finished 1st round this whole book on 14 - July - 09 3:49 AM

decoupled - there is one class between 2 different class
.
loosecoupled - two class know each other by abstract or interface
.
tightcoupled - two classes know each other by class reference variable

2 the Observer Pattern

Keeping your Objects in the know

Hey Jerry, I'm notifying everyone that the Patterns Group meeting moved to Saturday night. We're going to be talking about the Observer Pattern. That pattern is the best! It's the BEST, Jerry!

Don't miss out when something interesting happens! We've got a pattern that keeps your objects in the know when something they might care about happens. Objects can even decide at runtime whether they want to be kept informed. The Observer Pattern is one of the most heavily used patterns in the JDK, and it's incredibly useful. Before we're done, we'll also look at one to many relationships and loose coupling (yeah, that's right, we said coupling). With Observer, you'll be the life of the Patterns Party.

Congratulations!

Your team has just won the contract to build Weather-O-Rama, Inc.'s next generation, Internet-based Weather Monitoring Station.

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 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 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,

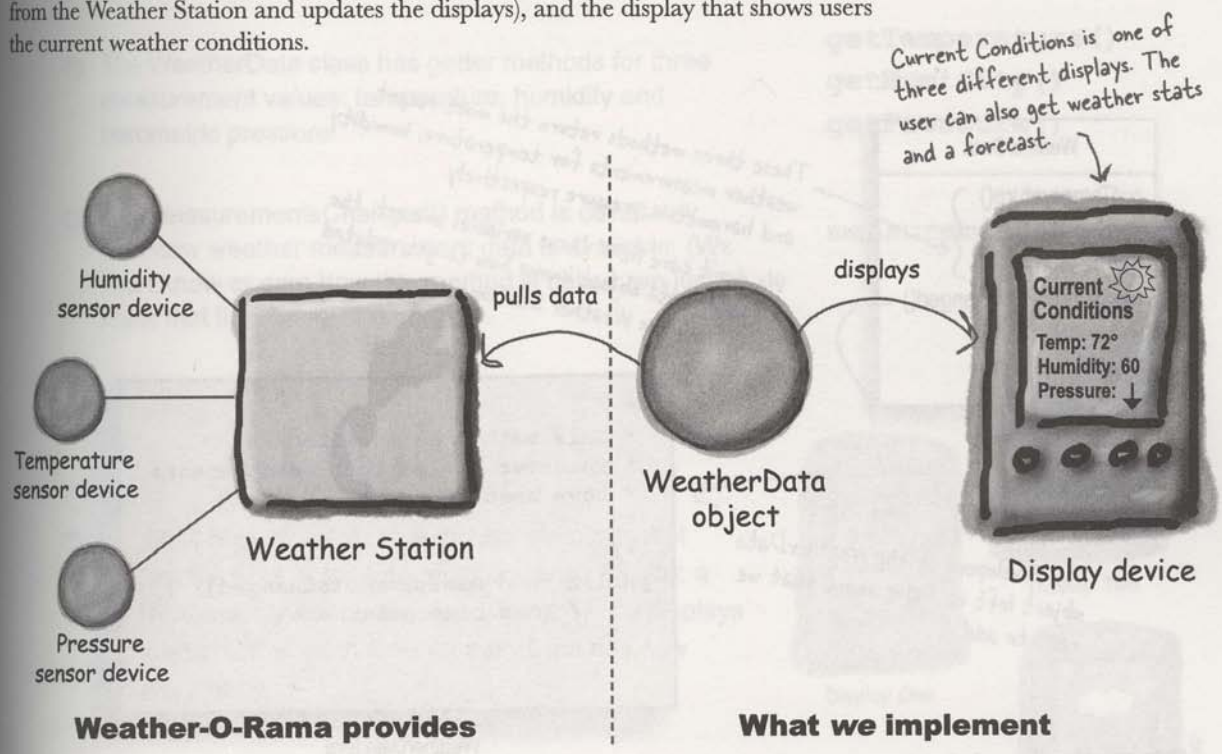
Johnny Hurricane

Johnny Hurricane, CEO

P.S. We are overnighting the WeatherData source files to you.

The Weather Monitoring application overview

The three players in the system are the weather station (the physical device that acquires the actual weather data), the WeatherData object (that tracks the data coming from the Weather Station and updates the displays), and the display that shows users the current weather conditions.

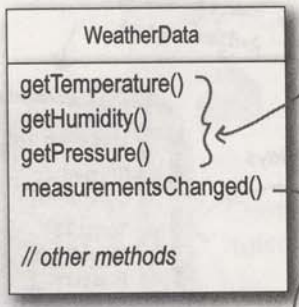


The WeatherData object knows how to talk to the physical Weather Station, to get updated data. The WeatherData object then updates its displays for the three different display elements: Current Conditions (shows temperature, humidity, and pressure), Weather Statistics, and a simple forecast.

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.

Unpacking the WeatherData class

As promised, the next morning the WeatherData source files arrive. Peeking inside the code, things look pretty straightforward:



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

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

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

WeatherData.java

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

Remember, this Current Conditions is just ONE of three different display screens.

Display device

Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.

What do we know so far?

The spec from Weather-O-Rama wasn't all that clear, but we have to figure out what we need to do. So, what do we know so far?

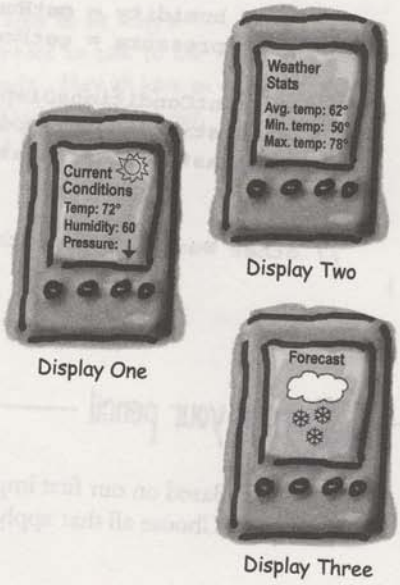
- The WeatherData class has getter methods for three measurement values: temperature, humidity and barometric pressure.

```
getTemperature()
getHumidity()
getPressure()
```

- The measurementsChanged() method is called any time new weather measurement data is available. (We don't know or care how this method is called; we just know that it *is*.

```
measurementsChanged()
```

- We 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 each time WeatherData has new measurements.



- The system must be expandable—other developers can create new custom display elements and users can add or remove as many display elements as they want to the application. Currently, we know about only the initial *three* display types (current conditions, statistics and forecast).

Future displays

Taking a first, misguided SWAG at the Weather Station

Here's a first implementation possibility—we'll take the hint from the Weather-O-Rama developers and add our code to the measurementsChanged() method:

```
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
}
```

Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

Sharpen your pencil

Based on our first implementation, which of the following apply? (Choose all that apply.)

- ❑ A. We are coding to concrete implementations, not interfaces.
- ❑ B. For every new display element we need to alter code.
- ❑ C. We have no way to add (or remove) display elements at run time.
- ❑ D. The display elements don't implement a common interface.
- ❑ E. We haven't encapsulated the part that changes.
- ❑ F. We are violating encapsulation of the WeatherData class.

Definition of SWAG: Scientific Wild A** Guess

What's wrong with our implementation?

Think back to all those Chapter 1 concepts and principles...

