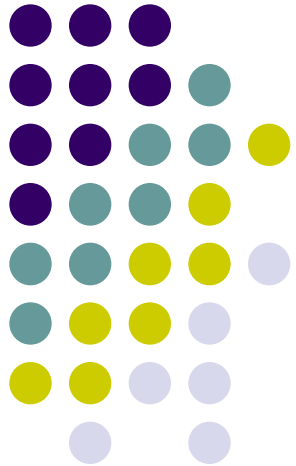
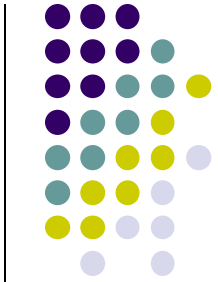
The Observer Pattern

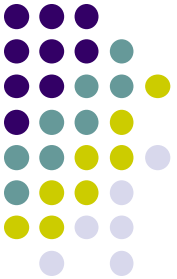


Project



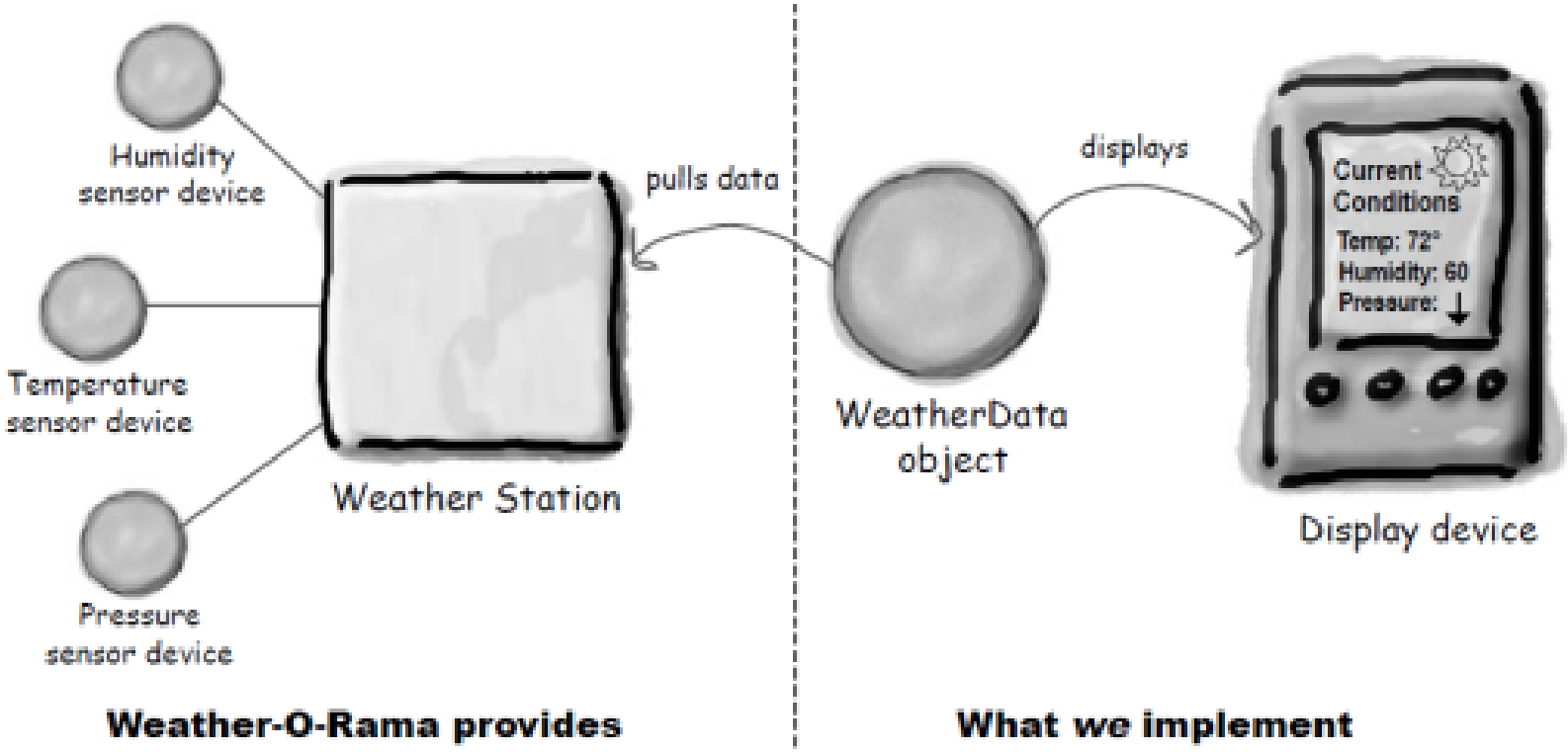
- Congratulations on being selected to build our next generation Internet-based Weather Monitoring Station! The weather station will be based on our patent pending WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We'd like or you to create an application that initially provides three display elements: current conditions, weather statistics and a simple forecast, all updated in real time as the WeatherDataobject acquires the most recent measurements.

Project

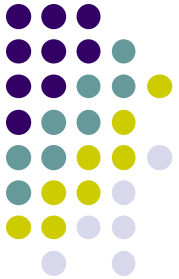


- Further, this is an expandable weather station. We want to release an API so that other developers can write their own weather displays and plug them right in. We'd like for you to supply that API!

