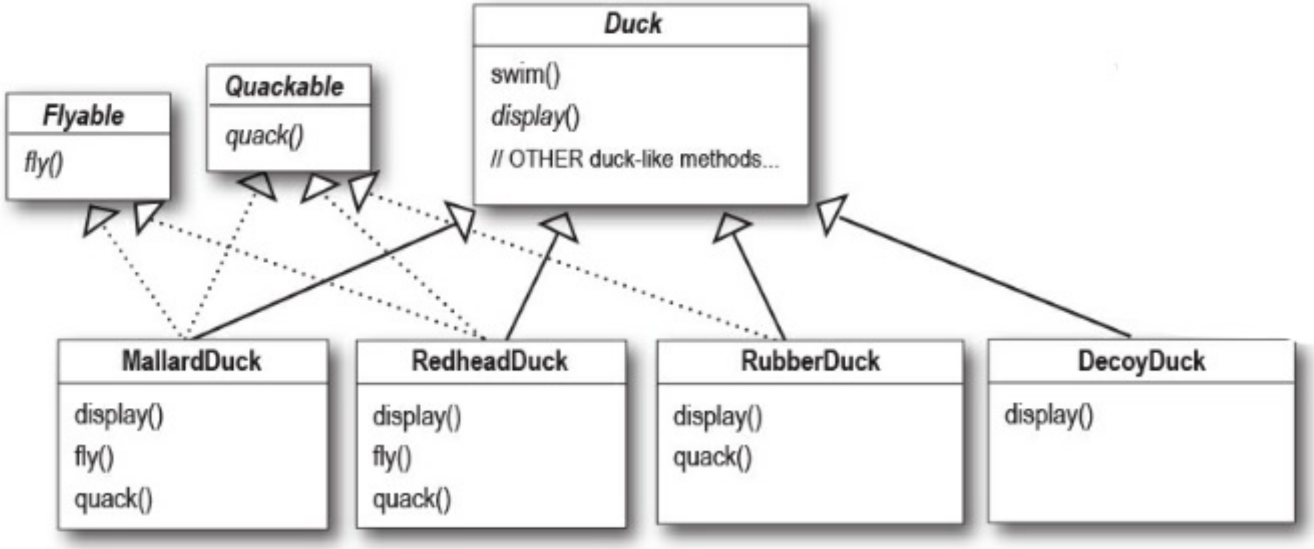
Learnings

- A localized update to the code caused a non-local side effect (flying rubber ducks)!

- A great use of inheritance for the purpose of reuse hasn’t turned out so well when it comes to maintenance.

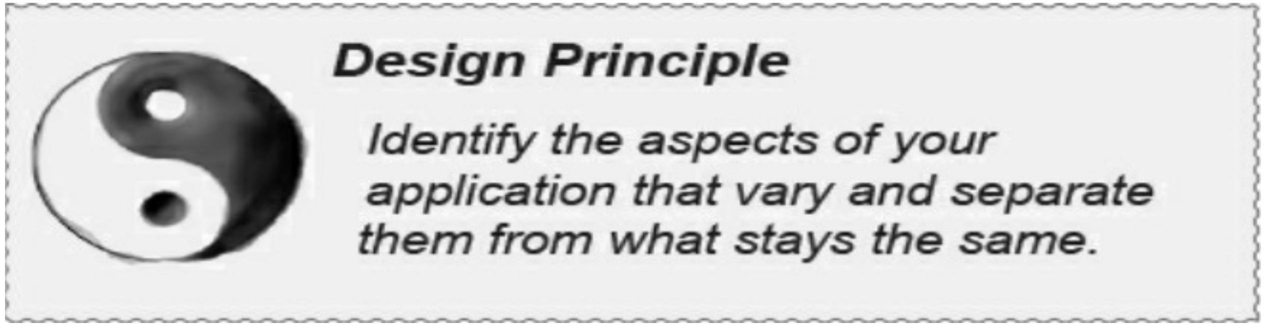
How to solve it

- What other OOP feature is best suitable to ensure that some ducks can fly while others cannot
    - Inheritance was not the answer
- Does it solve the problem
    - What if small change is needed in the flying behavior



Design Principle

- Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t.



Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

Take what varies and “encapsulate” it so it won’t affect the rest of your code.

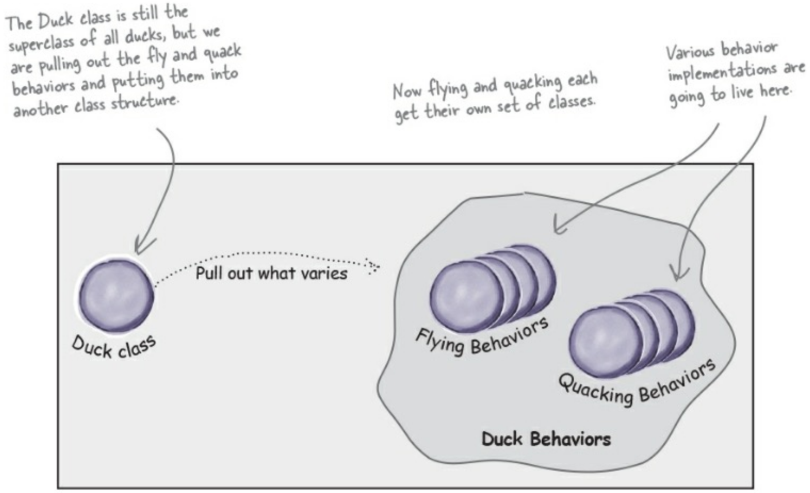
The result? Fewer unintended consequences from code changes and more flexibility in your systems!

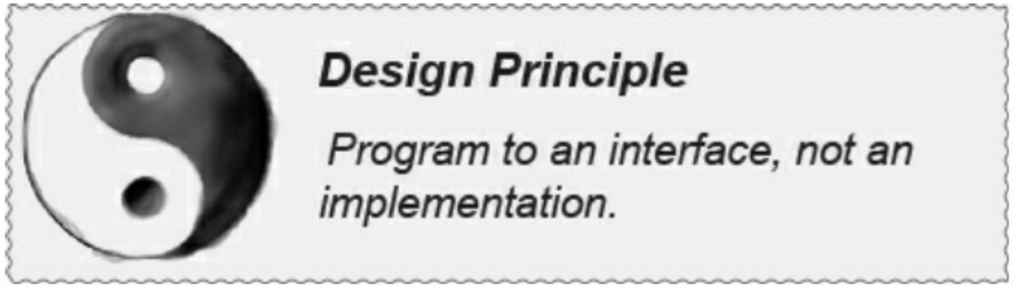
We know that fly() and quack() are the parts of the Duck class that vary across ducks.

