Observer Pattern

Objekte auf dem Laufenden halten

Werkzeugkasten

OO Basics

Abstraction

Encapsulation

Polymorphism

Inheritance

We assume you know the OO basics of using classes polymorphically, how inheritance is like design by contract, and how encapsulation works. If you are a little rusty on these, pull out your Head First Java and review, then skim this chapter again.

OO Principles

Encapsulate what varies.

Favor composition over inheritence.

Program to interfaces, not implementations.

We'll be taking a closer look at these down the road and also adding a few more to the list

OO Patterns

Strategy – defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

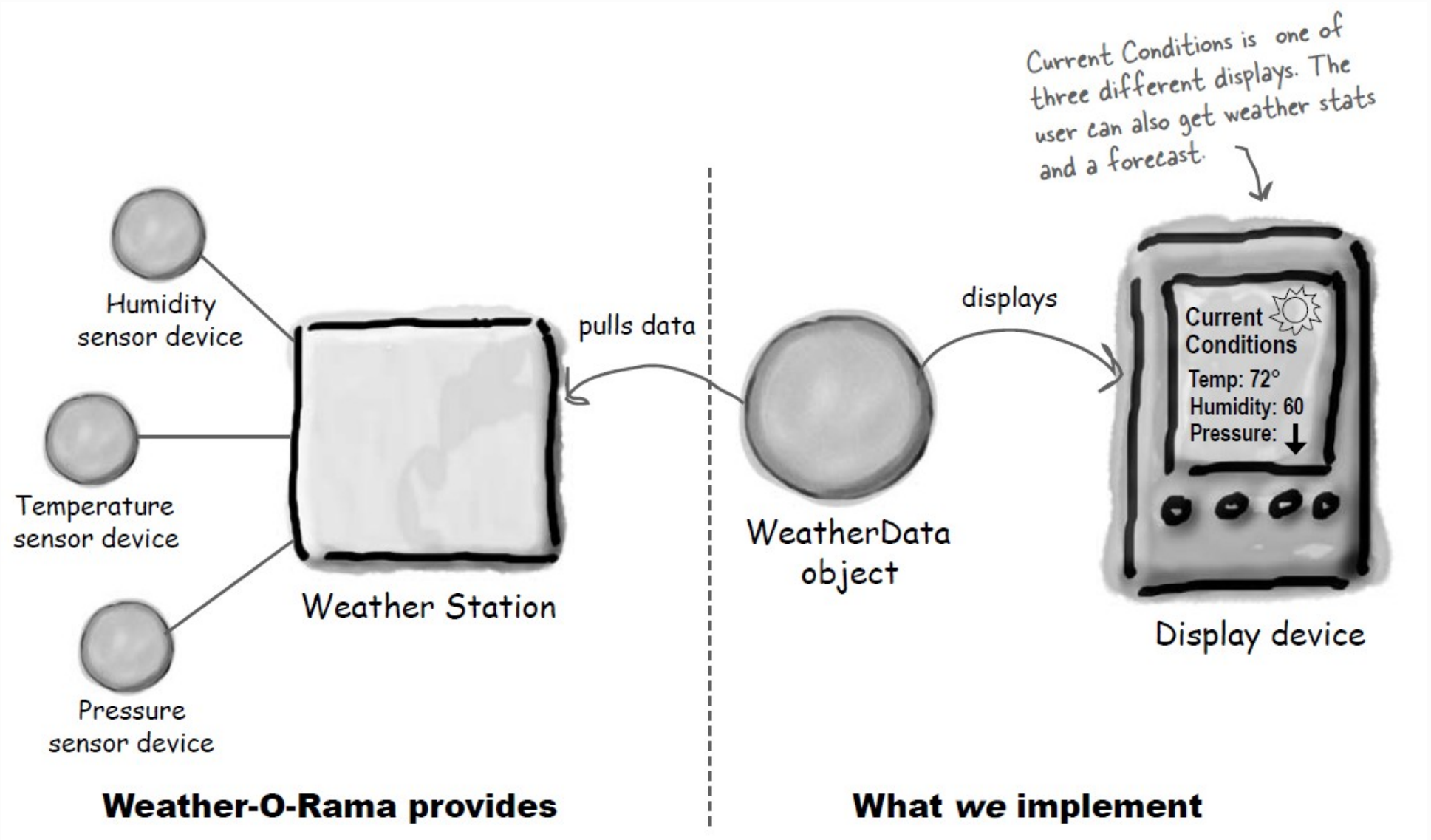
Throughout the book think about how patterns rely on OO basics and principles.

One down, many to go!

Projekt „Wetterstation“

- Basiert auf patentierten Wetterdaten-Objekt
- 3 Anzeige-Elemente
  (aktuell, Statistik und Vorhersage)
- Aktualisierung in Echtzeit
- API für zukünftige Erweiterungen
  (eigene Anzeige-Elemente)