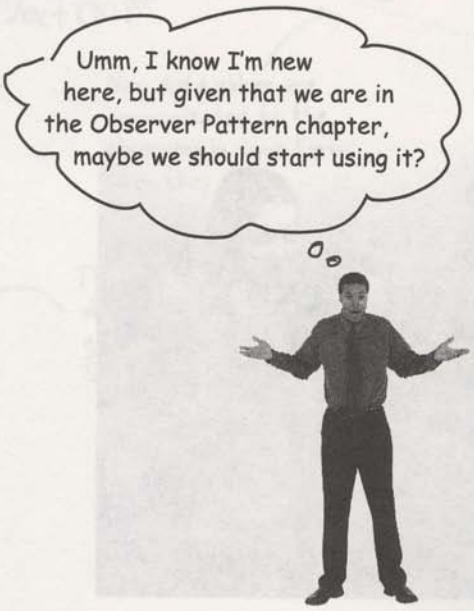
```
public void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

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 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 program.



We'll take a look at Observer, then come back and figure out how to apply it to the weather monitoring app.

Meet the Observer Pattern

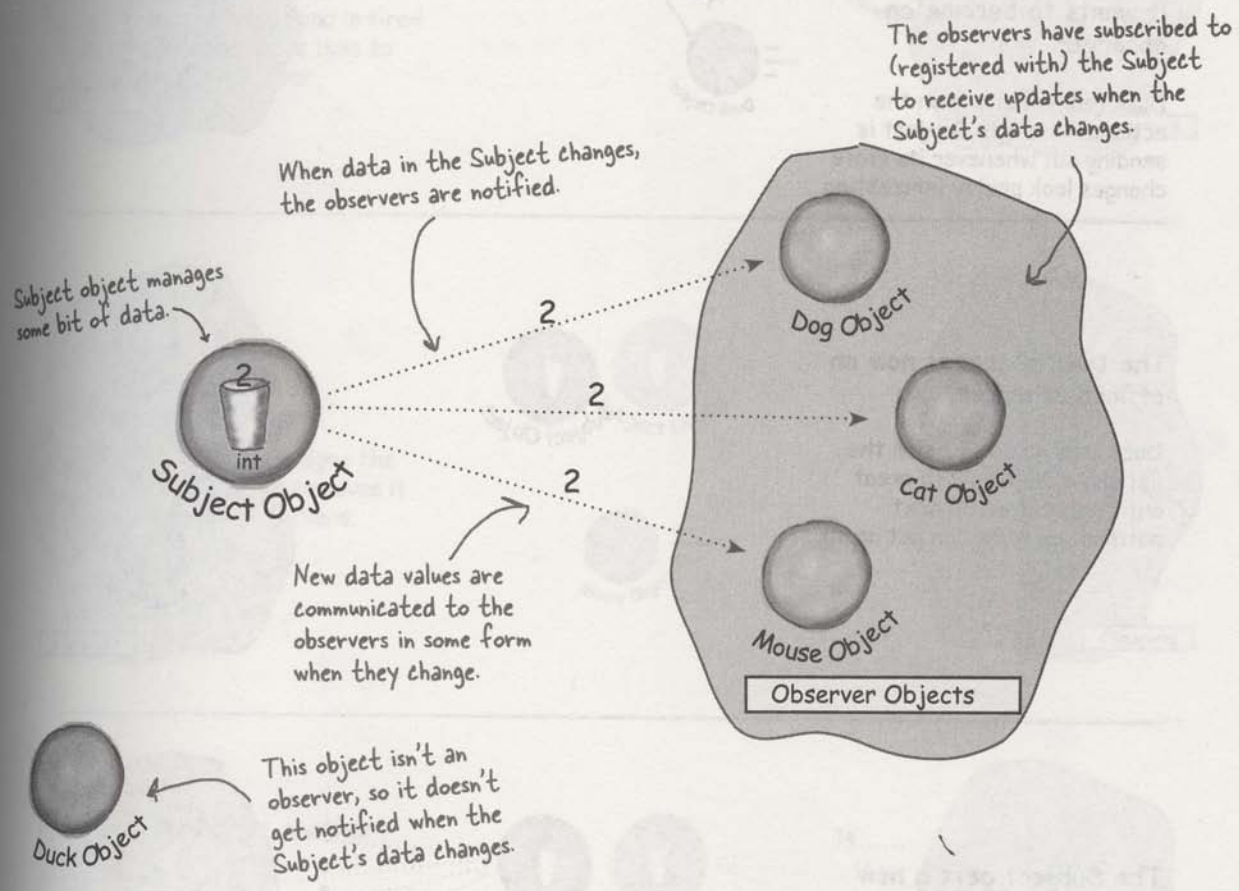
You know how newspaper or magazine subscriptions work:

1. A newspaper publisher goes into business and begins publishing newspapers.
2. 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.
3. You unsubscribe when you don't want papers anymore, and they stop being delivered.
4. While the publisher remains in business, people, hotels, airlines and other businesses constantly subscribe and unsubscribe to the newspaper.

Publishers + Subscribers = Observer Pattern

If you understand newspaper subscriptions, you pretty much understand the Observer Pattern, only we call the publisher the SUBJECT and the subscribers the OBSERVERS.

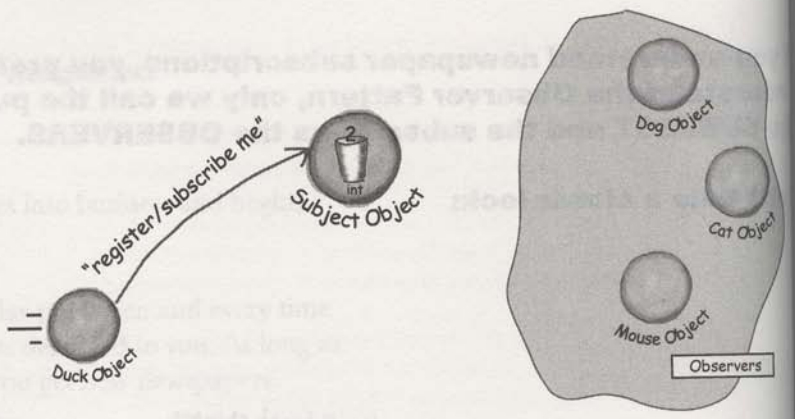
Let's take a closer look:



A day in the life of the Observer Pattern

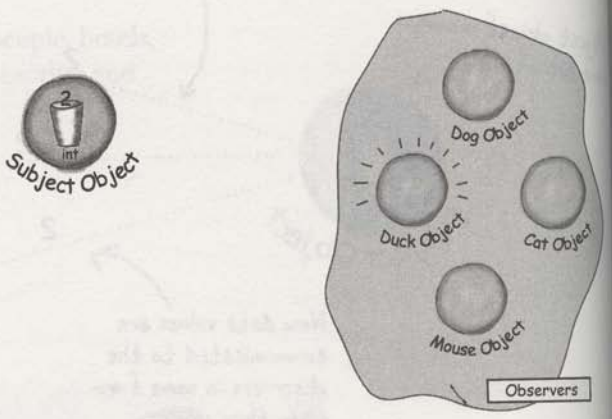
A Duck object comes along and tells the Subject that it wants to become an observer.

Duck really wants in on the action; those ints Subject is sending out whenever its state changes look pretty interesting...



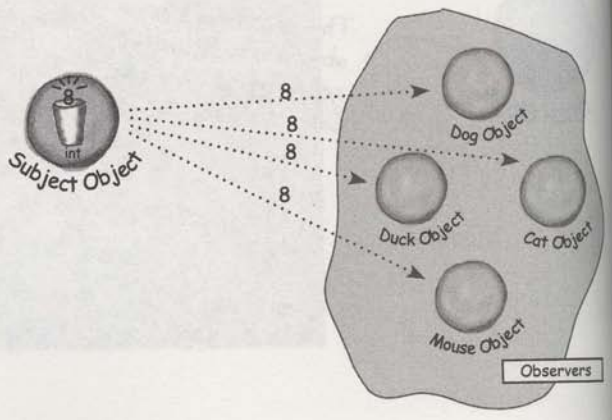
The Duck object is now an official observer.

Duck is psyched... he's on the list and is waiting with great anticipation for the next notification so he can get an int.



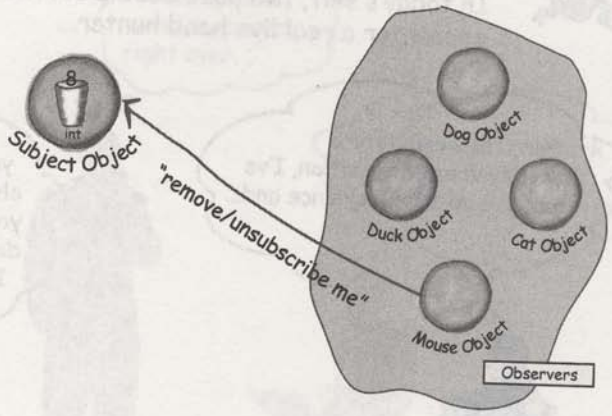
The Subject gets a new data value!

Now Duck and all the rest of the observers get a notification that the Subject has changed.



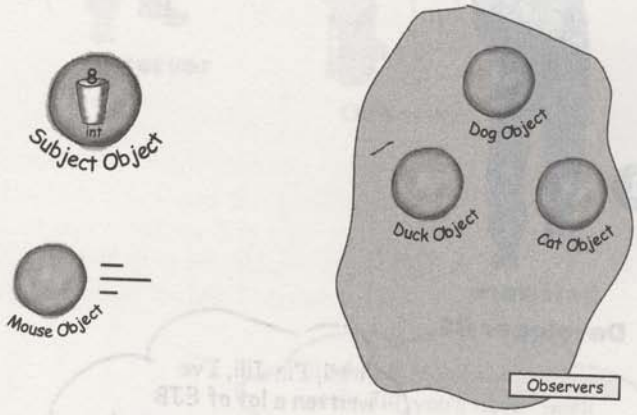
The Mouse object asks to be removed as an observer.

The Mouse object has been getting ints for ages and is tired of it, so it decides it's time to stop being an observer.



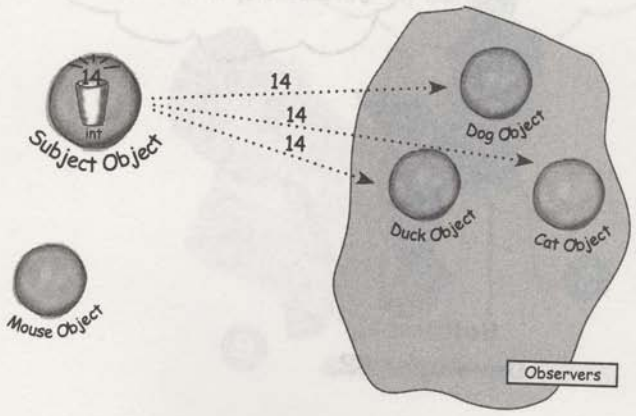
Mouse is outta here!

The Subject acknowledges the Mouse's request and removes it from the set of observers.



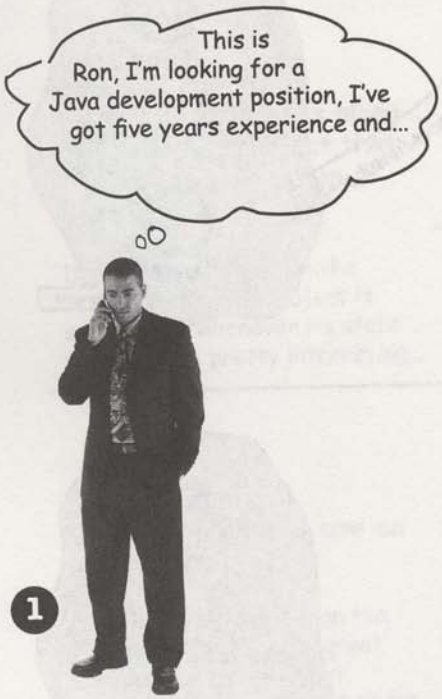
The Subject has another new int.

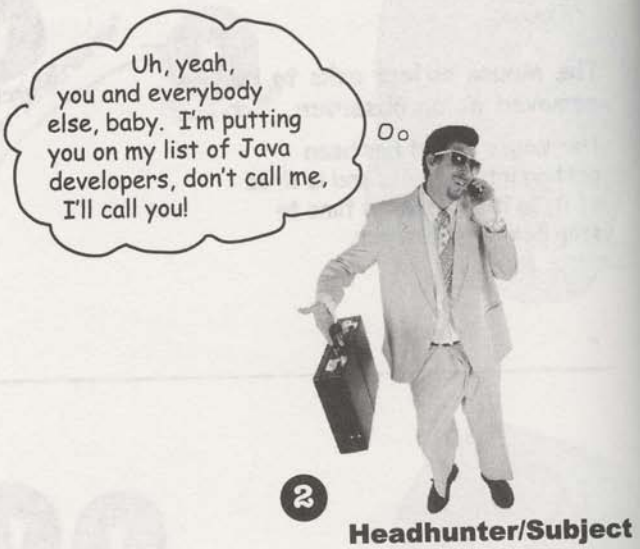
All the observers get another notification, except for the Mouse who is no longer included. Don't tell anyone, but the Mouse secretly misses those ints... maybe it'll ask to be an observer again some day.

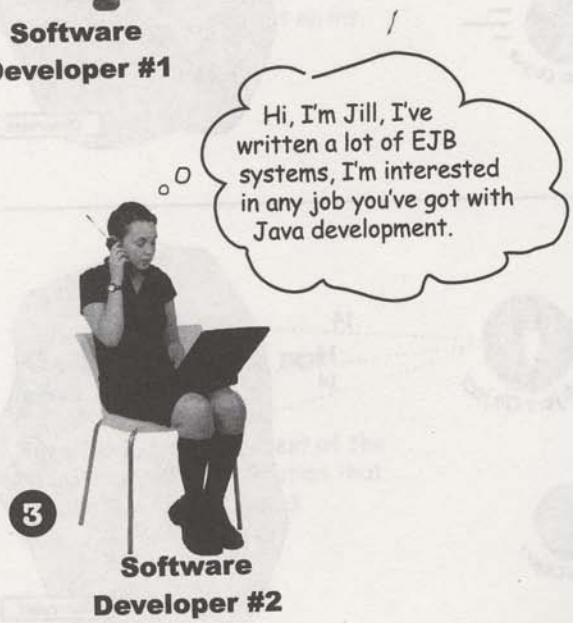


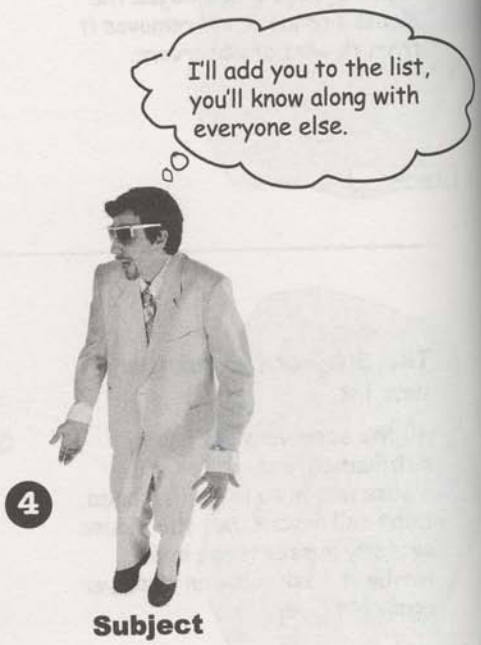
Five minute drama: a subject for observation

In today's skit, two post-bubble software developers encounter a real live head hunter...

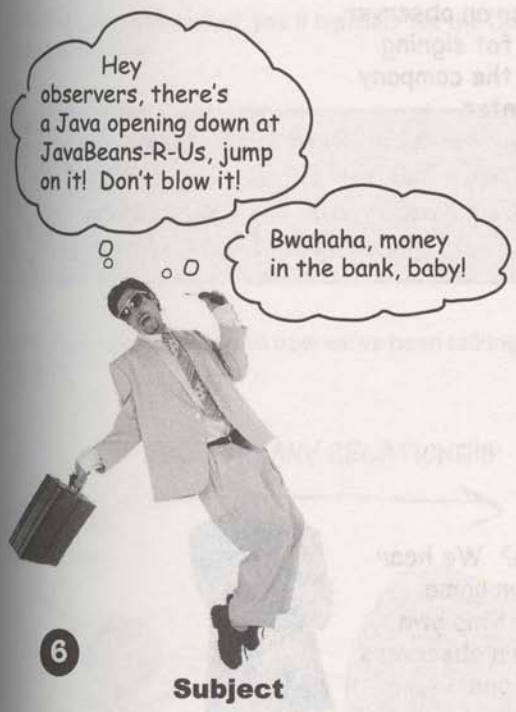


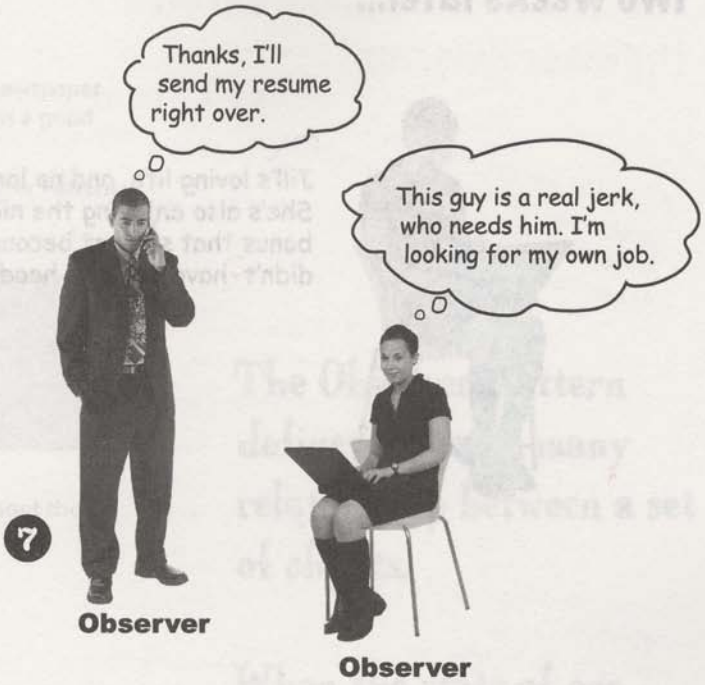




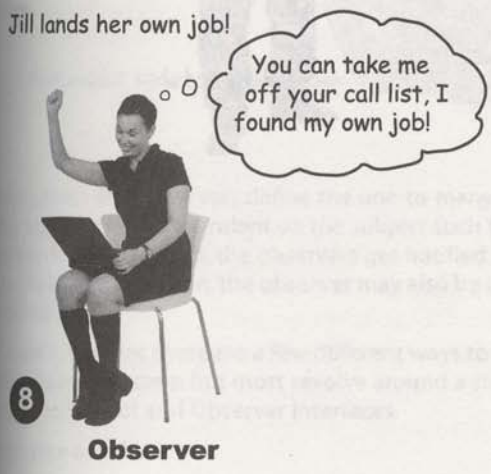


5 Meanwhile for Ron and Jill life goes on; if a Java job comes along, they'll get notified, after all, they are observers.





Jill lands her own job!



Two weeks later...

Jill's loving life, and no longer an observer. She's also enjoying the nice fat signing bonus that she got because the company didn't have to pay a headhunter.

But what has become of our dear Ron? We hear he's beating the headhunter at his own game. He's not only still an observer, he's got his own call list now, and he is notifying his own observers. Ron's a subject and an observer all in one.

The Observer Pattern defined

When you're trying to picture the Observer Pattern, a newspaper subscription service with its publisher and subscribers is a good way to visualize the pattern.

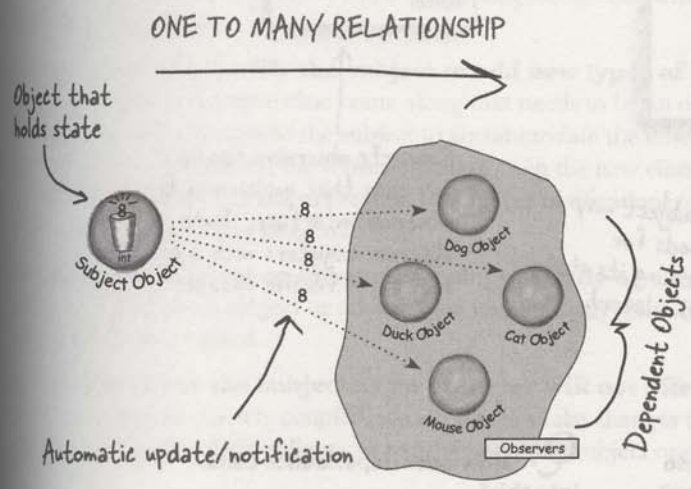
In the real world however, you'll typically see the Observer Pattern defined like this:

> **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 Observer Pattern defines a one-to-many relationship between a set of objects.

When the state of one object changes, all of its dependents are notified.

Let's relate this definition to how we've been talking about the pattern:

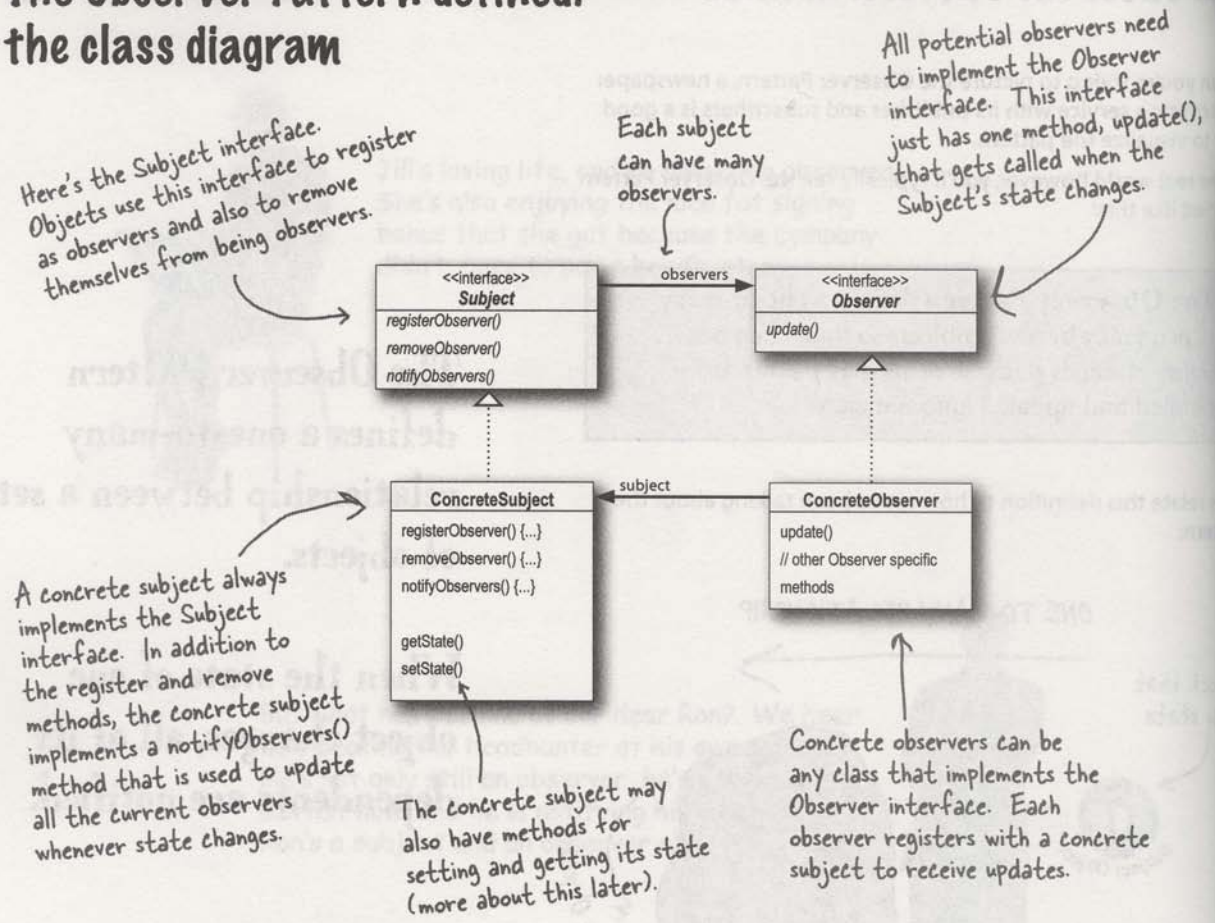


The subject and observers define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the style of notification, the observer may also be updated with new values.

As you'll discover, there are a few different ways to implement the Observer Pattern but most revolve around a class design that includes Subject and Observer interfaces.

Let's take a look...

