

# **ENTWURFSMUSTER**

Maurice Müller

2024-10-07

# **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
  - größere Bausteine helfen den Überblick zu behalten
  - s. z.B. integrierte Schaltkreise in der Elektronik
- Beginn einer höherwertigen Sprache unter Entwicklern
  - vereinfachte Kommunikation zwischen Entwicklern

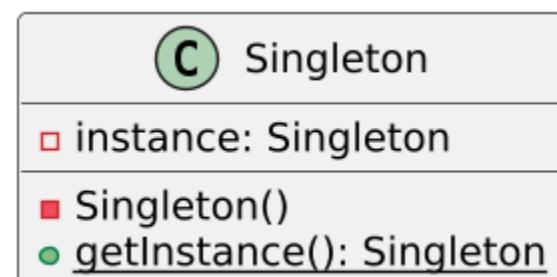
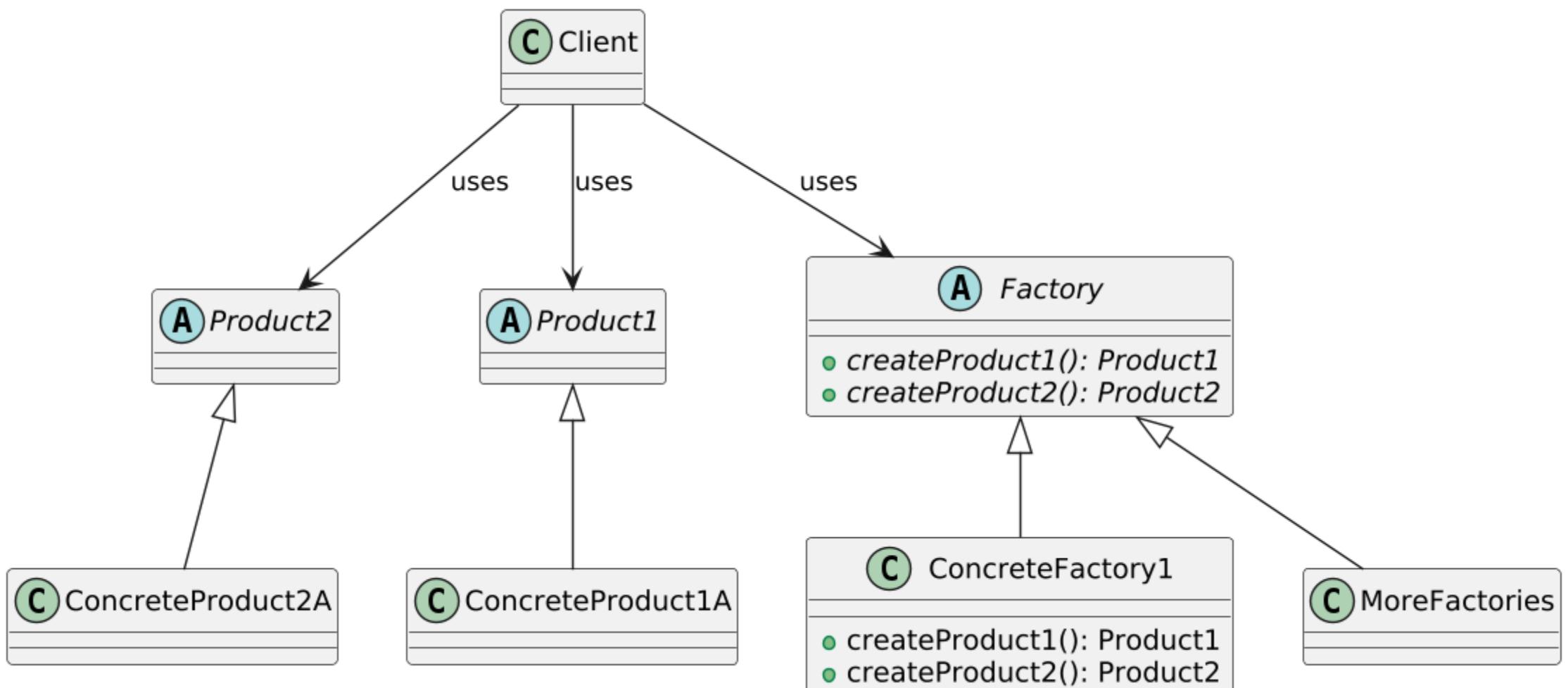
# Kategorien von Entwurfsmustern

