Entwurfsmuster

2021-03-23

Einleitung

Was sind Entwurfsmuster?

- Elemente wiederverwendbarer Software
- Lösungsansätze für typische Probleme
- kein fertiger Code → muss auf das konkrete Problem adaptiert werden

Warum sind Entwurfsmuster sinnvoll?

- kein "Neu-Erfinden" des Rads
- einfacheres Verständnis des Codes
- Reduktion der Komplexität
- Beginn einer höherwertigen Sprache unter Entwicklern

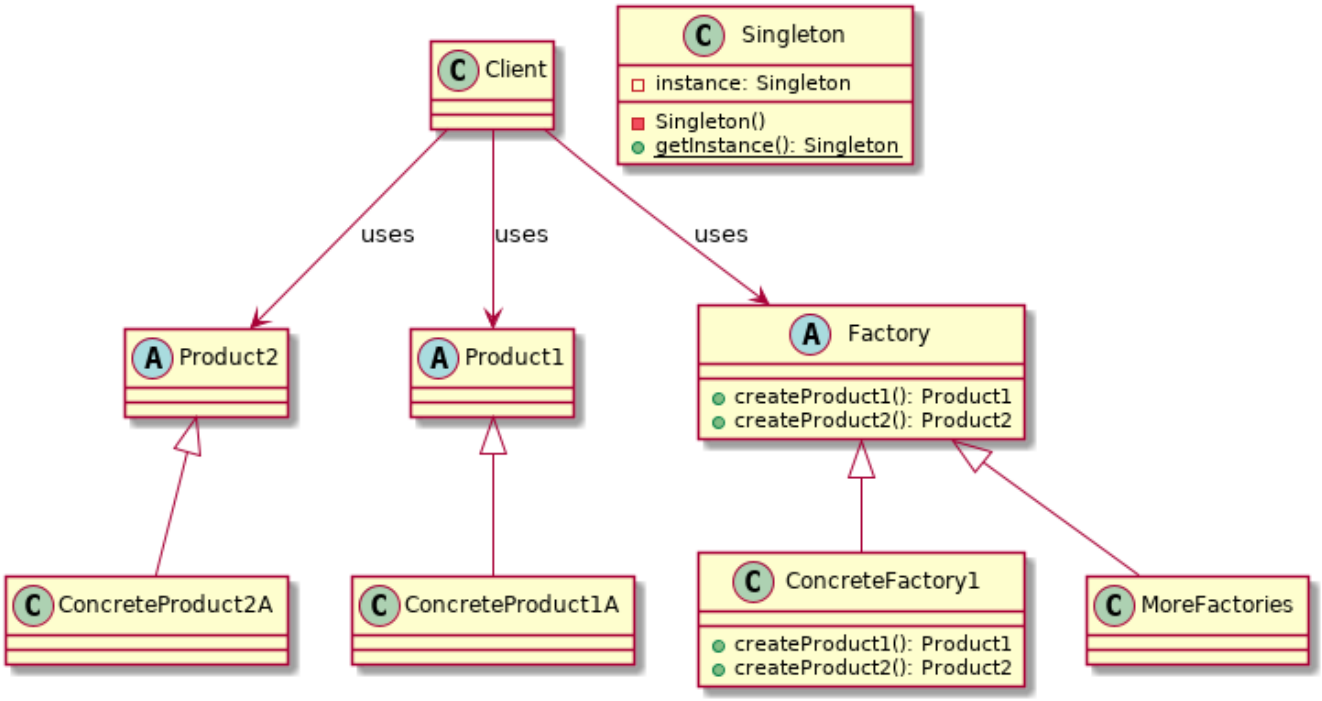
Kategorien von Entwufsmustern

- Erzeugungsmuster
- Strukturmuster
- Verhaltensmuster
- Nebenläufigkeitsmuster
  - typische Lösungen für Multithread-Programmierung

Kategorie: Erzeugungsmuster

- kümmern sich um die Erzeugung von Instanzen
- sinnvoll, wenn die Instanziierung komplex und/oder fehleranfällig ist
- Grundidee
  - Verstecken des konkreten Typs (in Zusammenhang mit Polymorphie)
  - Verstecken der Instanziierung

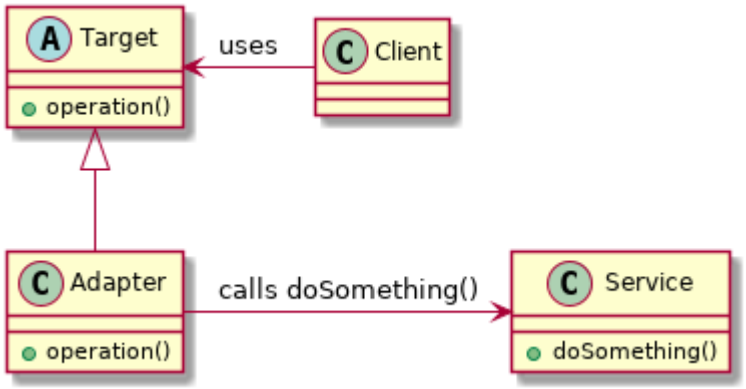
Erzeugungsmuster Beispiele

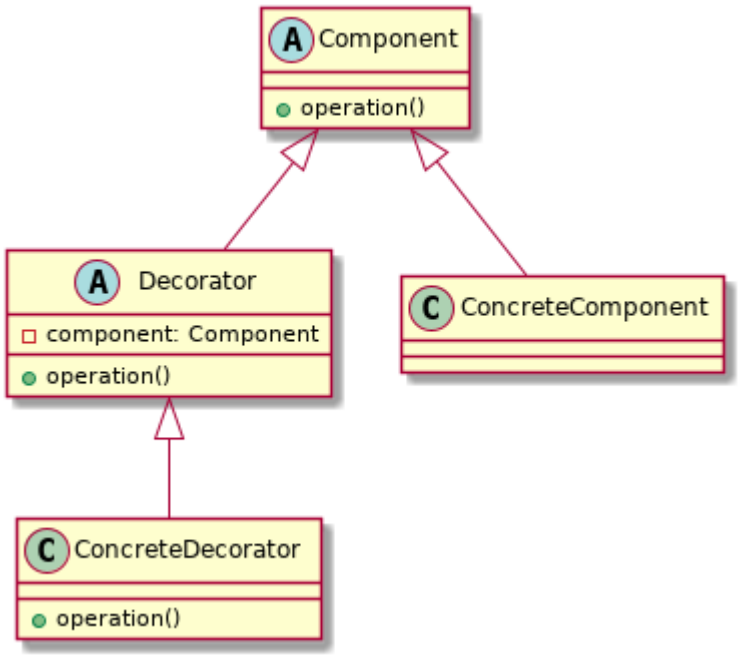


Kategorie: Strukturmuster

- Komposition von Klassen / Objekten
- übergeordnete Strukturen
- Hauptwerkzeuge
  - Vererbung zwischen Klassen
  - Assoziationen zu anderen Objekten