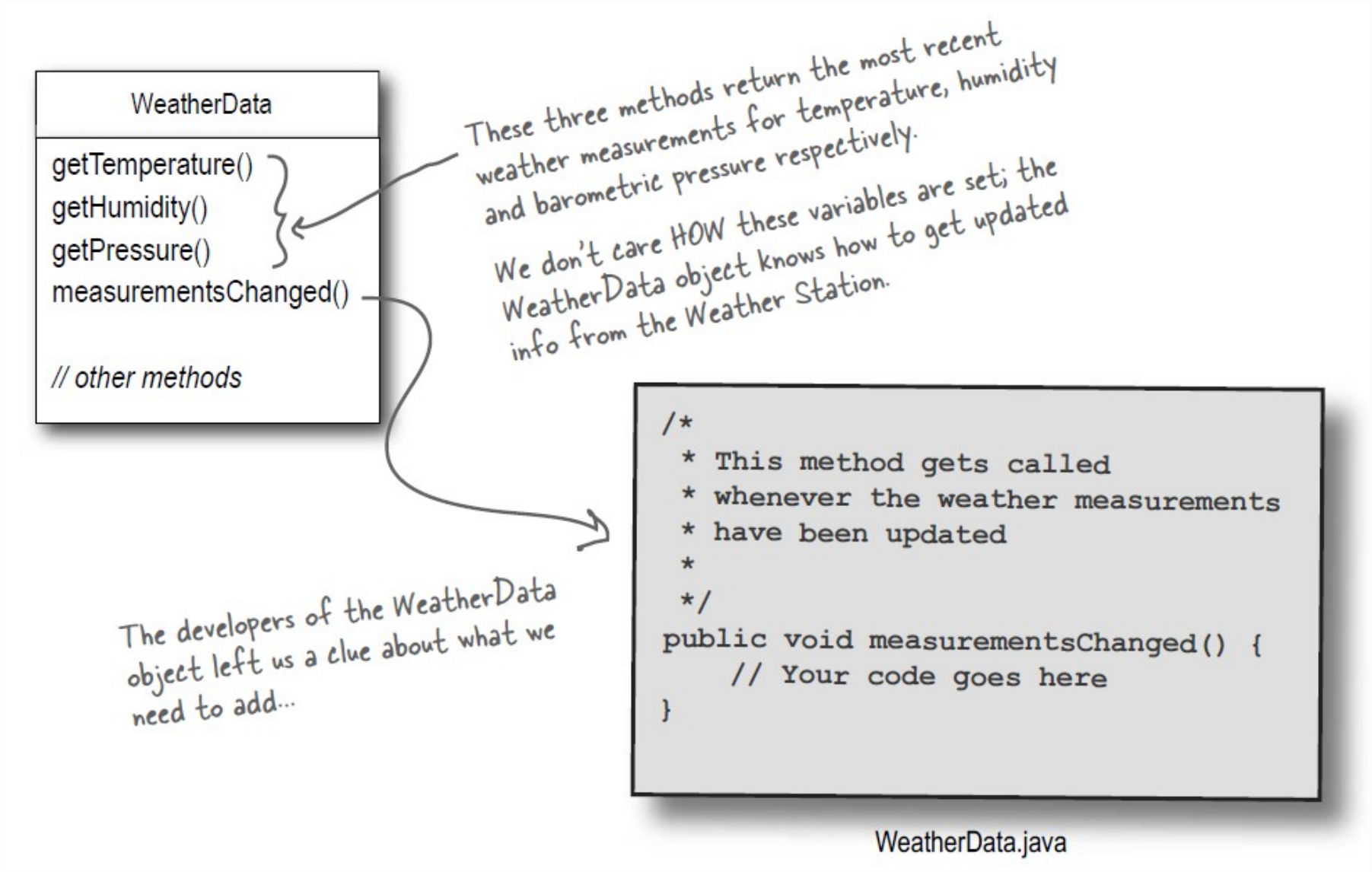
„Wetterstation“ im Überblick



Unsere Aufgabe

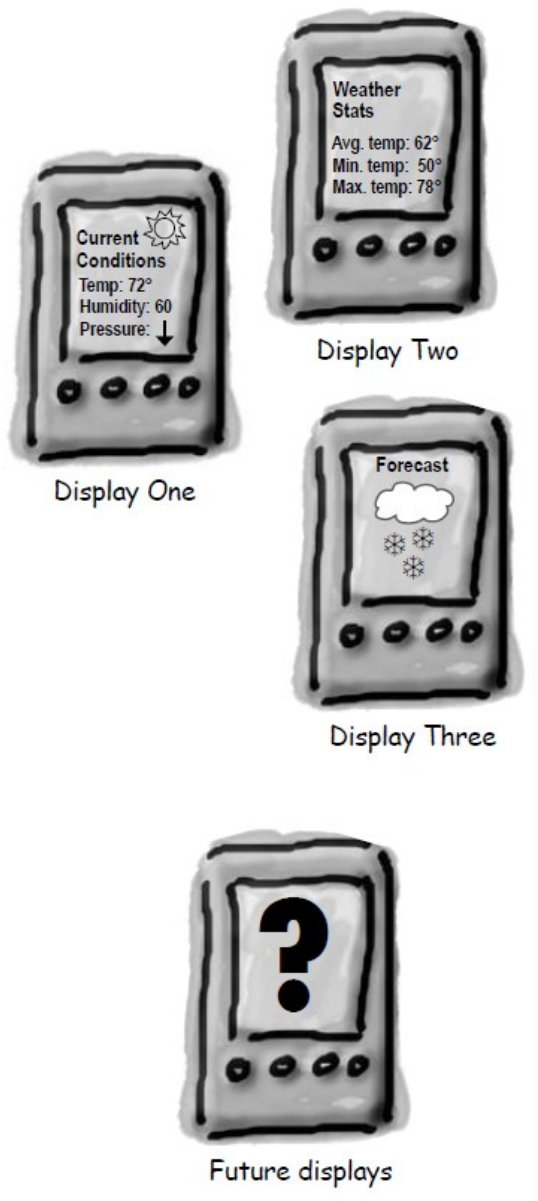
- Wetterdaten-Objekt einsetzen
- Verfügbare Anzeigen aktualisieren
  - Aktuelle Wetterbedingungen
  - Wetterstatistik
  - Wettervorhersage

Die „WetterDaten"-Klasse



Was wir bisher wissen

- WetterDaten hat getter-Methoden für alle 3 Messwerte
- *measurementsChanged()* wird nach jeder Aktualisierung aufgerufen
- Es müssen 3 Anzeigeelemente implementiert werden
- Das System muss erweiterbar sein



Unser erster Entwurf

```
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.

Probleme!

```
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.

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

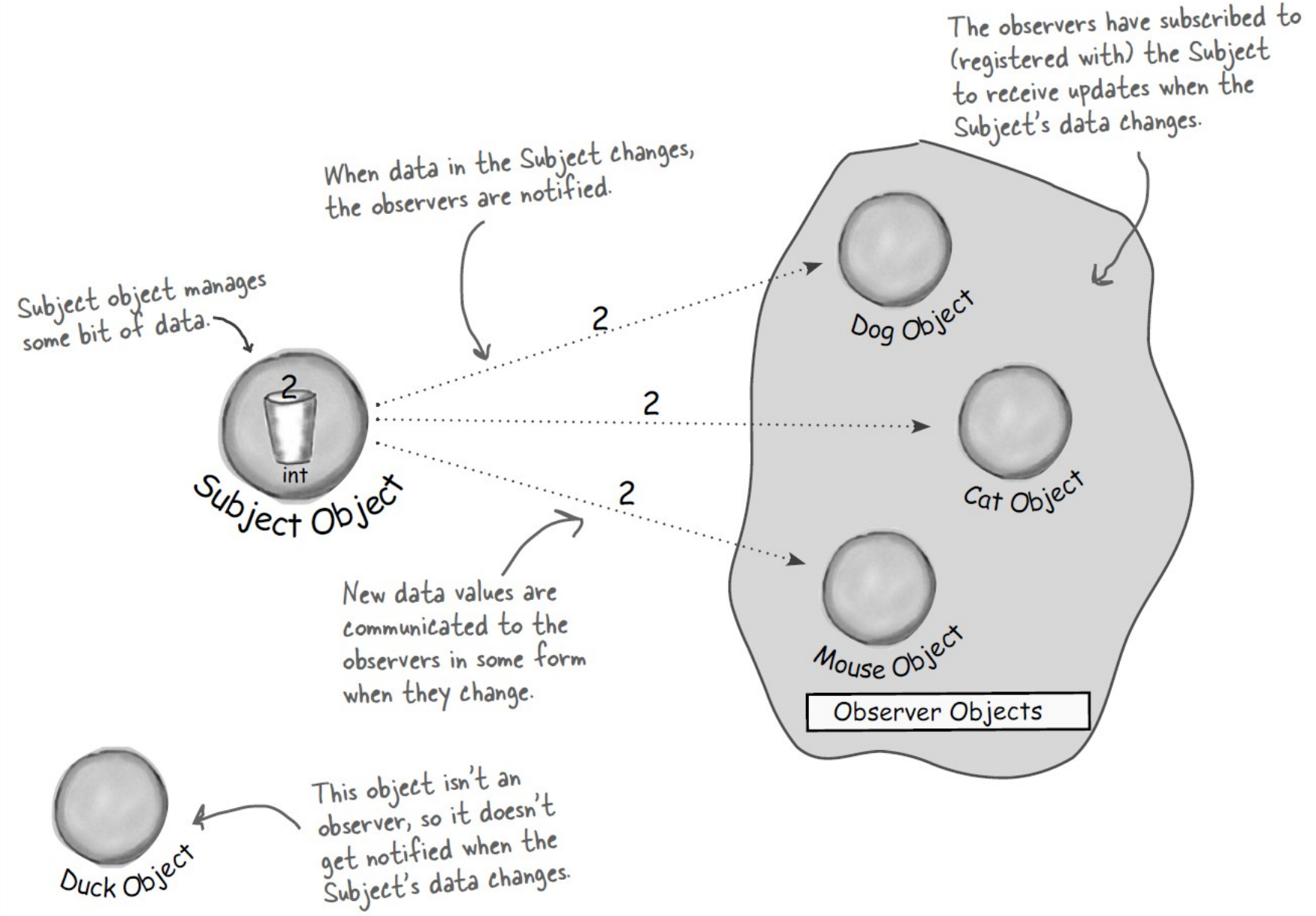
At least we seem to be using a common interface to talk to the display elements... they all have an update() method takes the temp, humidity, and pressure values.

