



Beobachter -Observer (Entwurfsmuster)

Das **Beobachter-Muster** ([englisch](#) *observer pattern*, auch *listener pattern*) ist ein [Entwurfsmuster](#) aus dem Bereich der [Softwareentwicklung](#). Es gehört zur Kategorie der [Verhaltensmuster](#) (engl. *behavioral patterns*) und dient der Weitergabe von Änderungen an einem [Objekt](#) an von diesem Objekt abhängige Strukturen.^[1] Das Muster ist eines der sogenannten GoF-Muster (*Gang of Four*).

Was ist das Observer Pattern (Beobachter-Entwurfsmuster)?

Das Observer Design Pattern, kurz Observer Pattern bzw. deutsch Beobachter-Entwurfsmuster, ist eine der beliebtesten Mustervorlagen für das Design von Computersoftware. Es gibt eine einheitliche Möglichkeit an die Hand, eine **Eins-zu-eins-Abhängigkeit zwischen zwei oder mehreren Objekten** zu definieren, um sämtliche **Änderungen** an einem bestimmten Objekt auf möglichst unkomplizierte und schnelle Weise **zu übermitteln**. Zu diesem Zweck können sich beliebige Objekte, die in diesem Fall als Observer bzw. Beobachter fungieren, bei einem anderen Objekt **registrieren**. Letzteres Objekt, das man in diesem Fall auch als Subjekt bezeichnet, **informiert** die registrierten Beobachter, sobald es sich verändert bzw. angepasst wird.

Das Observer Pattern zählt, wie eingangs bereits erwähnt, zu den 1994 in „Design Patterns: Elements of Reusable Object-Oriented Software“ veröffentlichten **GoF-Mustern**. Die über 20 beschriebenen Musterlösungen für das Softwaredesign spielen bis heute eine wichtige Rolle in der Konzeptionierung und Ausarbeitung von Computeranwendungen.

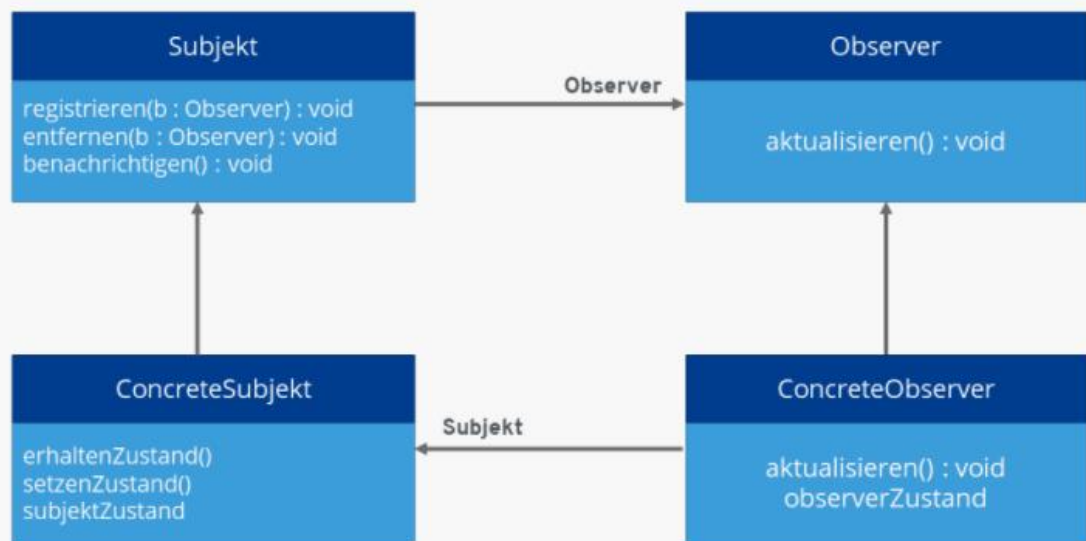
Zweck und Funktionsweise des Observer Patterns

Das Beobachter-Entwurfsmuster arbeitet mit zwei Typen von Akteuren: Auf der einen Seite steht das **Subjekt**, also das Objekt, dessen Status langfristig unter Beobachtung stehen soll. Auf der anderen Seite stehen die **beobachtenden Objekte** (Observer bzw. Beobachter), die über sämtliche Änderungen des Subjekts in Kenntnis gesetzt werden wollen.

Grafische Darstellung des Observer Patterns (UML-Diagramm)

Die Funktionsweise und der Nutzen von [Design Patterns](#) wie dem Observer Pattern sind für Außenstehende häufig nur schwer zu verstehen. Für ein leichteres Verständnis bietet sich daher eine **grafische Darstellung der Entwurfsmuster** an. Insbesondere die weit verbreitete Modellierungssprache [UML](#) ([Unified Modeling Language](#)) eignet sich hierfür, da sie die Zusammenhänge gleichermaßen für Anwender und Anwendungsexperten anschaulich und greifbar macht. Aus diesem Grund haben wir auch für die nachfolgende, abstrakte Darstellung des Observer Patterns auf UML als Darstellungssprache zurückgegriffen.

