Design Patterns

The Observer Design Pattern

The Observer Design Pattern

“I always feel like - Somebody’s watching me…”

The Observer pattern allows one object (or more) - the observer - to watch another - the subject. The Observer pattern allows the subject and observer to form a publish-subscribe relationship. Through the Observer pattern, observers can register to receive events from the subject. When the subject needs to inform its observers of an event, it simply sends the event to each registered observer.

The benefit: it decouples the observer from the subject. The subject doesn't need to know anything special about its observers. Instead, the subject simply allows observers to subscribe. When the subject generates an event, it simply passes it to each of its observers.

The Observer pattern is a behavioral design pattern. The observer pattern is also known as the *broadcaster-listener* or *publisher-subscriber* pattern. It defines a way for classes to be loosely coupled and for one class (or many) to be notified when another is updated. Basically, this means that when something happens in one place, you notify anyone who is observing and that is interested in that one place.

The Observer design pattern lets several observer objects be notified when a subject object is changed in some way. Observers register with the subject, and when a change occurs, the subject notifies them all. Each of the observers is notified in parallel (that is, at the same time).

The Observer Design Pattern

More Details…

The Observer Design Pattern allows one class - the *Subject* (also known as the *Observable*) - to notify interested classes - called *Observers* - of events that occur in it.

The *Subject* does not care about what the *Observers* do with this notification, or even if there are any interested Observers. The capability is simply provided in the design of the *Subject* class for this notification to take place if certain conditions in the *Subject* class occur. This object-oriented design scheme ensures that an observer object or a set of observer objects automatically perform appropriate actions when required to do so by an observable (subject) object.

Observers register their interest with one or more observable objects and are notified when an event that satisfies this interest is recognized by an observable. The observer pattern also promotes a fundamental object-oriented design heuristic by facilitating loose coupling between communicating objects.

Complications

Recall that the Subject maintains reference(s) to the Observer(s). Until the Subject releases the reference, the Observer cannot be removed by the garbage collector. Be aware of this possibility and remove observers when appropriate.

Also note that the set of Observer objects can be maintained in an unordered collection. You don't necessarily know if the first registered listener is notified first or last.

The Observer Design Pattern

Basic Observer Example: "Weather" Class

Assume we are working on an application that contains many entities that need to know when certain aspects of the weather changes. (Vehicles need to modify their movement to account for snow, buildings might adjust their climate controls if the temperature increases, traffic signs might need to change their message to alert drivers of bad weather, etc.)

Rather than have each of these entities (which could number in the thousands, or perhaps millions) constantly asking the Weather object for the current weather conditions (which may not have changed since the last time they asked), we'll put the Observer pattern in place. In this way the entities interested in Weather conditions will register themselves with the "subject" – the Weather object. Weather object will notify the "observers" (the Vehicles, Buildings, Traffic Signs, etc.) when the weather changes. This Observer implementation follows the *push* paradigm – where the updated data is *pushed* out to the observers.

Creating an Observer Interface

First, we will build an *observer* interface that is implemented by the weather observers (to enable them to get notifications). Minimally, we need a method that will be called by the Subject when a new weather notification is ready (we will call that method *updateWeather*). In this example, we pass the precipitation type ("Snow", "Rain", "Sleet", "None", etc.) and the current temperature to the *updateWeather* method.

```
public interface WeatherObserver
{
      public void updateWeather(String precip, double temp);
}
```

The Observer Design Pattern

Creating a Subject Interface

Since we do not want to tightly couple the weather *observers* (the Vehicles, Buildings, etc.) to one specific Weather *subject* class, we will set up an interface for the subject called *WeatherSubject*. The WeatherSubject interface lists the methods any potential subject must implement.

We first need a *registerWeatherObserver* method here so that the subject can keep track of observers that want to be registered. Besides registering observers, you should have some way to un-register them, so add a *removeWeatherObserver* method.

```
public interface WeatherSubject
{
    public void registerWeatherObserver(WeatherObserver wo) throws SomeException;
    public void removeWeatherObserver(WeatherObserver wo) throws SomeException;
}
```

When observers implement the *updateWeather* method, the subject is able to pass them the updated weather information.