To separate these behaviors from the Duck class, we’ll pull both methods out of the Duck class and create a new set of classes to represent each

Design Principle

- We should be able to change the behavior of ducks at runtime


The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.
Now flying and quacking each get their own set of classes.
Various behavior implementations are going to live here.
Pull out what varies
Duck class
Flying Behaviors
Quacking Behaviors
Duck Behaviors

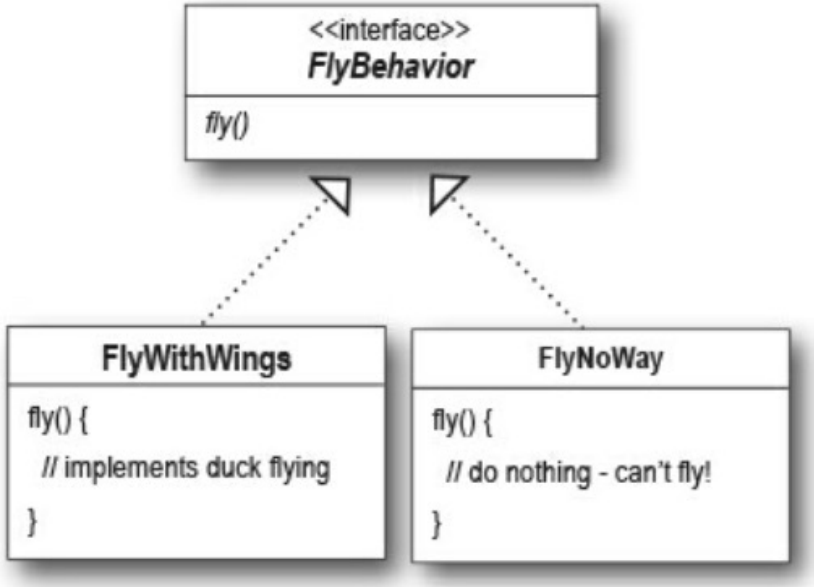


Design Principle

Program to an interface, not an implementation.

Solution

- Interface =/= Java Interface but a super type
- Sub-classes then implement the Interface
    - Exploit Poly-morphism



Points to note

Earlier

- Behavior came from
  - A concrete implementation in the superclass
  - Specialized implementation in the subclass

Now

- Subclass will use a behavior represented by an interface
- Actual implementation is not locked in the subclass

Problem

- You have different types of animals that make different types of sound, and move differently; but they also have similar characteristics
- Code: such that you can change the moving behavior while keeping the code maintenance as a goal, i.e., code can be added easily.

Reference

Erich, Gamma; Helm Richard; Johnson Ralph; Vlissides John. Design Patterns. Pearson Education. Kindle Edition.

Freeman, Eric; Robson, Elisabeth. Head First Design Patterns. O'Reilly Media. Kindle Edition.

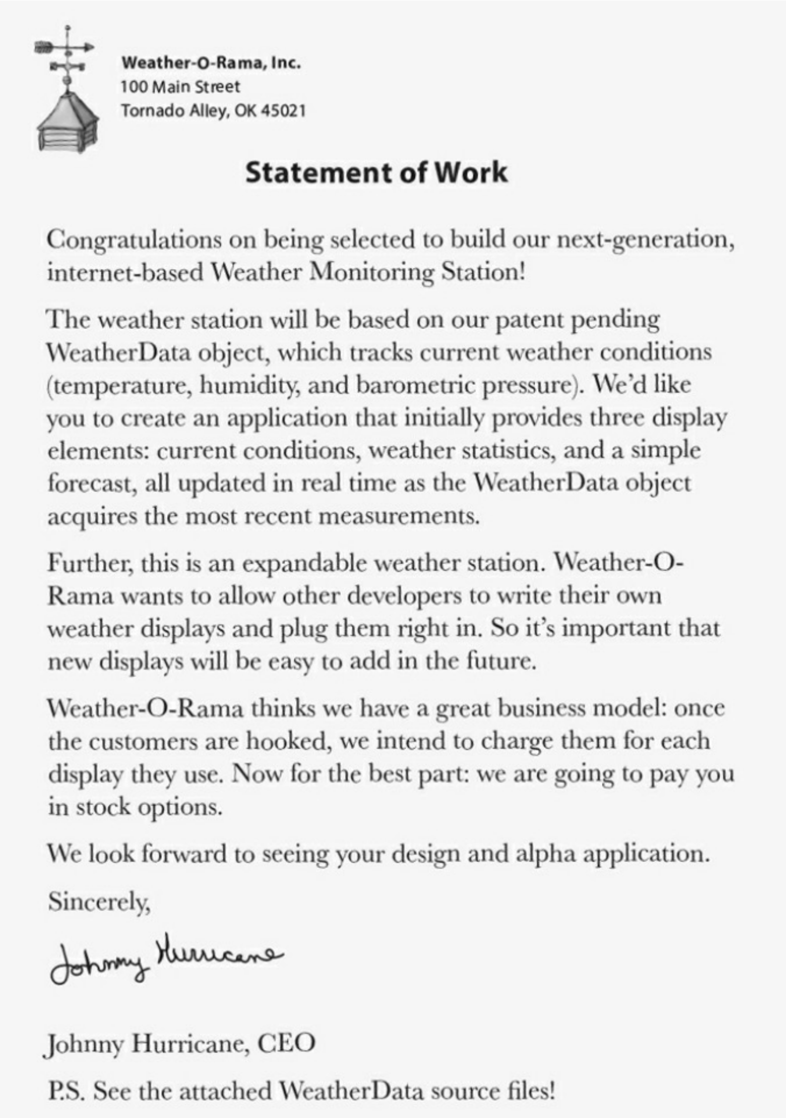
Observer Pattern

Observer Pattern

- You don’t want to miss out when something interesting happens, do you?
- Observer pattern keeps your objects in the know when something they care about happens.

Problem that we will be solving

- Break it down
  - Each client display should receive the data notification whenever there is a change
  - New client displays should be easily added
  - Not all display need to display all the information



Weather-O-Rama, Inc.
100 Main Street
Tornado Alley, OK 45021

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 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 allow other developers to write their own weather displays and plug them right in. So it's important that new displays will be easy to add in the future.

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.

