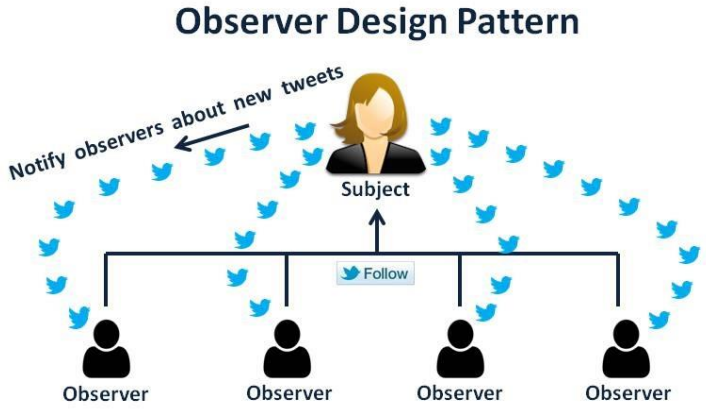
Observer

1 Observer

Das Designpattern Observer bzw. Beobachter zählt zu den **Verhaltensmustern**. Es ist das wohl am häufigsten anzutreffende Pattern.
Es geht im Wesentlichen darum, dass Objekte miteinander kommunizieren können, dass also bei der Änderung des Zustands eines Objekts die anderen Objekte diese Änderung bemerken und darauf reagieren können.
Um die Aufgabenstellung zu lösen gäbe es mehrere Möglichkeiten:

1. **Push**: das Auslöserobjekt hat auf jedes andere Objekt eine Referenz und ruft direkt eine Methode dieser Objekte auf
2. **Pull/Polling**: die interessierten Objekte haben eine Referenz auf das Auslöserobjekt und holen sich die Änderung in bestimmten Zeitabständen.
3. **Publish + Subscibe**: die interessierten Objekte registrieren sich (subscribe) beim Auslöserobjekt. Ändert sich dann der Zustand informiert das Auslöserobjekt die anderen Objekte (publish).

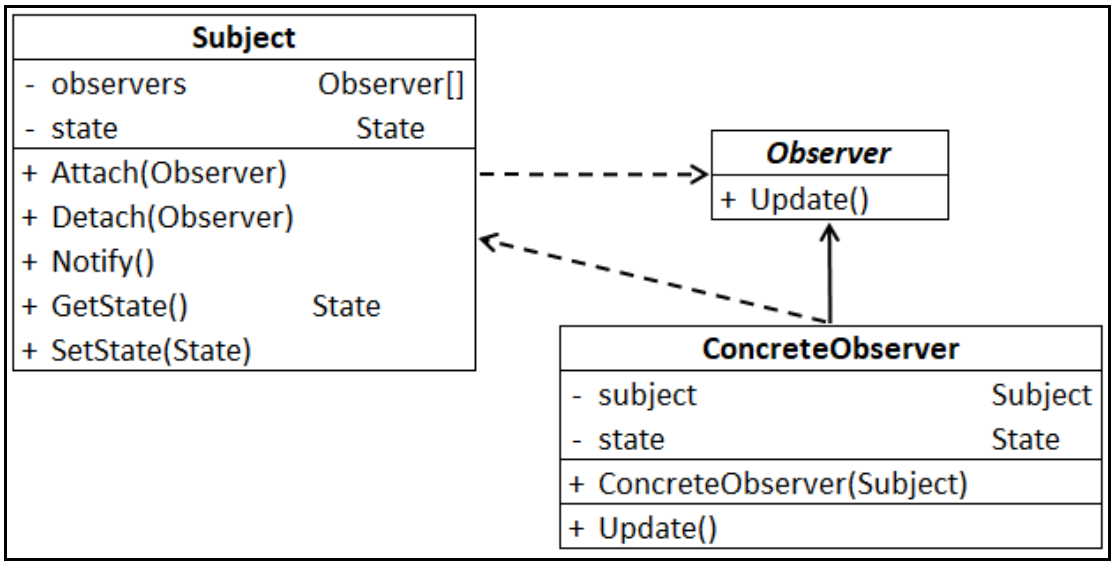
Das Observerpattern implementiert im Wesentlichen die Variante Publish+Subscribe. D.h. der Auslöser kennt seine Interessenten nicht und weiß auch nicht, ob es überhaupt welche gibt.