The Observer Design Pattern

Creating a Subject

The Subject has to allow observers to register, and has to notify them all when an event occurs.

According to the *WeatherSubject* interface, the methods a subject has to implement in these examples are: *registerWeatherObserver* and *removeWeatherObserver*.

That's what the *Weather* class will do in this example.

To keep track of the *WeatherSubjects*, our subject class – Weather – needs a list of *WeatherObserver* objects.

```
import java.util.*;

public class Weather implements WeatherSubject
{
      private String precipitation;
      private double temperature;
      private ArrayList<WeatherObserver> weatherObservers =
                                                new ArrayList<WeatherObserver>();

      public Weather ()
      {
         …
      }
      .
      .
      .
}
```

The Observer Design Pattern

Creating a Subject (cont.)

When a weather observer wants to register, it calls the weather subject's *registerWeatherObserver* method, passing itself as the parameter. The subject — an object of our Weather class — just has to add that weather observer to the *weatherObservers* list.

```
import java.util.*;

public class Weather implements WeatherSubject
{
      private String precipitation;
      private double temperature;
      private ArrayList<WeatherObserver> weatherObservers =
                                                new ArrayList<WeatherObserver>();

      public Weather ()
      {
         …
      }

      public void registerWeatherObserver(WeatherObserver wo) throws SomeException
      {
            if (wo == null) throw new SomeException(…);

            weatherObservers.add(wo);
      }
      .
      .
      .
}
```

The Observer Design Pattern

Creating a Subject (cont.)

To remove a weather observer from the Weather subject's *weatherObservers* list, the weather observer in question simply calls the weather subject's *removeWeatherObserver* method, passing itself as the parameter. The subject then has to remove that weather observer from the *weatherObservers* list.

```
import java.util.*;

public class Weather implements WeatherSubject
{
      private String precipitation;
      private double temperature;
      private ArrayList<WeatherObserver> weatherObservers =
                                          new ArrayList<WeatherObserver>();
      public Weather ()
      {
         …
      }

      public void registerWeatherObserver(WeatherObserver wo) throws SomeException
      {
            if (wo == null) throw new SomeException(…);

            weatherObservers.add(wo);
      }

      public void removeWeatherObserver(WeatherObserver wo) throws SomeException
      {
            if (wo == null) throw new SomeException(…);

            weatherObservers.remove(wo);
      }
      .
      .
      .
}
```

The Observer Design Pattern

Creating a Subject (cont.)

When something occurs that updates the precipitation or the temperature within the Weather object (the subject), the Weather object can then inform all currently registered weather observers of that change by calling it's own *notifyWeatherObservers* method.

Each weather observer implements the WeatherObserver interface (which means it has an *updateWeather* method) so *notifyWeatherObservers* just has to loop over all registered weather observers in the *weatherObservers* list, calling each one's *updateWeather* method with the updated weather information.

```
import java.util.*;

public class Weather implements WeatherSubject
{
      private String precipitation;
      private double temperature;
      private ArrayList<WeatherObserver> weatherObservers =
                                          new ArrayList<WeatherObserver>();
      .
      .
      .

      public void notifyWeatherObservers()
      {
         for (WeatherObserver wo : weatherObservers)
         {
               wo.updateWeather(precipitation, temperature);
         }
      }


}
```

The Observer Design Pattern

Creating an Observer

Any number of classes can implement the Observer interface (Vehicles, Buildings, Traffic Signs, etc.). The following is an example of how an Observer might be implemented.