Strukturmuster Beispiele

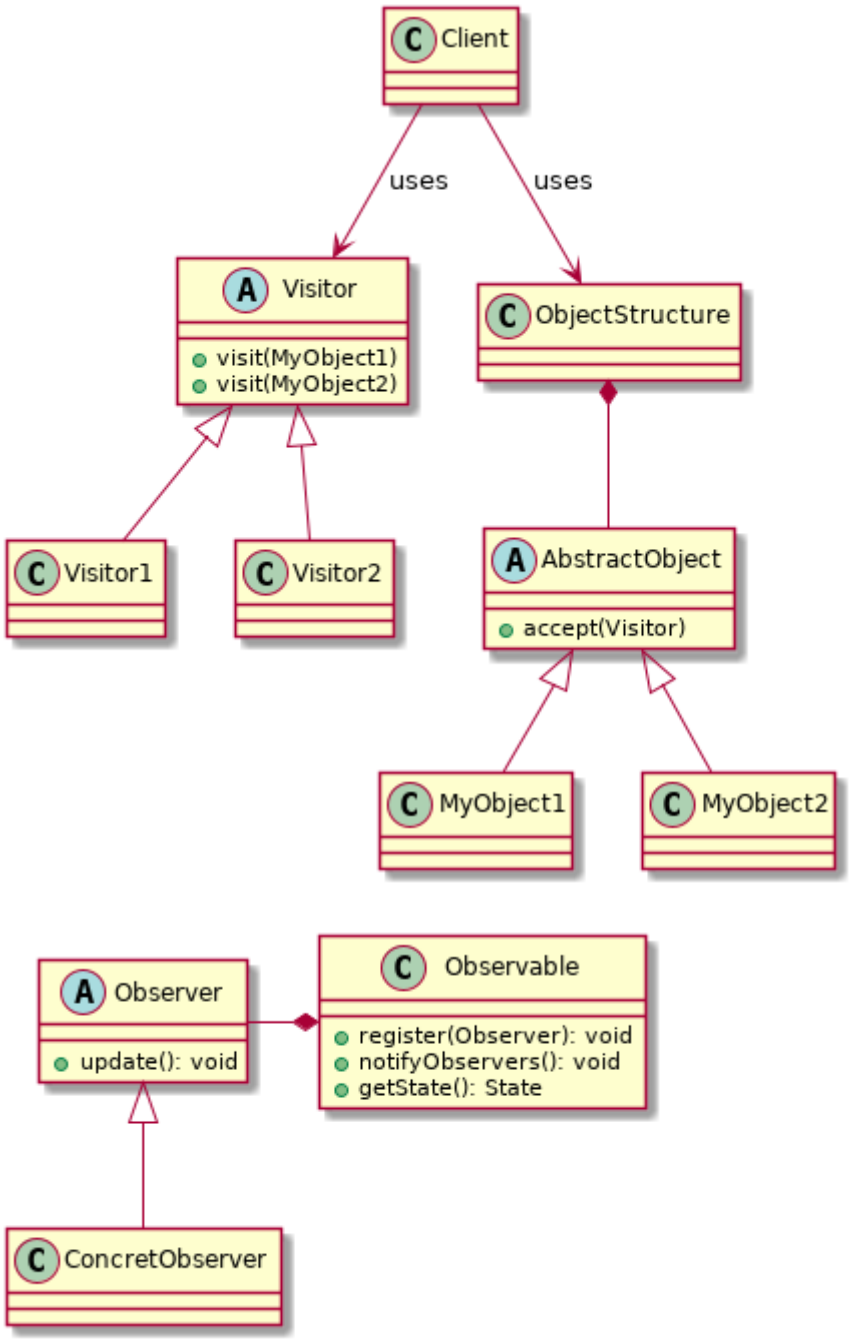




Kategorie: Verhaltensmuster

- Zusammenarbeit zwischen Objekten
- flexibleres Verhalten der Software
- Hauptwerkzeuge
  - polymorphe Methodenaufrufe
  - dynamisch änderbare Assoziation

Verhaltensmuster Beispiele



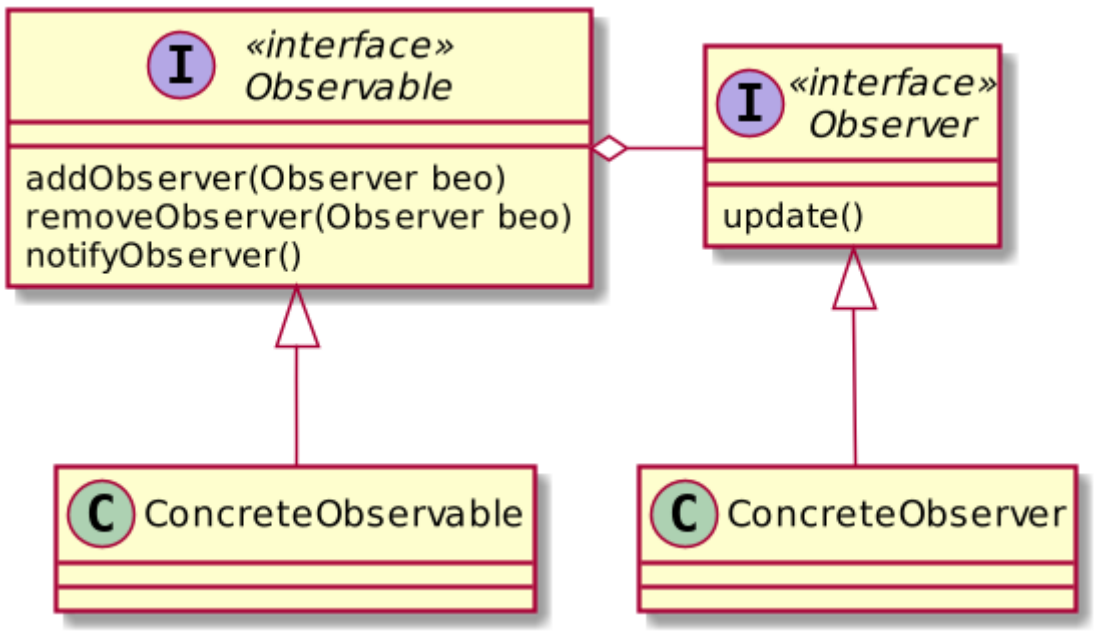
Beobachter (Observer)

auch **Listener** *oder* **Publish-Subscribe**

Steckbrief

- Art: Verhaltensmuster (behavioral pattern)
- Zweck
  - automatische Reaktion auf Zustandsänderung / Aktionen
  - 1:n Abhängigkeit zwischen Objekten ohne hohe Kopplung