Overview

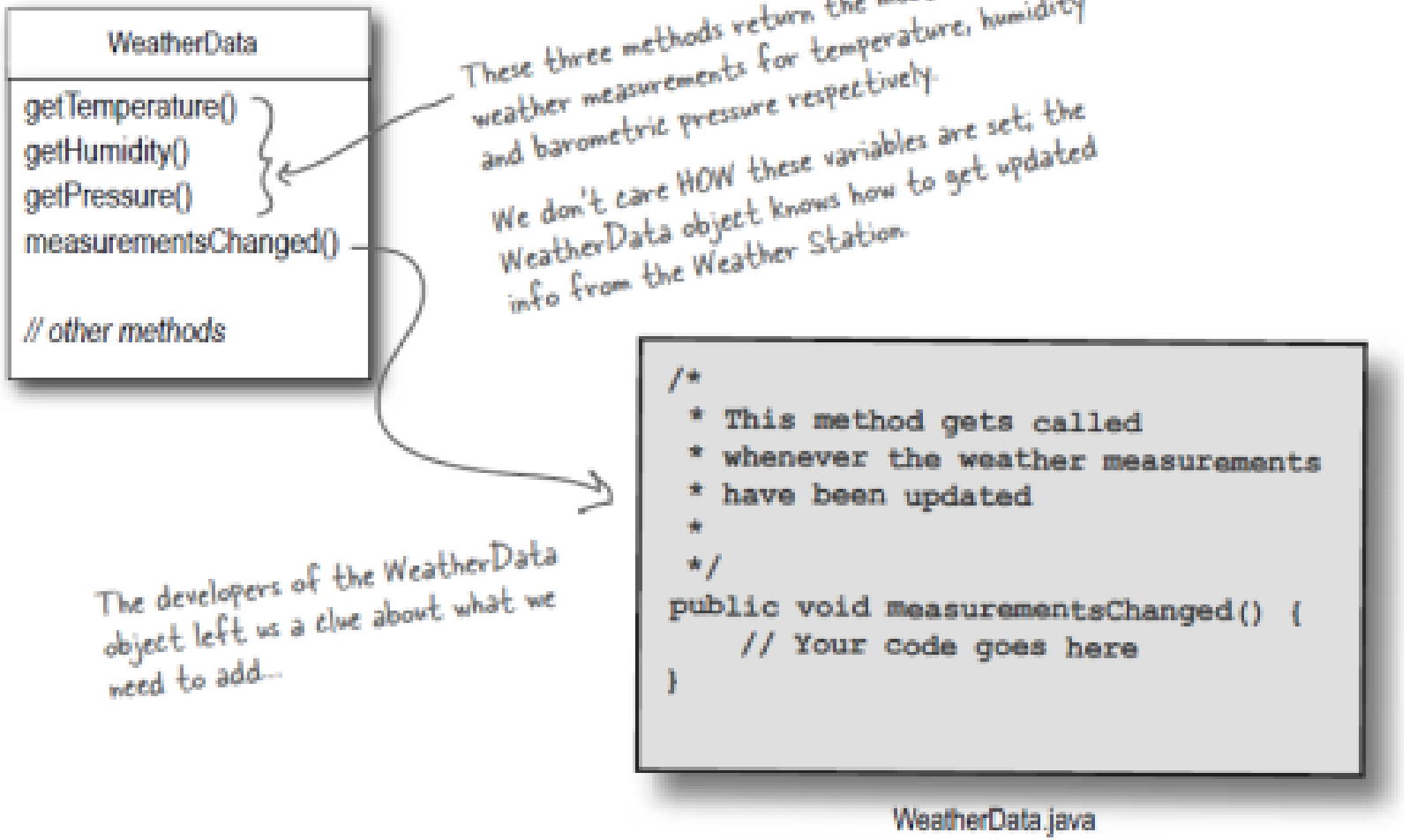

Humidity sensor device
Temperature sensor device
Pressure sensor device
Weather Station
pulls data
WeatherData object
displays
Current Conditions
Temp: 72°
Humidity: 60
Pressure:
Display device
Weather-O-Rama provides
What we implement

Our job

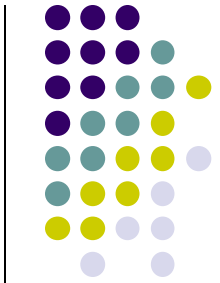


- Create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.

WeatherData class

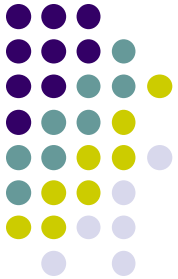


What we do



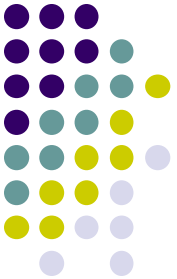
- Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.

What do we know so far?



- The WeatherData class has getter methods for three measurement values: temperature, humidity and barometric pressure.
- The measurementsChanged() method is called any time new weather measurement data is available.
- 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 elements and users can add or remove as many displays elements as they want to the application. Currently, we know about only the initial three display types (current conditions, statistics and forecast).

The first solution



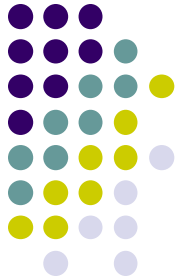
```
public class WeatherData {

    // 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.

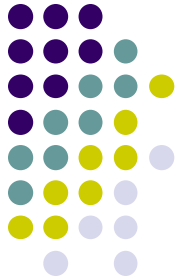
Think about the first solution



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.

What's wrong



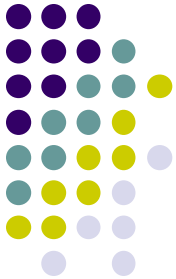
```
public void measurementsChanged() {

    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.

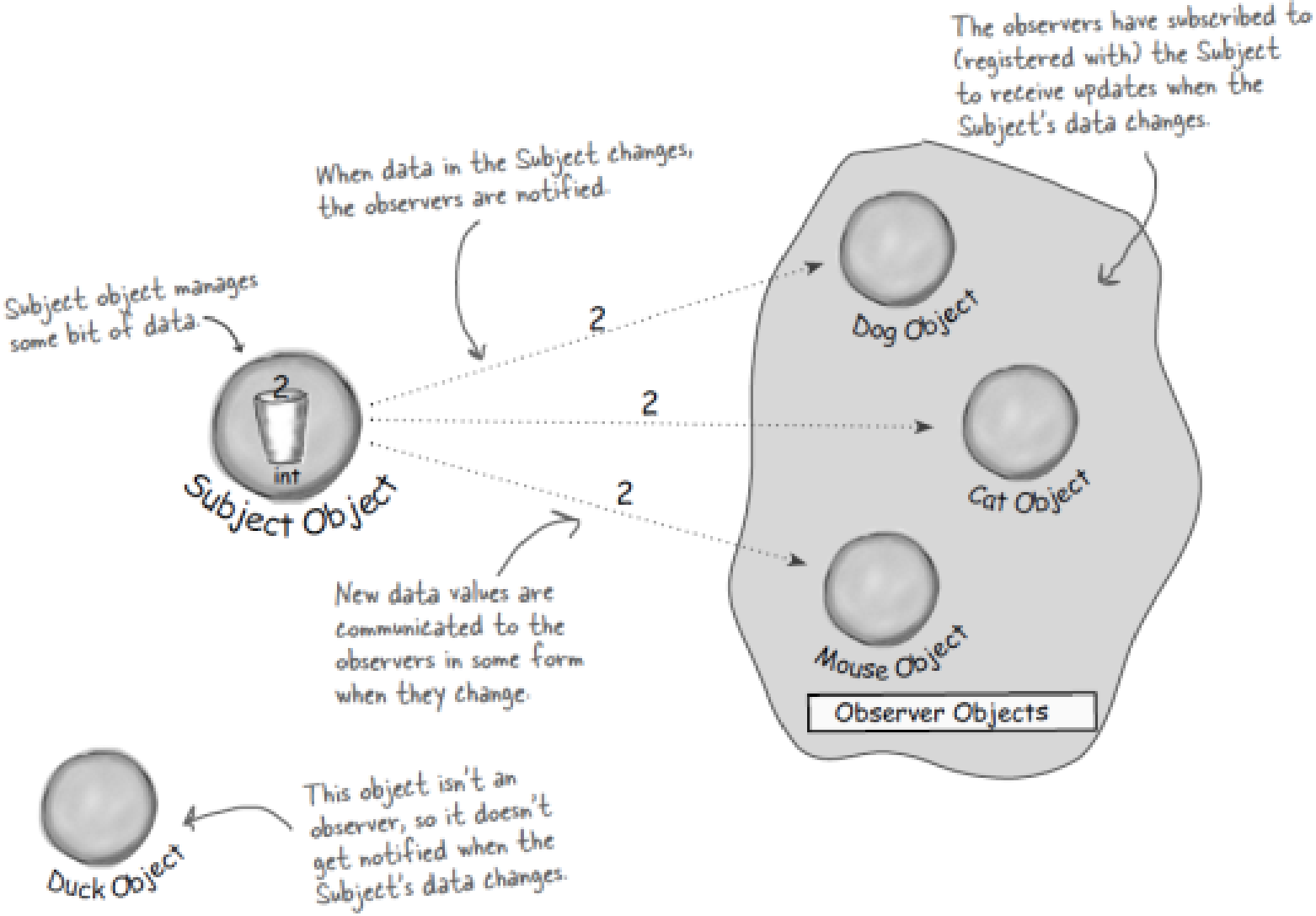
By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