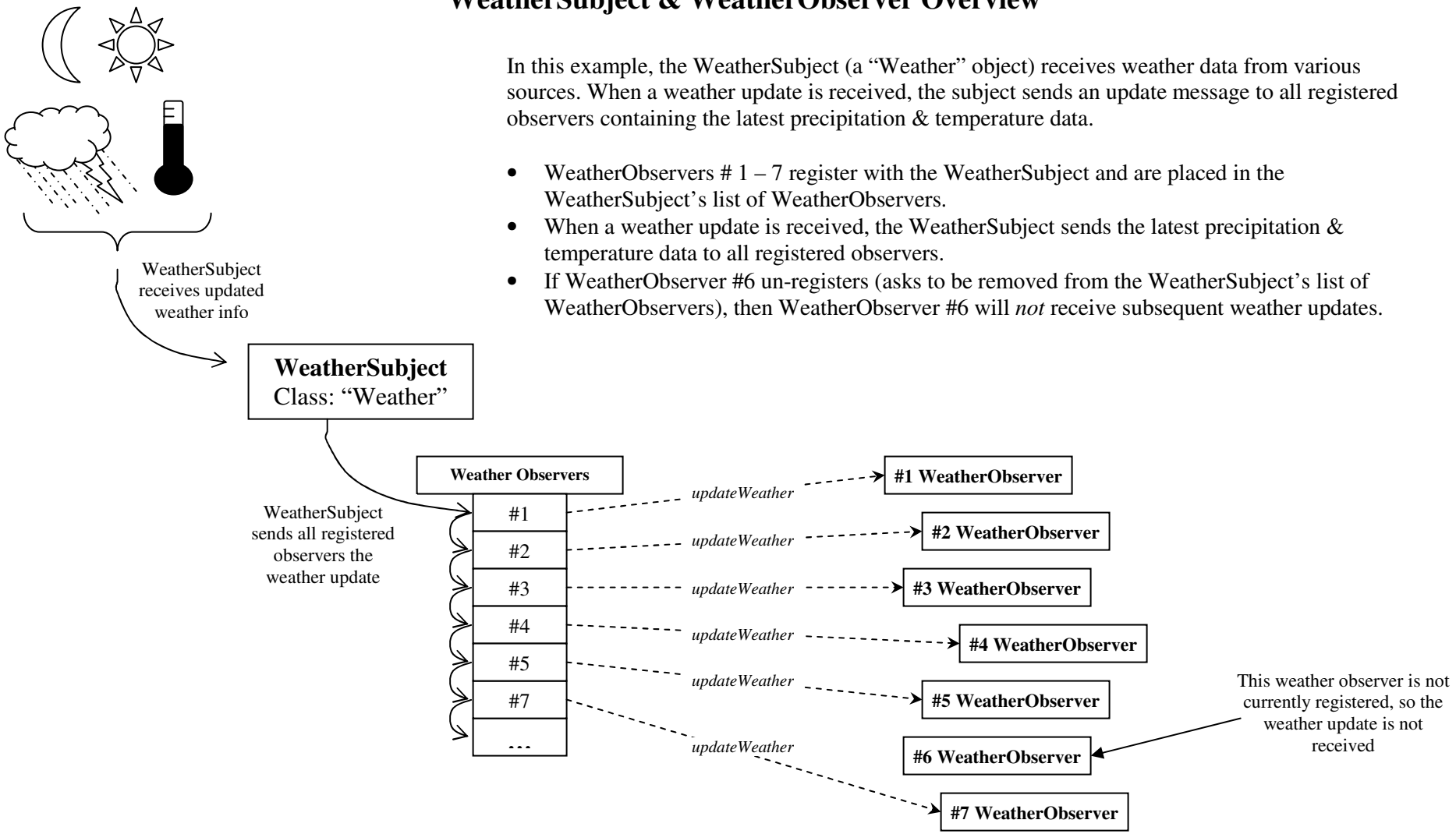
```
import java.util.*;

public class AutomatedTrafficSign implements WeatherObserver
{
	private WeatherSubject subject; // A reference to the subject
	private String currentPrecip; // Latest precipitation (from last update)
	private double currentTemp;   // Latest temperature (from last update)

	public AutomatedTrafficSign(WeatherSubject ws) throws SomeException
	{
	   if (ws == null) throw new SomeException(…); // Check for null
	   subject = ws;  // Setup the Subject reference
	   subject.registerWeatherObserver(this); // Register with Subject!
	}

	public void updateWeather(String precip, double temp) // Called by the Subject
	{
		currentPrecip = precip; // Update the latest precipitation
		currentTemp = temp;     // Update the latest temperature
		reactToNewWeather(); // Now, perform "domain" behavior based upon the weather
	}

	private void reactToNewWeather()
	{
		.  // Update a sign message…
		.  // Flash a yellow light…
		.  // Etc…
	}

	public void shutdown() throws SomeException
	{
		subject.removeWeatherObserver(this); // Shutting down - de-register from subject
	}
}
```

The Observer Design Pattern

WeatherSubject & WeatherObserver Overview

In this example, the WeatherSubject (a "Weather" object) receives weather data from various sources. When a weather update is received, the subject sends an update message to all registered observers containing the latest precipitation & temperature data.

