Einführung Observer/Observable

Das Beobachter-Design-Pattern (Observer/Observable)

Wir wollen uns nun mit dem Observer-Pattern beschäftigen, welches seine Ursprünge in Smalltalk-80 hat. Dort ist es etwas erweitert unter dem Namen MVC (Model-View-Controller) bekannt, ein Kürzel, womit auch wir uns noch näher beschäftigen müssen, da dies ein ganz wesentliches Konzept bei der Programmierung grafischer Bedieneroberflächen mit Swing ist.
Stellen wir uns eine Party mit einer netten Gesellschaft vor. Hier finden sich zurückhaltende passive Gäste und aktive Erzähler. Die Zuhörer sind interessiert an den Gesprächen der Unterhalter. Da die Erzähler nun von den Zuhörern beobachtet werden, bekommen sie den Namen **Beobachtete**, auf Englisch auch **Observable** (beobachtbar) genannt. Die Erzähler jedoch interessieren sich überhaupt nicht dafür, wer ihnen zuhört. Sie machen keine Unterscheidung zwischen ihren Zuhörern, sie halten allerdings den Mund, wenn ihnen überhaupt keiner zuhört. Die Zuhörer reagieren auf Witze der Unterhalter und werden dadurch zum **Beobachter** (engl. **observer**).

Die Klasse Observable und die Schnittstelle Observer

Unser Beispiel mit den Erzählern und Zuhörern können wir auf Datenstrukturen übertragen. Die Datenstruktur lässt sich beobachten und wird zum Beobachteten. Sie wird in Java als Exemplar der Bibliotheksklasse Observable repräsentiert. Der Beobachter wird durch die Schnittstelle Observer abgedeckt und ist der, der informiert werden will, wenn sich die Datenstruktur ändert. Jedes Exemplar der Observable-Klasse informiert nun alle seine Horcher, wenn sich sein Zustand ändert. Denken wir wieder an unser ursprüngliches Beispiel mit der Visualisierung. Wenn wir nun zwei Sichten auf die Datenstruktur haben, etwa die Eingabemaske und ein Balkendiagramm, ist es der Datenstruktur egal, wer an den Änderungen interessiert ist. Ein anderes Beispiel: Die Datenstruktur enthält einen Wert, der durch einen Schieberegler und ein Textfeld angezeigt wird. Beide Bedienelemente wollen informiert werden, wenn sich dieser Wert ändert. Es gibt viele Beispiele für diese Konstellation, so dass die Java-Entwickler die Klasse Observable und die Schnittstelle Observer mit in die Standardbibliothek aufgenommen haben. Noch besser wäre die Entscheidung gewesen, die Funktionalität in die oberste Klasse `Object` aufzunehmen, so wie es Smalltalk macht.

Die Klasse Observable

Eine Klasse, deren Exemplare sich beobachten lassen, muss jede Änderung des Objektzustands nach außen hin mitteilen. Dazu bietet die Klasse `Observable` die Methoden `setChanged()` und `notifyObservers()` an.
Wir wollen nun das Party-Szenario in Java implementieren. Dazu schreiben wir eine Klasse `Witzeerzaehler`, deren Objekte einen Witz erzählen können, mit `setChanged()` auf eine Änderung ihres Zustands aufmerksam machen und dann mit `notifyObservers()` die Zuhörer mit dem Witz in Form einer Zeichenkette versorgen.

```
class Witzeerzaehler extends Observable {
   public void erzähleWitz(String witz) {
         setChanged();
         notifyObservers(witz);
   }
}
```

`setChanged()` setzt intern ein Flag, das von `notifyObservers()` abgefragt wird. Nach dem Aufruf von `notifyObservers()` wird dieses Flag wieder gelöscht. Dies kann auch manuell mit `clearChanged()` geschehen. `notifyObservers()` sendet nur dann eine Benachrichtigung an die Zuhörer, wenn auch das Flag gesetzt ist. So kommen folgende Programmzeilen häufig zusammen vor, da sie das Flag setzen und alle Zuhörer informieren.

```
setChanged();             // Eine Änderung ist aufgetreten
notifyObservers(Object);  // Informiere Observer über Änderung
```