Meet the Observer Pattern

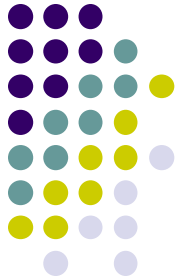


- You know how newspaper or magazine subscriptions work:
  - A newspaper publisher goes into business and begins publishing newspapers.
  - You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
  - You unsubscribe when you don't want papers any more, 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

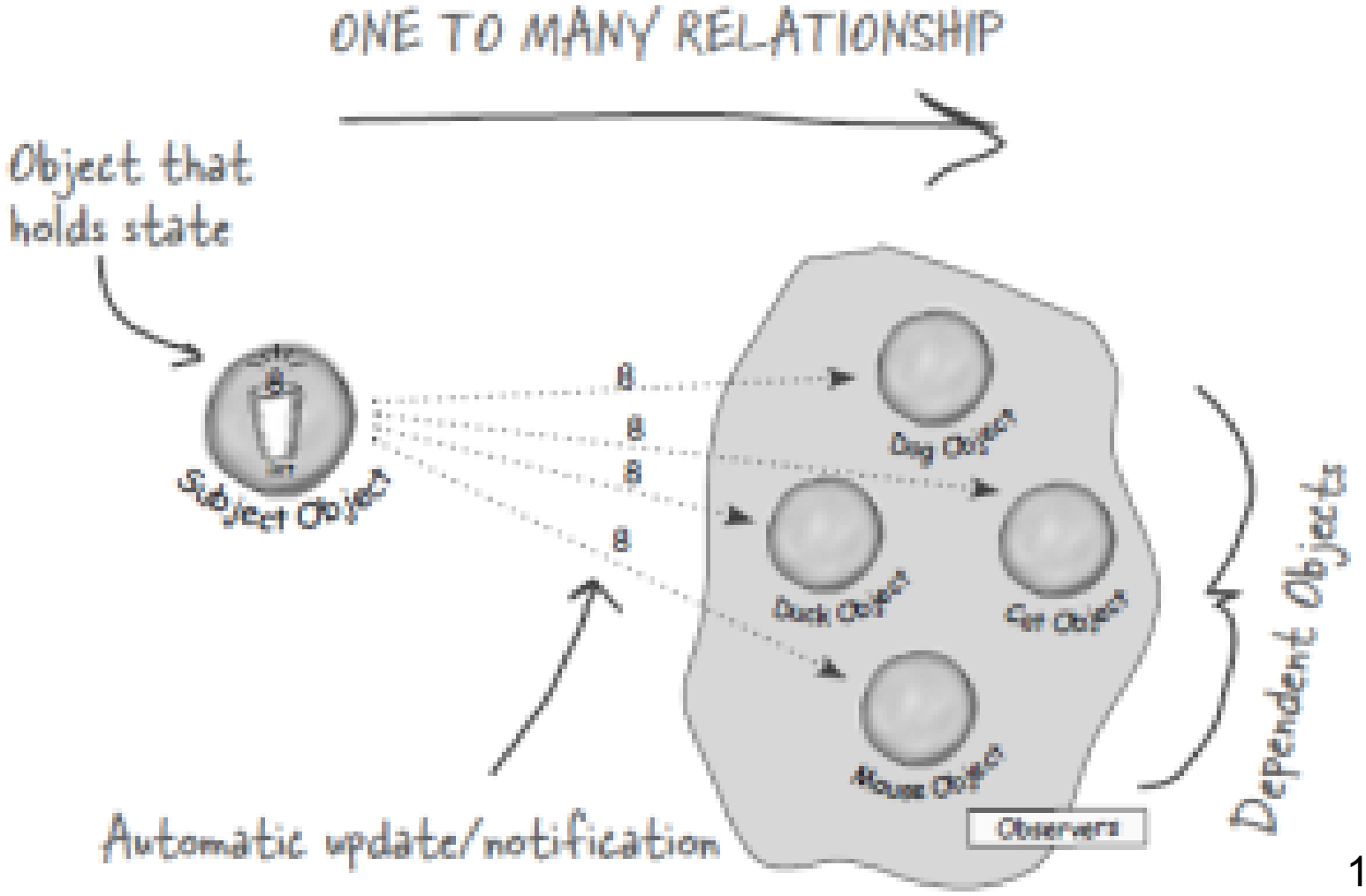


The Observer Pattern Defined

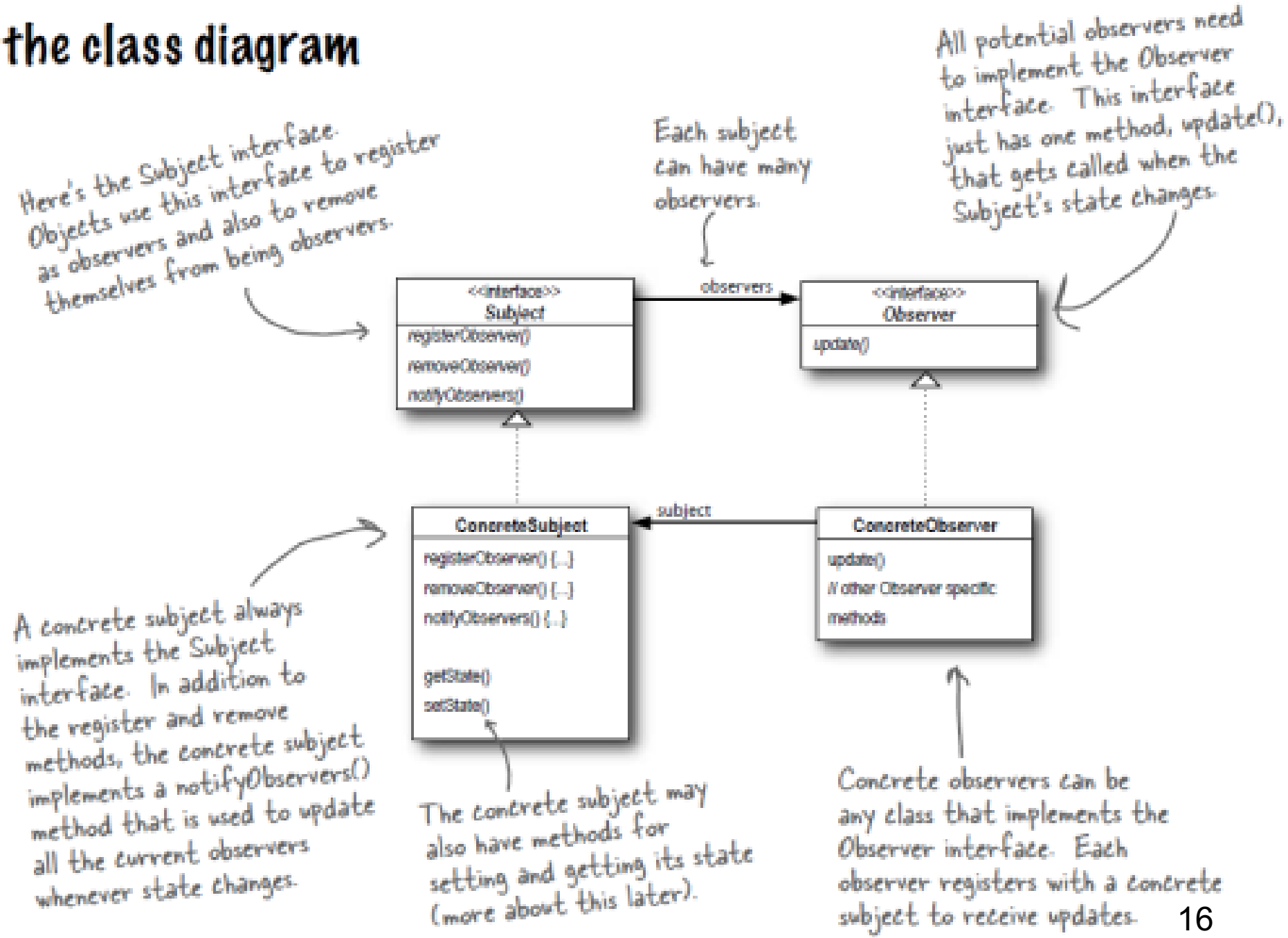


- The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

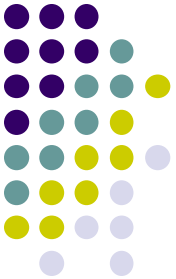
Relation to the example



the class diagram

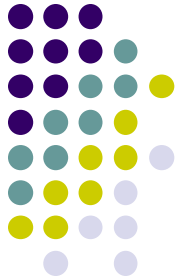


Q & A



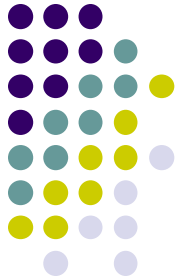
- what does this have to do with one-to-many relationships?
- With the Observer pattern, the Subject is the object that contains the state and controls it. So, there is ONE subject with state. The observers, on the other hand, use the state, even if they rely on the Subject to tell them when its state changes. So there is a relationship between the ONE subject to MANY Observers.