- WeatherObservers # 1 – 7 register with the WeatherSubject and are placed in the WeatherSubject's list of WeatherObservers.
- When a weather update is received, the WeatherSubject sends the latest precipitation & temperature data to all registered observers.
- If WeatherObserver #6 un-registers (asks to be removed from the WeatherSubject's list of WeatherObservers), then WeatherObserver #6 will *not* receive subsequent weather updates.



The Observer Design Pattern

Alternate Implementation

A variant of the Observer pattern exists that allows the Observers to *decide* if the want the latest update from the Subject they are registered with. This variant is useful when observers will make use of the updates under some conditions, but will *not* make use of the updates under others.

Rather then have the observers register and de-register and register and de-register based upon whether they want the latest updates, the subject informs the observers that there is an update *available*. Observers can then call back to the subject to obtain the update, if they so choose. This Observer implementation follows the *pull* paradigm – where the Observers *pull* the updated data from the Subject.

This is useful in distributed systems where registering and de-registering involves establishing & terminating external connections (an expensive process) or where the updated data to be sent to the observers is very large and sending that data to observers who may not even want it would incur a lot of overhead.

This variant requires the addition of a *getWeatherUpdate* method to the WeatherSubject interface. This is the method observers will call to obtain the update, if they so choose.

```
public interface WeatherSubject
{
    public void registerWeatherObserver(WeatherObserver wo) throws SomeException;
    public void removeWeatherObserver(WeatherObserver wo) throws SomeException;
    public WeatherDTO getWeatherUpdate();
}
```

The Observer Design Pattern

Alternate Implementation (cont.)

The WeatherObserver interface would need a change as well. In this variant, no data will be sent when the observers are notified that an update is available, since in this case they specifically request the update if they want it.

```
public interface WeatherObserver
{
      public void updateWeather(); // No parameters sent this time
}
```

Modified Subject

In this variant, the Subject's *notifyWeatherObservers* method is similar to the original, except that now the *updateWeather* method called on the observers will send no parameters.

```
import java.util.*;
public class Weather implements WeatherSubject
{
      private String precipitation;
      private double temperature;
      private ArrayList<WeatherObserver> weatherObservers =
                                    new ArrayList<WeatherObserver>();
      .
      . // The same "register" & "remove" code is here.
      .
      public void notifyWeatherObservers()
      {
         // Only indicates that a weather update is available
         for (WeatherObservers wo : weatherObservers)
         {
               wo.updateWeather(); // No parameters sent this time
         }
      }
      .
      .
      .
```

The Observer Design Pattern

Modified Subject (cont.)

Additionally, the Subject will need to implement the new *getWeatherUpdate* method.

"*getWeatherUpdate*" is the method that observers will call if they decide that they *do* want the weather update associated with a call to their *updateWeather* method. In this example, the *getWeatherUpdate* method returns a Weather *Data Transfer Object* (shown in the inset) that contains all the weather data elements associated with the latest update.

```
import java.util.*;
public class Weather implements WeatherSubject
{
      private String precipitation;
      private double temperature;
      private ArrayList<WeatherObserver> weatherObservers =
                                    new ArrayList<WeatherObserver>();
      .
      . // The same "register" & "remove" code is here.
      .
      public void notifyWeatherObservers()
      {
         // Only indicates that a weather update is available
         for (WeatherObservers wo : weatherObservers)
         {
               wo.updateWeather();// No parameters sent this time
         }
      }

      public WeatherDTO getWeatherUpdate()
      {
         // Return the DTO to those who specifically request the update
         return new WeatherDTO(precipitation, temperature);
      }
}
```

```
public class WeatherDTO
{
    public String precipitation;
    public double temperature;

    public WeatherDTO(String precip, double temp)
    {
          precipitation = precip;
          temperature = temp;
    }
}
```