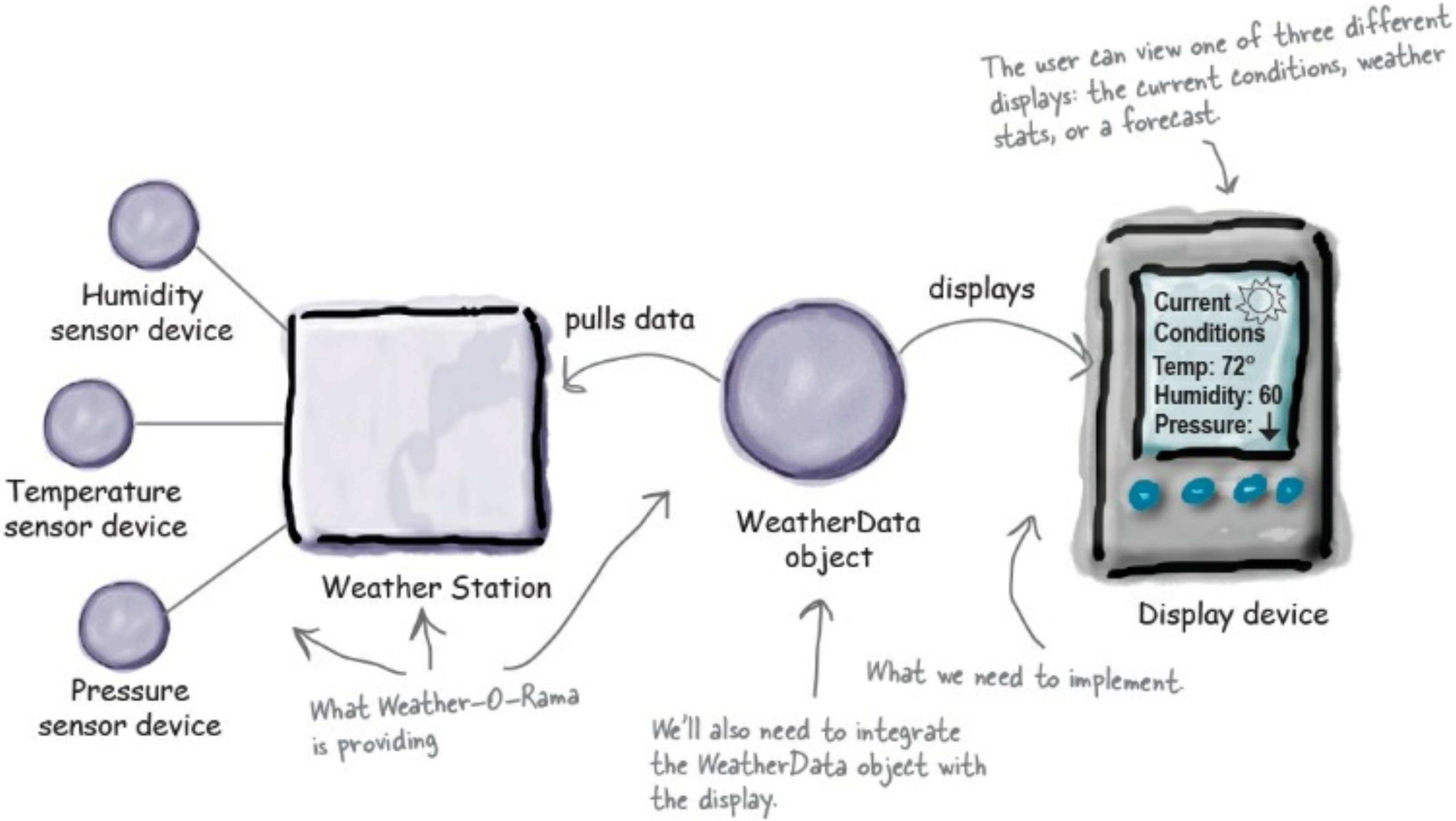
We look forward to seeing your design and alpha application.

Sincerely,

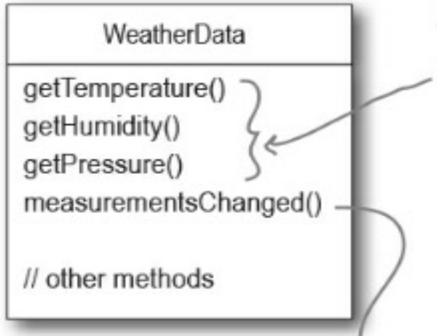
Johnny Hurricane

Johnny Hurricane, CEO

P.S. See the attached WeatherData source files!


The user can view one of three different displays: the current conditions, weather stats, or a forecast.
Humidity sensor device
Temperature sensor device
Pressure sensor device
Weather Station
pulls data
WeatherData object
displays
Current Conditions
Temp: 72°
Humidity: 60
Pressure: ↓
Display device
What Weather-O-Rama is providing
We'll also need to integrate the WeatherData object with the display.
What we need to implement

Here is our WeatherData class.



WeatherData

getTemperature()
getHumidity()
getPressure()
measurementsChanged()

// other methods

These three methods return the most recent weather measurements for temperature, humidity, and barometric pressure, respectively.

We don't care right now HOW it gets this data, we just know that the WeatherData object gets updated info from the Weather Station.

Note that whenever WeatherData has updated values, the measurementsChanged() method is called.

Let's looks at the measurementsChanged() method, which, again, gets called anytime the WeatherData obtains new values for temp, humidity, and pressure.

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

WeatherData.java

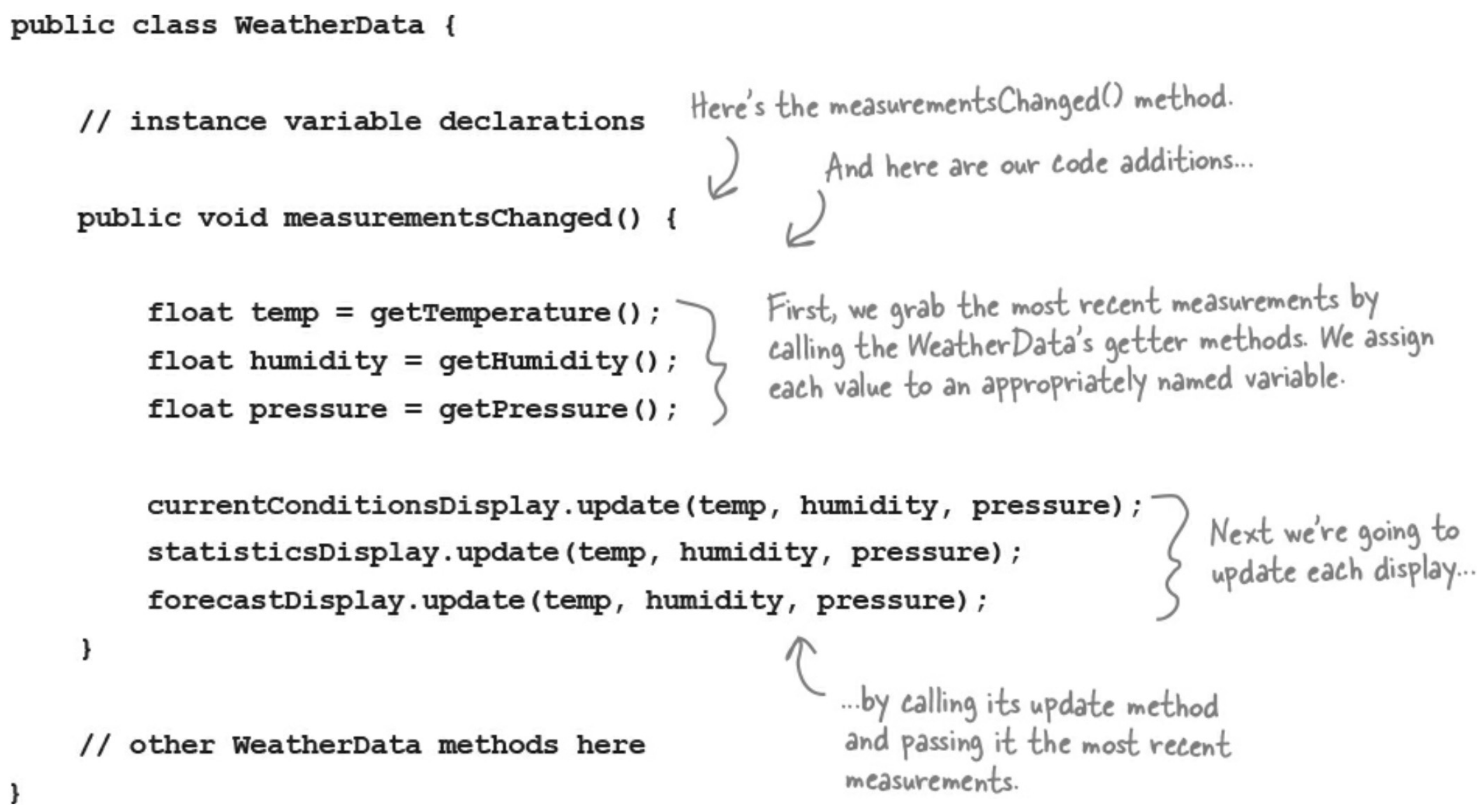
It looks like Weather-O-Rama left a note in the comments to add our code here. So perhaps this is where we need to update the display (once we've implemented it)