The Observer Pattern defined: the class diagram



Q: What does this have to do with one-to-many relationships?

A: With the Observer pattern, the Subject is the object that contains the state and controls it. So, there is ONE subject with state. The observers, on the other hand, use the state, even if they don't own it. There are many observers and they rely on the Subject to tell them when its state changes. So there is a relationship between the ONE Subject to the MANY Observers.

Q: How does dependence come into this?

A: Because the subject is the sole owner of that data, the observers are dependent on the subject to update them when the data changes. This leads to a cleaner OO design than allowing many objects to control the same data.

The power of Loose Coupling

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.

Why?

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.

How many different kinds of change can you identify here?

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.

Design Principle

Strive for loosely coupled designs between objects that interact.

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

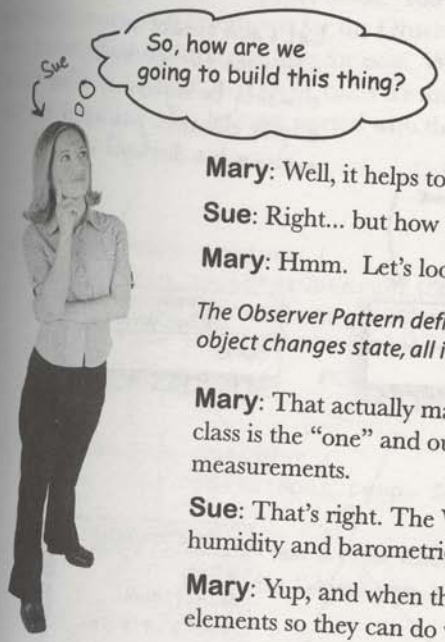
Sharpen your pencil

Before moving on, try sketching out the classes 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.

If you need a little help, read the next page; your teammates are already talking about how to design the Weather Station.

Cubicle conversation

Back to the Weather Station project, your teammates have already started thinking through the problem...



Mary: Well, it helps to know we're using the Observer Pattern.

Sue: Right... but how do we apply it?

Mary: Hmm. Let's look at the definition again:

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

Mary: That actually makes some sense when you think about it. Our WeatherData class is the "one" and our "many" is the various display elements that use the weather measurements.

Sue: That's right. The WeatherData class certainly has state... that's the temperature, humidity and barometric pressure, and those definitely change.

Mary: Yup, and when those measurements change, we have to notify all the display elements so they can do whatever it is they are going to do with the measurements.

Sue: Cool, I now think I see how the Observer Pattern can be applied to our Weather Station problem.

Mary: There are still a few things to consider that I'm not sure I understand yet.

Sue: Like what?

Mary: For one thing, how do we get the weather measurements to the display elements?

Sue: Well, looking back at the picture of the Observer Pattern, if we make the WeatherData object the subject, and the display elements the observers, then the displays will register themselves with the WeatherData object in order to get the information they want, right?

Mary: Yes... and once the Weather Station knows about a display element, then it can just call a method to tell it about the measurements.

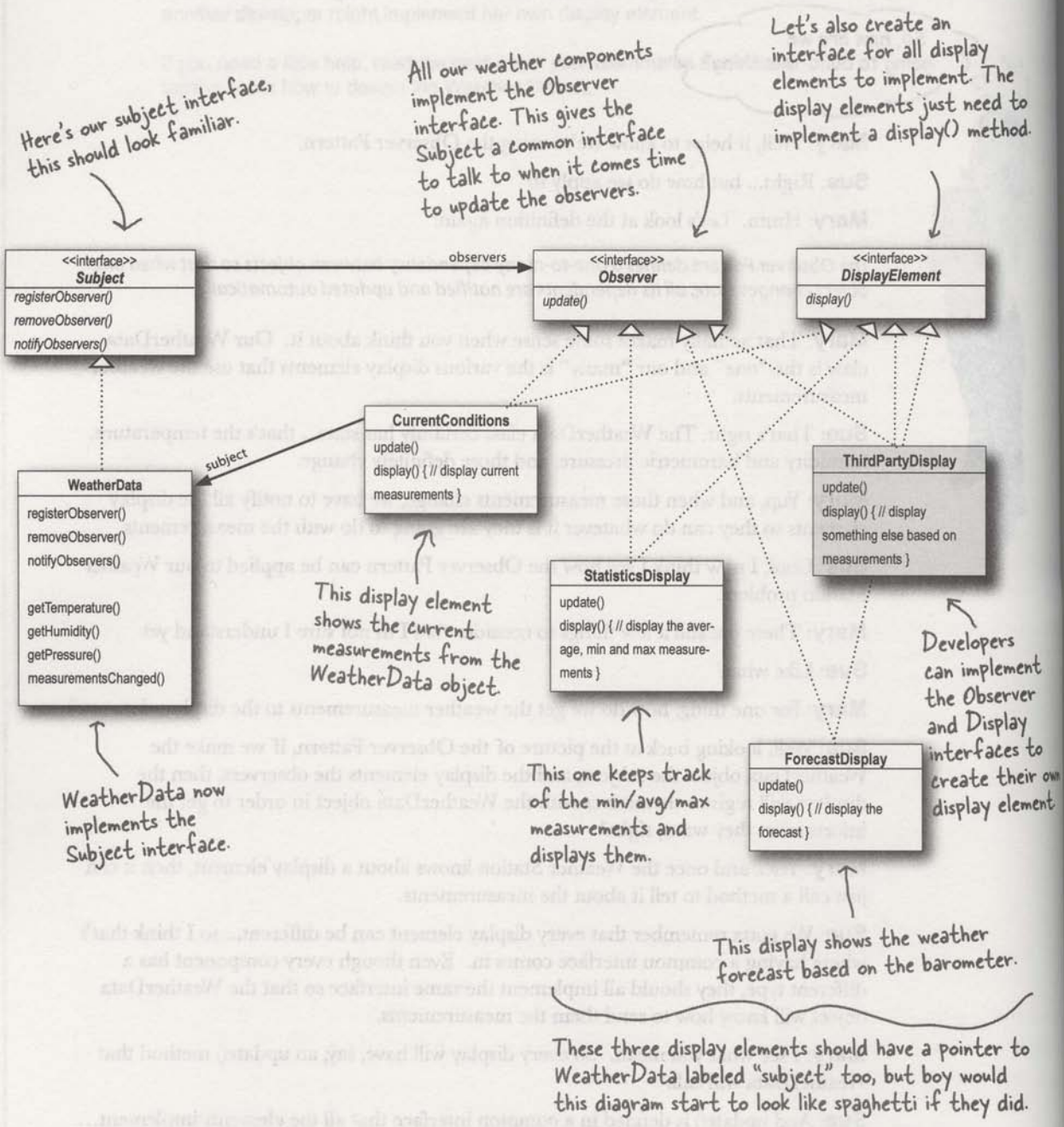
Sue: We gotta remember that every display element can be different... so I think that's where having a common interface comes in. Even though every component has a different type, they should all implement the same interface so that the WeatherData object will know how to send them the measurements.

Mary: I see what you mean. So every display will have, say, an update() method that WeatherData will call.

Sue: And update() is defined in a common interface that all the elements implement...

Designing the Weather Station

How does this diagram compare with yours?



Implementing the Weather Station

We're going to start our implementation using the class diagram and following Mary and Sue's lead (from a few pages back). You'll see later in this chapter that Java provides some built-in support for the Observer pattern, however, we're going to get our hands dirty and roll our own for now. While in some cases you can make use of Java's built-in support, in a lot of cases it's more flexible to build your own (and it's not all that hard). So, let's get started with the interfaces:

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

These are the state values the Observers get from the Subject when a weather measurement changes

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we're following Mary and Sue's lead and passing the measurements to the observers.