Q & A



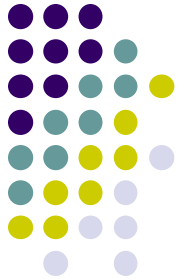
- How does dependence come into this?
- Because the Subject is the sole owner of that data, the observers are dependent on the subject to update them when the data changes. This leads to a cleaner OO design than allowing many objects to control the same data.

The power of loose coupling



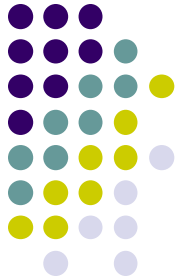
- When two objects are loosely coupled, they can interact, but have very little knowledge of each other.
- The Observer Pattern provides an object design where subjects and observers are loosely coupled.

Why

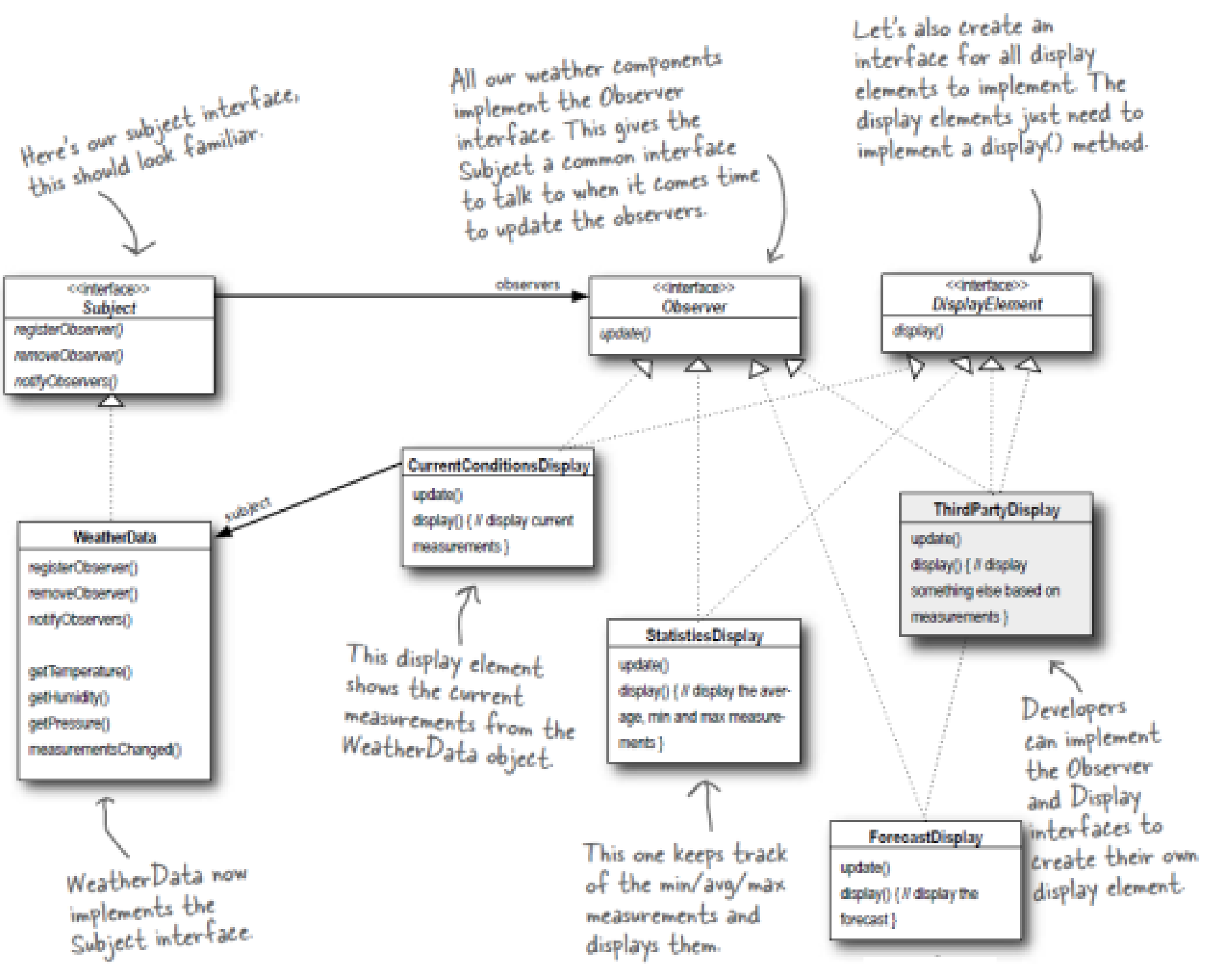


- The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface).
- We can add new observers at any time.
- We never need to modify the subject to add new types of observers.
- We can reuse subjects or observers independently of each other.
- Changes to either the subject or an observer will not affect the other.