Our soon-to-be-implemented display.

Break it down

- WeatherData class has getter methods for three measurement values: temperature, humidity, and barometric pressure.
- measurementsChanged() method is called anytime new weather measurement data is available.
  - Again, we don’t know or care how this method is called; we just know that it is called.
- We’ll need to implement three display elements that use the weather data: a current conditions display, a statistics display, and a forecast display.
- These displays must be updated as often as the WeatherData has new measurements.
- To update the displays, we’ll add code to the measurementsChanged() method.

First Approach



```
public class WeatherData {

    // instance variable declarations

    public 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);
    }

    // other WeatherData methods here
}
```

Here's the measurementsChanged() method.

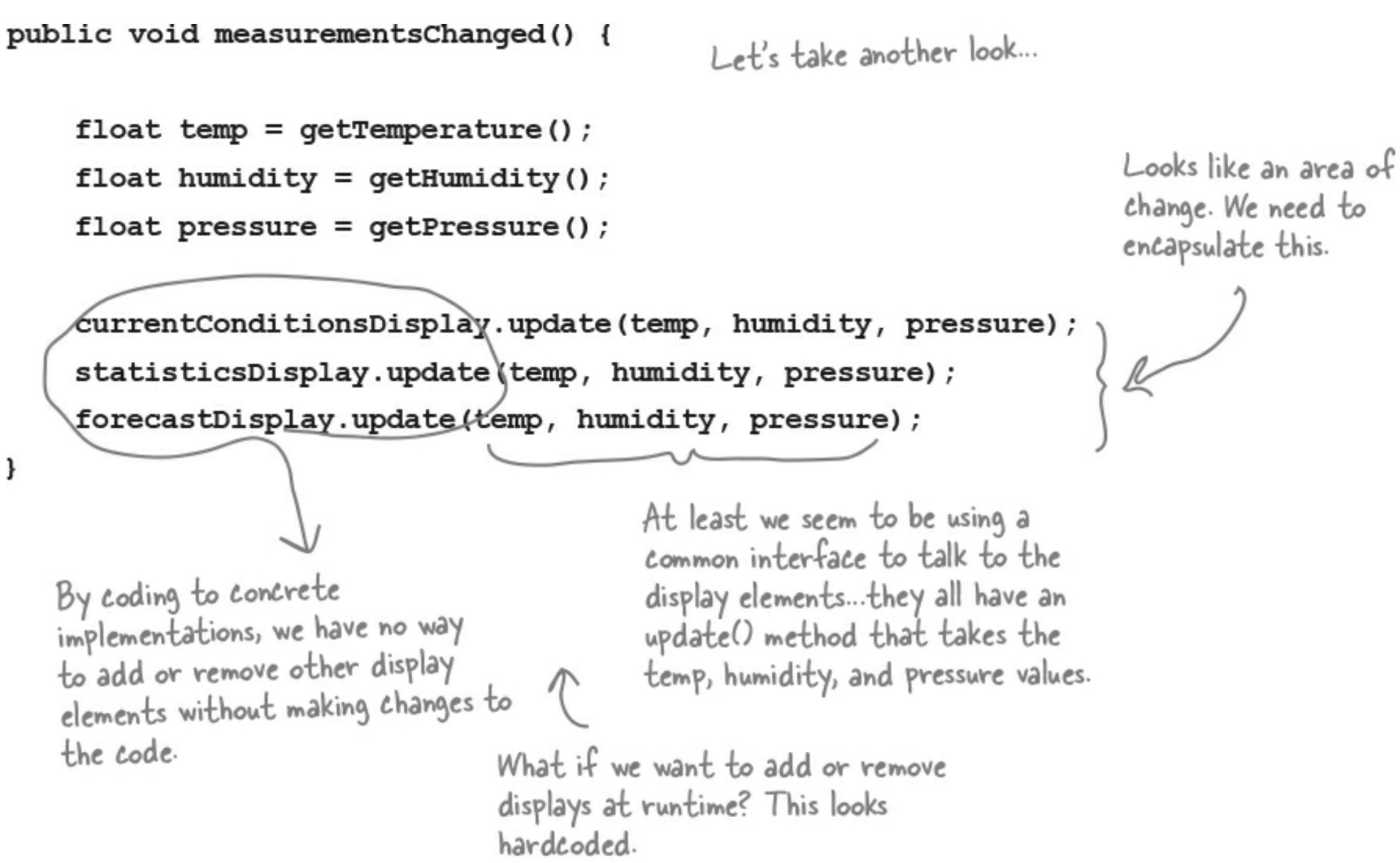
And here are our code additions...

First, we grab the most recent measurements by calling the WeatherData's getter methods. We assign each value to an appropriately named variable.

Next we're going to update each display...

...by calling its update method and passing it the most recent measurements.

First Approach



```
public 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);
}
```

Let's take another look...

Looks like an area of change. We need to encapsulate this.

At least we seem to be using a common interface to talk to the display elements...they all have an update() method that takes the temp, humidity, and pressure values.