The Observer Design Pattern

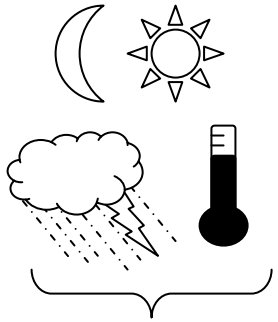
Modified Observer

In this variant, the observer will be informed that there is an update available. The observer will then determine if the update is needed, and will call back to the subject to obtain the update if needed.

```
import java.util.*;

public class AutomatedTrafficSign implements WeatherObserver
{
        .
        . // Same data attributes & constructor as the previous AutomatedTrafficSign example
        .

        public void updateWeather()  // No parameters sent this time
        {
                if (we determine that want this weather update)
                {
                        WeatherDTO wdto = ws.getWeatherUpdate(); // Get the update from the subject

                        currentPrecip = wdto.precipitation; // Update the latest precipitation
                        currentTemp = wdto.temperature;     // Update the latest temperature
                        reactToNewWeather(); // Now, perform "domain" behavior based upon the weather
                }
        }

        private void reactToNewWeather()
        {
                .  // Update a sign message
                .  // Flash a yellow light
                .  // Etc…
        }

        public void shutdown()
        {
                subject.removeWeatherObserver(this); // Shutting down - un-register from subject
        }
}
```

The Observer Design Pattern

Alternate Implementation: WeatherSubject & WeatherObserver Overview

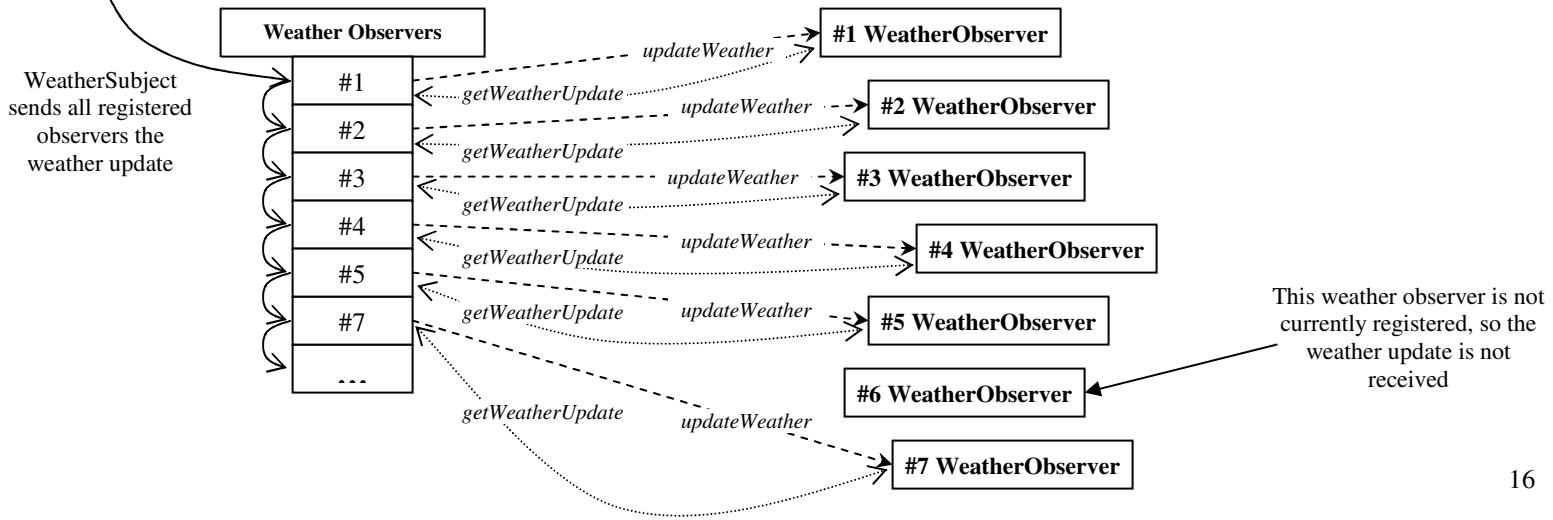


WeatherSubject receives updated weather info

WeatherSubject
Class: "Weather"

In this *alternate* example, the WeatherSubject (a "Weather" object) receives weather data from various sources. When a weather update is received, the subject sends an update message to all registered observers *that contains no data.* If the observer wants the update, they make a call to the WeatherSubject's *getWeatherUpdate* method to obtain it.