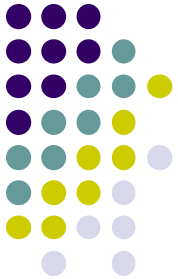
Design Principle



- 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.


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.
<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
<<interface>>
DisplayElement
display()
CurrentConditionsDisplay
update()
display() { // display current measurements }
subject
WeatherData
registerObserver()
removeObserver()
notifyObservers()
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
ThirdPartyDisplay
update()
display() { // display something else based on measurements }
This display element shows the current measurements from the WeatherData object.
StatisticsDisplay
update()
display() { // display the average, min and max measurements }
Developers can implement the Observer and Display interfaces to create their own display element.
ForecastDisplay
update()
display() { // display the forecast }
WeatherData now implements the Subject interface.
This one keeps track of the min/avg/max measurements and displays them.

Implementation -- subject



```
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.

Subject interface in WeatherData

```
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);
        }
    }
```

WeatherData now implements the Subject interface.

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

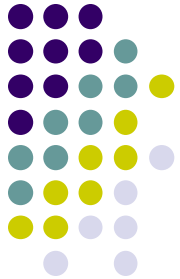
When an observer registers, we just add it to the end of the list.

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.

Cont'd



```
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
```

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.

Display elements

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.

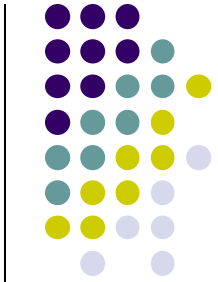
```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    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.

Q & A



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

Test

```
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);
    }
}
```

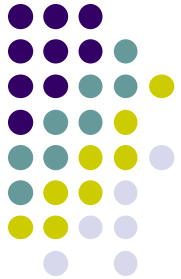
First, create the WeatherData object

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

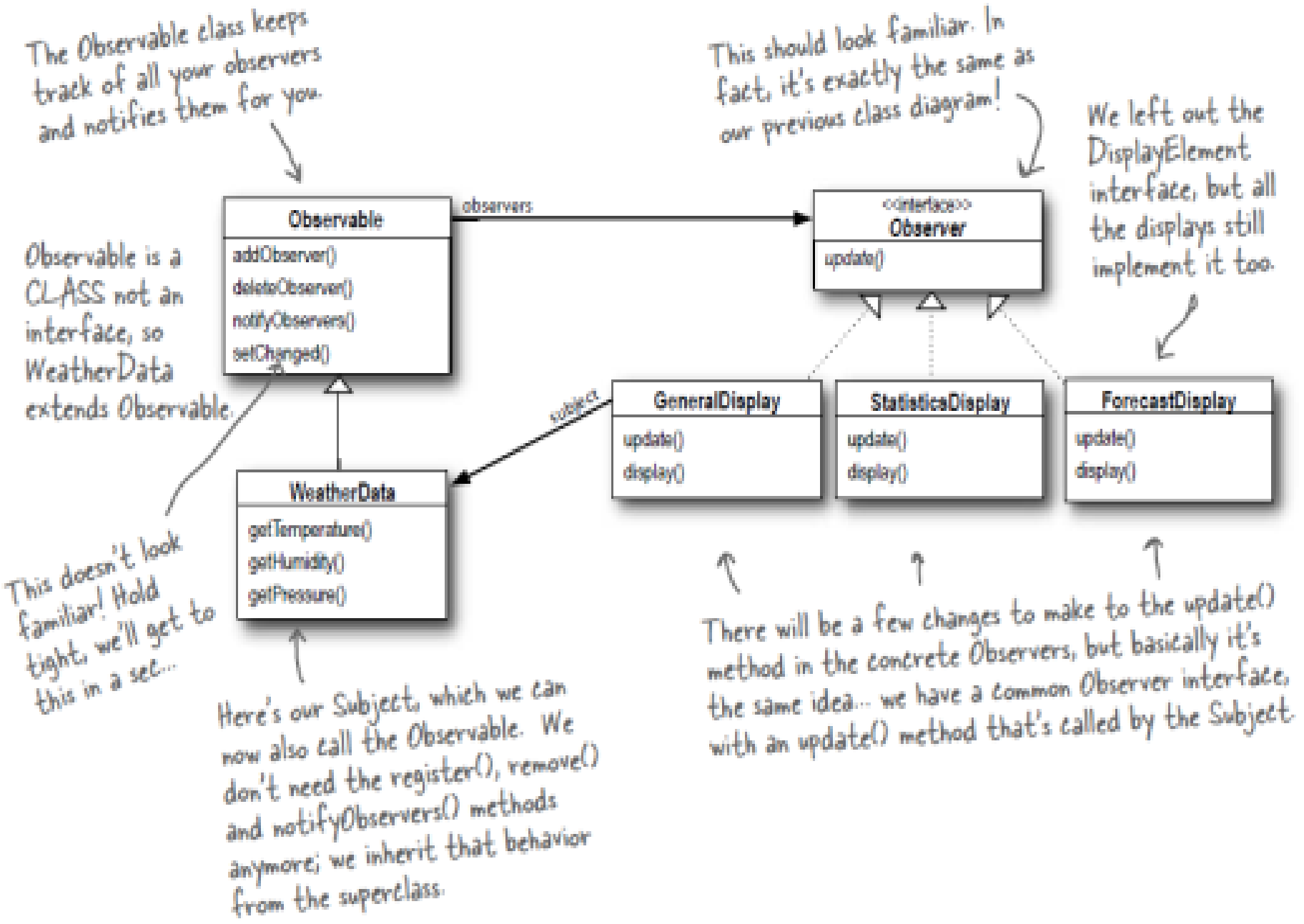
Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.

Using Java's built-in Observer Pattern



- Observable class and Observer interface in java.util package.
- With java's built-in support, all you have to do is extend Observable and tell it when to notify the Observers. The API does the rest for you.


The Observable class keeps track of all your observers and notifies them for you.
This should look familiar. In fact, it's exactly the same as our previous class diagram!
We left out the DisplayElement interface, but all the displays still implement it too.
Observable
addObserver()
deleteObserver()
notifyObservers()
setChanged()
observers
<<interface>>
Observer
update()
Observable is a CLASS not an interface, so WeatherData extends Observable.
subject
GeneralDisplay
update()
display()
StatisticsDisplay
update()
display()
ForecastDisplay
update()
display()
WeatherData
getTemperature()
getHumidity()
getPressure()
This doesn't look familiar! Hold tight, we'll get to this in a sec...
There will be a few changes to make to the update() method in the concrete Observers, but basically it's the same idea... we have a common Observer interface, with an update() method that's called by the Subject
Here's our Subject, which we can now also call the Observable. We don't need the register(), remove() and notifyObservers() methods anymore; we inherit that behavior from the superclass.

