

# ENTWURFSMUSTER

Maurice Müller

2023-02-28

# **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?

- keine "Neu-Erfindung" des Rads
- Wissensvermittlung auf abstrakten Niveau
- einfacheres Verständnis des Codes

- helfen, komplexer werdende Softwaresysteme zu verstehen
  - grösere Bausteine helfen den Überblick zu behalten
  - s. z.B. integrierte Schaltkreise in der Elektronik
- Beginn einer höherwertigen Sprache unter Entwicklern
  - vereinfachte Kommunikation zwischen Entwicklern

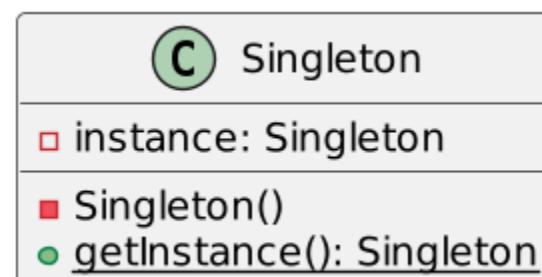
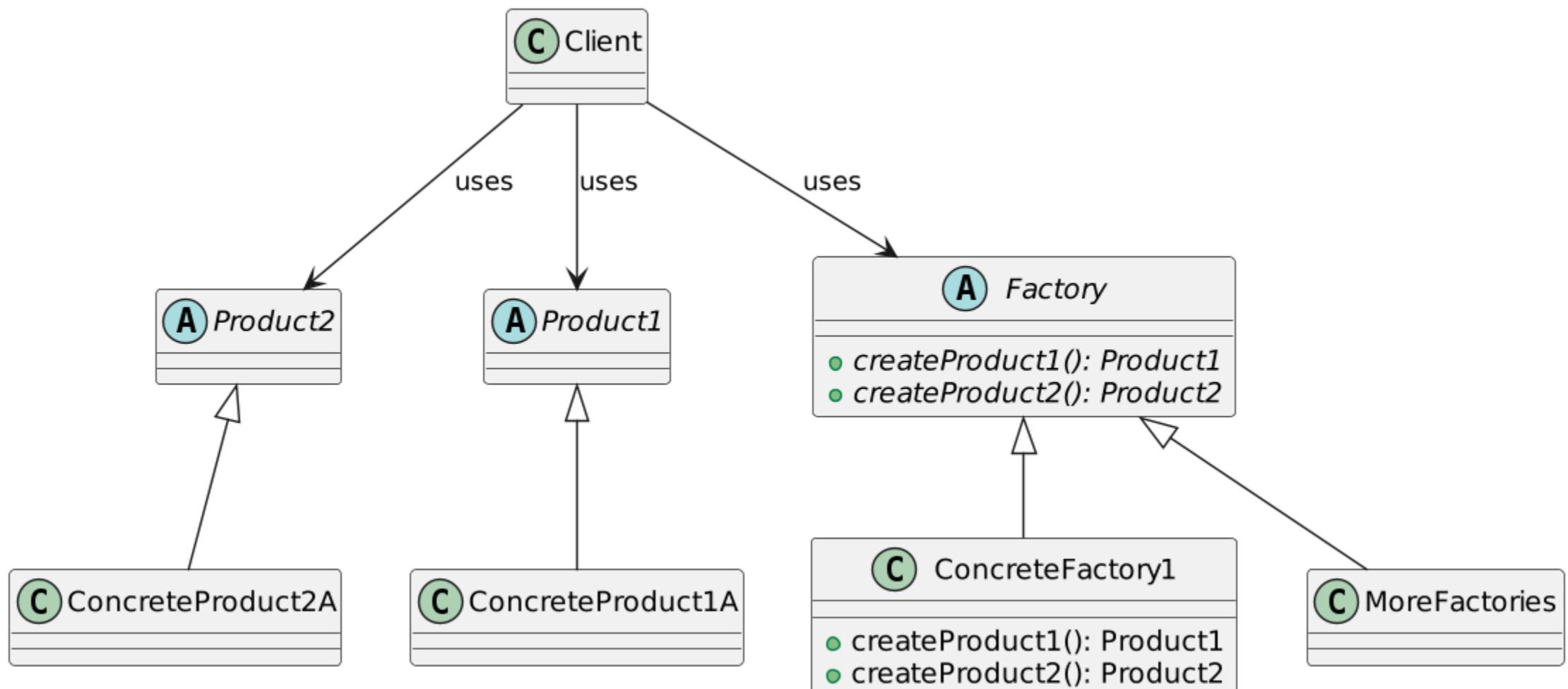
# Kategorien von Entwurfsmustern