Gestatten: das Observer-Pattern

Ähnlich wie ein Zeitungsabo

- Verlag startet sein Geschäft und veröffentlicht Zeitungen
- Man abonniert ein bestimmtes Produkt
  - Jedes Mal, wenn das Produkt aktualisiert wird, bekommt man eine neue Ausgabe
- Das Abo kann jederzeit gekündigt werden
  - Man erhält auch keine Aktualisierungen mehr

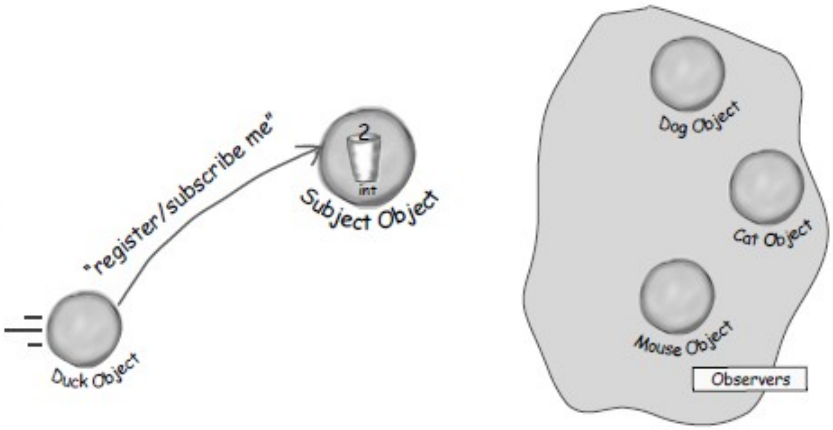
Herausgeber + Abonnenten = Observer-Pattern



Observe ...

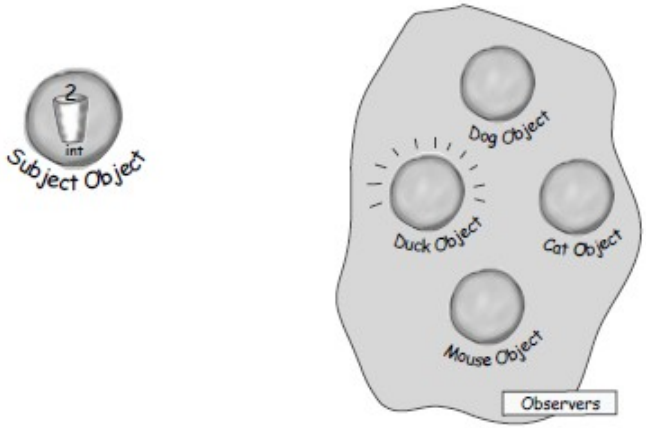
A Duck object comes along and tells the Subject that it wants to become an observer.

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



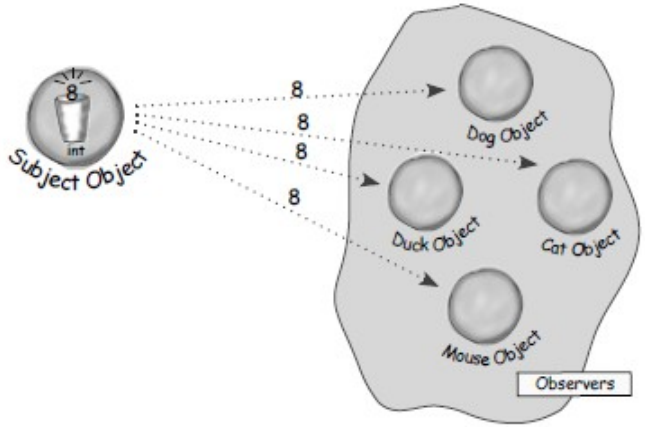
The Duck object is now an official observer.

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!

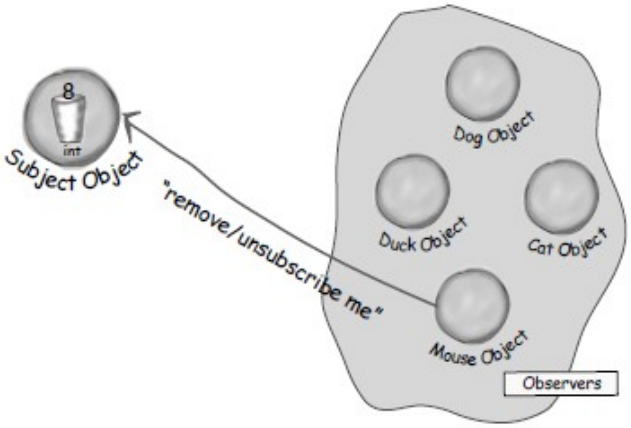
Now Duck and all the rest of the observers get a notification that the Subject has changed.



Observe ...

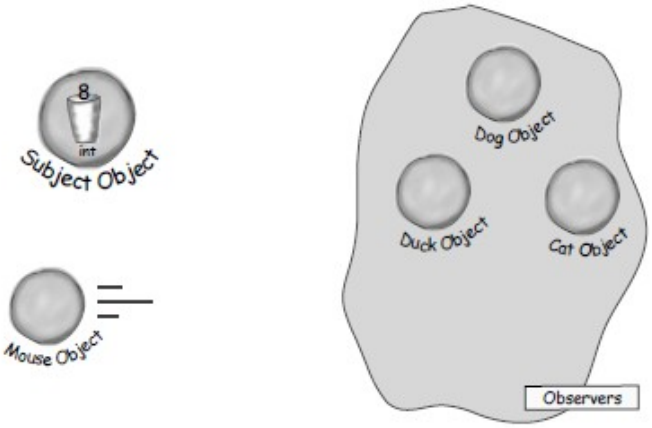
The Mouse object asks to be removed as an observer.

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.



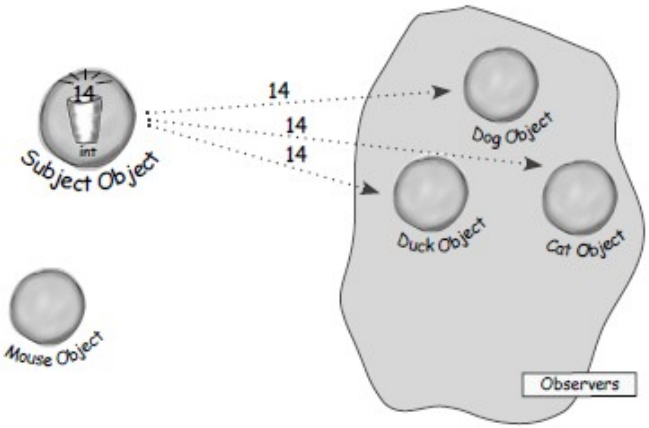
Mouse is outta here!