- WeatherObservers # 1 – 7 register with the WeatherSubject and are placed in the WeatherSubject's list of WeatherObservers.
- When a weather update is received, the WeatherSubject sends an update message (without data) to all registered observers.
- If WeatherObserver #6 un-registers (asks to be removed from the WeatherSubject's list of WeatherObservers), then WeatherObserver #6 will *not* receive subsequent weather update messages.
- If the observers determine that they *want* the latest weather update, they call the WeatherSubject's *getWeatherUpdate* method to get the latest weather data packaged in a Weather Data Transfer Object



The Observer Design Pattern

Another Observer Variant

Another variant of the Observer pattern allows the observers to specify (upon registration with the subject) the *amount* of change that must occur in the subject's data before they want to be updated.

For example, in our first Observer example (*the Push model*), *WeatherObservers* will receive updated weather data *whenever* the weather changes. Even if the temperature changes by a tenth of a degree, they will be notified. Some observers might want that level of granularity in their updates, while others might not.

This observer variant would allow the observers to specify a percent-change in the *temperature* that must occur before they desire notification. One observer might want an update only when the temperature changes at least 1% from the last time they were updated. Another might only want updates if the temperature has changed by 10% or more. Still another might still want *all* temperature updates.

This variant requires the subject to maintain a little more information on each observer. At a minimum, the subject must maintain the change percentage requested for each observer (the amount of change that must occur before notification) and the last temperature value sent to that observer.

Updated Subject Interface

```
public interface WeatherSubject
{
   // Here, the register method allows the specification of a percent change value to
   // limit the updates they will receive,
   public void registerWeatherObserver(WeatherObserver wo, double percentChange) throws SomeException;

   // The original registration method still exists - assumes the observer wants all updates
   public void registerWeatherObserver(WeatherObserver wo) throws SomeException;

   public void removeWeatherObserver(WeatherObserver wo) throws SomeException; // No changes here
}
```

The Observer Design Pattern

Updated Subject

The Weather subject will now store the Observers, as well as the percent change desired by each observer and the last temperature value sent to the observer using a *WeatherObserverHolder* object – an inner class (shown in the inset).

```
public class Weather implements WeatherSubject
{
   …
   private ArrayList<WeatherObserverHolder> weatherObservers = new ArrayList<WeatherObserverHolder>();

   // This is the new registration method that accepts the percent change value desired by the observer
   public void registerWeatherObserver(WeatherObserver wo, double percentChange) throws SomeException
   {
      if (wo == null) throw new SomeException("...");

      WeatherObserverHolder woh = new WeatherObserverHolder(wo, percentChange);
      woh.observer.updateWeather(precipitation, temperature);
      woh.lastTempUpdate = temperature;
      weatherObservers.add(woh);
   }

   // This is the original registration method - assumes observer wants all updates
   public void registerWeatherObserver(WeatherObserver wo)
                                          throws SomeException
   {
      if (wo == null) throw new SomeException("...");

      WeatherObserverHolder woh =
                        new WeatherObserverHolder(wo, 0.0);
      woh.observer.updateWeather(precipitation, temperature);
      woh.lastTempUpdate = temperature;
      weatherObservers.add(woh);
   }
```

```
class WeatherObserverHolder
{
   private WeatherObserver observer;
   private double percentChange;
   private double lastTempUpdate;

   public WeatherObserverHolder(WeatherObserver woIn,
                                                  double pcIn)
   {
          observer = woIn;
          percentChange = pcIn;
   }
}
```

The Observer Design Pattern

Updated Subject

When the Subject's weather data changes, the subject must check to see if the chance exceeds the percent-change that the observer has specified.

If it does, the value is sent out to the observer and that value is recorded for subsequent percent change calculations.