UML



CODE BEISPIEL

Observer

```
interface Observer<T extends Observable> {
    void update(T observable);
}
```

Observable

```
interface Observable {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObserver();
}
```

ConcreteObserver

```
public class ConcreteObserver implements Observer<ConcreteObservable> {

    @Override
    public void update(ConcreteObservable observable) {
        System.out.println("Received udpate: " + observable.isState());
    }

}
```

ConcreteObservable

```
class ConcreteObservable implements Observable {
    private boolean state = false;
    private Set<Observer> observers = new HashSet<Observer>();

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObserver() {
        for (Observer observer : observers) {
           observer.update(this);
        }
    }

    public boolean isState() {
        return state;
    }

    public void setState(boolean state) {
        this.state = state;
        this.notifyObserver();
    }
}
```

- Wie kommt der Beobachter an den aktuellen Zustand?
  - `new ConcreteObserver(ConcreteObservable obs)`
  - `Observer::update(ConcreteObserverable obs)`
  - `Observer::update(T value)`
  - generell: **push** oder **pull**
  - abhängig vom Anwendungsfall

Bewertung: *pull*

- (+) *Observable* braucht keine Informationen über *Observer*
  - lose Kopplung
- (-) *Observer* muss ggf. selbst das Delta bestimmten
  - u.U. kostenintensiv

Bewertung: *push*

- (+) *Observable* informiert gezielt über spezifische Änderungen
- (-) *Observable* muss Informationen über *Observer* haben
- Wer löst die Methode `Observable::notifyObserver` aus?
  - *Observable* selbst
    - (+) keine Änderung wird übersehen
    - (-) jede Änderung löst aus
  - Benutzer von *Observable*
    - (+) Transaktionen möglich
    - (-) _notifyObserver kann leicht übersehen werden

Weitere _notifyObserver-Alternativen:

- Observable::setValue(T value, boolean notify)
  - notify = true, falls benachrichtigt werden soll
- Observable::setValueWithNotify(T value)
  - alternative *setter*-Methode, die benachrichtigt
- Observable::setValueWithoutNotify(T value)
  - normale *setter*-Methode benachrichtigt, diese extra Methode nicht

Probleme des Observers

Problem 1: Ungewollte Rekursion

1. ein *Observer* ändert den Zustand des *Observables*, nachdem er informiert wurde
2. ein anderer *Observer* empfängt dieses Änderungen und ändert auch das *Observable*
3. der erste *Observer* wird informiert und es geht von vorne los

CODE BEISPIEL

Problem 2: Unvorhersehbare Reihenfolge

- Aufruf-Reihenfolge der einzelnen Observer ist nicht garantiert (bei 1 Observable zu n Observers)
- Reihenfolge, wer den Observer zu erst aufruft, ist nicht garantiert (bei n Observables zu 1 Observer)

Beispiel: n Observers, 1 Observable

```
class Observable {

    private Set<Observer> observers = new HashSet<>();

    void addObserver(Observer observer) {
        observers.add(observer);
    }

    void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
```

```
interface Observer {
    void update();
}
```

```
class Observer_1 implements Observer {
    @Override
    public void update() {
        System.out.println("Observer_1 wurde informiert.");
    }
}
```

```
        Observable observable = new Observable();
        observable.addObserver(new Observer_1());
        observable.addObserver(new Observer_2());
        observable.addObserver(new Observer_3());
        observable.addObserver(new Observer_11());
        observable.addObserver(new Observer_12());
        observable.notifyObservers();
```

Beispiel Ausgabe:

```
Observer_12 wurde informiert.
Observer_3 wurde informiert.
Observer_2 wurde informiert.
Observer_11 wurde informiert.
Observer_1 wurde informiert.
```

Verbesserungsvorschläge

- eine Liste verwenden, die die Observer in der *Add*-Reihenfolge aufruft
  - (-) Thread-übergreifendes *add* ist immer noch ein Problem
  - (-) *remove* wird teurer, aber evtl nur ein Randfall
  - (-) *add* ist fehleranfälliger (gleicher Observer 2x hinzufügen)
- Gewichtung der Observer angeben (höhere Gewichtung = frühere Benachrichtigung)
  - `observable.register(observer, 1000);`

Beispiel Thread-übergreifendes *add*:

- Thread 1 fügt 3 Observer hinzu (*o1*, *o2*, *o3*)
- Thread 2 fügt aber zwischen *o1* und *o2* einen weiteren Observer hinzu (*o4*)
- Reihenfolge: *o1*, *o4*, *o2*, *o3*

Beispiel: Zeichenprogramm

1 Observer, n Observables

- Klick auf ein Element selektiert dieses
- Klick nicht auf das Element deselektiert dieses
- wenn nichts selektiert ist, ist der Mauszeiger ein Pfeil
- wenn ein Element selektiert ist, ist der Mauszeiger ein *X*

Implementierung mit Beobachter-Muster

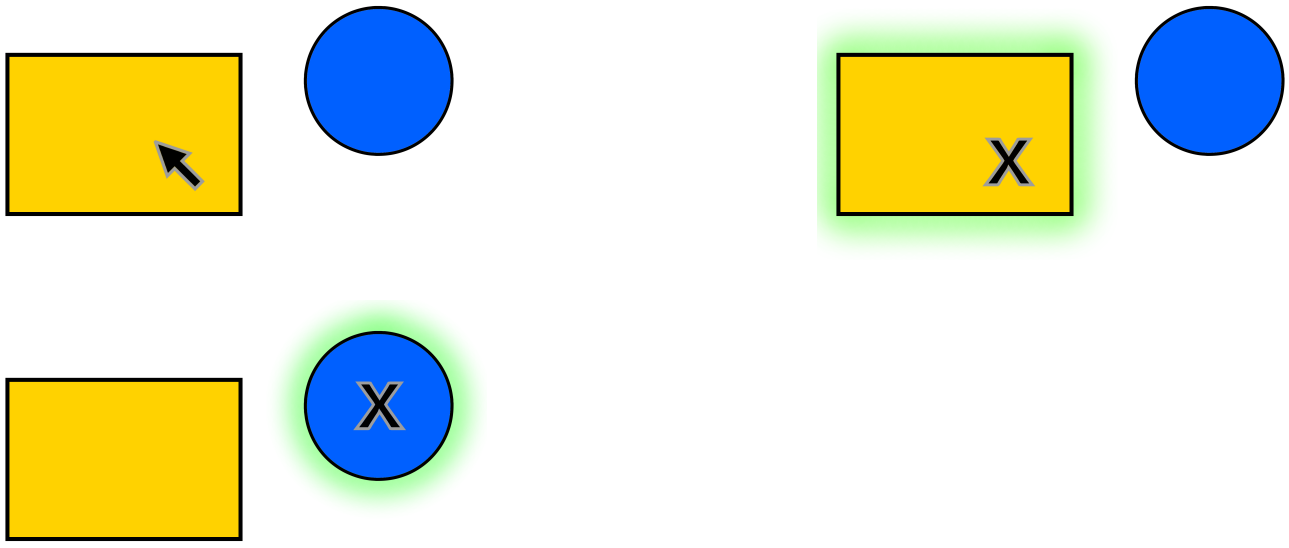
```
interface SelectionListener {
    void select(Element element);
    void deselect(Element element);
}
```