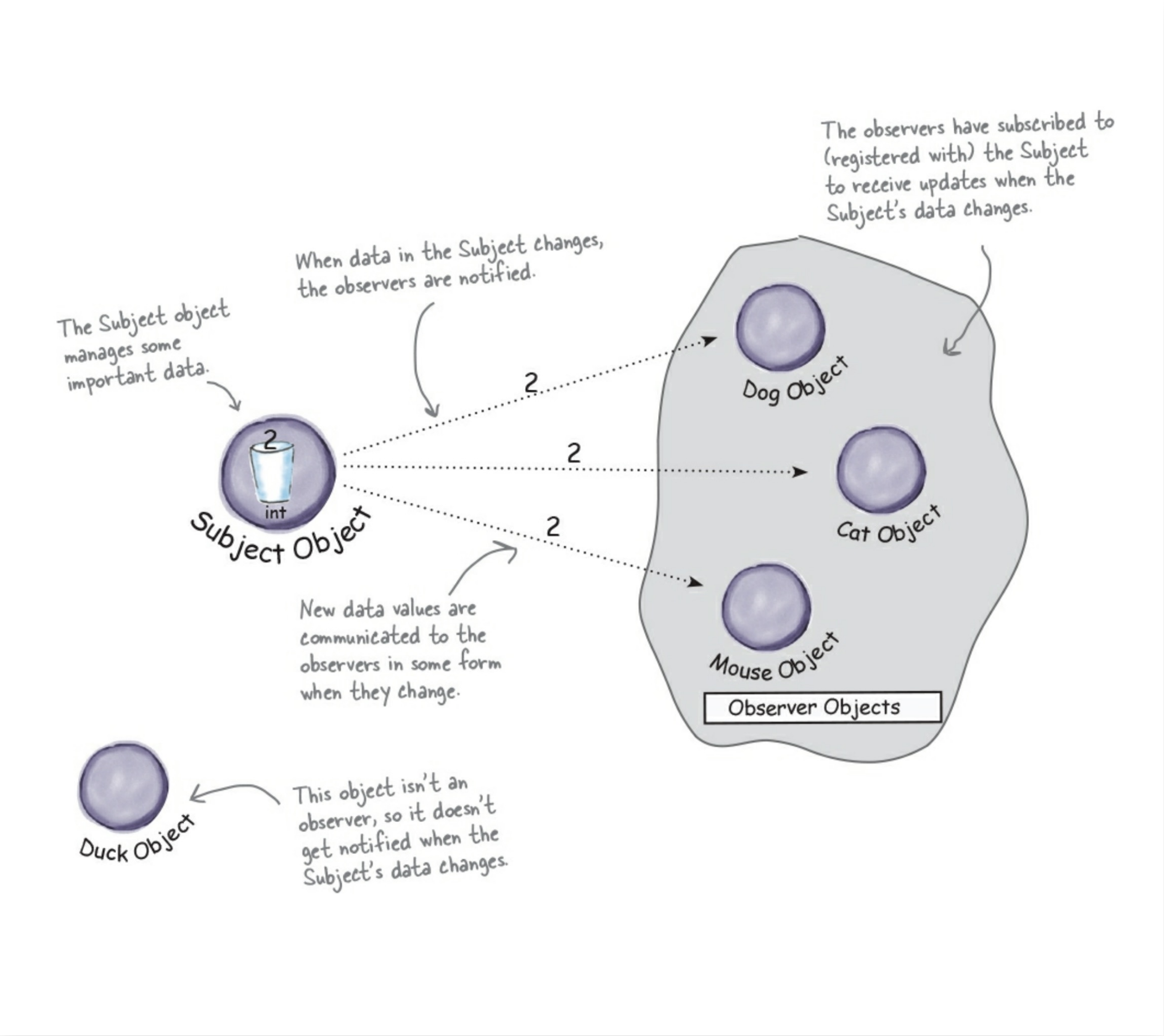
By coding to concrete implementations, we have no way to add or remove other display elements without making changes to the code.

What if we want to add or remove displays at runtime? This looks hardcoded.

Observer Pattern

- How newspaper subscription works
  - 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 get new newspapers.
  - You unsubscribe when you don’t want papers 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.

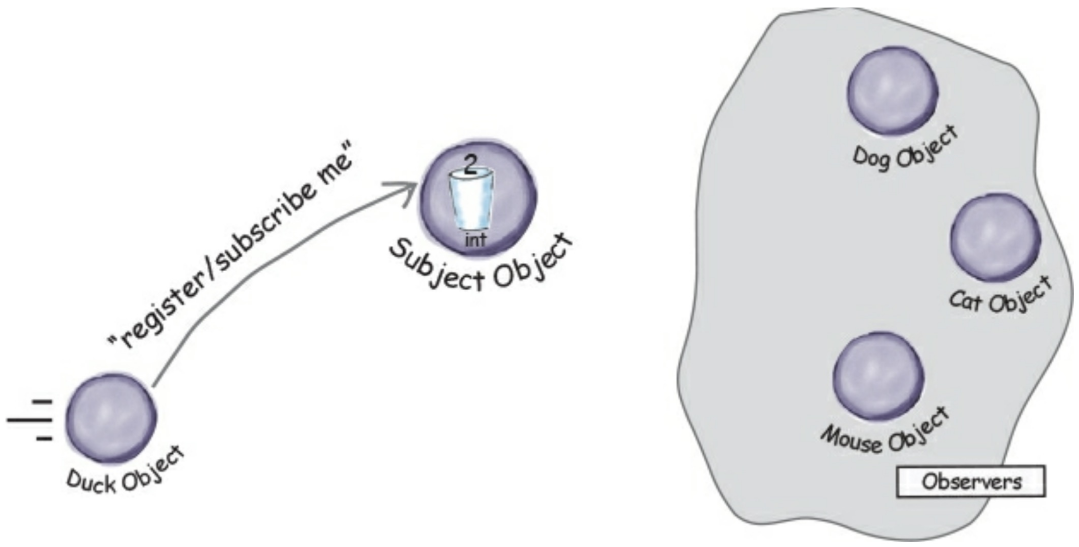
Publishers + Subscribers = Observer Pattern



The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.
When data in the Subject changes, the observers are notified.
The Subject object manages some important data.
2
2
2
2
int
Subject Object
Dog Object
Cat Object
Mouse Object
Observer Objects
New data values are communicated to the observers in some form when they change.
Duck Object
This object isn't an observer, so it doesn't get notified when the Subject's data changes.