```
public interface DisplayElement {
    public void display();
}
```

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.

BRAIN POWER

Mary and Sue thought that passing the measurements directly to the observers was the most straightforward method of updating state. Do you think this is wise? Hint: is this an area of the application that might change in the future? If it did change, would the change be well encapsulated, or would it require changes in many parts of the code?

Can you think of other ways to approach the problem of passing the updated state to the observers?

Don't worry, we'll come back to this design decision after we finish the initial implementation.

Implementing the Subject interface in WeatherData

Remember our first attempt at implementing the WeatherData class at the beginning of the chapter? You might want to refresh your memory. Now it's time to go back and do things with the Observer Pattern in mind...

REMEMBER: we don't provide import and package statements in the code listings. Get the complete source code from the wickedlysmart web site. You'll find the URL on page xxxiii in the Intro.

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // other WeatherData methods here
}
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

When an observer registers, we just add it to the end of the list.

Likewise, when an observer wants to un-register, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

Here we implement the Subject Interface.

We notify the Observers when we get updated measurements from the Weather Station.

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the web.

Now, let's build those display elements

Now that we've got our WeatherData class straightened out, it's time to build the Display Elements. Weather-O-Rama ordered three: the current conditions display, the statistics display and the forecast display. Let's take a look at the current conditions display; once you have a good feel for this display element, check out the statistics and forecast displays in the head first code directory. You'll see they are very similar.

This display implements Observer so it can get changes from the WeatherData object.

It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

The constructor is passed the weatherData object (the Subject) and we use it to register the display as an observer.

When update() is called, we save the temp and humidity and call display().

The display() method just prints out the most recent temp and humidity.

there are no Dumb Questions

Q: Is update() the best place to call display?

A: In this simple example it made sense to call display() when the values changed. However, you are right, there are much better ways to design the way the data gets displayed. We are going to see this when we get to the model-view-controller pattern.

Q: Why did you store a reference to the Subject? It doesn't look like you use it again after the constructor?

A: True, but in the future we may want to un-register ourselves as an observer and it would be handy to already have a reference to the subject.

Power up the Weather Station

❶ First, let's create a test harness

The Weather Station is ready to go, all we need is some code to glue everything together. Here's our first attempt. We'll come back later in the book and make sure all the components are easily pluggable via a configuration file. For now here's how it all works:

```
public class WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

First, create the WeatherData object.

If you don't want to download the code, you can comment out these two lines and run it.

Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.

❷ Run the code and let the Observer Pattern do its magic

```
File Edit Window Help StormyWeather
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

Sharpen your pencil

Johnny Hurricane, Weather-O-Rama's CEO just called, they can't possibly ship without a Heat Index display element. Here are the details:

The heat index is an index that combines temperature and humidity to determine the apparent temperature (how hot is actually feels). To compute the heat index, you take the temperature, T, and the relative humidity, RH, and use this formula:

$$
\begin{aligned}
\text{heatindex} = {} & 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T * RH \\
& + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * RH^2 + 3.45372 * 10^{-4} * T^2 * RH - 8.14971 * 10^{-4} * T * RH^2 + 1.02102 * 10^{-5} * T^2 * RH^2 - \\
& 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * RH^3 + 1.42721 * 10^{-6} * T^3 * RH \\
& + 1.97483 * 10^{-7} * T * RH^3 - 2.18429 * 10^{-8} * T^3 * RH^2 + 8.43296 * 10^{-10} * T^2 * RH^3 - 4.81975 * 10^{-11} * T^3 * RH^3
\end{aligned}
$$

So get typing!

Just kidding. Don't worry, you won't have to type that formula in; just create your own HeatIndexDisplay.java file and copy the formula from heatindex.txt into it.

You can get heatindex.txt from wickedlysmart.com

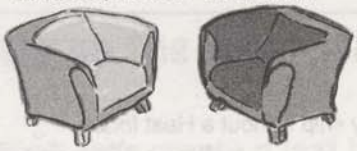
How does it work? You'd have to refer to *Head First Meteorology*, or try asking someone at the National Weather Service (or try a Google search).

When you finish, your output should look like this:

Here's what changed in this output.