Interface-Implementierung

```
class CursorMonitor implements SelectionListener {

    private final Set<Element> elements = new HashSet<>();

    @Override
    public void select(Element element) {
        if (elements.add(element)) updateCursor();
    }

    @Override
    public void deselect(Element element) {
        if(elements.remove(element)) updateCursor();
    }

    private void updateCursor() {
        if (elements.isEmpty()) {
            System.out.println("Cursor ist jetzt ein Pfeil.");
            return;
        }
        System.out.println("Cursor ist jetzt ein X.");
    }
}
```

Testfall

1. nichts ist selektiert
2. Klick auf Rechteck
3. Klick auf Kreis



Wie verhält sich unsere Implementierung?

- es kann zum Flackern des Cursors kommen (jenachdem, welches Element zuerst feuert)

Auswertung

1. Alternative 1
   a. Rechteck wird selektiert: Cursor wird zum *X*
   b. Kreis wird selektiert: Cursor bleibt *X*
   c. Rechteck wird deselektiert: Cursor bleibt *X*
2. Alternative 2
   a. Rechteck wird selektiert: Cursor wird zum *X*
   b. Rechteck wird deselektiert: Cursor wird zum Pfeil
   c. Kreis wird selektiert: Cursor wird zum *X*

Kompositionalität ist nicht gegeben: das Programm kann sich unterschiedlich verhalten bei gleicher Verknüpfung der einzelnen Teile.

Verbesserungsvorschläge

- Timer einbauen, der bei Deselektierung das Update um ein paar ms verzögert
  - (-) Deselektierung ist verzögert
- Transaktion händisch einbauen
  - (-) fehleranfällig
  - (-) sehr aufwändig
- die Reihenfolge der Events garantieren
  - (-) sehr schwierig und umständlich
- Listener mit Prioritäten versehen (sehr umständlich)
  - geht nur bei mehreren Listener

Problem 3: Verpasste Ereignisse

Beispiel: Verbindungsaufbau

```
public class Connection {
  void addListener(Listener listener);
  void requestConnection();
  // ...

  interface Listener {
    void online(Session session);
  }
}
```

```
class Client implements Connection.Listener {
    private final Connection connection;

    Client(Connection connection) {
       this.connection = connection;
       connection.addListener(this);
    }

    void connect() {
       connection.requestConnection();
    }

    void online(Session session) {
       // ...
    }
}
```

```
Connection connection = new Connection();
// [...]
// andere Clients bekommen auch diese Connection (evtl. auch in anderen Threads)
// [...]
Client client = new Client(connection);
client.connect();
```

Verbindungsaufbau ist häufig asynchron, weil niemand so lange warten möchte.

- Connection: Verbindung zu Server
- Session: konkrete Session
- Client: das eigentliche Modul, das mit dem Server kommuniziert
  - registriert sich als Listener, um über online Ereignisse informiert zu werden

Auswertung

- Alternative 1
  - *Client* registriert sich als Listener
  - Verbindung wird aufgebaut
  - *Client* wird informiert
- Alternative 2
  - Verbindung wird aufgebaut (z.B. ausgelöst durch andere Clients)
  - *Client* registriert sich als Listener
  - *Client* wird **nicht** informiert (Verbindung schon da)

Fehlersuche sehr nervenraubend, denn alles funktioniert (es kommen keine Fehler), aber der Client hat scheinbar keine Verbindung.

Verbesserungsvorschläge

- bei Listener-Registrierung den aktuellen Zustand schicken
  - (-) ungewollte Seiteneffekte möglich

Problem 4: Ungewollte Benachrichtigungen

im Normalfall wird jeder Observer immer informiert, auch wenn ihn eine Änderung nicht interessiert

CODE-BEISPIEL

```
class Observable {

    private final Set<Observer> observers = new HashSet<>();
    private boolean state1;
    private boolean state2;

    void setState1(boolean state) {
        this.state1 = state;
        notifyObservers();
    }

    void setState2(boolean state) {
        this.state2 = state;
        notifyObservers();
    }

    boolean getState1() { return this.state1; }

    boolean getState2() { return this.state2; }

    void addObserver(Observer observer) { this.observers.add(observer); }

    void notifyObservers() { observers.forEach(it -> it.update(this)); }
}
```

```
class Observer_Simple implements Observer {
    @Override
    public void update(Observable observable) {
        System.out.println("Observable hat sich ge  ndert. State1 = "
            + observable.getState1());
    }
}
```

```
        Observable observable = new Observable();
        observable.addObserver(new Observer_Simple());

        observable.setState1(true);
        observable.setState2(true);
        observable.setState1(true);
```

Ausgabe

```
Observable hat sich ge  ndert. State1 = true
Observable hat sich ge  ndert. State1 = true
Observable hat sich ge  ndert. State1 = true
```

Mögliche Lösungen:

- Zustand im Observer merken
- gezielte Benachrichtigung durch Observable

Fix: Zustand im Observer

```
class Observer_Fixed implements Observer {

    private boolean oldState = false;

    @Override
    public void update(Observable observable) {
        if (oldState == observable.getState1())
            return;
        oldState = observable.getState1();
        System.out.println("Observable hat sich ge  ndert. State1 = "
            + oldState);
    }
}
```

Observer verpasst so doppeltes gleiches Setzen des *State1*.

Fix: Gezielte Benachrichtigung

```
class Observable {

    private final Set<Observer> observers = new HashSet<>();
    private boolean state1;
    private boolean state2;

    void setState1(boolean state) {
        this.state1 = state;
        notifyObserversState1();
    }

    void setState2(boolean state) {
        this.state2 = state;
        notifyObserversState2();
    }

    boolean getState1() { return this.state1; }

    boolean getState2() { return this.state2; }

    void addObserver(Observer observer) { this.observers.add(observer); }

    void notifyObserversState1() { observers.forEach(it -> it.onState1(this)); }
    void notifyObserversState2() { observers.forEach(it -> it.onState2(this)); }
}
```

```
interface Observer {
    void onState1(Observable observable);
    void onState2(Observable observable);
}
```

Observer könnte um einen Zustand erweitert werden (wie im vorherigen Fix) → doppelte Benachrichtigung beim Setzen des gleichen Wertes entfällt.