- 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

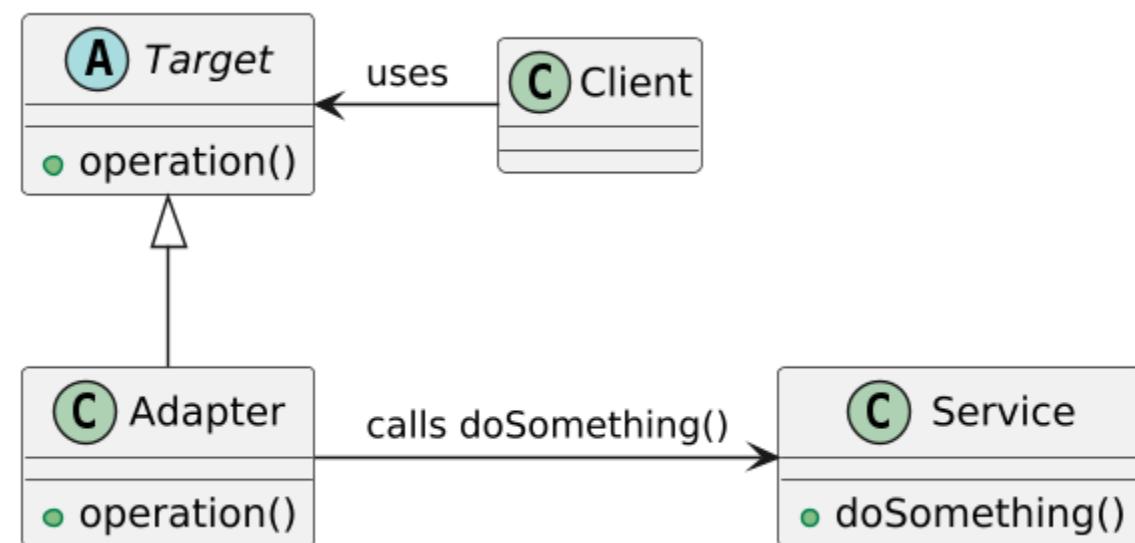
# Erzeugungsmuster Beispiele

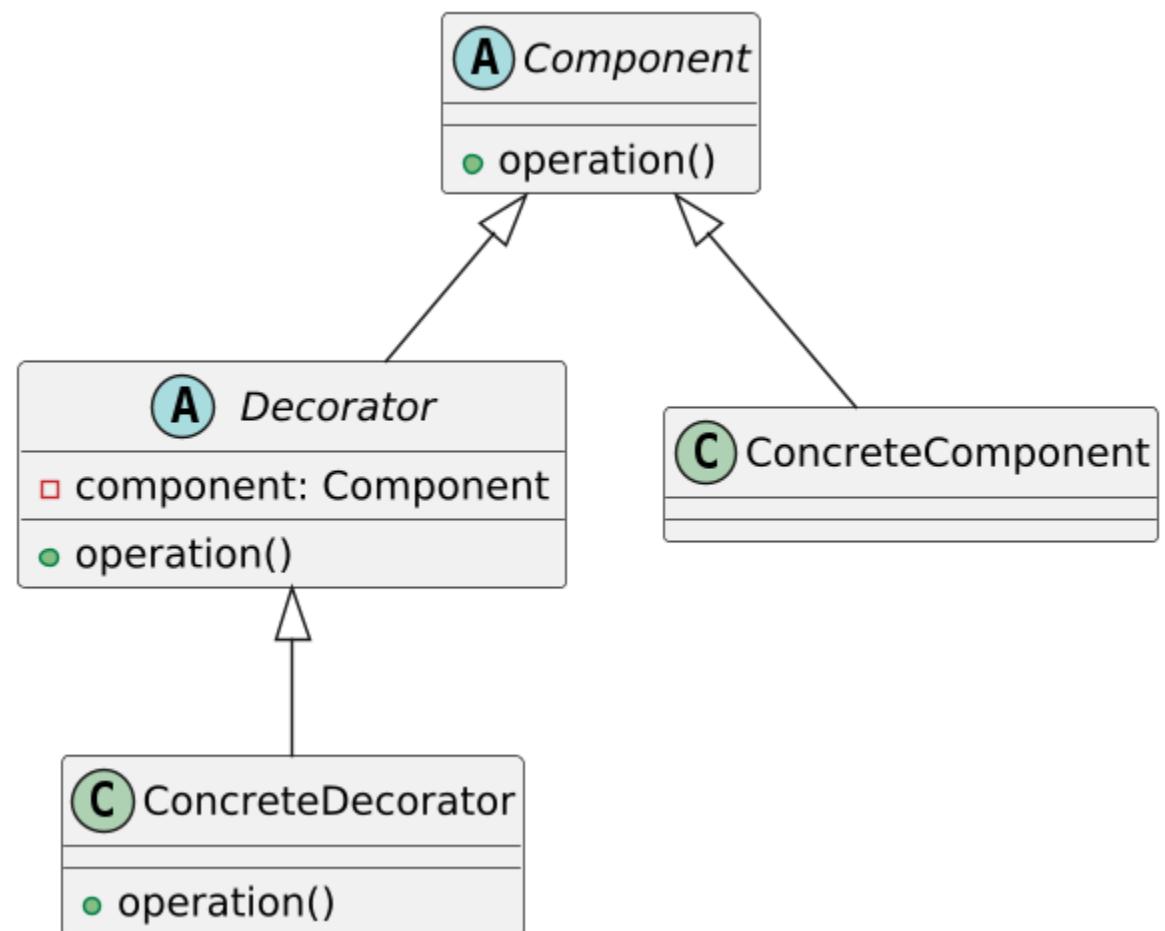


# Kategorie: Strukturmuster

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

# Strukturmuster Beispiele

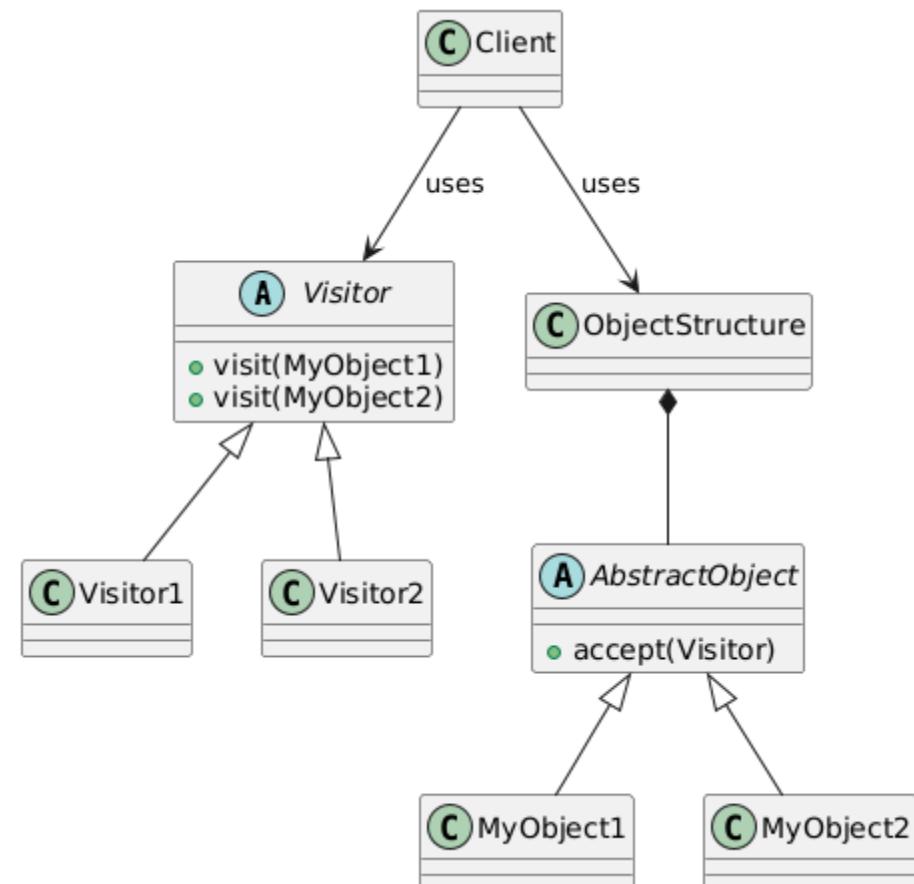


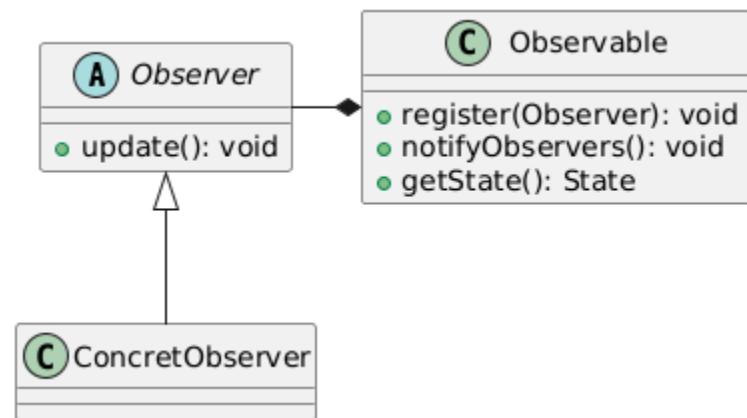


# Kategorie: Verhaltensmuster

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

# Verhaltensmuster Beispiele





# Beobachter (Observer)

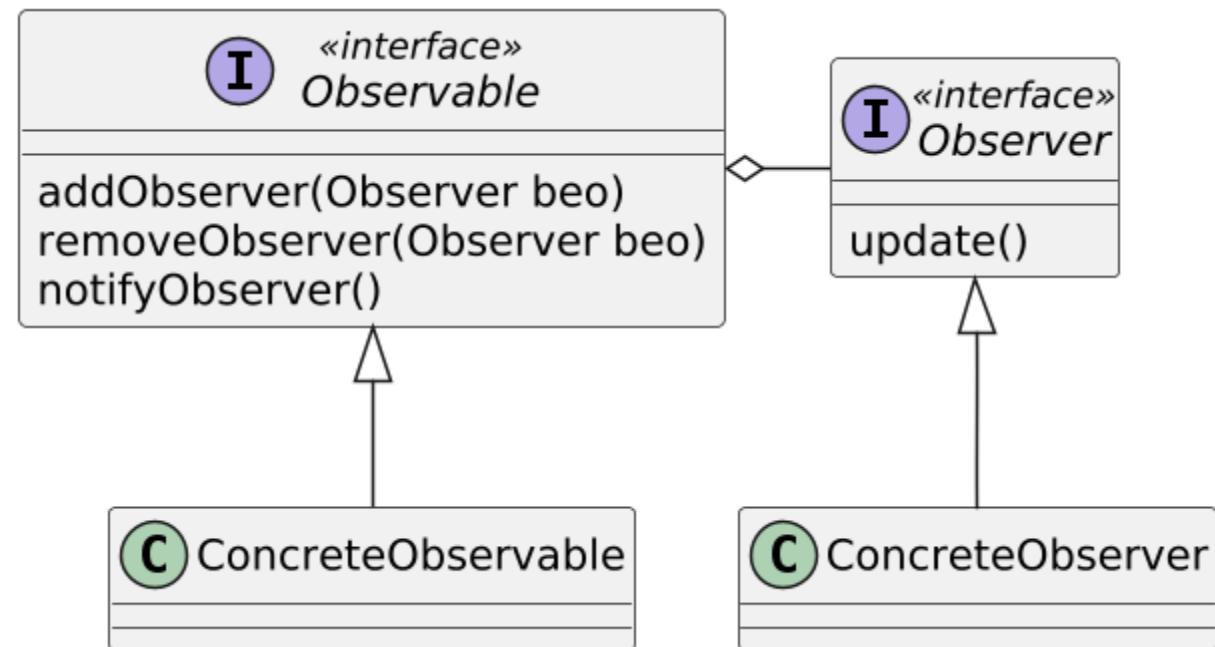
auch Listener oder Publish-Subscribe



# 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.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);  
        }  
    }  
}
```