```
File Edit Window Help OverdaRainbow
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Heat index is 82.95535
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Heat index is 86.90124
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
Heat index is 83.64967
%
```

Fireside Chats



Tonight's talk: **A Subject and Observer spar over the right way to get state information to the Observer.**

Subject

I'm glad we're finally getting a chance to chat in person.

Well, I do my job, don't I? I always tell you what's going on... Just because I don't really know who you are doesn't mean I don't care. And besides, I do know the most important thing about you—you implement the Observer interface.

Oh yeah, like what?

Well *excuuuse* me. I have to send my state with my notifications so all you lazy Observers will know what happened!

Well... I guess that might work. I'd have to open myself up even more though to let all you Observers come in and get the state that you need. That might be kind of dangerous. I can't let you come in and just snoop around looking at everything I've got.

Observer

Really? I thought you didn't care much about us Observers.

Well yeah, but that's just a small part of who I am. Anyway, I know a lot more about you...

Well, you're always passing your state around to us Observers so we can see what's going on inside you. Which gets a little annoying at times...

Ok, wait just a minute here; first, we're not lazy, we just have other stuff to do in between your oh-so-important notifications, Mr. Subject, and second, why don't you let us come to you for the state we want rather than pushing it out to just everyone?

Subject

Yes, I could let you **pull** my state. But won't that be less convenient for you? If you have to come to me every time you want something, you might have to make multiple method calls to get all the state you want. That's why I like **push** better... then you have everything you need in one notification.

Well, I can see the advantages to doing it both ways. I have noticed that there is a built-in Java Observer Pattern that allows you to use either push or pull.

Great... maybe I'll get to see a good example of pull and change my mind.

Observer

Why don't you just write some public getter methods that will let us pull out the state we need?

Don't be so pushy! There's so many different kinds of us Observers, there's no way you can anticipate everything we need. Just let us come to you to get the state we need. That way, if some of us only need a little bit of state, we aren't forced to get it all. It also makes things easier to modify later. Say, for example, you expand yourself and add some more state, well if you use pull, you don't have to go around and change the update calls on every observer, you just need to change yourself to allow more getter methods to access our additional state.

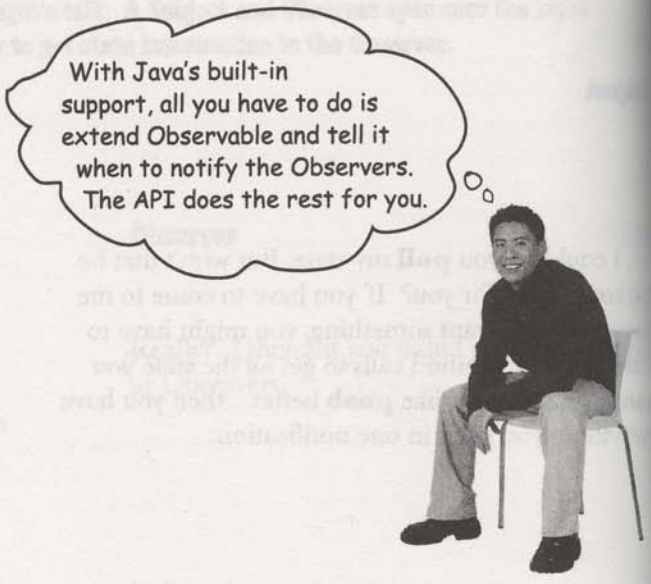
Oh really? I think we're going to look at that next....

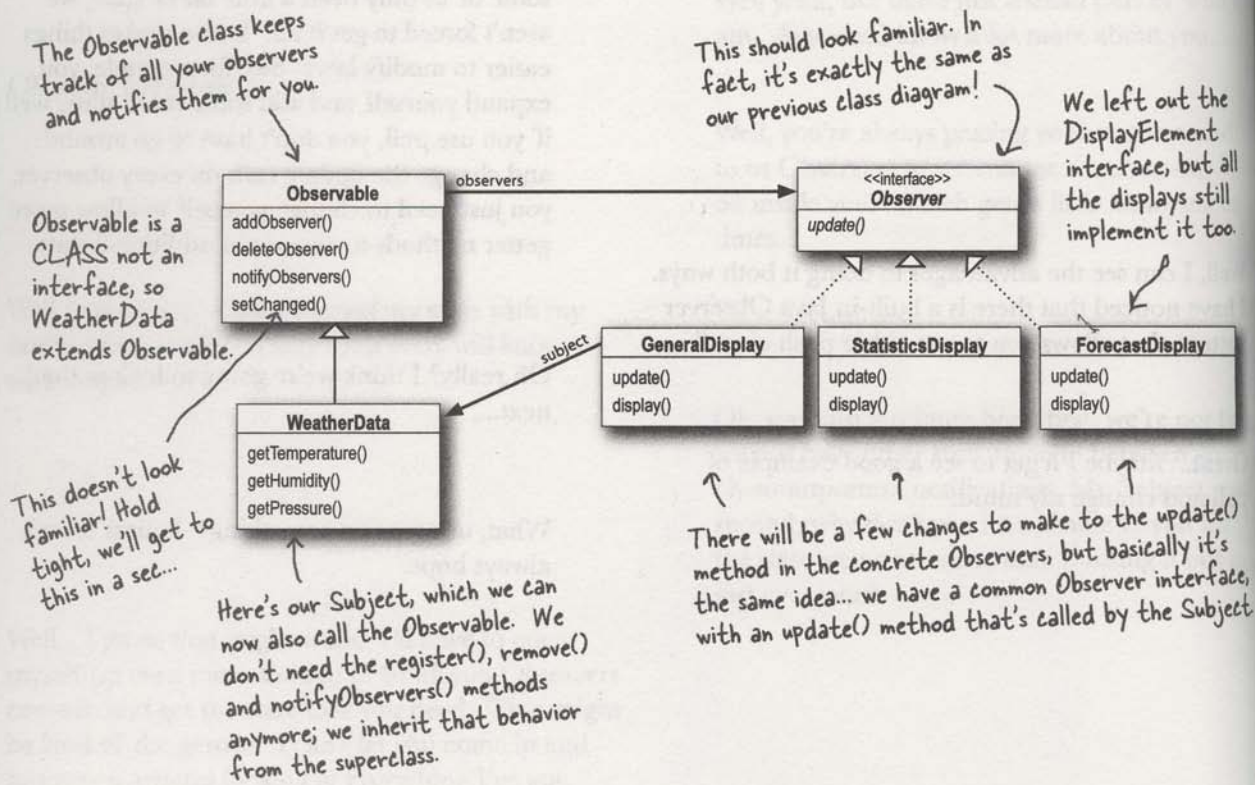
What, us agree on something? I guess there's always hope.

Using Java's built-in Observer Pattern

So far we've rolled our own code for the Observer Pattern, but Java has built-in support in several of its APIs. The most general is the Observer interface and the Observable class in the java.util package. These are quite similar to our Subject and Observer interface, but give you a lot of functionality out of the box. You can also implement either a push or pull style of update to your observers, as you will see.

To get a high level feel for java.util.Observer and java.util.Observable, check out this reworked OO design for the WeatherStation:





How Java's built-in Observer Pattern works

The built in Observer Pattern works a bit differently than the implementation that we used on the Weather Station. The most obvious difference is that WeatherData (our subject) now extends the Observable class and inherits the add, delete and notify Observer methods (among a few others). Here's how we use Java's version:

For an Object to become an observer...

As usual, implement the Observer interface (this time the java.util.Observer interface) and call addObserver() on any Observable object. Likewise, to remove yourself as an observer just call deleteObserver().

For the Observable to send notifications...

First of all you need to be Observable by extending the java.util.Observable superclass. From there it is a two step process:

1. **You first must call the setChanged() method to signify that the state has changed in your object**

2. **Then, call one of two notifyObservers() methods:**

either `notifyObservers()` **or** `notifyObservers(Object arg)`

This version takes an arbitrary data object that gets passed to each Observer when it is notified.

For an Observer to receive notifications...

It implements the update method, as before, but the signature of the method is a bit different:

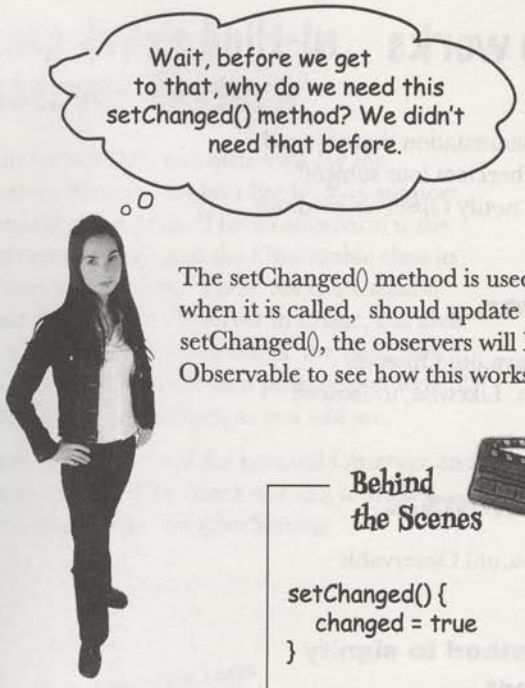
data object