Problem 5: Threadsicherheit

- bei mehreren Observables / Observern über Threads verteilt

CODE-BEISPIEL

```
class Observable {
    private final Set<Observer> observers = new HashSet<>();

    void addObserver(Observer observer) {
        this.observers.add(observer);
    }

    void removeObserver(Observer observer) {
        this.observers.add(observer);
    }

    void notifyObservers() {
        for(Observer observer : observers) {
            Sleep.milliseconds(1000);
            observer.update();
        }
    }
}
```

```
class Observer {

    private final int number;

    Observer(int number) {
        this.number = number;
    }

    void update() {
        System.out.println("Observer " + number + " aktualisiert.");
    }
}
```

```
        Observable observable = new Observable();

        observable.addObserver(new Observer(1));
        observable.addObserver(new Observer(2));
        observable.addObserver(new Observer(3));
        observable.addObserver(new Observer(4));
        observable.addObserver(new Observer(5));

        CompletableFuture future1 = CompletableFuture.runAsync(() -> {
            observable.notifyObservers();
        });

        Sleep.milliseconds(300);

        CompletableFuture future2 = CompletableFuture.runAsync(() -> {
            observable.addObserver(new Observer(6));
            System.out.println("Observer 6 hinzugef  gt.");
        });

        CompletableFuture.allOf(future1, future2).get();
```

```
Observer 6 hinzugef  gt.
Observer 5 aktualisiert.
Exception in thread "main" java.util.concurrent.ExecutionException: java.util
.ConcurrentModificationException
    at java.base/java.util.concurrent.CompletableFuture.reportGet(CompletableFuture
.java:395)
    at java.base/java.util.concurrent.CompletableFuture.get(CompletableFuture.java
:1999)
    at patterns.oo.listener.threadsicherheit.ohne_synchronized.ThreadSicherheitMain
.main(ThreadSicherheitMain.java:31)
Caused by: java.util.ConcurrentModificationException
    at java.base/java.util.HashMap$HashIterator.nextNode(HashMap.java:1489)
    at java.base/java.util.HashMap$KeyIterator.next(HashMap.java:1512)
    at patterns.oo.listener.threadsicherheit.ohne_synchronized.Observable
.notifyObservers(Observable.java:21)
    at patterns.oo.listener.threadsicherheit.ohne_synchronized.ThreadSicherheitMain
.lambda$main$0(ThreadSicherheitMain.java:21)
```

Problemlösung?

synchronized benutzen

```
class Observable {

   private final Set<Observer> observers = new HashSet<>();

   synchronized void addObserver( Observer observer ) {
      this.observers.add(observer);
   }

   synchronized void removeObserver( Observer observer ) {
      this.observers.remove(observer);
   }

   synchronized void notifyObservers() {
      for ( Observer observer : observers ) {
         observer.update(this);
      }
   }
}
```

neues Problem: Deadlocks

```
class Observer {

    private final int number;

    Observer(int number) {
        this.number = number;
    }

    void update(Observable observable) {
        System.out.println(number + " updated.");
        observable.removeObserver(this);
    }
}
```

```
    public static void main(String[] args) {
        Observable observable = new Observable();

        observable.addObserver(new Observer(1));
        observable.addObserver(new Observer(2));
        observable.addObserver(new Observer(3));

        observable.notifyObservers();
    }
```

Der Deadlock ensteht, weil ein Observer sich selbst entfernt (und damit auf die *synchronized* Methode *removeObserver* zugreift. Zur gleichen Zeit läuft aber noch die *synchronized* Methode *notifyObservers*.

Immer, wenn **synchronized** benutzt wird, sollte man sich Gedanken über **Deadlocks** machen.

Lösung des Deadlock

```
class Observable {

   private final Set<Observer> observers = new HashSet<>();

   synchronized void addObserver( Observer observer ) {
      this.observers.add(observer);
   }

   synchronized void removeObserver( Observer observer ) {
      this.observers.remove(observer);
   }

   void notifyObservers() {
      for ( Observer observer : new HashSet<>(observers) ) {
         observer.update(this);
      }
   }
}
```

Thread-Probleme entstehen durch einen geteilten veränderbaren Zustand (*shared mutable state*). Hätte man diesen nicht - d.h., alle Threads sind unabhängig von anderen Threads - wäre man automatisch threadsicher.

Alternative Lösung: ConcurrentHashSet

```
class Observable {

    // basically a ConcurrentHashSet
    private final Set<Observer> observers = ConcurrentHashMap.newKeySet();

    void addObserver(Observer observer) {
        this.observers.add(observer);
    }

    void removeObserver(Observer observer) {
        this.observers.remove(observer);
    }

    void notifyObservers() {
        for (Observer observer : observers) {
            Sleep.milliseconds(1000);
            observer.update(this);
        }
    }
}
```

- (+): während dem Iterieren können weitere Observer hinzugefügt werden, über die ggf. auch iteriert wird

Problem 6: Zustandschaos

Beispiel: Verbindungsabbau

```
interface Listener {
  void online(Session session);
  void offline(Session session);
  void tearDown(Session session, TearDownCallback callback);

  interface TearDownCallback {
    void tornDown();
  }
}
```

Ereignisse	Zustände

• Verbindungsaufbau anfordern • Verbindungsabbau anfordern • Verbindung hergestellt • Verbindung fehlgeschlagen • TearDown Bestätigung der Clients	• ONLINE • OFFLINE • CONNECTING • TEARING_DOWN

5 (Zustände) * 4 (Ereignisse) = 20 mögliche Kombinationen → viele davon ungültig

Problem

- viele Ereignisse passen nicht zu allen Zuständen
- Lösung
  - Zustand merken
  - 20 Möglichkeiten abbilden (und dabei nichts vergessen)
- Randfälle
  - Verbindung ist schneller aufgebaut als die Methode beendet, die dies gestartet hat
  - TEAR_DOWN der Clients schneller als die Methode, die dies ausgelöst hat

Problem 7: Transaktionen

es sollen mehr als 1 Operation auf dem Observable ausgeführt werden, bevor die Observer benachrichtigt werden

CODE-BEISPIEL

```
        Observable observable = new Observable();
        observable.addObserver(new Observer());

        observable.setState1("Hello");
        observable.setState2("world!");
        observable.notifyObservers();
```

Bewertung

- (+) einfach
- (-) *notifyObservers* **muss** extern aufgerufen werden (und darf nicht vergessen werden)
- (-) nicht threadsicher
  - andere Threads können die State-Felder auch setzen und *notifyObservers* propagiert dann ungewollte Werte