- 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

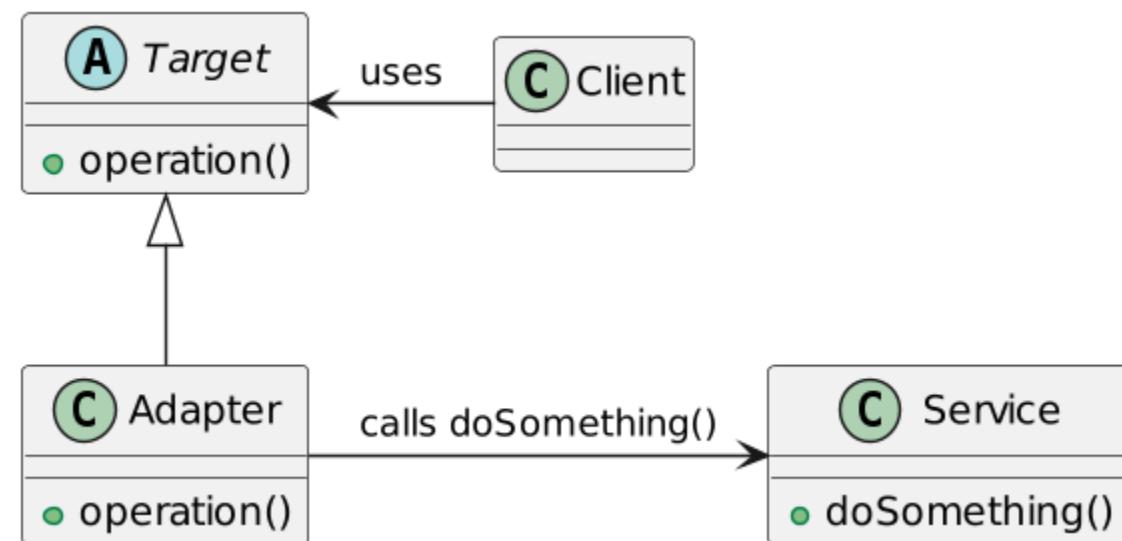
# Erzeugungsmuster Beispiele

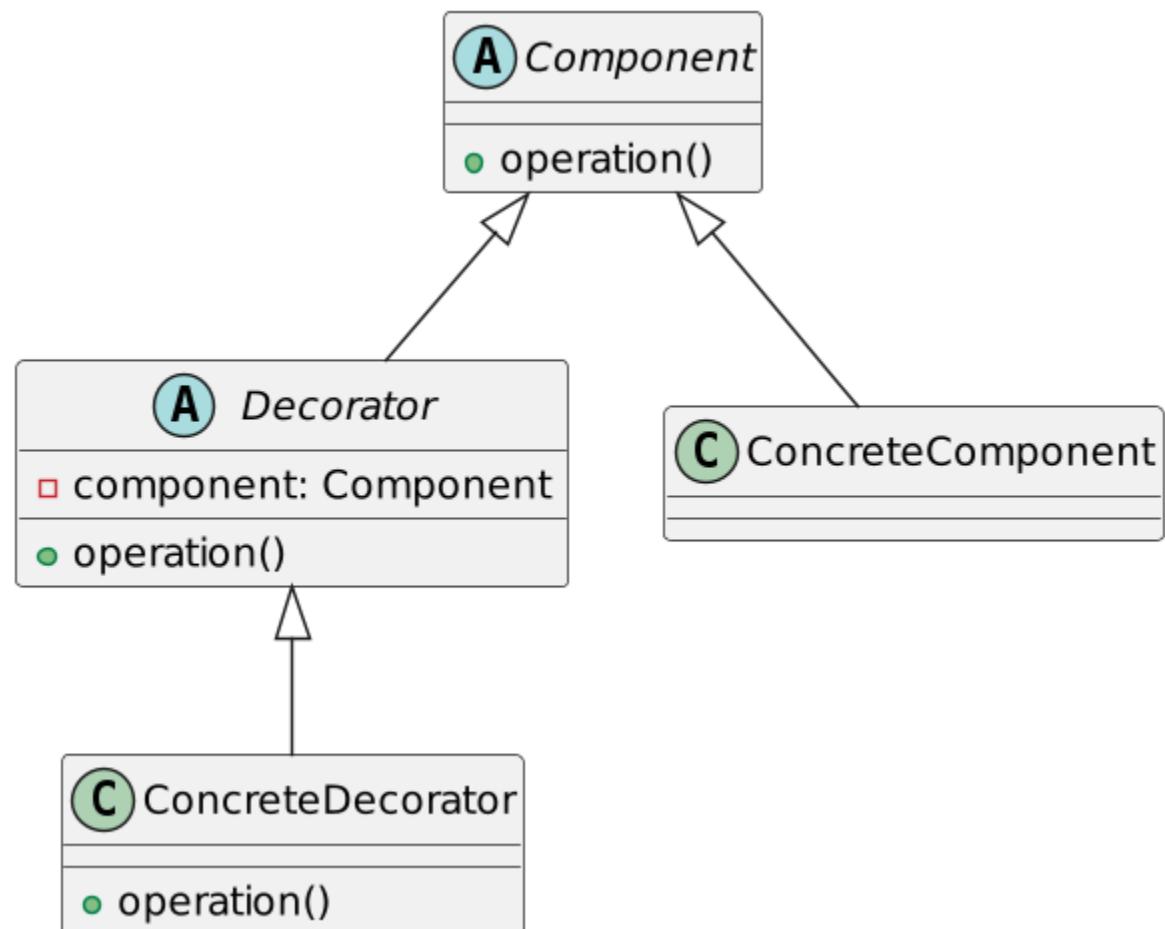


# Kategorie: Strukturmuster

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

