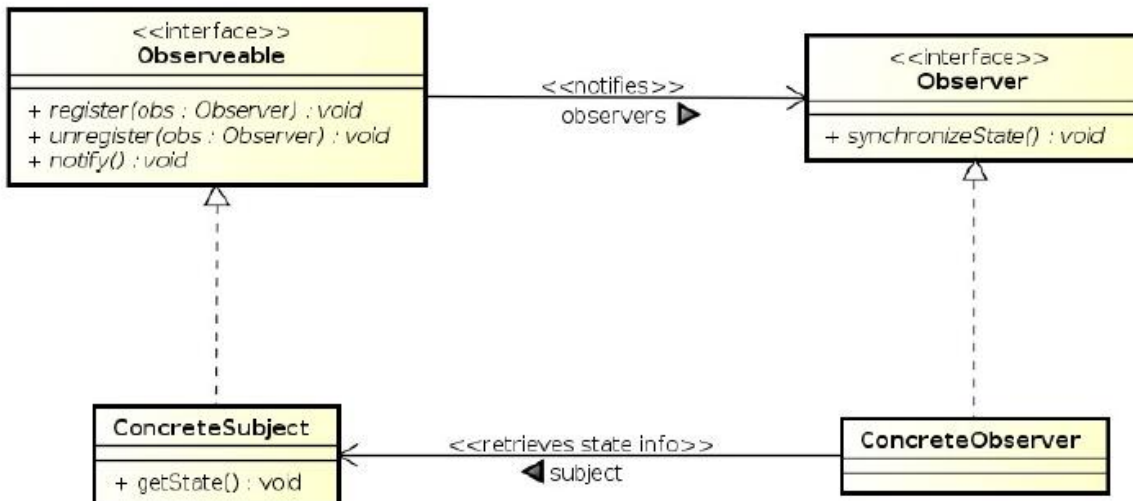


Observer Pattern

- 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.



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

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

2 We are now subclassing Observable.

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

    public WeatherData() { }

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

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

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

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

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.

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()`.

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.

Strebe für lose gekoppelte Designs zwischen Objekten, die interagieren.

- Locker gebunden = Interaktion mit wenig Detailwissen
- Lose gekoppelte Designs ermöglichen das Bauen von flexiblen OO Systemen, die die Änderung bewältigen können, weil sie die wechselseitige Abhängigkeit minimieren.
- **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 sind unabhängig verwendbar
- **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

Frage 2

Man könnte verwenden:

- `BlockingQueue`
- `ConcurrentMap`
- `ConcurrentNavigationMap`

Wenn `ArrayLists` verwendet werden:

Zugriffsfehler können dabei auftreten, wenn mehrere `Threads` auf einen Datensatz zugreifen wollen.

Frage 3

1) `log` auf `synchronized` setzen:

```
Public static synchronized void log() {}
```

2) Konstruktor von `logger` auf `private` setzen.

Eine Instanz von `logger` in `logger` speichern (statisch)

Eine Statische Methode erstellen um Instanz zu holen: `getInstanz(): logger`

Frage 4

Nach welcher Zeile wäre der richtige Zeitpunkt, um einen neuen Thread zu erzeugen?

(Erweitert) Geben Sie den Java-Code an, den Sie hier einfügen würden, um diesen Thread zu erzeugen und zu starten!

Beispiel II

Betrachten Sie die folgenden Codezeilen in Java:

```
1. try (ServerSocket serverSocket = new ServerSocket(10101)) {
2.     while (true) {
3.         Socket socket = null;
4.         try {
5.             socket = serverSocket.accept();
6.             try (PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
7.                 BufferedReader in = new BufferedReader(
8.                     new InputStreamReader(socket.getInputStream()));) {
9.                 String answer;
10.                while ((answer = in.readLine()) != null) {
11.                    System.out.println(answer);
12.                }
13.                socket.close();
14.            } catch (IOException e) {
15.                e.printStackTrace();
16.            }
17.        } finally {
18.            if (socket != null)
19.                socket.close();
20.        }
21.    }
22. } catch (IOException e) {
23.     System.err.println("Could not listen!");
24.     System.exit(-1);
25. }
```

Nach Zeile 5

Thread tnew = new Thread(MyRunImpl(socket));

Tnew.start();

Frage 5