Observer pattern

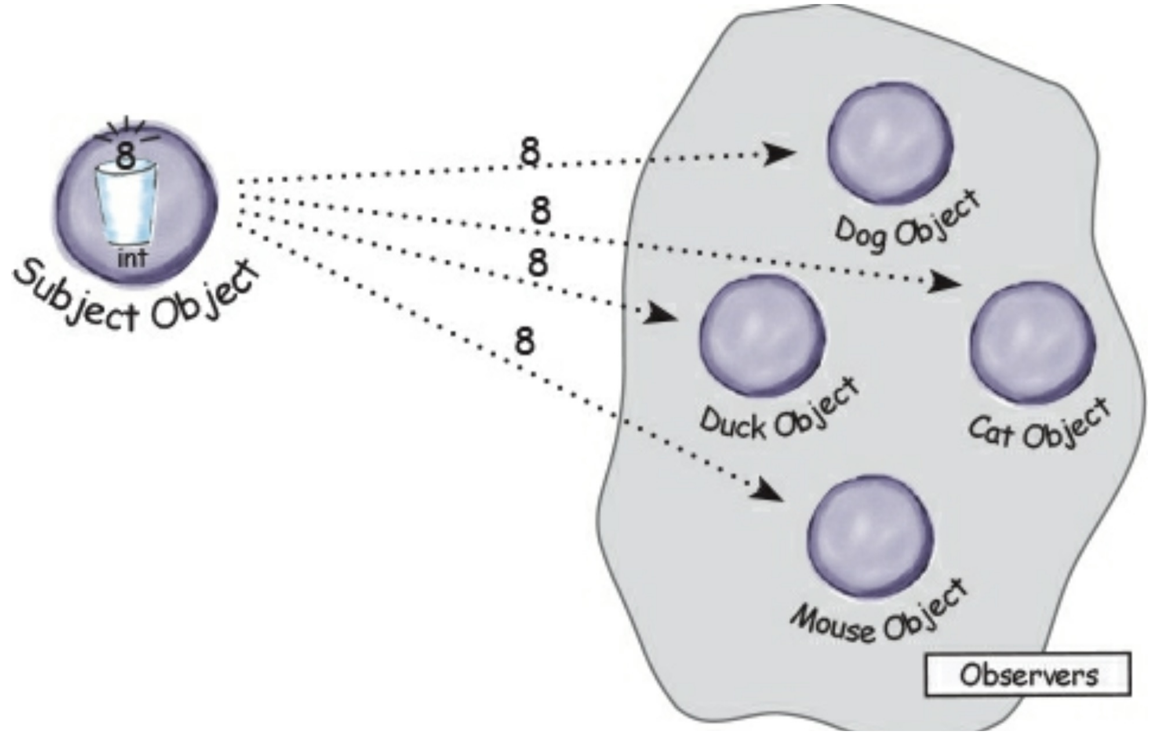
- A Duck object comes along and tells the Subject that he wants to become an observer.



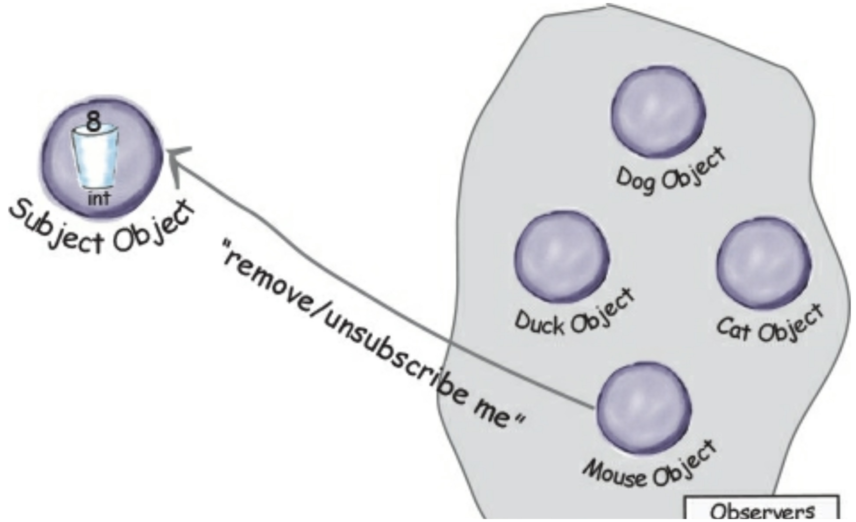
Observer pattern

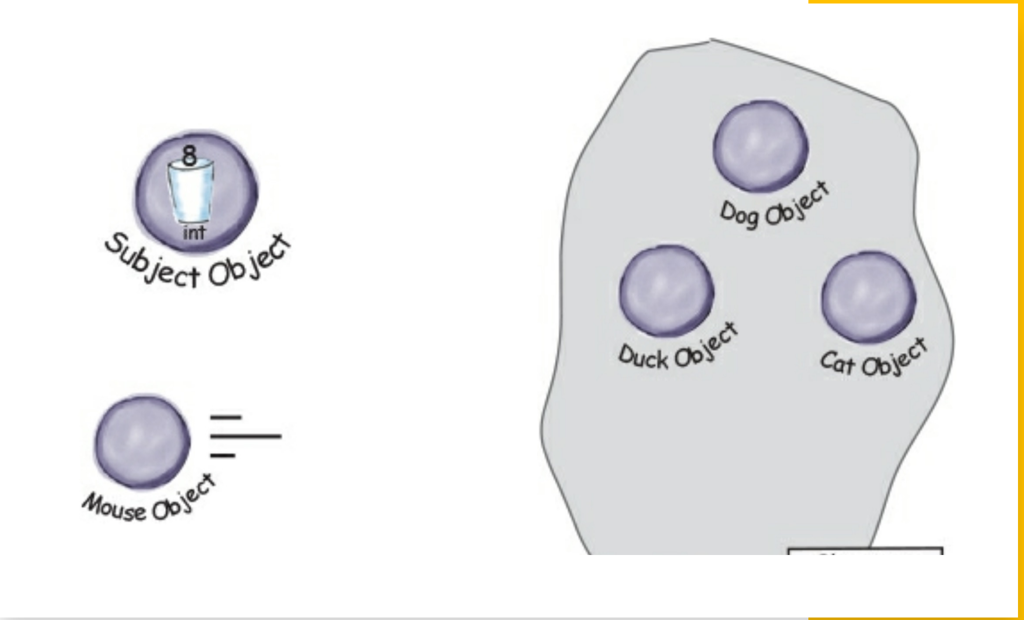
- Duck object starts receiving the new values