UML-Diagramm: Observer Design Pattern



Welche Vor- und Nachteile hat das Observer Design Pattern?

Das Observer Pattern in der Software-Entwicklung einzusetzen, kann sich in vielen Situationen auszahlen. Der große Vorteil, den das Konzept bietet, ist dabei **der hohe Unabhängigkeitsgrad** zwischen einem beobachteten Objekt (Subjekt) und den beobachtenden Objekten, die sich an dem aktuellen Zustand dieses Objekts orientieren. Das beobachtete Objekt muss beispielsweise keinerlei Informationen über seine Beobachter besitzen, da die Interaktion unabhängig über die Observer-Schnittstelle erledigt wird. Die **beobachtenden Objekte erhalten die Updates derweil automatisch**, wodurch ergebnislose Anfragen im Observer-Pattern-System (weil sich das Subjekt nicht geändert hat) gänzlich wegfallen.

Dass das Subjekt alle registrierten Beobachter automatisch über jegliche Änderungen informiert, ist jedoch nicht immer von Vorteil: Die **Änderungsinformationen** werden nämlich auch dann übermittelt, wenn sie für einen der Observer **irrelevant** sein sollten. Das wirkt sich insbesondere dann negativ aus, wenn die Zahl an registrierten Beobachtern sehr groß ist, da an dieser Stelle durch das Observer-Schema eine Menge Rechenzeit verschenkt werden kann. Ein weiteres Problem des Beobachter-Entwurfsmusters: Häufig ist im Quellcode des Subjekts nicht ersichtlich, welche Beobachter mit Informationen versorgt werden.

Observer Pattern: Wo wird es eingesetzt?

Das Observer Design Pattern ist insbesondere in Anwendungen gefragt, die auf Komponenten basieren, deren **Status**

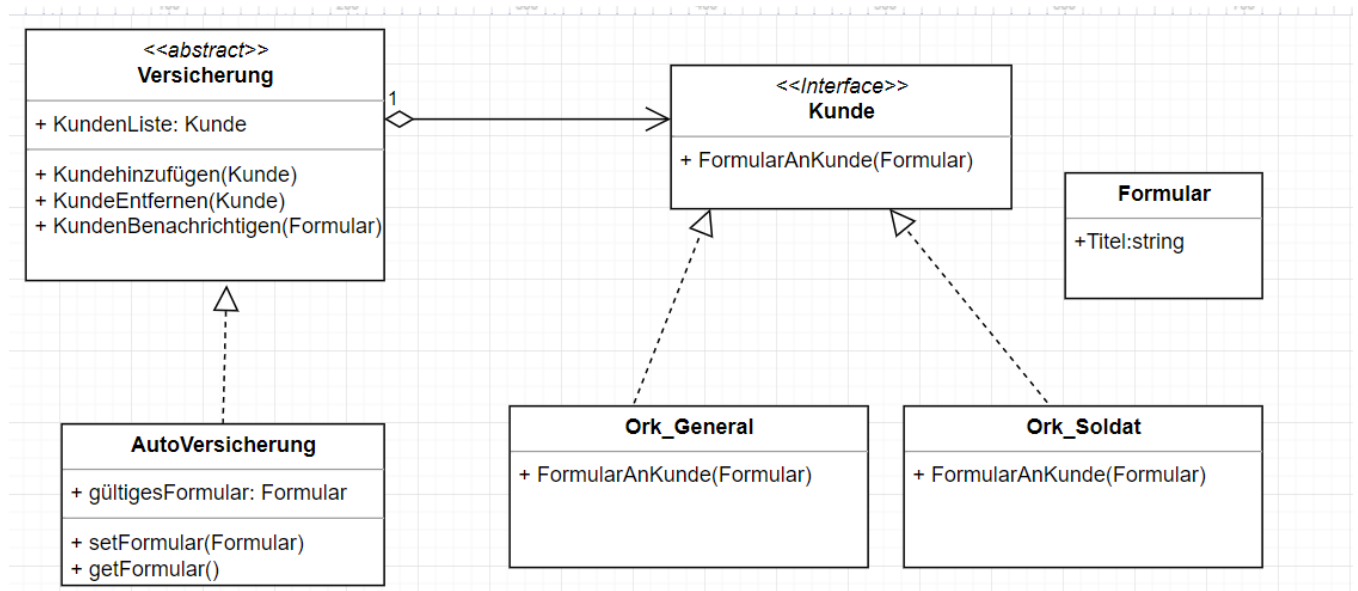
- einerseits **stark von anderen Komponenten beobachtet** wird,
- andererseits **regelmäßigen Änderungen** unterliegt.