```
update(Observable o, Object arg)
```

The Subject that sent the notification is passed in as this argument.

This will be the data object that was passed to notifyObservers(), or null if a data object wasn't specified.

If you want to "push" data to the observers you can pass the data as a data object to the notifyObserver(arg) method. If not, then the Observer has to "pull" the data it wants from the Observable object passed to it. How? Let's rework the Weather Station and you'll see.



The setChanged() method is used to signify that the state has changed and that notifyObservers(), when it is called, should update its observers. If notifyObservers() is called without first calling setChanged(), the observers will NOT be notified. Let's take a look behind the scenes of Observable to see how this works:

Behind the Scenes

```
setChanged() {
   changed = true
}

notifyObservers(Object arg) {
   if (changed) {
      for every observer on the list {
         call update (this, arg)
      }
      changed = false
   }
}

notifyObservers() {
   notifyObservers(null)
}
```

Pseudocode for the Observable Class.

The setChanged() method sets a changed flag to true.

notifyObservers() only notifies its observers if the changed flag is TRUE.

And after it notifies the observers, it sets the changed flag back to false.

Why is this necessary? The setChanged() method is meant to give you more flexibility in how you update observers by allowing you to optimize the notifications. For example, in our weather station, imagine if our measurements were so sensitive that the temperature readings were constantly fluctuating by a few tenths of a degree. That might cause the WeatherData object to send out notifications constantly. Instead, we might want to send out notifications only if the temperature changes more than half a degree and we could call setChanged() only after that happened.

You might not use this functionality very often, but it's there if you need it. In either case, you need to call setChanged() for notifications to work. If this functionality is something that is useful to you, you may also want to use the clearChanged() method, which sets the changed state back to false, and the hasChanged() method, which tells you the current state of the changed flag.

Reworking the Weather Station with the built-in support

First, let's rework WeatherData to use java.util.Observable

❶ Make sure we are importing the right Observer/Observable.

❷ We are now subclassing Observable.

❸ We don't need to keep track of our observers anymore, or manage their registration and removal, (the superclass will handle that) so we've removed the code for register, add and notify.

```
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }

    public void measurementsChanged() {
        setChanged();
        notifyObservers(); *
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

❹ Our constructor no longer needs to create a data structure to hold Observers.

* Notice we aren't sending a data object with the notifyObservers() call. That means we're using the PULL model.

❺ We now first call setChanged() to indicate the state has changed before calling notifyObservers().

❻ These methods aren't new, but because we are going to use "pull" we thought we'd remind you they are here. The Observers will use them to get at the WeatherData object's state.

Now, let's rework the CurrentConditionsDisplay

1 Again, make sure we are importing the right Observer/Observable.

2 We now are implementing the Observer interface from java.util

```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

3 Our constructor now takes a Observable and we use this to add the current conditions object as an Observer.

4 We've changed the update() method to take both an Observable and the optional data argument

5 In update(), we first make sure the observable is of type WeatherData and then we use its getter methods to obtain the temperature and humidity measurements. After that we call display().

Code Magnets

The ForecastDisplay class is all scrambled up on the fridge. Can you reconstruct the code snippets to make it work? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

```
public ForecastDisplay(Observable
observable) {
```

```
display();
```

```
observable.addObserver(this);
```

```
if (observable instanceof WeatherData) {
```

```
public class ForecastDisplay implements
Observer, DisplayElement {
```

```
public void display() {
    // display code here
}
```

```
lastPressure = currentPressure;
currentPressure = weatherData.getPressure();
```

```
private float currentPressure = 29.92f;
private float lastPressure;
```

```
WeatherData weatherData =
(WeatherData)observable;
```

```
public void update(Observable observable,
    Object arg) {
```

```
import java.util.Observable;
import java.util.Observer;
```

Running the new code

Just to be sure, let's run the new code...

```
File Edit Window Help TryTihisAtHome
%java WeatherStation
Forecast: Improving weather on the way!
Avg/Max/Min temperature = 80.0/80.0/80.0
Current conditions: 80.0F degrees and 65.0% humidity
Forecast: Watch out for cooler, rainy weather
Avg/Max/Min temperature = 81.0/82.0/80.0
Current conditions: 82.0F degrees and 70.0% humidity
Forecast: More of the same
Avg/Max/Min temperature = 80.0/82.0/78.0
Current conditions: 78.0F degrees and 90.0% humidity
%
```

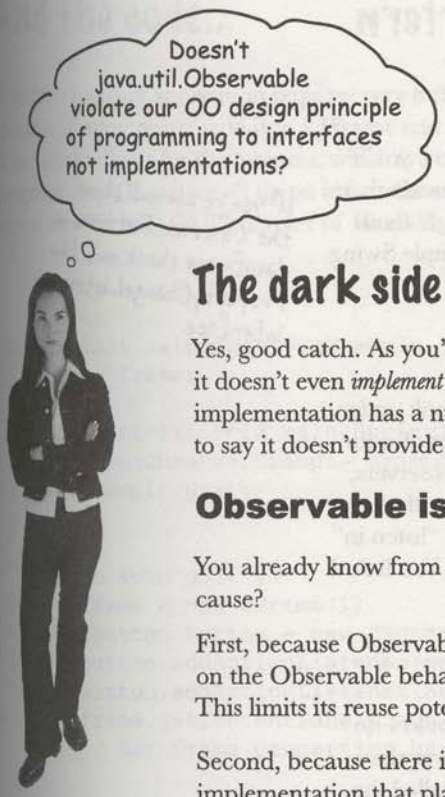
Hmm, do you notice anything different? Look again...

You'll see all the same calculations, but mysteriously, the order of the text output is different. Why might this happen? Think for a minute before reading on...

Never depend on order of evaluation of the Observer notifications

The java.util.Observable has implemented its notifyObservers() method such that the Observers are notified in a *different* order than our own implementation. Who's right? Neither; we just chose to implement things in different ways.

What would be incorrect, however, is if we wrote our code to *depend* on a specific notification order. Why? Because if you need to change Observable/Observer implementations, the order of notification could change and your application would produce incorrect results. Now that's definitely *not* what we'd consider loosely coupled.



The dark side of java.util.Observable

Yes, good catch. As you've noticed, Observable is a *class*, not an *interface*, and worse, it doesn't even *implement* an interface. Unfortunately, the java.util.Observable implementation has a number of problems that limit its usefulness and reuse. That's not to say it doesn't provide some utility, but there are some large potholes to watch out for.

Observable is a class

You already know from our principles this is a bad idea, but what harm does it really cause?

First, because Observable is a *class*, you have to *subclass* it. That means you can't add on the Observable behavior to an existing class that already extends another superclass. This limits its reuse potential (and isn't that why we are using patterns in the first place?).

Second, because there isn't an Observable interface, you can't even create your own implementation that plays well with Java's built-in Observer API. Nor do you have the option of swapping out the java.util implementation for another (say, a new, multi-threaded implementation).

Observable protects crucial methods

If you look at the Observable API, the setChanged() method is protected. So what? Well, this means you can't call setChanged() unless you've subclassed Observable. This means you can't even create an instance of the Observable class and compose it with your own objects, you *have* to subclass. The design violates a second design principle here…*favor composition over inheritance.*

What to do?

Observable *may* serve your needs if you can extend java.util.Observable. On the other hand, you may need to roll your own implementation as we did at the beginning of the chapter. In either case, you know the Observer Pattern well and you're in a good position to work with any API that makes use of the pattern.

Other places you'll find the Observer Pattern in the JDK

The java.util implementation of Observer/Observable is not the only place you'll find the Observer Pattern in the JDK; both JavaBeans and Swing also provide their own implementations of the pattern. At this point you understand enough about observer to explore these APIs on your own; however, let's do a quick, simple Swing example just for the fun of it.

If you're curious about the Observer Pattern in JavaBeans check out the PropertyChangeListener interface.

A little background...

Let's take a look at a simple part of the Swing API, the JButton. If you look under the hood at JButton's superclass, AbstractButton, you'll see that it has a lot of add/remove listener methods. These methods allow you to add and remove observers, or as they are called in Swing, listeners, to listen for various types of events that occur on the Swing component. For instance, an ActionListener lets you "listen in" on any types of actions that might occur on a button, like a button press. You'll find various types of listeners all over the Swing API.

A little life-changing application

Okay, our application is pretty simple. You've got a button that says "Should I do it?" and when you click on that button the listeners (observers) get to answer the question in any way they want. We're implementing two such listeners, called the AngelListener and the DevilListener. Here's how the application behaves:

Here's our fancy interface.

And here's the output when we click on the button.

```
File Edit Window Help HeMadeMeDoIt
%java SwingObserverExample
Come on, do it!
Don't do it, you might regret it!
%
```

Devil answer

Angel answer

And the code...

This life-changing application requires very little code. All we need to do is create a JButton object, add it to a JFrame and set up our listeners. We're going to use inner classes for the listeners, which is a common technique in Swing programming. If you aren't up on inner classes or Swing you might want to review the "Getting GUI" chapter of Head First Java.

Simple Swing application that just creates a frame and throws a button in it.