# Strukturmuster Beispiele

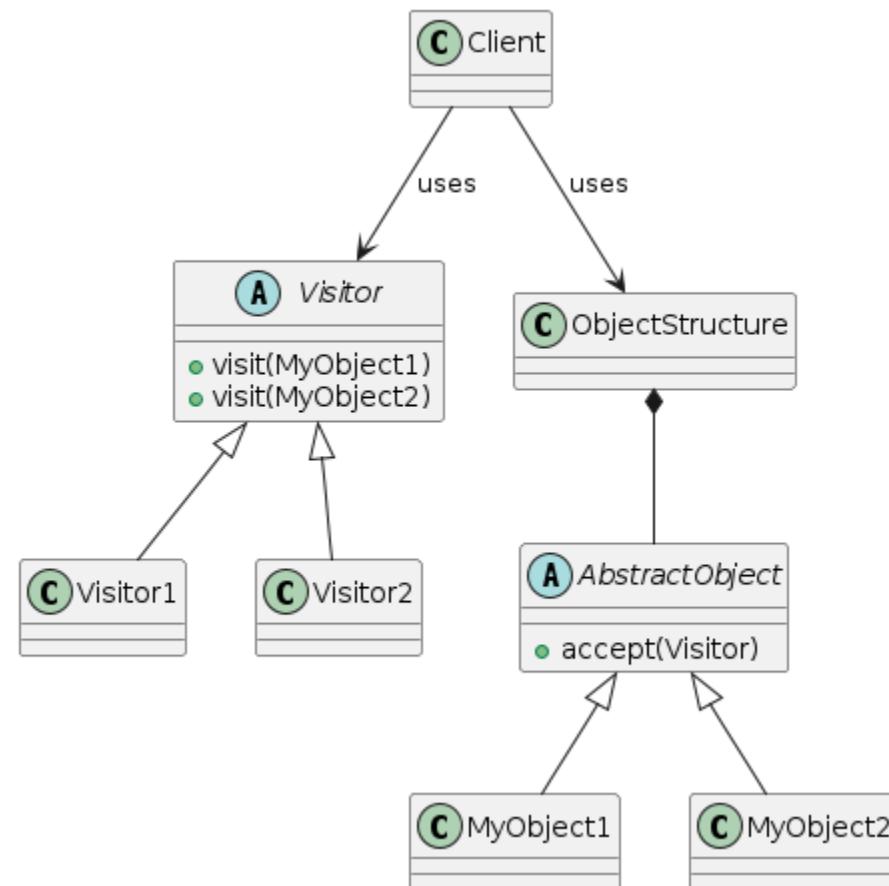


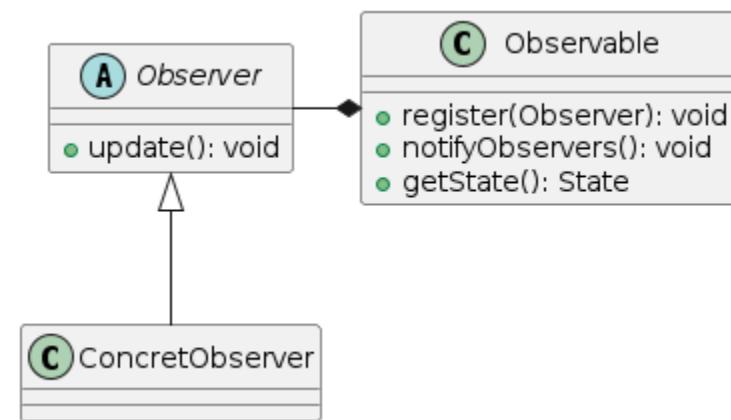


# Kategorie: Verhaltensmuster

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

# Verhaltensmuster Beispiele





# Beobachter (Observer)

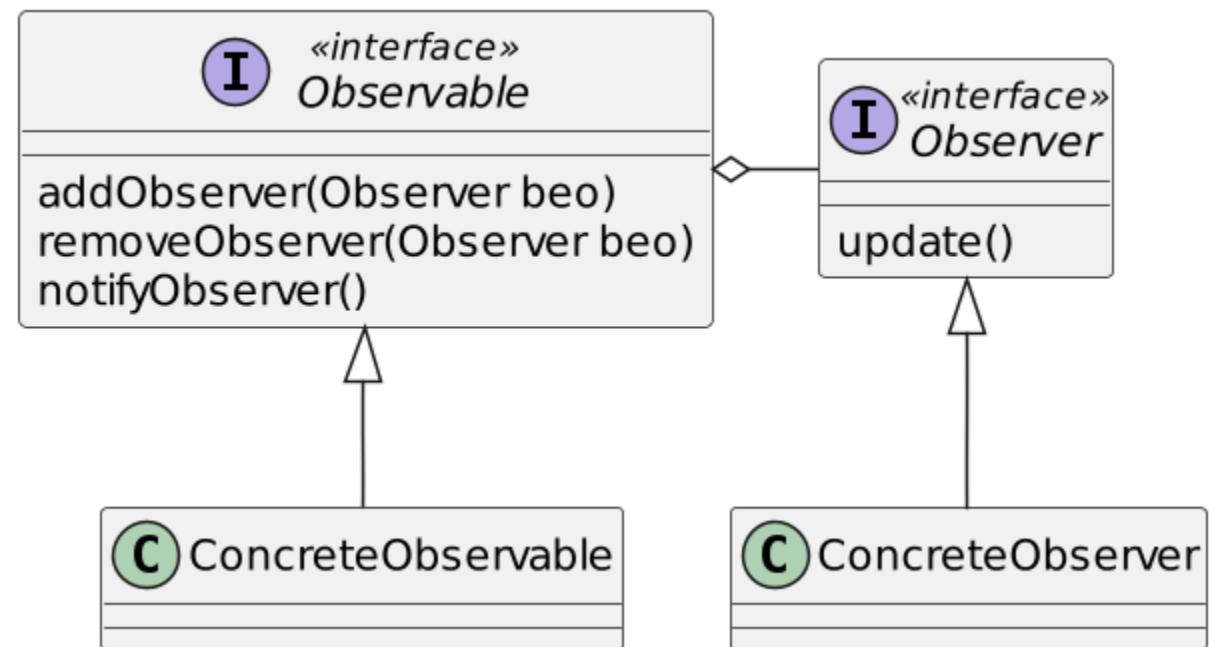
auch Listener oder Publish-Subscribe



# 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 update: " + observable.getState());  
    }  
  
}
```

# 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);  
        }  
    }  
}
```