How Java's built-in Observer Pattern works

- For an Object to become an observer
  - Extending the java.util.Observable superclass.
  - Call the setChanged() method to signify that the state has changed in your object.
  - Call one of two notifyObservers() methods:
    - notifyObservers() notifyObserver(Object arg)
- For an Observer to receive notifications…
  - Update(Observable o, Object arg)
    - If you want to "push", you can pass the data as a data object to the notifyObserver(arg) method. If not, the Observer has to "pull" the data.

setChanged()

Behind the Scenes

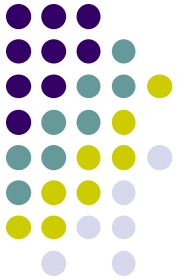
```
setChanged() {
    changed = true
}

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

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

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.

Explanation



- The setChanged() method give you more flexibility.
  - If you want to update the Observers when the temperature changes more than 0.5 degree, not 0.01 degree.

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

2 We are now subclassing Observable.

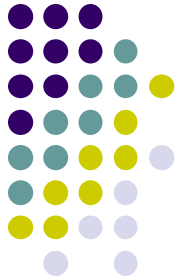
3 We don't need to keep track of our observers anymore, or manage their registration and removal, (the superclass will handle that) so we've removed the code for register, add and notify.

```
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();
    }
```

4 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.

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



```
    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

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

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.

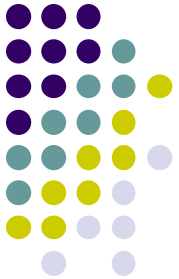
```
import java.util.Observable;
import java.util.Observer;

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 an 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.

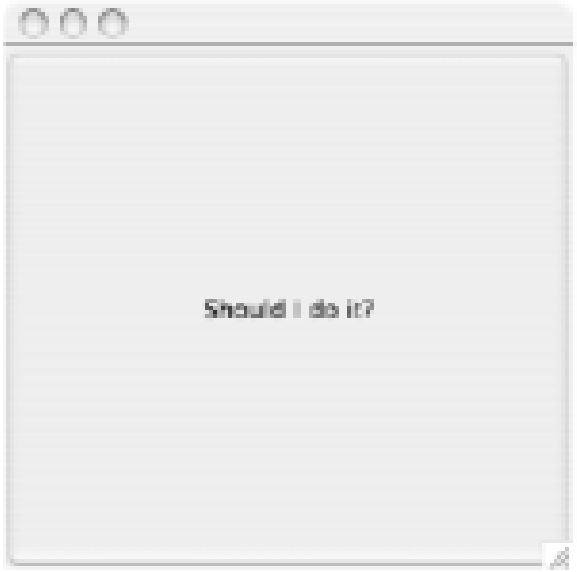
5 In update(), we first make sure the observable is of type WeatherData and then we use its getter methods to obtain the temperature and humidity measurements. After that we call display().

The dark side



- Observable is a class
- Observable protects crucial methods
- 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.

An Swing example



Here's our fancy interface

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

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

Devil answer

Angel answer

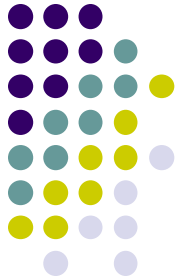
Simple Swing application that just creates a frame and throws a button in it

```
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.

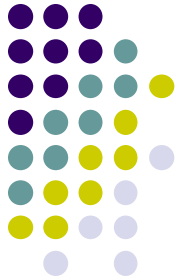
Here are the class definitions for the observers, defined as inner classes (but they don't have to be).

Tools for your Design Toolbox



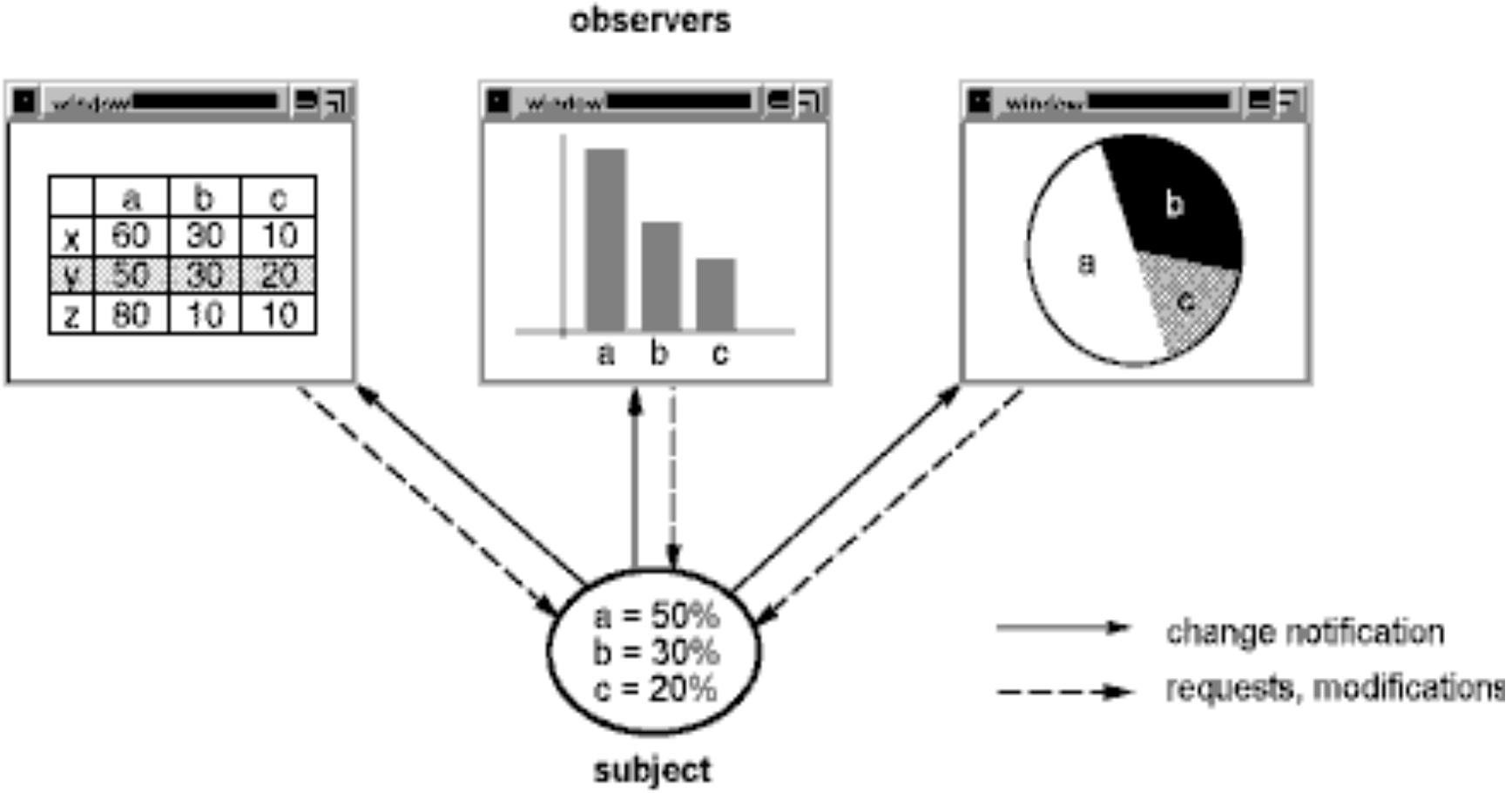
- OO Principles
  - Encapsulate what varies.
  - Favor composition over inheritance.
  - Program to interfaces, not implementations.
  - Strive for loosely coupled designs between objects that interact.
- OO Patterns
  - Observer