Alternative

synchronized Transaktionsmethode

```
        Observable observable = new Observable();
        observable.addObserver(new Observer());

        observable.setState1AndState2("Hello", "World");
```

Bewertung

- (o) für wenige Felder *OK*
- (-) Aufrufer muss den Unterschied kennen und ggf. die richtige Methode aufrufen

Alternative

start / *endTransaction*

```
// als extra Methoden
observable.startTransaction();
// [hier werden Werte ge  ndert]
observable.endTransaction();

// als Lambda
observable.transaction(() => {
   // [hier werden Werte ge  ndert]
});
```

Bewertung

- (+) für viele Felder eine (relativ) saubere Lösung
- (-) kompliziert umzusetzen
  - Threadsicherheit; was passiert bei Änderungen ohne Transaktion; …

Problem 8: Vergessene Observer

- *removeListener* wird vergessen
- schwierig den Zeitpunkt für *removeListener* zu finden

Alternativen zum Observer

- *Mediator* als *ChangeManager*
  - zwischen *Observable* und *Observer* sitzt der *ChangeManager*, der letztlich über Änderungen informiert
- Message Queue

    - eher im applikationsübergreifenden Kontext
- Callback-Methoden (in der funktionalen Programmierung)

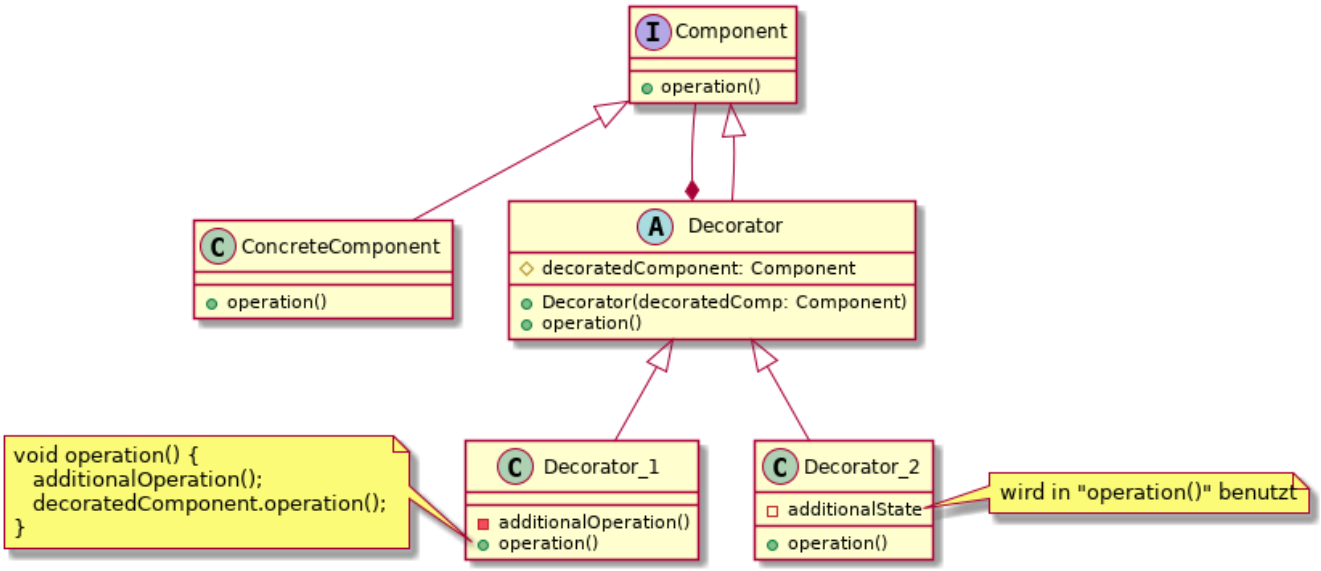
Decorator

auch ***Wrapper***

Steckbrief

- Art: Strukturmuster (structural pattern)
- Zweck
    - flexible Alternative zur Vererbung
    - dynamisches Hinzufügen von Funktionalität zur Laufzeit

Dekorierer: UML



Dekorierer: Beispiel

- Text soll durch den Nutzer zur Laufzeit formatiert werden können
  - *kursiv*
  - **fett**

Klassisch könnte die Klasse *Text* die Zustände halten:

```
class Text {
   boolean isItalic;
   boolean isBold;

   String unformattedText;

   String toHtml() {
      String html = "";
      if (isItalic) html += "<i>";
      if (isBold) html += "<b>";
      html += unformattedText;
      if (isBold) html += "</b>";
      if (isItalic) html += "</i>";
      return html;
   }
}
```

Problem: Bei neuen Formatierungsmöglichkeiten muss bestehende Logik angepasst werden.

```
interface Text {
   String toHtml();
}
```

```
class BasicText implements Text {
   String unformattedText;

   String toHtml() {
      return unformattedText;
   }
}
```

```
abstract class TextDecorator implements Text {

   protected final Text decoratee;

   protected TextDecorator(Text decoratee) {
      this.decoratee = decoratee;
   }
}
```

```
class BoldDecorator extends TextDecorator {

   BoldDecorator(Text decoratee) {
      super(decoratee);
   }

   String toHtml() {
      return "<b>" + decoratee.toHtml() + "</b>";
   }
}
```

```
class ItalicDecorator implements TextDecorator {

   ItalicDecorator(Text decoratee) {
      super(decoratee);
   }

   String toHtml() {
      return "<i>" + decoratee.toHtml() + "</i>";
   }
}
```

```
// Aufrufer
Text myText = new BasicText();
myText.unformattedText = "Hello world!";

myText = new BoldText(myText);
myText = new ItalicText(myText);
```

Vorteile

- vermeidet großen Klassen
- Funktionalität nur im Bedarfsfall hinzugefügt
- flexibler als Vererbung

Nachteile

- Entfernen von Dekorieren nicht vorgesehen
- viele Dekorierer können unübersichtlich werden
- erhöhter Debugging und Leseaufwand

Singleton

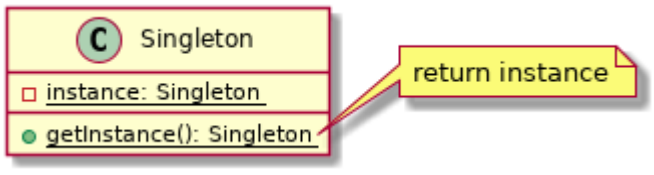
dt: Einzelstück

Die deutsche Übersetzung ist nicht gebräuchlich.