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



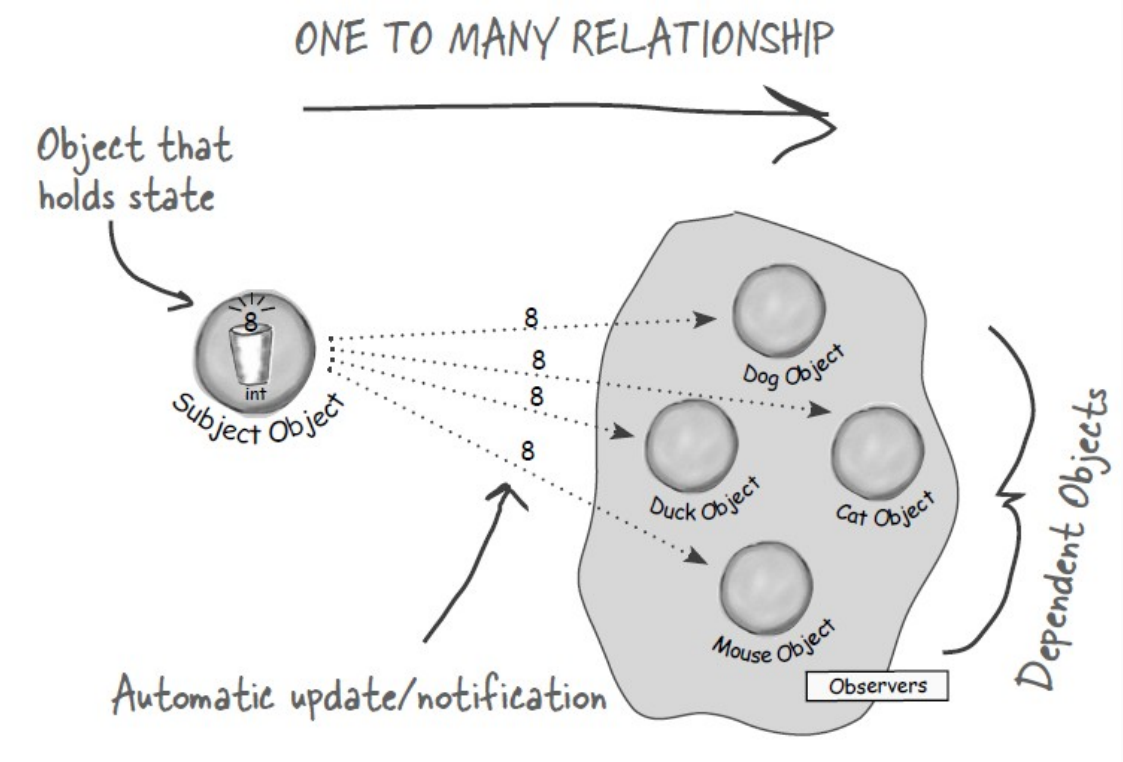
The Subject has another new int.

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.