If it does not exceed the percent-change that the observer has specified, no update is sent out.

```
import java.util.*;
public class Weather implements WeatherSubject
{
    .
    .
    .

   public void notifyWeatherObservers()
   {
      for (WeatherObserverHolder woh : weatherObservers)
      {
            // Determine the percent change since the last update sent to the observer
            double change = Math.abs(temperature - woh.lastTempUpdate);
            double percentChg = change /woh.lastTempUpdate;

            // If the value has changed more than the observer's specified percent change
            // value, then send the update and record the value sent out.
            if (percentChg > woh.percentChange)
            {
                  woh.observer.updateWeather(precipitation, temperature);
                  woh.lastTempUpdate = temperature;
            }
      }
   }
}
```

The Observer Design Pattern

Updated Observer

No changes need be made to the *WeatherObserver* interface, as the only change needed on the observer's side is the addition of the percent-change value to the constructor and to the registration. The updated *AutomatedTrafficSign* observer is shown below. Note, the observer can still register with the subject without specifying any percent-change value. This will result in the observer receiving *all* weather updates, matching the original functionality.

```
public class AutomatedTrafficSign implements WeatherObserver
{
	private WeatherSubject subject; // A reference to the subject
	private String currentPrecip; // Latest precipitation (from last update)
	private double currentTemp;   // Latest temperature (from last update)

	public AutomatedTrafficSign(WeatherSubject ws, double percentChange) throws SomeException
	{
	    if (ws == null) throw new SomeException(…); // Check for null
	    subject = ws;  // Setup the Subject reference
	    subject.registerWeatherObserver(this, percentChange); // Register with Subject!
	}

	public void updateWeather(String precip, double temp)
	{
		currentPrecip = precip; // Update the latest precipitation
		currentTemp = temp;     // Update the latest temperature
		reactToNewWeather(); // Now, perform "domain" behavior based upon the weather
	}

	private void reactToNewWeather()
	{
		.  // Update a sign message…
		.  // Flash a yellow light…
		.  // Etc…
	}

	public void shutdown() throws SomeException
	{
		subject.removeWeatherObserver(this); // Shutting down - un-register from subject
	}
}
```

The Observer Design Pattern

Java Observable/Observer Implementation

The Java programming language directly supports the observer pattern by providing a concrete *Observable* class (for the "subject") and an *Observer* interface (for the "observers"), both in package *java.util.* This implementation promotes a loosely coupled relationship where observers are referenced via an interface and observables (subjects) must extend a base class. Thus observables are coupled to an interface rather than directly to observer instances.

Subjects must *extend* the Observable class to inherit the *observable* data & behaviors. In this way, domain-specific code will reside in the *Subject*, while the observable-specific code is resides in the *Observable* class – distinct & separate.

Java Observable Class

```
public class Observable
{
   private boolean changed = false; // Set true when observers need to be notified of changed data
   private Vector obs; // The observers are maintained in a Vector

   public Observable() // Constructor – initializes the "obs" Vector
   public synchronized void addObserver(Observer o) // Adds the specified observer to "obs" – ignores duplicates
   public synchronized void deleteObserver(Observer o) // Removes the specified observer from "obs"
   public void notifyObservers() // Notification to observers – no data sent (the Pull model)
   public void notifyObservers(Object arg) // Notification to observers – data is sent (the Push model)
   public synchronized void deleteObservers() // Removes all observers from the "obs" Vector
   protected synchronized void setChanged() // Sets the "changed" Boolean data attribute to true
   protected synchronized void clearChanged() // Sets the "changed" Boolean data attribute to false
   public synchronized boolean hasChanged() // Returns the value of the "changed" Boolean data attribute
   public synchronized int countObservers() // Returns the number of observers in the "obs" Vector

}
```

The Observer Design Pattern

The Java Observer Interface

Any observer wishing to interact with a Java *Observable*-based subject must implement the *Observer* interface. This interface contains only a single method – update – much like our original *WeatherObserver* example.

```
public interface Observer
{
     // The "Observable" reference is also sent with the update for "pull" model
     // implementations where the observer might want to call back to the Observable
     // to get the data update. The "Object" reference is the actual update - usually
     // some sort of DTO.
    void update(Observable o, Object arg);
}
```

Observable Subjects

Subjects extending the Java Observable class can make use of all or only parts of the provided functionality. For example, a Subject might use all of the registration, deregistration & notification functionality, but might define a custom technique to determine when an update might become available.