## Wie kommt der Beobachter an den aktuellen Zustand?

- new ConcreteObserver(ConcreteObservable obs)
- Observer::update(ConcreteObservable 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: Zeichenprogramm

1 Observer, n Observables

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

# Implementierung mit Beobachter-Muster

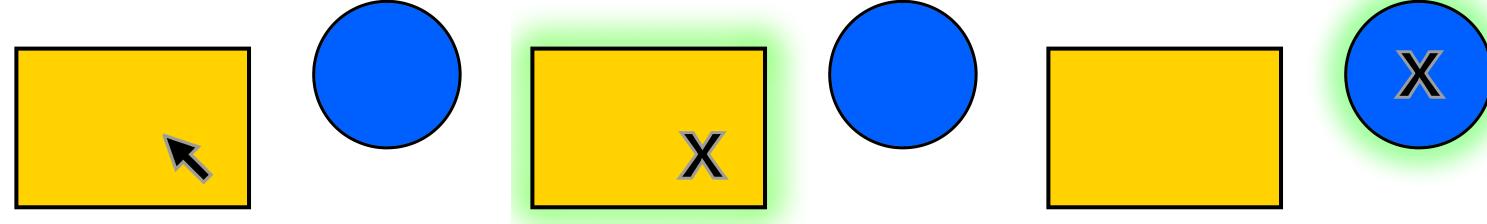
```
interface SelectionListener {  
    void select(Element element);  
    void deselect(Element element);  
}
```

# 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



# Auswertung

## 1. Alternative 1

1. Rechteck wird selektiert: Cursor wird zum X
2. Kreis wird selektiert: Cursor bleibt X
3. Rechteck wird deseletktiert: Cursor bleibt X

## 1. Alternative 2

1. Rechteck wird selektiert: Cursor wird zum X
2. Rechteck wird deseletktiert: Cursor wird zum Pfeil
3. Kreis wird selektiert: Cursor wird zum X

# 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 Thr  
// [...]  
Client client = new Client(connection);  
client.connect();
```

# 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)

# 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 geändert. State1 = "  
    }  
}
```

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

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

## Ausgabe

```
Observable hat sich geändert. State1 = true
Observable hat sich geändert. State1 = true
Observable hat sich geändert. 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 geändert. 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);  
}
```

# 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.remove(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 hinzugefügt.");
});

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

```

Observer 6 hinzugefügt.

Observer 5 aktualisiert.

Exception in thread "main" java.util.concurrent.ExecutionException: java.ut

at java.base/java.util.concurrent.CompletableFuture.reportGet(Completable

at java.base/java.util.concurrent.CompletableFuture.get(CompletableFuture

at patterns.oo.listener.threadsicherheit.ohne\_synchronized.ThreadSicherhe

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.not

at patterns.oo.listener.threadsicherheit.ohne\_synchronized.ThreadSicherhe

# 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();  
}
```

# 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);  
        }  
    }  
}
```

# 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

- Verbindungsaufbau anfordern
- Verbindungsabbau anfordern
- Verbindung hergestellt
- Verbindung fehlgeschlagen
- TearDown Bestätigung der Clients

## Zustände

- ONLINE
- OFFLINE
- CONNECTING
- TEARING\_DOWN

# 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 geändert]  
observable.endTransaction();  
  
// als Lambda  
observable.transaction(() => {  
    // [hier werden Werte geändert]  
});
```

## 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)

```
#include::oo/decorator/decorator.adoc[] #
#include::oo/singleton/singleton.adoc[] #
#include::oo/adapter/adapter.adoc[] #
#include::oo/builder/builder.adoc[] #
#include::oo/strategy/strategy.adoc[] # #include::books.adoc[] #
```