# Das Observer Pattern

## Ziel, Inhalt

- Das letzte mal angekündigt, diesmal kommts. Ein Objekt verwaltet Daten, die von verschiedenen anderen Objekten geändert werden können und angezeigt werden. Diese Objekte müssen häufig über Änderungen in den Daten informiert werden.

# Das Observer Pattern

## Einführung

**Problemstellung**

Eine gute Software-Architektur führt dazu, dass gewisse Daten an einem Ort gesammelt werden und für andere Objekte zur Verfügung stehen. Das kann ein Objekt sein, dass in einem Container Messwerte speichert, oder ein Objekt, dass Personendaten aus einer Datenbank beziehen kann. Es können genauso die Daten in den Zellen einer Tabellenkalkulation sein. Solche Daten werden im Allgemeinen für den Benutzer dargestellt, manchmal auch auf verschiedene Arten gleichzeitig. So kann man Daten in einer Tabellenkalkulation einerseits in den Zellen betrachten, andererseits können die Werte dazu dienen eine Grafik anzuzeigen. Der Benutzer erwartet bei einer Änderung der Daten in der Tabellenzelle, dass die Grafik nachgeführt wird. Diese Problemstellung findet sich an vielen Orten wieder. Irgendwo befinden sich Daten, die von mehreren Programmteilen angezeigt werden und auch von verschiedenen Programmteilen aus geändert werden können.

**Begriffe**

Die Daten werden in der Literatur als „Subject“ bezeichnet, im Gegensatz zu den Beobachtern der Daten, die man als „Observer“ bezeichnet. Ein Beobachter hat also ein Subjekt, das er beobachtet. Eine andere Bezeichnung für diese Rollenverteilung ist auch „Publisher/Subscriber“. Die Daten oder der Datencontainer ist der „Publisher“, also derjenige der Daten veröffentlicht. Der Subscriber ist der Abonnent für die Änderungen an den Daten.
Eine weitere Bezeichnung für dieses Pattern ist Model/View/Controller. Das Model ist die Datenbasis, also das „Subject“. Für diese Daten interessiert sich ein View-Objekt, also etwas, das die Daten zur Ansicht aufbereitet, wobei Ansicht nicht unbedingt ein Fenster sein muss. Hier wird auch ein Controller eingeführt. Ein Controller ist ein Objekt, das die Daten ändern kann. Manchmal ist eine View auch ein Controller, denn der Benutzer kann über einen Dialog, der die Daten anzeigt, diese Daten auch ändern.
Daraus leitet sich auch die „Document/View“- Architektur ab, die beispielsweise von der MFC (Microsoft Foundation Classes) verwendet wird und nichts anderes als eine Umsetzung des Observer-Patterns ist. Hier ist die Idee, dass die Daten das Dokument bilden, und die Views verschiedene Ansichten auf die Daten darstellen. Über diese Ansichten lassen sich diese Daten auch ändern.

### Mechanik

Der Observer kennt das Subject, das heisst meistens, dass im Code des Observers ein #include vorkommt, das die Klasse des Subject bekannt macht. Der Observer ruft eine Methode des Subject auf, mit dem er sich für Änderungsbenachrichtigungen einträgt, er abonniert (subscribe) sich also für Mitteilungen über Änderungen. Der Publisher/Das Subject merkt sich den Observer und ruft eine Methode auf um diesen zu benachrichtigen wenn sich etwas an seinen Daten ändert.

## Implementation

Es gibt natürlich viele Möglichkeiten dieses Pattern zu implementieren und je nach Anwendung kann man das Pattern anpassen. Einige Dinge sollten wir aber immer beachten.

### Lose Kopplung durch abstrakte Interfaces

Die Klasse mit den Daten, die für andere Objekte bereitstehen wird im Laufe einer Entwicklung vermutlich mehrere Änderungen erfahren. Genauso sind die Klassen, die als Observer in Frage kommen meist recht unterschiedlich. Für beide kann der Entwickler jedoch schon bald festlegen, wie die Methoden zum abonnieren aussehen, oder was der Publisher bei den Subscribern aufruft, wenn eine Änderung stattfindet.
Man kann also die Methoden mit grosser Sicherheit bald festlegen aber lässt die weitere Implementation der Klassen offen. Hier sind abstrakte Klassen sehr sauber und hilfreich. Man kann nicht direkt ein Objekt von einer abstrakten Klasse erzeugen, aber man die Schnittstelle zu einem Objekt, das von einer solchen Klasse ableitet trotzdem kennen. In unserem Fall führt es auch zu einer losen Kopplung zwischen den verschiedenen Klassen. Ein Implementationsänderung an einer Klasse führt im Allgemeinen nicht zu einer Änderung in einer anderen.
Hier ein Beispiel für eine Interface-Klasse

```
#ifndef INTERFACE_H
#define INTERFACE_H

class Interface
{
public:
    // virtueller Destruktor
    // da von dieser Klasse bestimmt
    // abgeleitet wird
    virtual ~Interface() {};

    // abstrakte Methode
    virtual void doIt() = 0;

};

#endif
```