Note, that since the Observable class contains a Vector of Observables, more work would be needed to create the observer pattern variant in which the observers specify a percent-change that must occur before they desire notification. Much of the Observable behavior would need to be over-ridden in the extending subject class.

The Observer Design Pattern

Creating a Subject using the Java Observable class

In this *Weather* subject variant, the observer management-related data & behavior are inherited from the parent *Observable* class, so the subject implementation is smaller. The WeatherDTO - a *Data Transfer Object* (shown in the inset) contains the weather data elements associated with the latest update

```
import java.util.Observable;

public class Weather extends Observable
{
    private String precipitation;
    private double temperature;

    public Weather()
    {
         …
    }

    public void notifyWeatherObservers()
    {
        // Calling the parent method "setChanged()" sets the "changed" indicator
        // to true, which indicates to the parent class that updates should be
        // sent (won't send if "false")
        setChanged();

        notifyObservers(new WeatherDTO(precipitation, temperature)); // Parent "Observable" method

        // Calling the parent method "clearChanged()" sets the "changed" indicator
        // back to false.
        clearChanged();
    }
}
```

```
public class WeatherDTO
{
    public String precipitation;
    public double temperature;

    public WeatherDTO(String precip, double
temp)
    {
        precipitation = precip;
        temperature = temp;
    }
}
```

The Observer Design Pattern

Creating an Observer using the Java Observer interface

The *AutomatedTrafficSign* class requires a few changes to work with the modified *Weather* subject:

1) The Observer interface is implemented rather than the custom-created *WeatherObserver* interface used in the previous examples.
2) References to the subject are of the generic type *Observable*, rather than the *WeatherSubject* interface used earlier.
3) The Observable class methods *addObserver* and *deleteObserver* are used now, rather than the *registerWeatherObserver* & *removeWeatherObserver* methods used earlier.
4) Finally, the *updateWeather* method has been changed to *update,* to conform to the Observable interface requirements.

```
import java.util.Observable;
import java.util.Observer;
public class AutomatedTrafficSign implements Observer
{
    private Observable subject;
    private String currentPrecip; // Latest precipitation (from last update)
    private double currentTemp;   // Latest temperature (from last update)

    public AutomatedTrafficSign(Observer ws) throws SomeException {
        if (ws == null) throw new SomeException("..."); // Check for null
        subject = ws;
        subject.addObserver(this); // Register with Subject!
    }

    public void update(Observable ws, Object data) {
        WeatherDTO w = (WeatherDTO) data;
        currentPrecip = w.precipitation; // Update the latest precipitation
        currentTemp = w.temperature;     // Update the latest temperature
        reactToNewWeather(); // Now, perform "domain" behavior based upon the weather
    }

    private void reactToNewWeather() {
        System.out.println("Update a sign message…");
        System.out.println("Flash a yellow light…");
        System.out.println("Etc…");
    }

    public void shutdown() throws SomeException {
        subject.deleteObserver(this); // Shutting down - un-register from subject
    }
}
```

The Observer Design Pattern

Summary

- The Observer pattern allows one object (or more) - the observer - to watch another - the subject.
- The Observer pattern decouples the observer from the subject. The subject doesn't need to know anything special about its observers. Instead, the subject simply allows observers to subscribe. When the subject generates an event, it simply passes it to each of its observers.
- The Observer pattern can follow the *push* paradigm – where the updated data is *pushed* out to the observers, or the *pull* paradigm, where the Observers pull the updated data from the Subject.
- Another variant of the Observer pattern allows the observers to specify (upon registration with the subject) the amount of change that must occur in the subject's data before they want to be updated.
- The Java programming language directly supports the observer pattern by providing a concrete Observable class (for the "subject") and an Observer interface (for the "observers"), both in package java.util. This implementation promotes a loosely coupled relationship where observers are referenced via an interface and observables (subjects) must extend a base class. Thus observables are coupled to an interface rather than directly to observer instances.
- Subjects must extend the Observable class to inherit the observable data & behaviors. In this way, domain-specific code will reside in the Subject, while the observable-specific code is resides in the Observable class – distinct & separate.