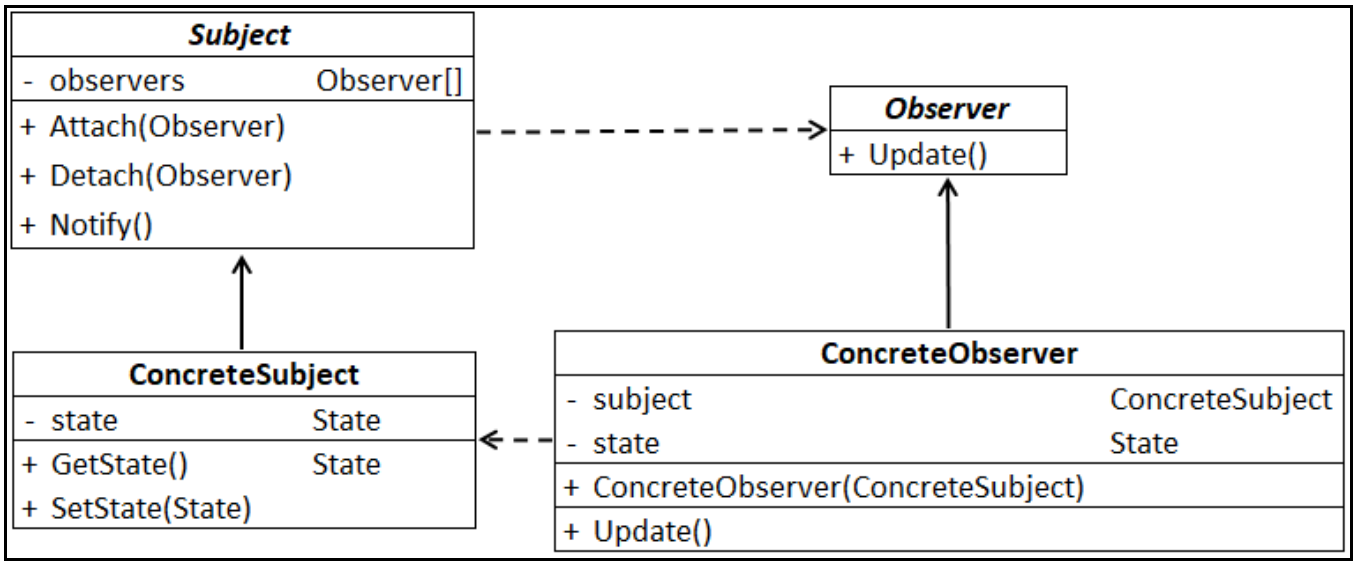
(Bild-Quelle http://devman.pl/pl/techniki/wzorce-projektowe-obserwatorobserver/)

1.1 UML

Einfach:



Mit konkretem Subject:



Hinweise:

- Die **`Subject`**-Basisklasse ist meist abstrakt, muss sie aber nicht unbedingt sein. Die Methoden Attach, Detach und Notify sind praktisch unabhängig von der Aufgabenstellung gleich zu programmieren.
- **`Observer`** ist üblicherweise ein Interface (daher in C# der Name üblicherweise **`IObserver`**).
- Das **`ConcreteSubject`** ist das oben erwähnte Auslöserobjekt. Es speichert den Zustand.
- Der **`ConcreteObserver`** meldet sich im Konstruktor beim Subject an (Attach).
- Es gibt meistens genau ein ConcreteSubject.
- Es kann beliebig viele Observer-Objekte geben.
- GetState()/SetState() wird in C# oft als Property programmiert.
- Das Subject weiß weder, ob es überhaupt Observer gibt, noch wer diese genau sind.
- Notify ruft von allen registrierten Observer-Objekten die Methode Update() auf.
- Notify ist üblicherweise protected
- Es bietet sich an, Subject und Observer in einer eigenen **Library** zusammenzufassen, weil diese oft ohne Änderung in verschiedenen Projekten einsetzbar sind.

1.2 Erkennungsmerkmale

An folgenden Merkmalen kann man erkennen, dass der Einsatz eines Observer:

- Man möchte auf Änderungen eines Objekts reagieren, wobei man nicht weiß, zu welchem Zeitpunkt diese auftreten können.
- Zustandsänderungen müssen sofort erkannt werden.
- Viele unterschiedliche Objekte zeigen gleichzeitig denselben Zustand an.
- Ein Objekt muss eine bestimmte Methode eines anderen Objekts aufrufen, ohne aber eine explizite Referenz darauf zu haben. So darf z.B. einem ViewModel in WPF kein Window, ListBox, … übergeben werden.

1.3 Beispiele

In folgenden Beispielen würde sich ein Observer anbieten:

- Chatter aller Art
- Events in C#. Dabei ist das Objekt vom Typ **`event`** das Subject des UML-Diagramms.
- MVC bzw. MVVM
- Angular: Die einzelnen Elemente im HTML sind Observer auf die verschiedenen Variablen in der dahinterliegenden Typescript-Klasse.
- RxJs
- SignalR

1.4 Vor-/Nachteile

1.4.1 Vorteile

- Man vermeidet enge Kopplung. Ändert sich der konkrete Beobachter, beeinflusst das das Subject nicht.
- Das Subject kennt den Beobachter nicht. Daher erhöht sich sehr stark die Wiederverwendbarkeit.
- Die Grundstruktur ist praktisch unabhängig von der Aufgabenstellung.

1.4.2 Nachteile

- Man darf nicht vergessen, sich vom Subject abzumelden.
- In einem System mit vielen Subjects bzw. Observern kann man durch Setzen eines neuen Zustands eine Aktualisierungslawine auslösen. Im schlimmsten Fall rufen sich Aktualisierungen rekursiv auf.