Steckbrief

- Art: Erzeugungsmuster (creator pattern)
- Zweck
  - Einschränkung der Instanziierung
    - üblicherweise auf 1 Objekt
  - globale Variable / Zustand / Service

Singleton: UML



Singleton: Implementierung

```
public final class MySingleton {

   private static final MySingleton instance = new MySingleton();

   private MySingleton() {}

   public static MySingleton getInstance() {
      return instance;
   }
}
```

Eager Instantiation (frühe Instanziierung)

- Vorteile: einfache Implementierung, Laufzeitverhalten bekannt
- Nachteil: Erzeugungsarbeit immer beim Systemstart (= längere Startzeit)

Lazy Instantiation

```
public final class MySingleton {

   private static final MySingleton instance;

   private MySingleton() {}

   public static MySingleton getInstance() {
      if (instance == null) {
         instance = new MySingleton();
      }
      return instance;
   }
}
```

Fallstrick: Threadsicherheit!

Lazy Instantiation: Synchronized

```
public final class MySingleton {

   private static final MySingleton instance;

   private MySingleton() {}

   public static synchronized MySingleton getInstance() {
      if (instance == null) {
         instance = new MySingleton();
      }
      return instance;
   }
}
```

synchronized ohne eigenes *Lock* ist fehleranfällig → bei statischen Methoden wird das Klassenobjekt als Lock genutzt

Lazy Instantiation: Synchronized mit Custom Lock

```
public final class MySingleton {

   private static final MySingleton instance;
   private static final Object lock = new Object();

   private MySingleton() {}

   public static MySingleton getInstance() {
      synchronized ( lock ) {
         if (instance == null) {
            instance = new MySingleton();
         }
         return instance;
      }
   }
}
```

Viel Aufwand für einen geringen Effekt.

Singleton als Enum

```
public enum MySingleton {
   INSTANCE;
}
```

- Einzige korrekte Art Singletons zu erzeugen.

- *Enum*s sind ein Sprachfeature, dass die einzigartige Erzeugung garantiert.

- ohne Enum
  - über Reflection kann trotzdem eine weitere Instanz erzeugt werden
    - → Exception im Constructor werfen, wenn schon eine Instanz vorhanden
  - Serialization: alle Felder müssen *transient* sein und eine *readResolve* -Methode muss existieren → ansonsten wird jedes mal bei der Deserialisierung einer serialisierten Instanz eine neue Instanz angelegt
    - *readResolve* gibt einfach die (schon vorhandene) Instanz zurück

Vergleich der Varianten

- Eager Instantiation
  - einfache Implementierung
  - früher Ressourcenbedarf
- Lazy Instantiation
  - aufwendige Implementierung
  - spätestmöglicher Ressourcenbedarf
- Enum-basiert
  - Garantie durch Sprachfeature
  - früher Ressourcenbedarf
  - ungewohnt

Singleton: Vorteile

- systemweit nur eine Instanz
  - bei korrekter Implementierung
- einfacher Zugriff auf die Instanz
  - einfach zu verstehen
  - einfach zu verwenden
  - unmittelbarer "Erfolg"

Singleton: Nachteile

Nachteil: Globaler Zugriff

- globaler Zugriff auf die Instanz
- konkreter Klassenname in jedem Zugriff

    - keine Polymorphie möglich
- aus Architektursicht unvorteilhaft
    - jeder hat jeder Zeit auf alles Zugriff
- Race Conditions

```
// skipping instantiation and constructor
final public class MySingleton {
   private int state = 0;
   public void changeState() {
      state++;
   }
}
```

```
// Aufrufer 1
MySingleton.getInstance().changeState();
// parallel Aufrufer 2
MySingleton.getInstance().changeState();
```

Nicht thread-sicher, da *state* erst gelesen und dann gesetzt wird.

Rufen zwei Threads gleichzeitig *changeState* auf, könnte es passieren, dass sie beide den gleichen Wert lesen, bevor sie inkrementieren (und *state* damit effektiv nur um 1 inkrementiert wird anstatt um 2).

Nachteil: Starke Kopplung

- Kopplung an den konkreten Typ
- keine polymorphen Aufrufe zu haben bedeutet, den Singleton-Code nahezu zu „Inlinen“
- es gibt kaum eine stärkere Kopplung
    - guter Code ist lose gekoppelt

100% Kopplung

```
class MyClass {

   void myMethod() {
      // ...
      MySingleton.getInstance().operation();
      // ...
   }
}
```

Nachteil: Testen erschwert

- aufwendig ein Mock-Objekt für ein Singleton anzubieten
  - Singletons erlauben keine Unterklassen
  - Singletons werden im Normalfall "direkt" benutzt

Wie kann man in einem Test das Singleton mit einem Mock-Objekt ersetzen?

```
class MyClass {

   void myMethod() {
      // ...
      MySingleton.getInstance().operation();
      // ...
   }
}
```

Extract Interface* und *Parametrize Method

```
class MyClass {

   void myMethod(MySingletonInterface singleton) {
      // ...
      singleton.operation();
      // ...
   }
}
```

Jetzt kann ein Mock-Objekt das *MySingletonInterface* implementieren und im Test verwendet werden.

```
void test() {
   MyClass myClass = new MyClass();
   myClass.myMethod(new MockSingleton());
   //...
}
```

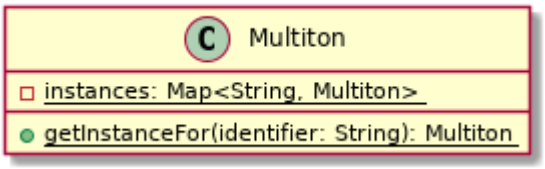
Nachteil:

- durch *Extract Interface* können nun beliebig viele andere Instanzen mit diesem Interface erzeugt werden.
- Ausblick: Sealed Classes
  - seit Java 15 in der Preview (JEP 360)
  - seit Java 16 in der Second Preview (JEP 397)
  - *sealed* Interfaces erlauben das Deklarieren der Implementierer → "Wildwuchs" nicht mehr

möglich

Singleton: Varianten

- Instanzpool / Factory
  - statt einer Instanz werden mehrere Instanzen erzeugt und vorgehalten
  - Herausgabe z.B. im "Round Robin"-Verfahren
- Multiton
  - mehrere Instanzen werden erzeugt und vorgehalten
  - jede Instanz besitzt einen eindeutigen Identifier



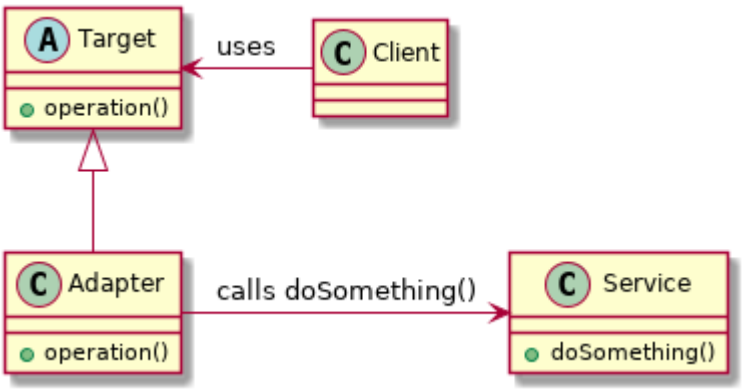
Adapter

auch: Hüllenklasse oder Wrapper

Steckbrief

- Art: Strukturmuster (structural pattern)
- Zweck
  - adaptieren einer Schnittstelle in eine andere
  - generell für Klassen, Methoden, Services, ...