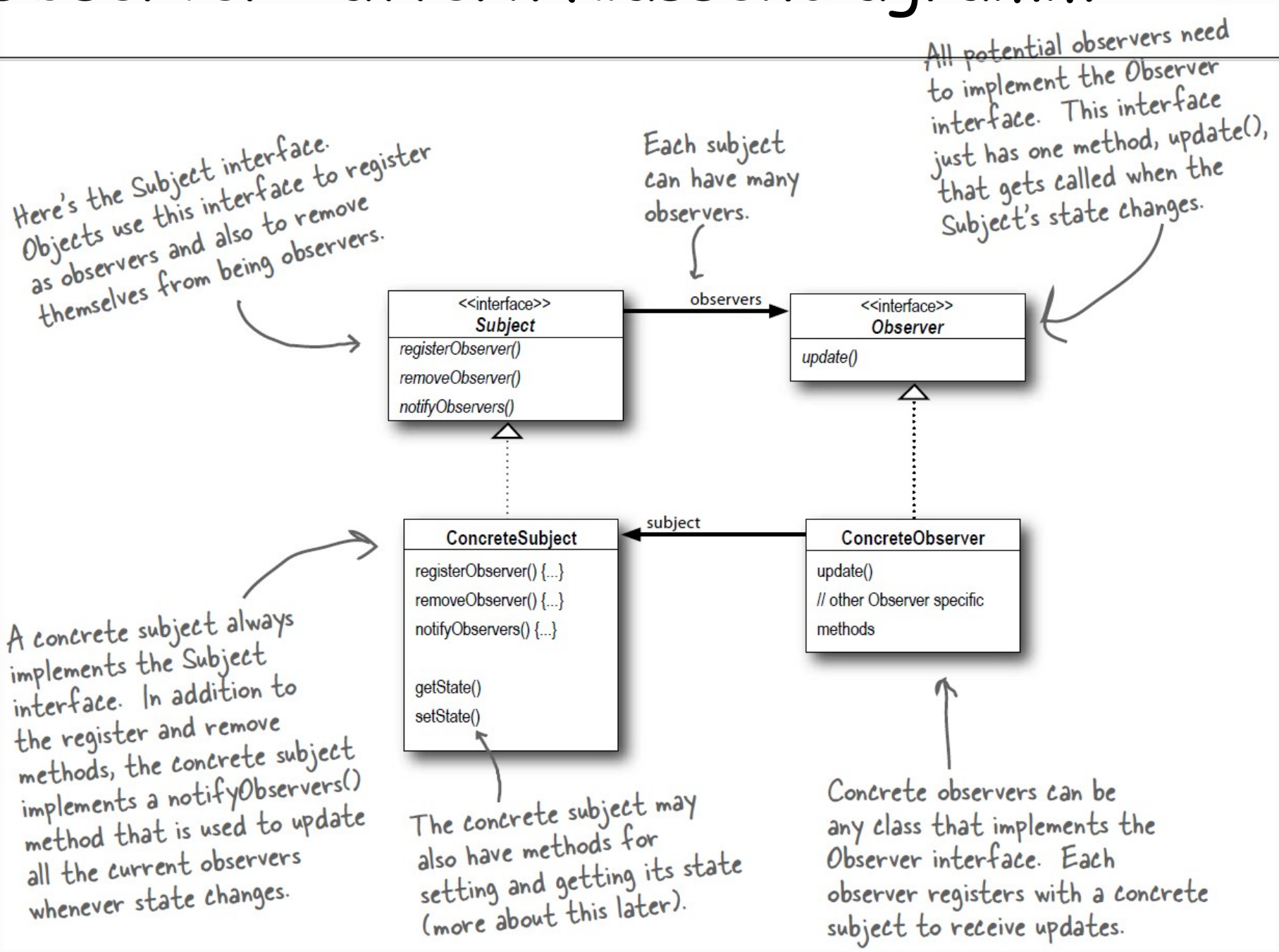


Definition des Observer-Patterns

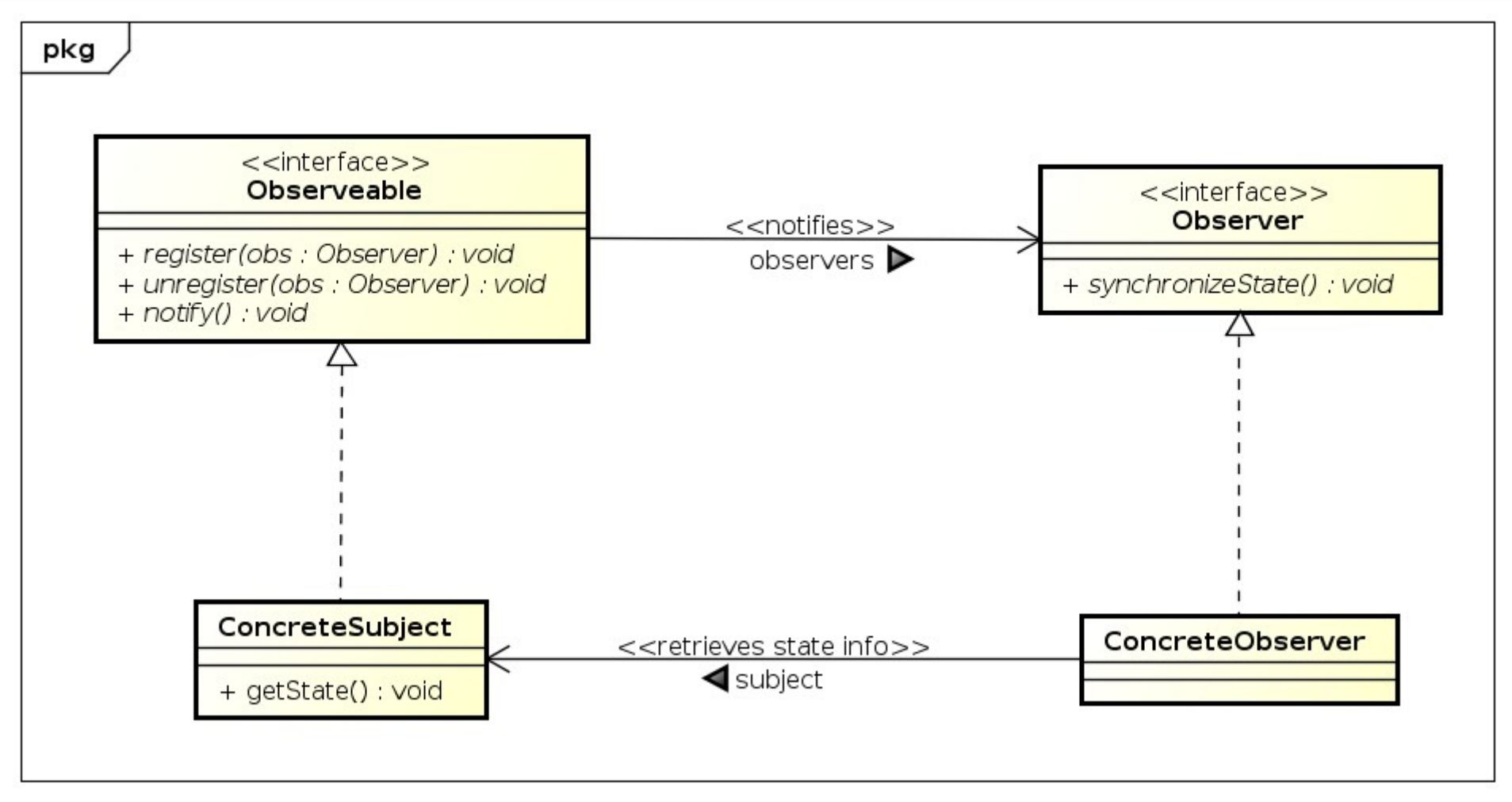
- Das Observer-Pattern definiert eine Eins-zu-viele-Abhängigkeit zwischen Objekten in der Art, dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objekts ändert.



Observer-Pattern Klassendiagramm



Observer-Pattern Klassendiagramm



Die Macht der losen Kopplung

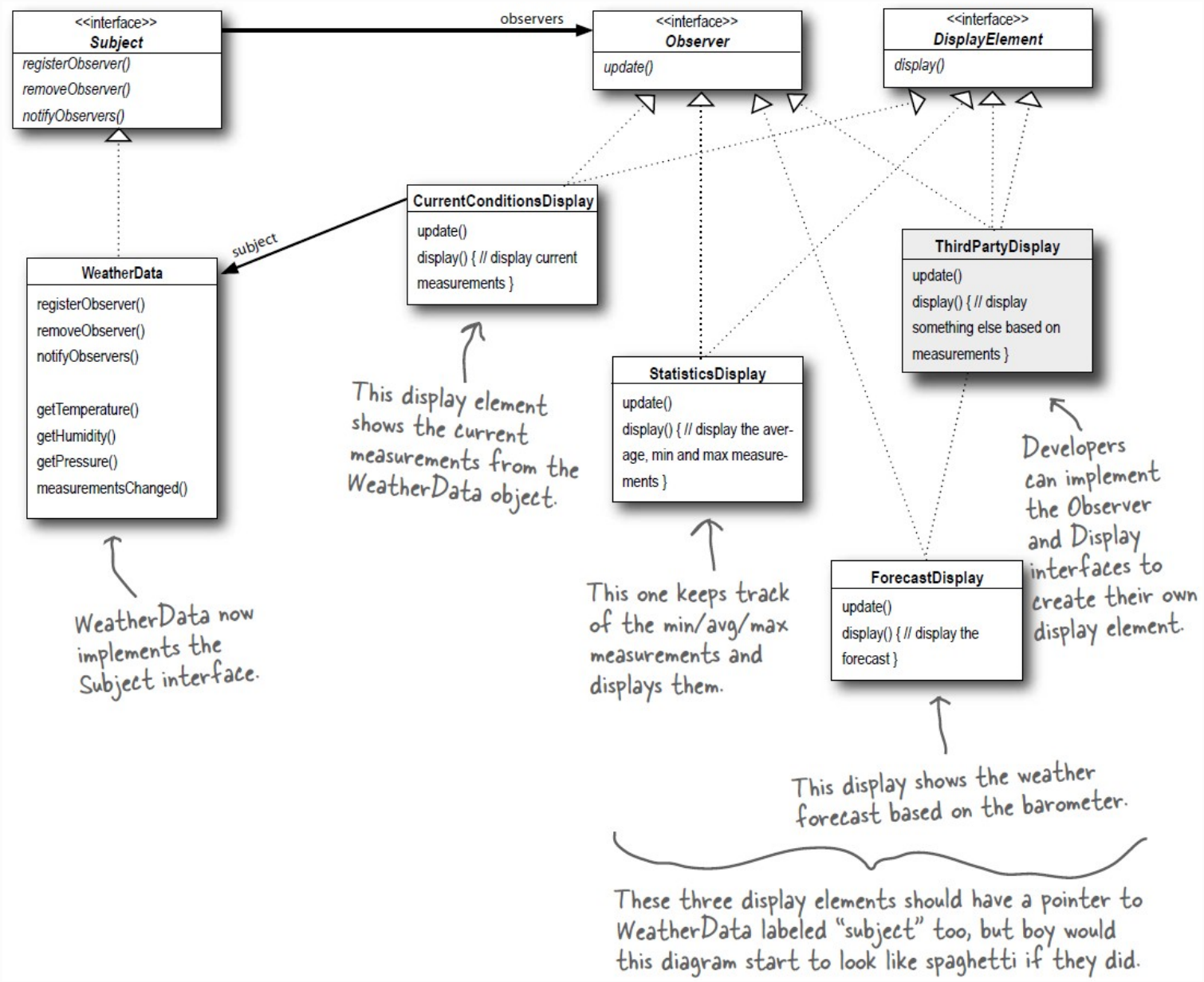
- Locker gebunden = Interaktion mit wenig Detailwissen
- Observer-Pattern: lockere Kopplung zwischen Subjekt und Beobachter
  - Subjekt kennt von einem Beobachter nur die Beobachter-Schnittstelle
  - Subjekt muss für neue Beobachter nicht verändert werden
  - Subjekt und Beobachter unabhängig verwendbar

Design Principle

Strive for loosely coupled designs between objects that interact.

Skizziere die Klassen

Skizziere die Klassen



Implementierung „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.

Implementierung „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.

Implementierung „WeatherData“

```
    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.

Implementierung eines Anzeigeelements

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.

Testlauf

```
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.