Intent

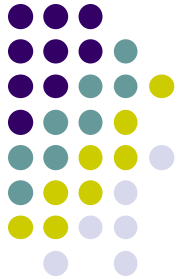


- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Also Known As**
  - Dependents, Publish-Subscribe

Motivation

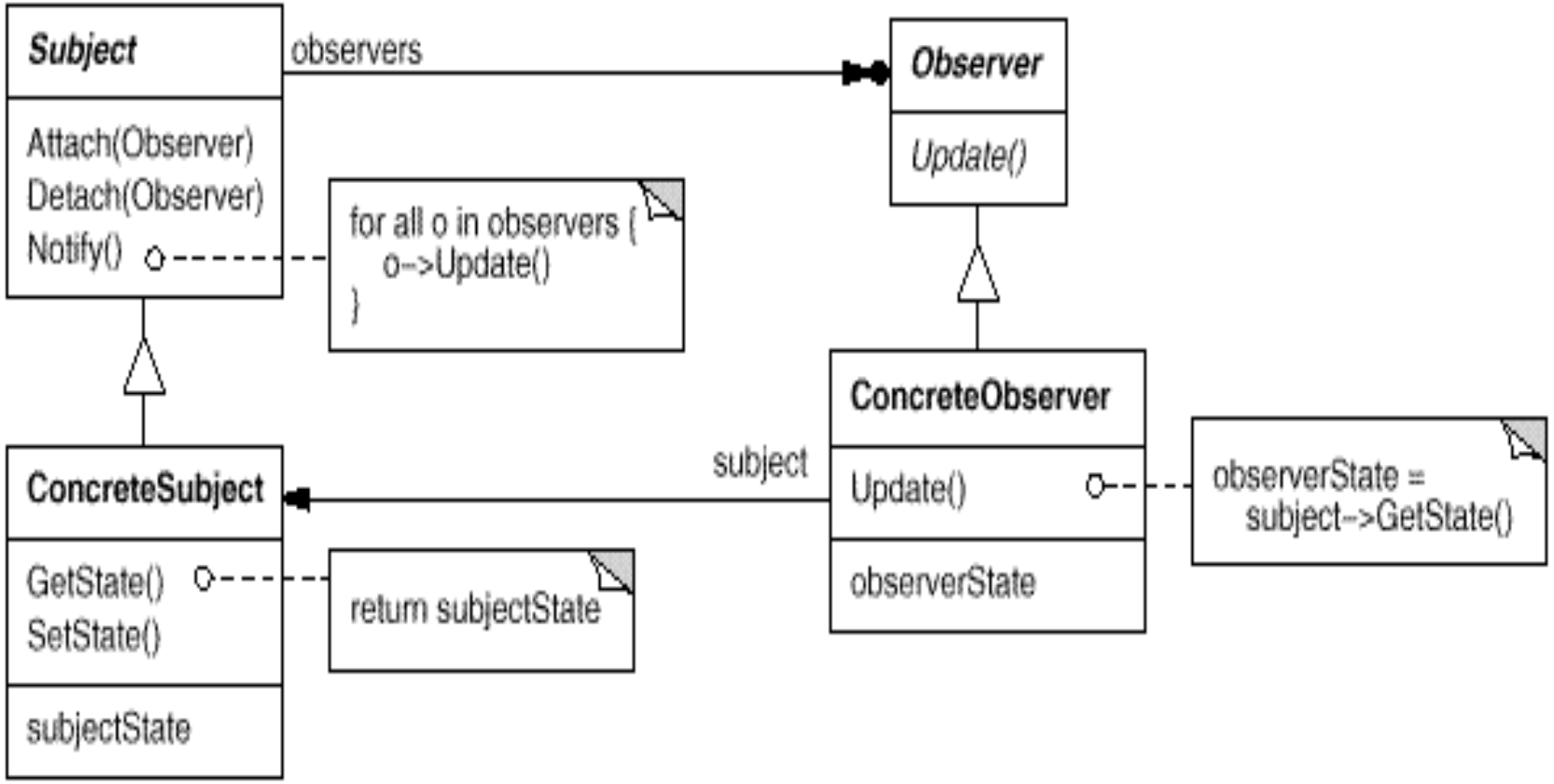

observers
a b c
x 60 30 10
y 50 30 20
z 80 10 10
a b c
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modifications

Applicability

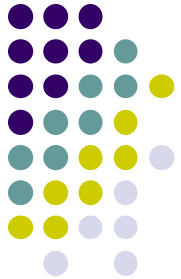


- Use the Observer pattern in any of the following situations:
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Update()
ConcreteObserver
subject
ConcreteSubject
Update()
observerState =
subject->GetState()
GetState()
SetState()
return subjectState
observerState
subjectState

Implementation



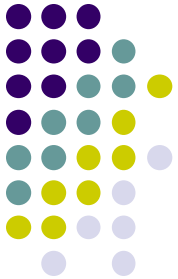
- *Mapping subjects to their observers.*

The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject. However, such storage may be too expensive when there are many subjects and few observers. One solution is to trade space for time by using an associative look-up (e.g., a hash table) to maintain the subject-to-observer mapping. Thus a subject with no observers does not incur storage overhead. On the other hand, this approach increases the cost of accessing the observers.

- *Observing more than one subject.*

It might make sense in some situations for an observer to depend on more than one subject. For example, a spreadsheet may depend on more than one data source. It's necessary to extend the Update interface in such cases to let the observer know *which* subject is sending the notification. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.

Implementation—cont'd

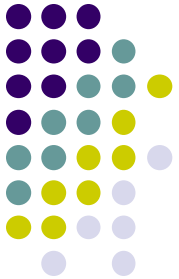


- *Who triggers the update?*

The subject and its observers rely on the notification mechanism to stay consistent. But what object actually calls Notify to trigger the update? Here are two options:

- Have state-setting operations on Subject call Notify after they change the subject's state. The advantage of this approach is that clients don't have to remember to call Notify on the subject. The disadvantage is that several consecutive operations will cause several consecutive updates, which may be inefficient.
- Make clients responsible for calling Notify at the right time. The advantage here is that the client can wait to trigger the update until after a series of state changes has been made, thereby avoiding needless intermediate updates. The disadvantage is that clients have an added responsibility to trigger the update. That makes errors more likely, since clients might forget to call Notify.