```
public class SwingObserverExample {
   JFrame frame;

   public static void main(String[] args) {
       SwingObserverExample example = new SwingObserverExample();
       example.go();
   }

   public void go() {
       frame = new JFrame();
       JButton button = new JButton("Should I do it?");
       button.addActionListener(new AngelListener());
       button.addActionListener(new DevilListener());
       frame.getContentPane().add(BorderLayout.CENTER, button);
       // Set frame properties here
   }

   class AngelListener implements ActionListener {
       public void actionPerformed(ActionEvent event) {
           System.out.println("Don't do it, you might regret it!");
       }
   }

   class DevilListener implements ActionListener {
       public void actionPerformed(ActionEvent event) {
           System.out.println("Come on, do it!");
       }
   }
}
```

Makes the devil and angel objects listeners (observers) of the button.

Here are the class definitions for the observers, defined as inner classes (but they don't have to be).

Rather than update(), the actionPerformed() method gets called when the state in the subject (in this case the button) changes.

Tools for your Design Toolbox

Welcome to the end of Chapter 2. You've added a few new things to your OO toolbox...

OO Basics

Abstraction
tion
hism
ce

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Here's your newest principle. Remember, loosely coupled designs are much more flexible and resilient to change.

OO Patterns

Stra
encap
inter
vary

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

A new pattern for communicating state to a set of objects in a loosely coupled manner. We haven't seen the last of the Observer Pattern – just wait until we talk about MVC!

BULLET POINTS

- The Observer Pattern defines a one-to-many relationship between objects.
- Subjects, or as we also know them, Observables, update Observers using a common interface.
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer Interface.
- You can push or pull data from the Observable when using the pattern (pull is considered more "correct").
- Don't depend on a specific order of notification for your Observers.
- Java has several implementations of the Observer Pattern, including the general purpose java.util.Observable.
- Watch out for issues with the java.util.Observable implementation.
- Don't be afraid to create your own Observable implementation if needed.
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.
- You'll also find the pattern in many other places, including JavaBeans and RMI.

Exercise

Design Principle Challenge

For each design principle, describe how the Observer Pattern makes use of the principle.

Design Principle

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

Design Principle

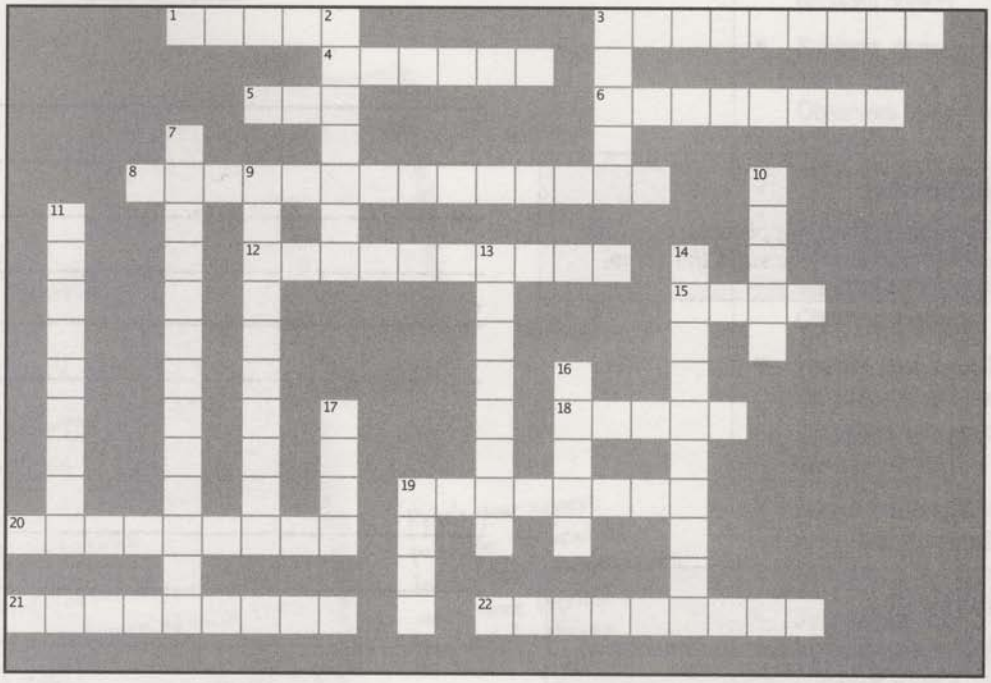
Program to an interface, not an implementation.

Design Principle

Favor composition over inheritance.

This is a hard one, hint: think about how observers and subjects work together.

Time to give your right brain something to do again!
This time all of the solution words are from chapter 2.



Across

1. Observable is a _____ not an interface
3. Devil and Angel are _____ to the button
4. Implement this method to get notified
5. Jill got one of her own
6. CurrentConditionsDisplay implements this interface
8. How to get yourself off the Observer list
12. You forgot this if you're not getting notified when you think you should be
15. One Subject likes to talk to _____ observers
18. Don't count on this for notification
19. Temperature, humidity and _____
20. Observers are _____ on the Subject
21. Program to an _____ not an implementation
22. A Subject is similar to a _____

Down

2. Ron was both an Observer and a _____
3. You want to keep your coupling _____
7. He says you should go for it
9. _____ can manage your observers for you
10. Java framework with lots of Observers
11. Weather-O-Rama's CEO named after this kind of storm
13. Observers like to be _____ when something new happens
14. The WeatherData class _____ the Subject interface
16. He didn't want any more ints, so he removed himself
17. CEO almost forgot the _____ index display
19. Subject initially wanted to ____ all the data to Observer

Exercise solutions

Sharpen your pencil

Based on our first implementation, which of the following apply? (Choose all that apply.)

- ☑ A. We are coding to concrete implementations, not interfaces.
- ☑ B. For every new display element we need to alter code.
- ☑ C. We have no way to add display elements at run time.
- ☐ D. The display elements don't implement a common interface.
- ☑ E. We haven't encapsulated what changes.
- ☐ F. We are violating encapsulation of the WeatherData class.

Design Principle Challenge

Design Principle

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

The thing that varies in the Observer Pattern is the state of the Subject and the number and types of Observers. With this pattern, you can vary the objects that are dependent on the state of the Subject, without having to change that Subject. That's called planning ahead!

Design Principle

Program to an interface, not an implementation.

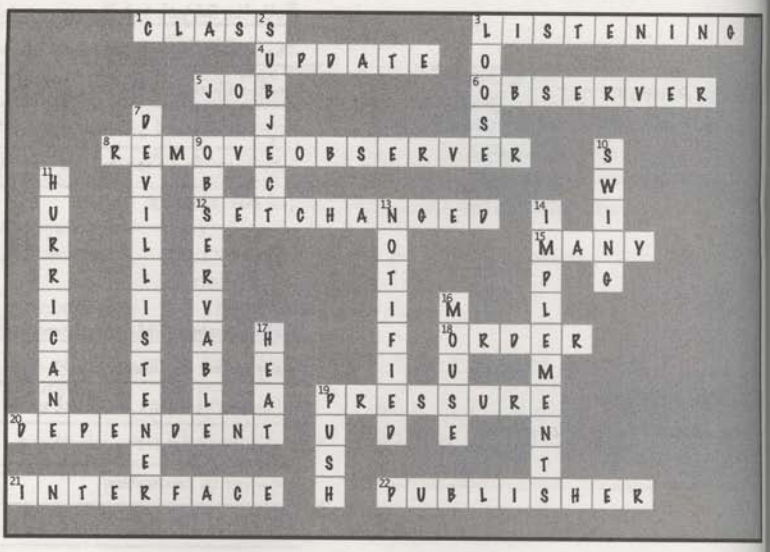
Both the Subject and Observer use interfaces. The Subject keeps track of objects implementing the Observer interface, while the observers register with, and get notified by, the Subject interface. As we've seen, this keeps things nice and loosely coupled.

Design Principle

Favor composition over inheritance.

The Observer Pattern uses composition to compose any number of Observers with their Subjects. These relationships aren't set up by some kind of inheritance hierarchy. No, they are set up at runtime by composition!

Code Magnets

Exercise solutions

```
import java.util.Observable;
import java.util.Observer;

public class ForecastDisplay implements
Observer, DisplayElement {

  Private float currentPressure = 29.92f;
  Private float lastPressure;

  public ForecastDisplay(Observable
  observable) {
      WeatherData weatherData =
      (WeatherData)observable;
    observable.addObserver(this);
  }
  public void update(Observable observable,
      Object arg) {
    if (observable instanceof WeatherData) {
      lastPressure = currentPressure;
      currentPressure = weatherData.getPressure();
      display();
    }
  }
  public void display() {
      // display code here
  }
}
```