Diese Klasse wird durch eine Header-Datei definiert, es braucht keine .cpp-Datei, da wir alle Methoden (hier *doIt*) rein virtuell definieren (also =0). Nur den Destruktor definieren wir auch, aber er tut nichts. Trotzdem ist es wichtig einen virtuellen Destruktor zu definieren, da ein Objekt dieser Klasse möglicherweise nur über ein Interface-Zeiger in einer Kollektion drin ist und auch über diesen Interface-Zeiger zerstört wird. Siehe zum Thema Vererbung und virtuelle Methoden auch den alten Stoff:
http://www.devmentor.ch/teaching/additional/01I/Semester3/Abend7/Abend7.pdf

### Das Interface für den Beobachter

Wir beginnen bei dieser abstrakten Klasse, da sie im Normalfall ziemlich einfach ist und nur eine Methode enthält.

```
class Observer
{
    public:
        virtual void update() = 0;
};
```

Mehr braucht es nicht um anzuzeigen, dass sich etwas geändert hat. Wir werden sehen, dass diese Methode erweitert werden kann, um dem Observer mehr Informationen zu geben.

### Das Interface für die Datenbasis/Subject

Ein solches Objekt muss einem potentiellen Beobachter die Möglichkeit bieten, sich für Änderungen zu abonnieren. Natürlich bietet ein solches Objekt Zugriff auf die Daten.

```
class Subject
{
    public:
        virtual ~Subject(){}
        virtual void addObserver(Observer* observer) = 0;
        virtual void removeObserver(Observer* observer) = 0;

        virtual const Data& getData() const = 0;
        virtual void setData(const Data& data) = 0;
};
```

## Übung Implementation

Versuchen wir nun diese abstrakten Klassen in einem einfachen Projekt zu implementieren.

### Neues Projekt und Interface - Klassen

Erzeuge ein Konsolenprojekt und erzeuge eine Headerdatei („Observer.h“), mit den Definitionen der beiden abstrakten Klassen *Observer* und *Subject*.

Du kannst selbstverständlich einen der anderen Begriffe wählen wie *Publisher/Subscriber*.

**Erzeuge eine neue Klasse *Database* oder *SubjectImpl*.**

Diese Klasse leitet von der abstrakten Basisklasse *Subject* ab. Die Klasse, die ich im Beispiel oben *Data* genannt habe kann auch ein einfacher *int* sein.

```
typedef int Data; // Neuer Typ Data entspricht einem int
```

Bei der Implementation brauchst du einen Container für die verschiedenen *Observer*-Objekte. Die Wahl eines solchen ist Dir überlassen. In diesen Container speicherst du jeweils die *Observer**, die sich mit einem Aufruf von *addObserver* eintragen. Bei *removeObserver* musst du diesen Zeiger wieder aus der Kollektion entfernen. Dabei gibt es Kollektionen, die eine *find*-Funktion anbieten. Für eine einfache Implementation rate ich Dir einen Blick auf die Kollektion *std::set* zu werfen.

**Erzeuge zwei oder drei Klassen, die von Observer ableiten**

Nun wollen wir also zwei oder drei verschiedene Klassen definieren, die etwas mit unseren Daten anfangen. Sie können diese Daten zum Beispiel einfach auf einer Zeile der Konsole ausgeben. Wenn sie das unterschiedlich tun, können wir uns vorstellen, dass das verschiedene Views sind.

**Erzeugen und verbinden der Subject und Observer-Objekte**

Ein main muss nun einige Objekte erzeugen und dafür sorgen, dass sich die verschiedenen Observer bei einem Subject eintragen.

## Weitere Varianten

Das Ziel ist es einige Spielarten des Observer-Patterns bereit zu haben, so dass wir je nach Problemstellung schnell eine passende bereit haben.