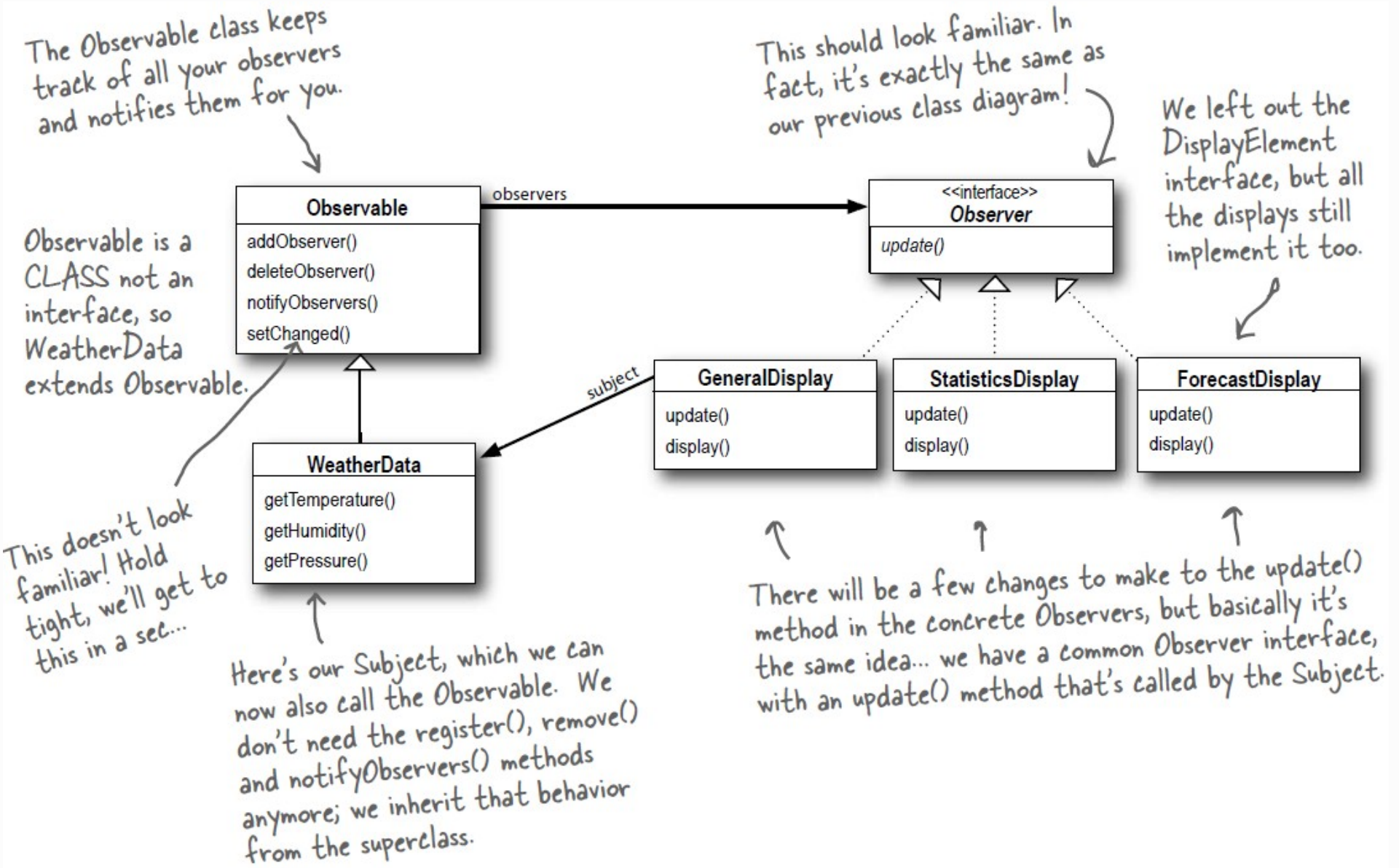
If you don't want to download the code, you can comment out these two lines and run it.

Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.

```
%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
%
```