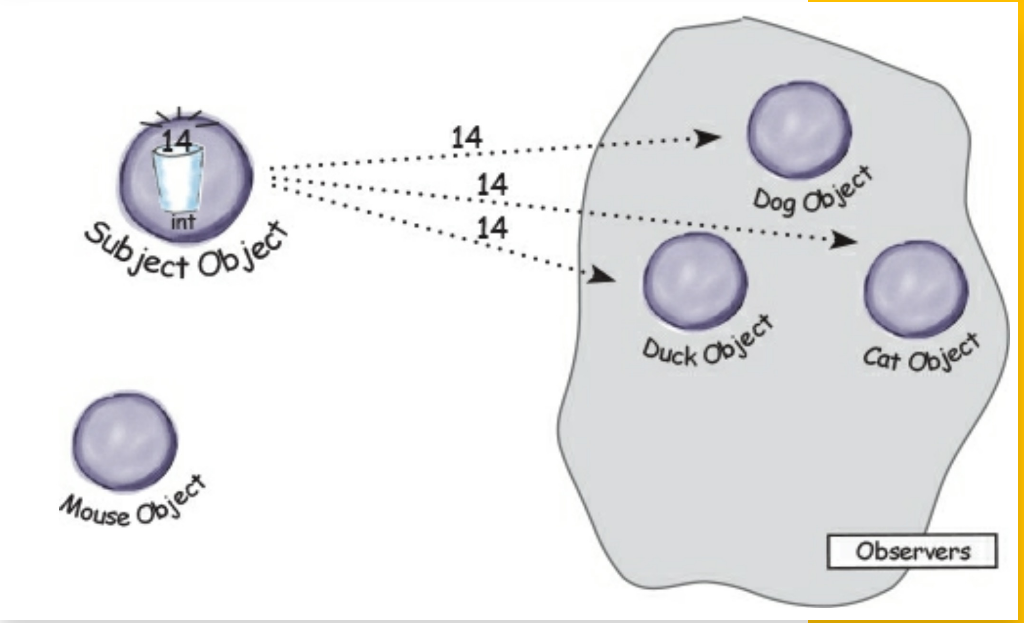
- Mouse object asks to be removed



Observer Pattern

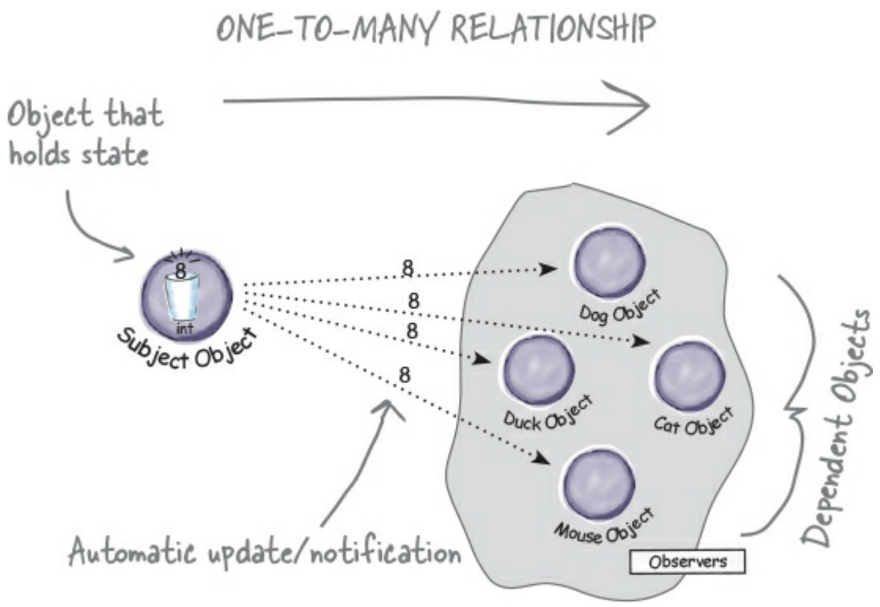




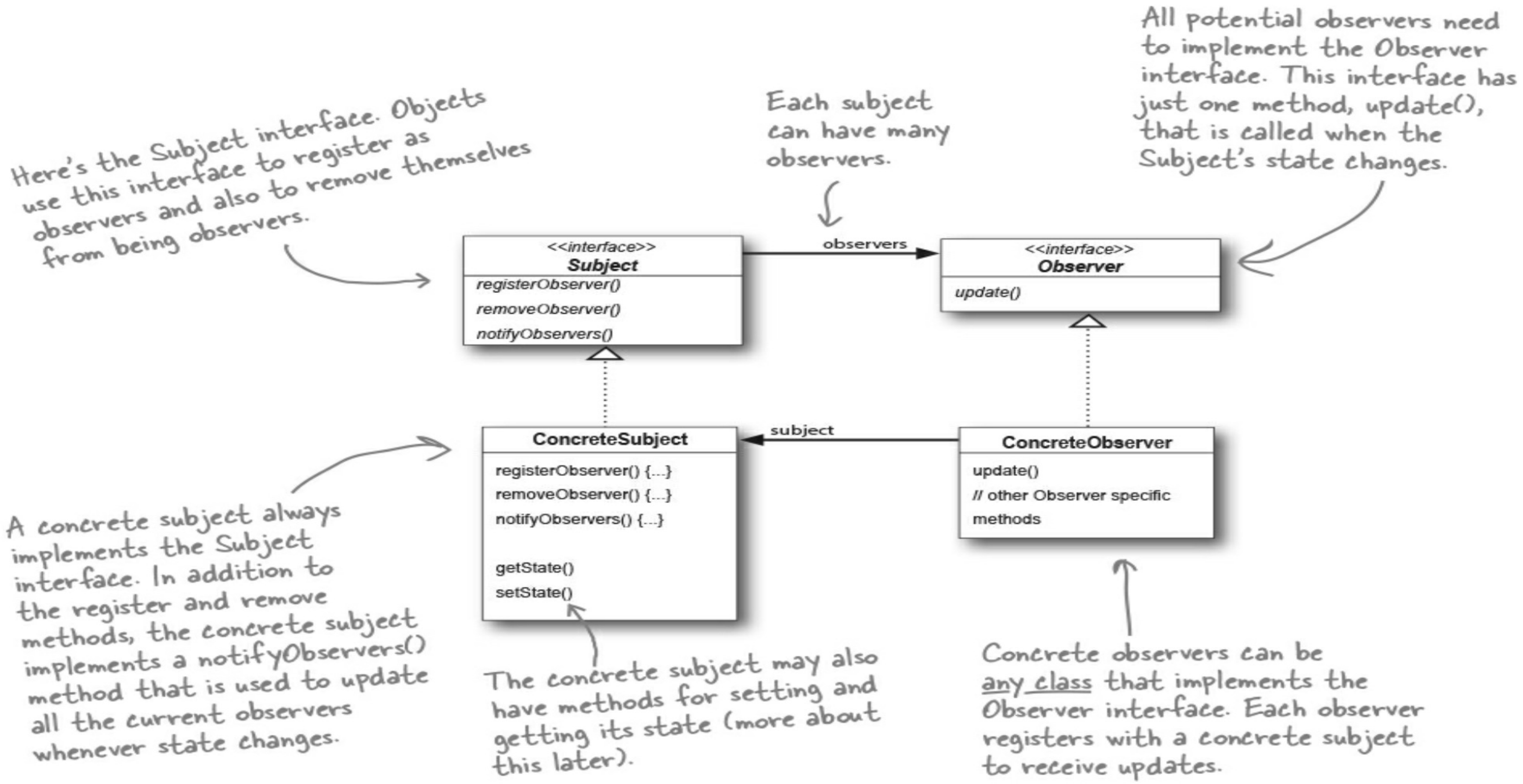


Observer pattern

- 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.



The Class diagram



Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers.