Zu den typischen Anwendungsfällen zählen [GUIs \(Graphical User Interfaces\)](#), die Nutzern als einfach zu bedienende **Schnittstelle** für die Kommunikation mit einer Software dienen. Sobald sie Daten verändern, müssen diese in allen GUI-Komponenten aktualisiert werden – ein Szenario, das optimal durch die Subjekt-Beobachter-Struktur des Observer Patterns abgedeckt wird. Auch Programme, die mit zu visualisierenden Datensätzen (ob klassischen Tabellen oder grafischen Diagrammen) arbeiten, profitieren von der Ordnung durch das Entwurfsmuster.

Hinsichtlich der verwendeten **Programmiersprache** gibt es prinzipiell keine spezifischen Einschränkungen für das Observer Design Pattern. Wichtig ist lediglich, dass das **objektorientierte Paradigma** unterstützt

wird, damit eine Implementierung des Musters auch sinnvoll ist. Sprachen, in denen der Einsatz des Patterns sehr beliebt ist, sind u. a. C#, C++, Java, JavaScript, Python und PHP.

Observer Pattern: Beispiel für den Einsatz des Beobachter-Entwurfsmusters



```
public abstract class Versicherung
{
    private List<Kunde> K_liste = new List<Kunde>();

    public virtual void KundeHinzufügen(Kunde kunde)
    {
        K_liste.Add(kunde);
    }
    public virtual void KundeEntfernen(Kunde kunde)
    {
        K_liste.Remove(kunde);
    }
    public virtual void KundenBenachrichtigen(Formular formular)
    {
        foreach (Kunde kunde in K_liste)
            kunde.FormularAnKunde(formular);
    }
}
```

```
public class Auto_Versicherung : Versicherung
{
    public Formular gültigesFormular;
    public void setFormular(Formular formular)
    {
        gültigesFormular = formular;
        KundenBenachrichtigen(formular);
    }
    public Formular getFormular()
    {
        return gültigesFormular;
    }
}
```

```
public interface Kunde
{
    public void FormularAnKunde(Formular formular);
}
```

```
public class OrkGeneral : Kunde
{
    public void FormularAnKunde(Formular formular)
    {
        Console.WriteLine("\nSchicke Formular "+ formular.name + " an OrkGeneral nach Zürich !!!!");
    }
}
```

```
public class OrkSoldat : Kunde
{
    public void FormularAnKunde(Formular formular)
    {
        Console.WriteLine("\nSchicke Formular " + formular.name + " an OrkSoldat nach Luzern !!!!");
    }
}
```

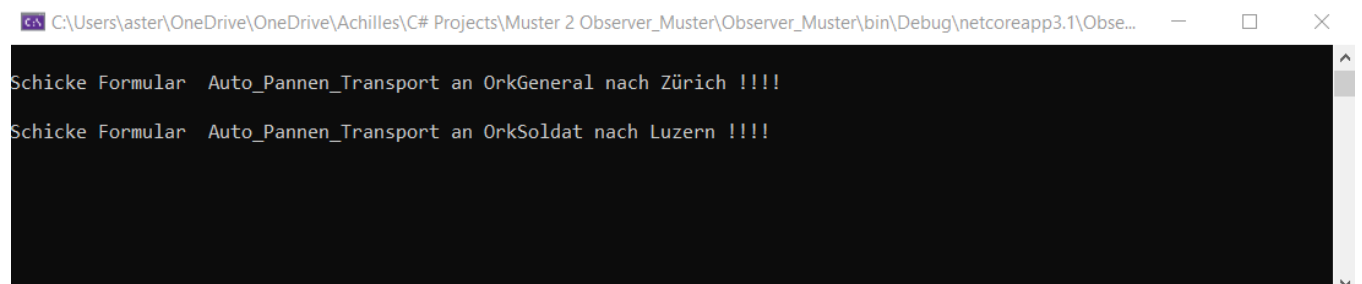
```
public class Formular
{
    public string name { get; }
    public Formular (string Formularname )
    {
        name = Formularname;
    }
}
```

```
static void Main(string[] args)
{
    Auto_Versicherung Premium = new Auto_Versicherung();
    Kunde Ork_General = new OrkGeneral();
    Kunde Ork_Soldat = new OrkSoldat();

    Premium.KundeHinzufügen(Ork_General);
    Premium.KundeHinzufügen(Ork_Soldat);

    Formular New_Formular = new Formular(" Auto_Pannen_Transport");
    Premium.setFormular(New_Formular);

    Console.WriteLine("Hello World!");
}
```



```
C:\Users\aster\OneDrive\OneDrive\Achilles\C# Projects\Muster 2 Observer_Muster\Observer_Muster\bin\Debug\netcoreapp3.1\Obse...
Schicke Formular Auto_Pannen_Transport an OrkGeneral nach Zürich !!!!
Schicke Formular Auto_Pannen_Transport an OrkSoldat nach Luzern !!!!
```