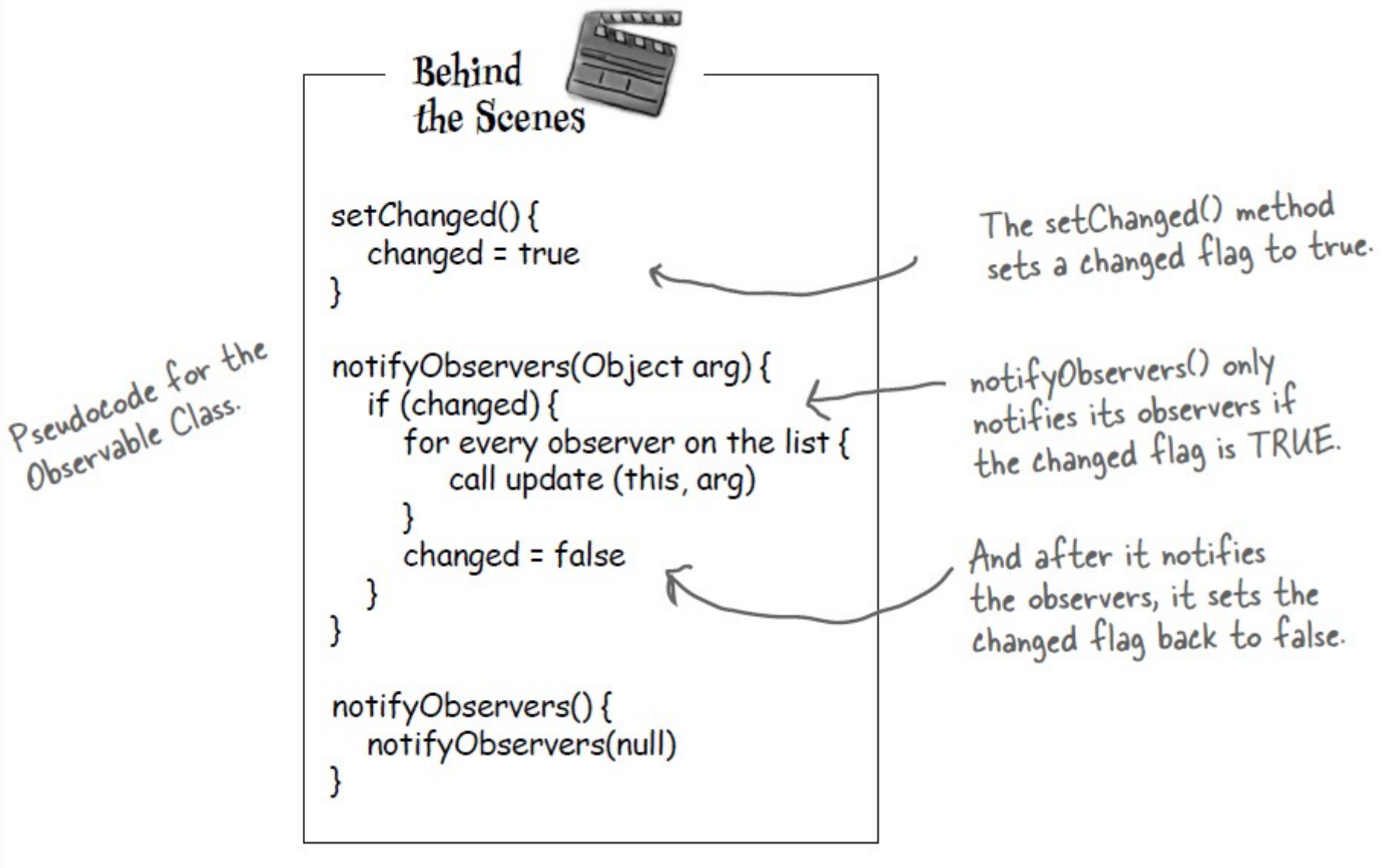
Java eingebautes Observer-Pattern



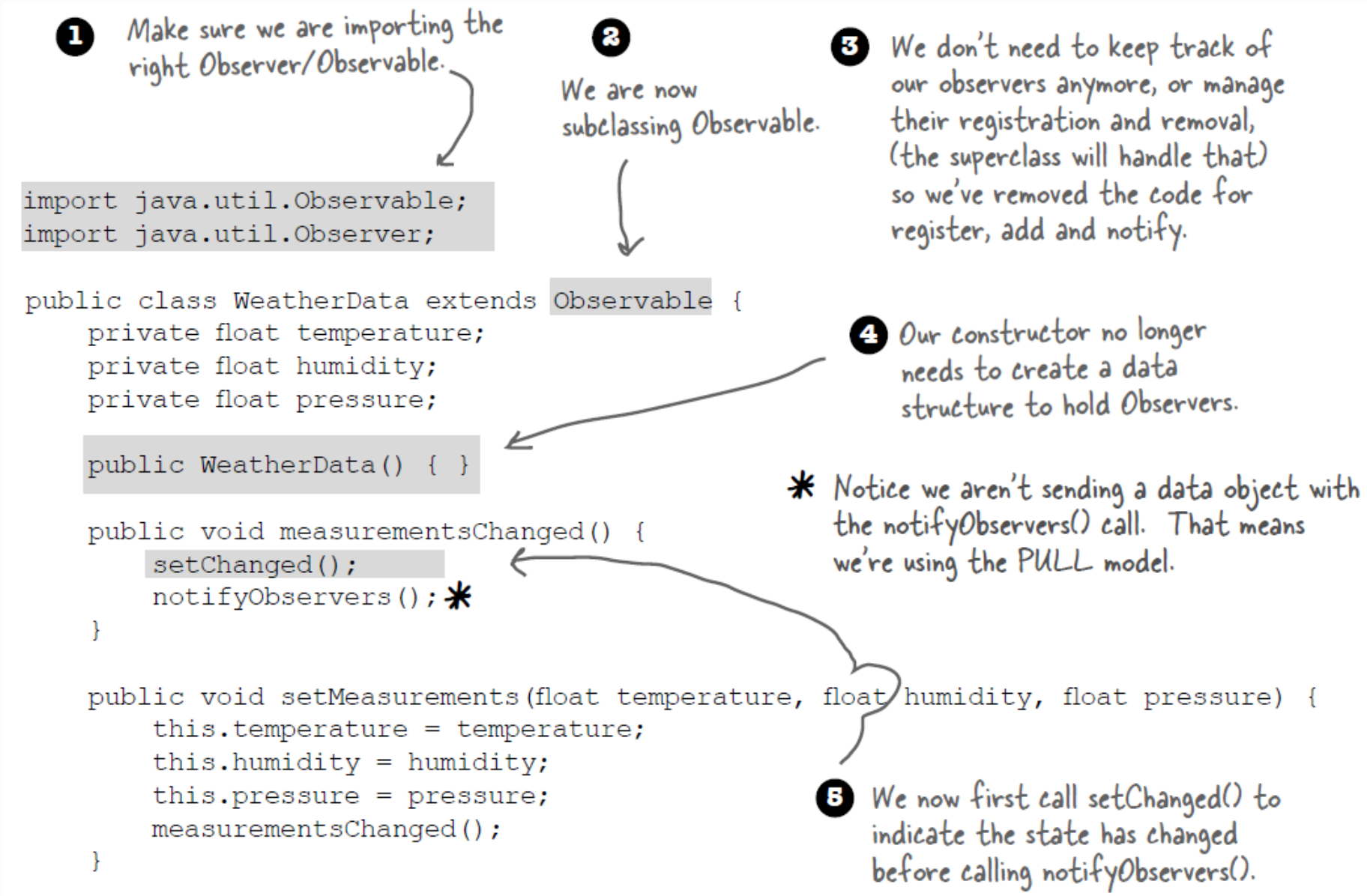
Funktionsweise

- Beobachter-Klassen implementieren java.util.Observer
- Die Subjekt-Klasse erweitert(!) java.util.Observable
- Nachrichten schicken:
  - setChanged() aufrufen
  - notifyObservers() oder notifyObservers(Object arg)
- Benachrichtigung erhalten:
  - update(Observable o, Object arg) implementieren

Hinter den Kulissen



Wetterstation überarbeiten



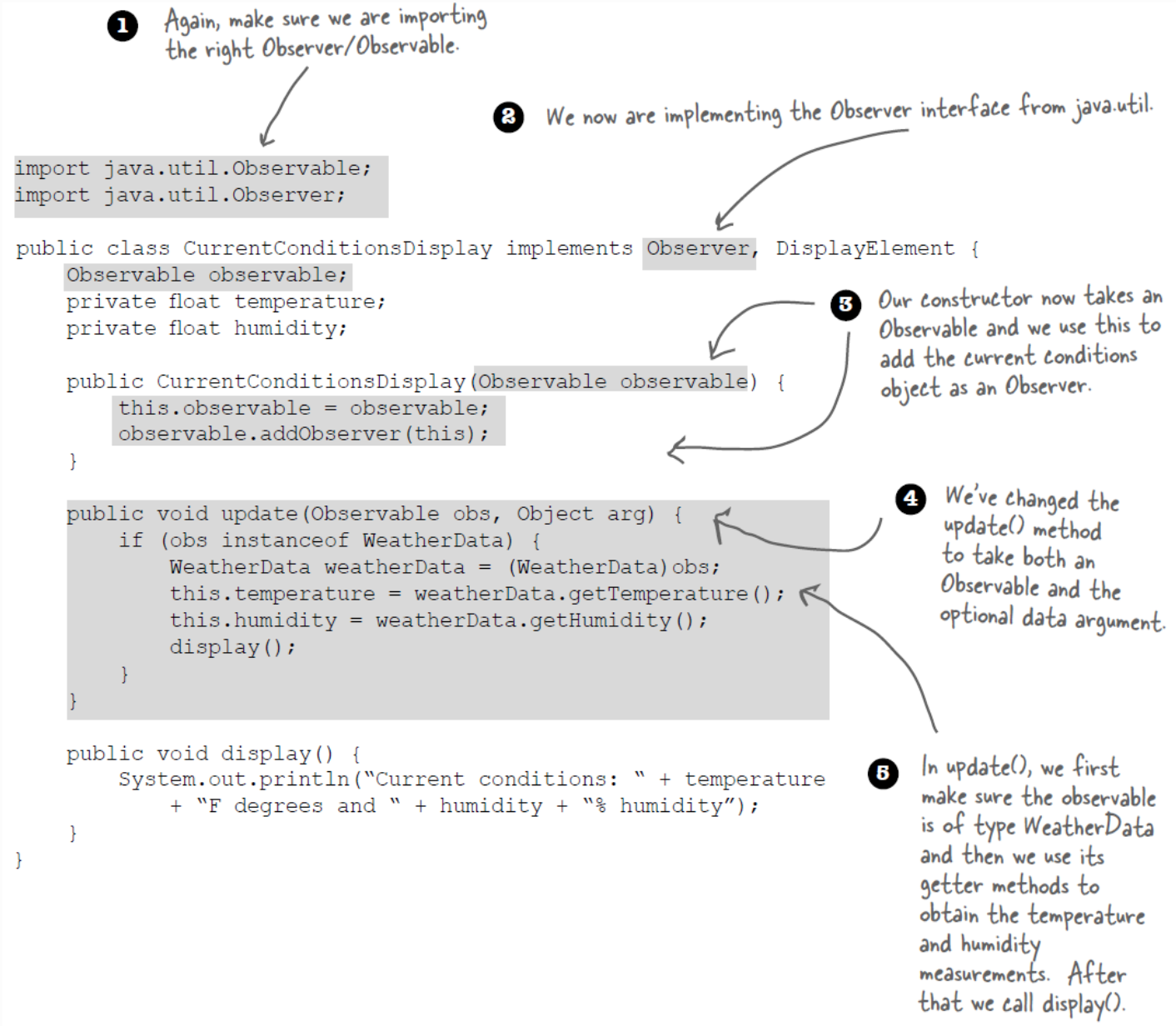
Wetterstation überarbeiten

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

    public float getHumidity() {
        return humidity;
    }

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

6

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.