Die `notifyObservers()`-Methode existiert auch ohne extra Parameter. Sie entspricht einem `notifyObservers(null)`. Mit der Methode `hasChanged()` können wir herausfinden, ob das Flag der Änderung gesetzt ist.
Interessierte Beobachter müssen sich am `Observerable`-Objekt mit der Methode `addObserver(Observer)` anmelden. Dabei sind aber nicht beliebige Objekte als Beobachter erlaubt, sondern nur solche, die die Schnittstelle `Observer` implementieren. Sie können sich mit `deleteObserver(Observer)` wieder abmelden. Die Anzahl der angemeldeten Observer bekommen wir mit `countObservers()`. Leider ist die Namensgebung etwas unglücklich, da Klassen mit der Endung »able eigentlich immer Schnittstellen sein sollen. Genau das ist hier aber nicht der Fall. Der Name `Observer` bezeichnet überraschenderweise eine Schnittstelle, und hinter dem Namen `Observable` verbirgt sich eine echte Klasse

Die Schnittstelle Observer

Das aktive Objekt, der Sender der Nachrichten, ist ein Exemplar der Klasse `Observable`, das Benachrichtigungen an angemeldete Objekte schickt. Das aktive Objekt informiert alle zuhörenden Objekte, die dazu die Schnittstelle `Observer` implementieren müssen. Damit versprechen sie, die Methode `update(Observable, Object)` zu implementieren, die bei Änderungen vom `Observable`-Objekt aufgerufen wird.
Die Schnittstelle `Observer` besteht im Übrigen nur aus dieser einen Methode. Der zweite Parameter ist genau die Nachricht, die mit `notifyObservers()` verschickt wurde. Bei der parameterlosen Variante `notifyObservers()` ist das Argument `null`. Somit können wir für die Party auch die Zuhörer implementieren.

```
class Zuhoerer implements Observer {
   private String name;
   Zuhoerer(String name) {
         this.name = name;
   }
   public void update(Observable o, Object obj) {
         System.out.println(name + " lacht über " + obj);
   }
}
```

Da auf einer echten Party die Zuhörer und Erzähler[1] nicht fehlen dürfen, baut die dritte Klasse `Party` nun echte Stimmung auf

```
import java.util.*;

public class Party {
   public static void main(String args[]) {
         Zuhoerer achim = new Zuhoerer("Achim");
         Zuhoerer michael = new Zuhoerer("Michael");
         Witzeerzaehler ulli = new Witzeerzaehler();
         ulli.addObserver(achim);
         ulli.erzähleWitz(
               "Sorry, aber du siehst so aus, wie ich " +
               "mich fühle.");
         ulli.erzähleWitz(
               "Eine Null kann ein bestehendes Problem " +
               "verzehnfachen.");
         ulli.addObserver(michael);
         ulli.erzähleWitz(
               "Wer zuletzt lacht, hat es nicht eher " +
               "begriffen.");
         ulli.erzähleWitz("Wer zuletzt lacht, stirbt
               wenigstens " + "fröhlich.");
         ulli.deleteObserver(achim);
         ulli.erzähleWitz(
               "Unsere Luft hat einen Vorteil: Man " + "sieht,
               was man einatmet.");
   }
}
```

Wir melden zwei Zuhörer nacheinander an und einen wieder ab. Dann ist die Ausgabe:

- Achim lacht über Sorry, aber du siehst so aus, wie ich mich fühle.
- Achim lacht über Eine Null kann ein bestehendes Problem verzehnfachen.
- Michael lacht über Wer zuletzt lacht, hat es nicht eher begriffen.
- Achim lacht über Wer zuletzt lacht, hat es nicht eher begriffen.
- Michael lacht über Wer zuletzt lacht, stirbt wenigstens fröhlich.
- Achim lacht über Wer zuletzt lacht, stirbt wenigstens fröhlich.
- Michael lacht über Unsere Luft hat einen Vorteil: Man sieht, was man einatmet.

Referenzen

[JavaInsel]	Java ist auch eine Insel (2. Aufl.) von Christian Ullenboom Programmieren für die Java 2-Plattform in der Version 1.4 Kapitel 11 Datenstrukturen und Algorithmen http://www.galileocomputing.de/openbook/javainsel3/javainsel_110012.htm