Adapter: UML



Adapter: Beispiel

```
class User {
   private MyBirthday birthday;

   void setBirthday(MyBirthday date) {
      birthday = date;
   }
}
```

```
interface MyBirthday {
   int getCurrentAge();
}
```

Jetzt wird ein Service eingebunden, der als Birthday ein *LocalDate* zurückliefert.

```
class AdaptedLocalDate implements MyBirthday {
   private final LocalDate birthday;

   AdaptedLocalDate(LocalDate date) {
      birthday = date;
   }

   int getCurrentAge() {
      return Period
                .between(birthday, LocalDate.now())
                .getYears();
   }
}
```

LocalDate kann nun in *User* als *MyBirthday* benutzt werden.

Vorteile

- lose Kopplung
- einfache Erweiterung / Adaptierung anderer Komponenten
- Unabhängigkeit von externen Komponenten
- unterstützt *Information Expert* und *High Cohesion*
- Einschränkung des Zugriffs auf nicht-benötigte Funktionalität

Nachteile

- "teure" Aufrufe nicht sichtbar
  - z.B. ein unerwarteter Remote Call

Adapter: Variante

Ohne Interface

```
class  MyBirthday {

   public static MyBirthday of(LocalDate date) {
      return new MyBirthday(date);
   }

   public static MyBirthday of(Date date) {
      return new MyBirthday(toLocalDate(date));
   }

   private final LocalDate birthday;

   private MyBirthday(LocalDate date) {
      birthday = date;
   }

   int getCurrentAge() {
      return Period
                .between(birthday, LocalDate.now())
                .getYears();
   }
}
```

- Vorteil: schnell implementiert und verständlich
- Nachteil: kann groß und unübersichtlich werden

Empfehlung: alles Externe adaptieren

- 3rd-Party-Libraries
- Services
- SDKs

Besser für einen konkreten Anwendungsfall adaptieren als eine weitere allgemeine Klasse.

Z.B.: LocalDate → MyBirthday

Nicht: ~~LocalDate → MyLocalDate~~

So bleibt man auf einfache Art unabhängig und man kann schneller auf Änderungen reagieren. Zudem werden die adaptierten Klassen zum *Information Expert* und haben eigene (sinnvolle) Funktionalität drin. Gleichzeitig kann der Zugriff auf nicht-benötigte Funktionalität der adaptierten Klasse eingeschränkt werden.

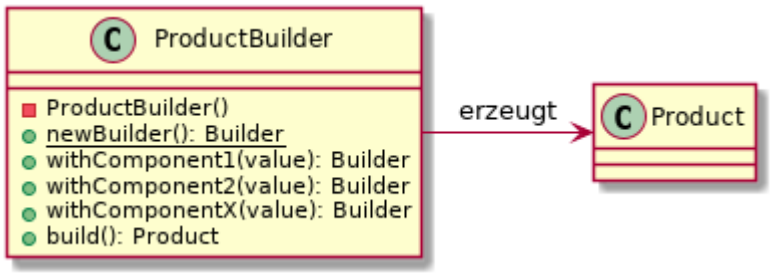
Builder

auch: Erbauer, Creator

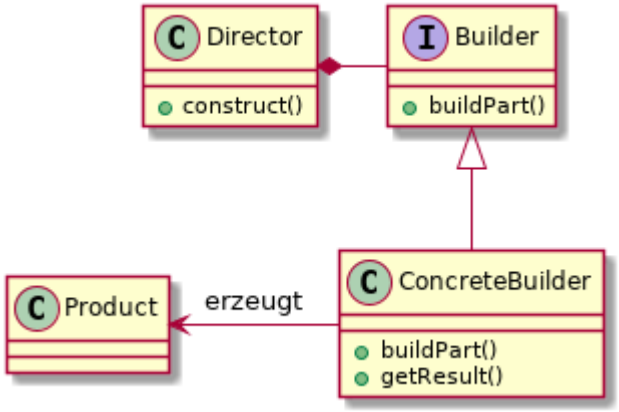
Steckbrief

- Art: Erzeugungsmuster (creational pattern)
- Zweck
  - Auslagern komplexer Erzeugungslogik
  - einfaches Bauen komplexer Objekte
  - Wiederverwendung der Erzeugungslogik für unterschiedliche Repräsentationen

Builder: UML



Builder (alternativ): UML



Das ist die Fassung wie sie auch in *Design Patterns* von Erich Gamma et. al. vorkommt.

Builder: Beispiel

```
class User {
   // ...
   User(String firstName, String lastName, Role role, String email, String
phoneNumber) {
      // ...
   }

   User(String firstName, String lastName, Role role) {
      this(firstName, lastName, role, null, null);
   }

   User(String firstName, String lastName, String phoneNumber) {
      this(firstName, lastName, Role.default(), null, phoneNumber);
   }
}
```

Probleme: unübersichtlich, mögliche Überschneidung der Methoden-Signatur (User(String, String, String)), fehleranfällig beim Aufruf (nur Strings)

```
class UserBuilder {
   // ...

   public static Builder withName(String firstName, String lastName) {
      return new UserBuilder(firstName, lastName);
   }

   private UserBuilder(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public Builder withEmail(String email) {
      this.email = email;
      return this;
   }

   // ...

   public User build() {
      return new User(firstName, lastName, ...);
   }
}
```

Builder: Vorteile

- Erzeugung wird vereinfacht
- sprechende Methodennamen

- schwieriger falsch zu benutzen
- *Seperation of Concerns* und *Information Hiding*
- einfach erweiterbar mit neuen Attributen

Builder: Nachteile

- (relativ) viel Code für die Objekterzeugung
- Duplizierung der Objektattribute im Builder
  - enge Kopplung zwischen Objekt und Builder

Builder: Varianten

- Builder kann Default-Werte festlegen, wenn die entsprechenden Attribute nicht gesetzt werden
- Builder kann vorbelegte Varianten vorhalten
  - z.B. *CarBuilder.**mercedes()**.withAC()* und *CarBuilder.**audi()**.withoutAC()*