Wetterstation überarbeiten

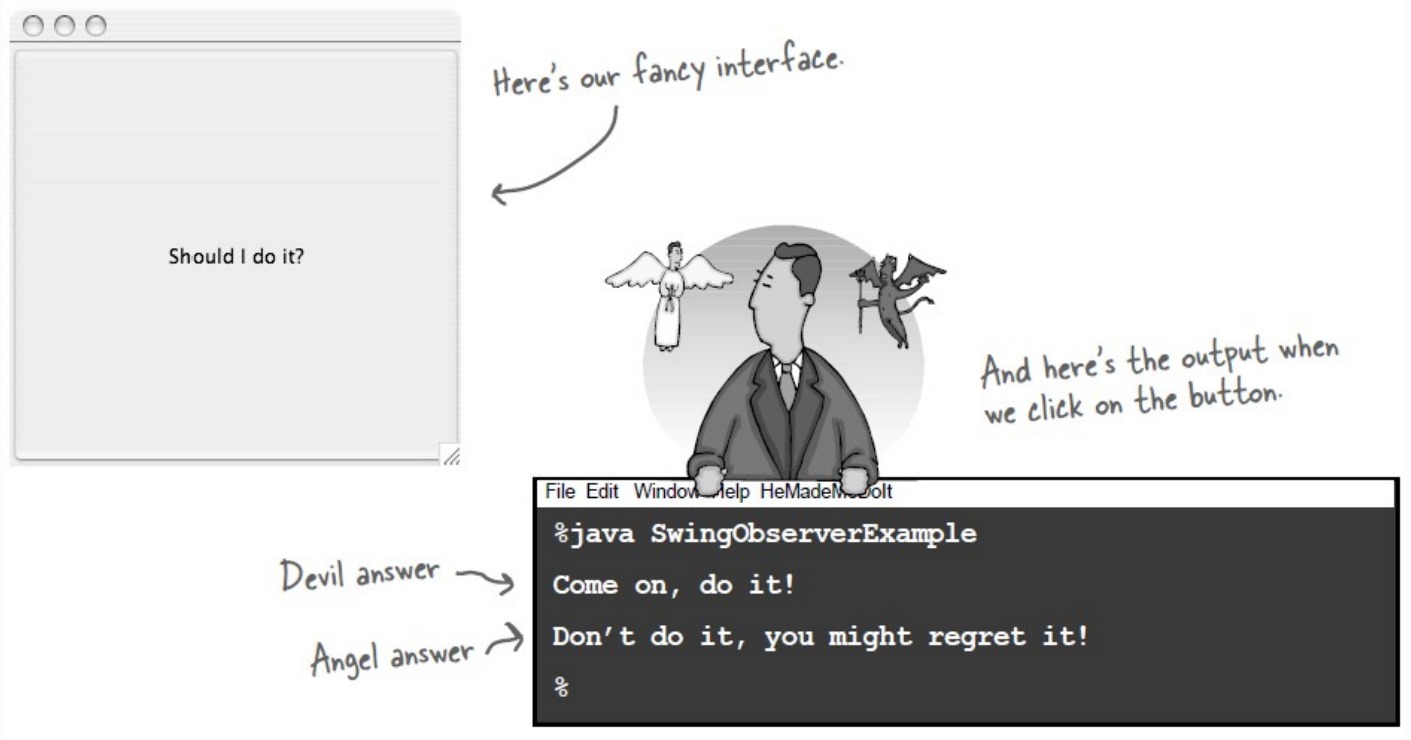



```
import java.util.Observable;
import java.util.Observer;

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");
    }
}
```

Die dunkle Seite von java.util.Observable

- Observable ist eine Klasse
  - Man muss sie erweitern (Wiederverwendbarkeit?)
  - Keine eigene Implementierung
- Observable schützt entscheidende Methoden
  - protected setChanged() nur in Unterklasse aufrufbar
  - Komposition ist aber der Vererbung vorzuziehen!!!

Engel- und Teufel-Listener



Engel- und Teufel-Listener

```
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!");
        }
    }
}
```

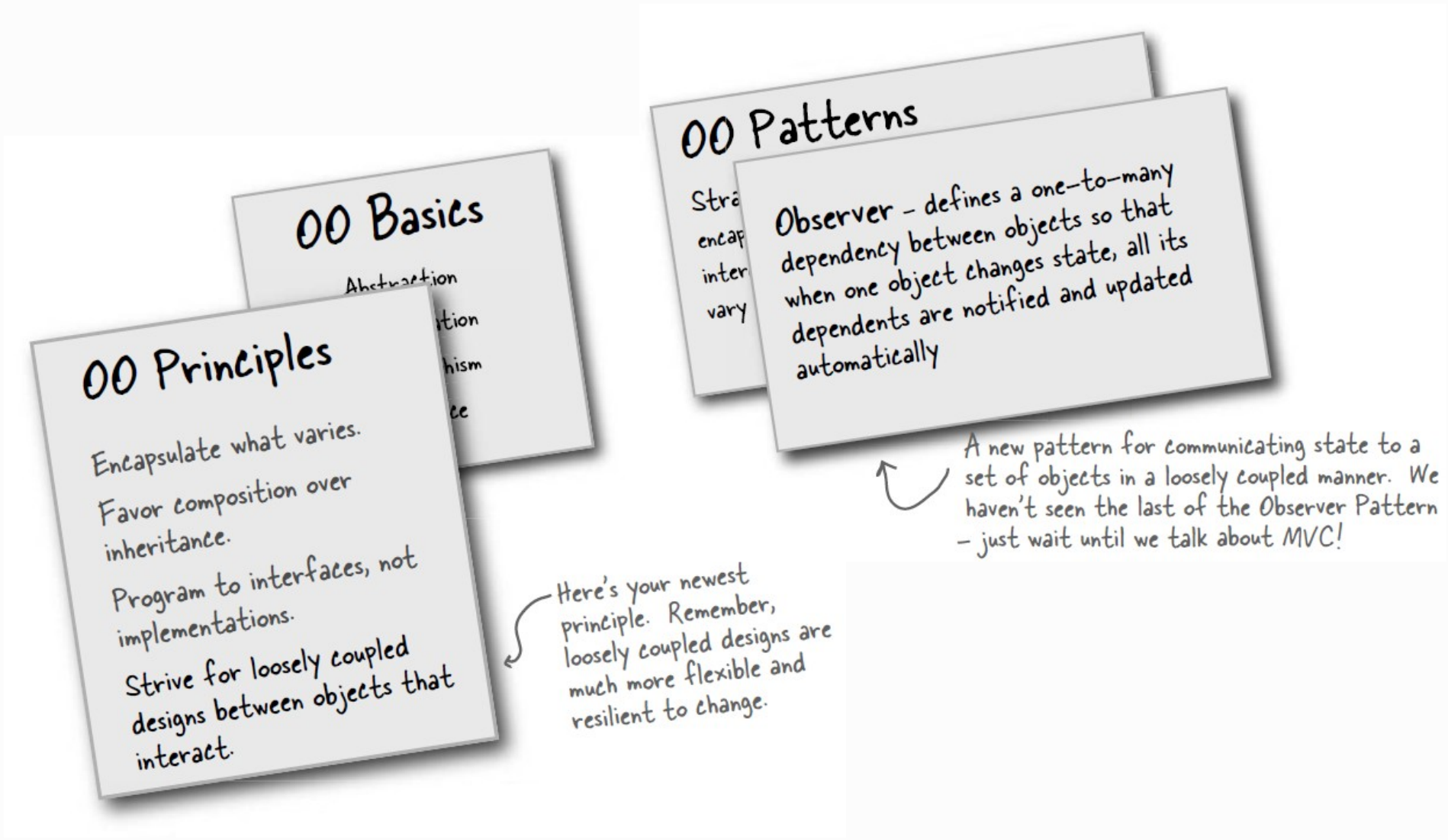
Simple Swing application that just creates a frame and throws a button in it.

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.

Neue Werkzeuge in der Design-Toolbox



to be continued ...