Implementation—cont'd

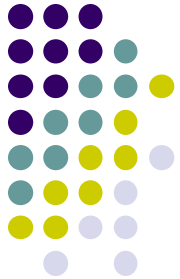


- Avoiding observer-specific update protocols: the push and pull models. Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to Update. The amount of information may vary widely.

- At one extreme, which we call the push model, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme is the pull model; the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.

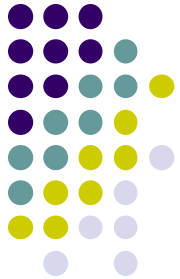
- The pull model emphasizes the subject's ignorance of its observers, whereas the push model assumes subjects know something about their observers' needs. The push model might make observers less reusable, because Subject classes make assumptions about Observer classes that might not always be true. On the other hand, the pull model may be inefficient, because Observer classes must ascertain what changed without help from the Subject.

Implementation—cont'd



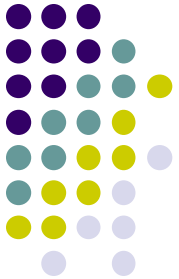
- Specifying modifications of interest explicitly.
- You can improve update efficiency by extending the subject's registration interface to allow registering observers only for specific events of interest. When such an event occurs, the subject informs only those observers that have registered interest in that event. One way to support this uses the notion of aspects for Subject objects.

Variation



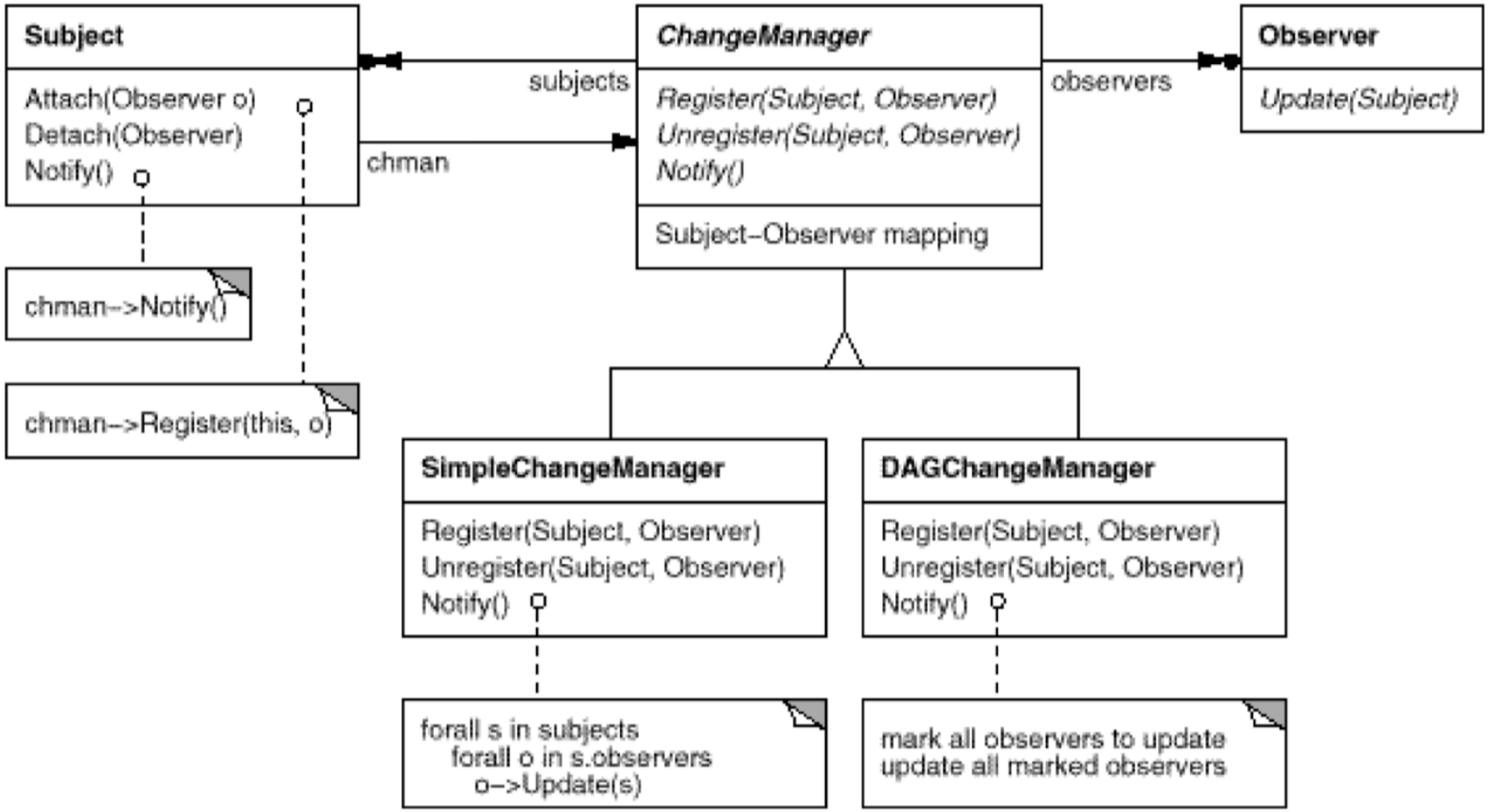
- When the dependency relationship between subjects and observers is particularly complex, an object that maintains these relationships might be required. We call such an object a **ChangeManager**. Its purpose is to minimize the work required to make observers reflect a change in their subject. For example, if an operation involves changes to several interdependent subjects, you might have to ensure that their observers are notified only after all the subjects have been modified to avoid notifying observers more than once.

ChangeManager

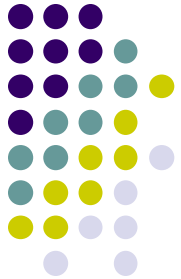


- ChangeManager has three responsibilities:
  - It maps a subject to its observers and provides an interface to maintain this mapping. This eliminates the need for subjects to maintain references to their observers and vice versa.
  - It defines a particular update strategy.
  - It updates all dependent observers at the request of a subject.

Variation

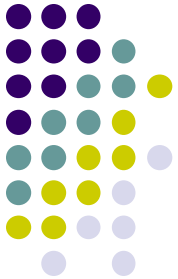

Subject
Attach(Observer o)
Detach(Observer)
Notify()
subjects
chman
ChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
Subject–Observer mapping
observers
Observer
Update(Subject)
chman->Notify()
chman->Register(this, o)
SimpleChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
DAGChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
forall s in subjects
forall o in s.observers
o->Update(s)
mark all observers to update
update all marked observers

Consequences



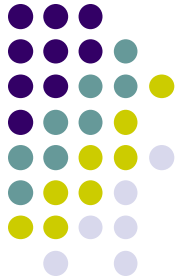
- The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.

Consequences



- Abstract coupling between Subject and Observer.
  - All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.
  - Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact. If Subject and Observer are lumped together, then the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).

Consequences



- Unexpected updates.
  - Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

Reviews 1

- 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.

Reviews 2

- 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.