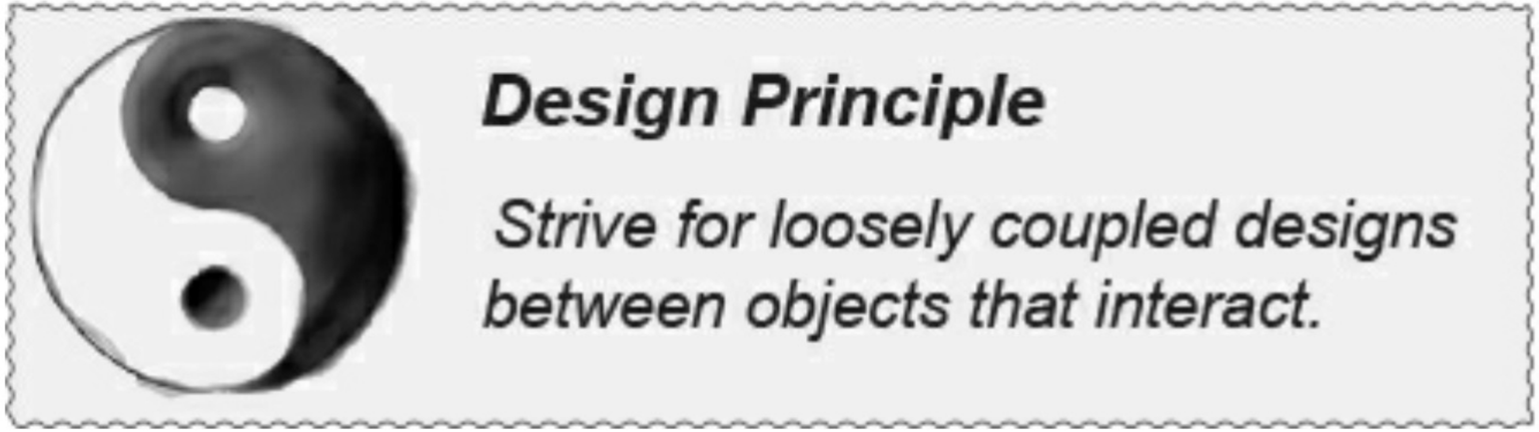
All potential observers need to implement the Observer interface. This interface has just one method, update(), that is called when the Subject's state changes.

A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

The concrete subject may also have methods for setting and getting its state (more about this later).

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Design Principle



Design Principle

Strive for loosely coupled designs between objects that interact.

Look! We have a new Design Principle!

The Power of Loose Coupling

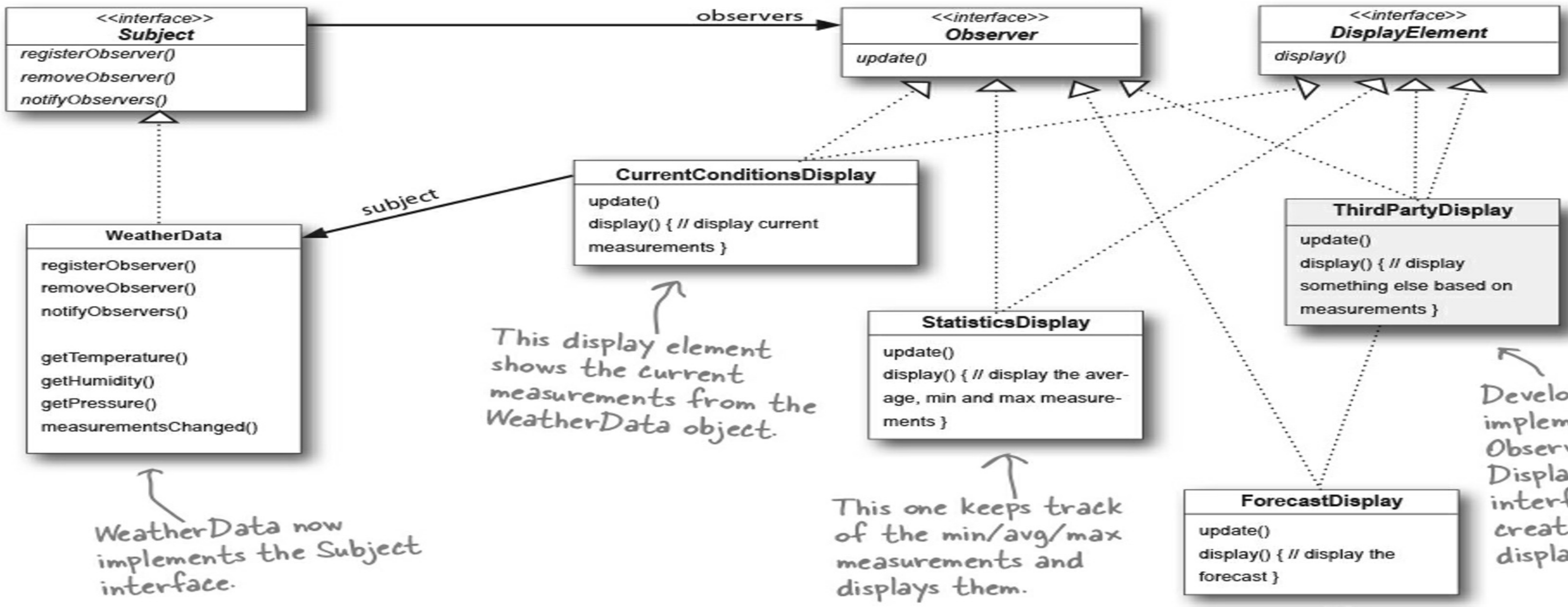
- When two objects are loosely coupled, they can interact, but they typically have very little knowledge of each other.
- Loosely coupled designs often give us a lot of flexibility

Loose Coupling in Observer Pattern

- First, the only thing the subject knows about an observer is that it implements a certain interface (the Observer interface).
    - It doesn’t need to know the concrete class of the observer, what it does, or anything else about it.
- We can add new observers at any time.
    - Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep purring along. Likewise, we can remove observers at any time.
- We never need to modify the subject to add new types of observers.
    - Let’s say we have a new concrete class come along that needs to be an observer. We don’t need to make any changes to the subject to accommodate the new class type; all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn’t care; it will deliver notifications to any object that implements the Observer interface.
- We can reuse subjects or observers independently of each other.
    - If we have another use for a subject or an observer, we can easily reuse them because the two aren’t tightly coupled.
- Changes to either the subject or an observer will not affect the other.
    - Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the Subject or Observer interfaces.

Going Back to the Weather Problem

- Is observer pattern the right choice here?
- What could be other programming problems where you can use the observer pattern?



Here's our Subject interface. This should look familiar.

All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.

Let's also create an interface for all display elements to implement. The display elements just need to implement a display() method.

This display element shows the current measurements from the WeatherData object.

WeatherData now implements the Subject interface.

This one keeps track of the min/avg/max measurements and displays them.

Developers can implement the Observer and DisplayElement interfaces to create their own display element.

This display shows the weather forecast based on the barometer.

These three display elements should have a pointer to WeatherData labeled "subject" too, but boy would this diagram start to look like spaghetti if they did.

Reference

Erich, Gamma; Helm Richard; Johnson Ralph; Vlissides John. Design Patterns. Pearson Education. Kindle Edition.

Freeman, Eric; Robson, Elisabeth. Head First Design Patterns. O'Reilly Media. Kindle Edition.