## 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<>();

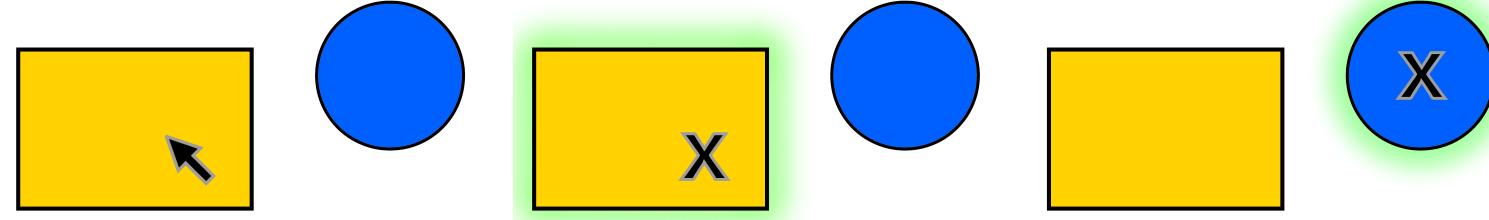
    @Override
    public void select(Element element) {
        if (elements.add(element)) updateCursor();
    }

    @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 Th  
// [...]  
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.getState1());  
    }  
}
```

```
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);  
    void onState2(Observable observable);  
}
```

# Fix: Gezielte Benachrichtigung

Alternative: unterschiedliche Listen mit Observern

```
class Observable {  
  
    private final Set<Observer> state1observers = new HashSet<>();  
    private final Set<Observer> state2observers = 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.onChange(this)); }  
    void notifyObserversState2() { observers.forEach(it -> it.onChange(this)); }  
}
```

# 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.util.ConcurrentModificationException  
   at java.base/java.util.concurrent.CompletableFuture.reportGet(CompletableFuture.java:397)  
   at java.base/java.util.concurrent.CompletableFuture.get(CompletableFuture.java:1394)  
   at patterns.oo.listener.threadsicherheit.ohne\_synchronized.ThreadSicherheit.main(ThreadSicherheit.java:12)  
 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:16)  
   at patterns.oo.listener.threadsicherheit.ohne\_synchronized.ThreadSicherheit.lambda\$0(ThreadSicherheit.java:12)

# 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)

# Decorator

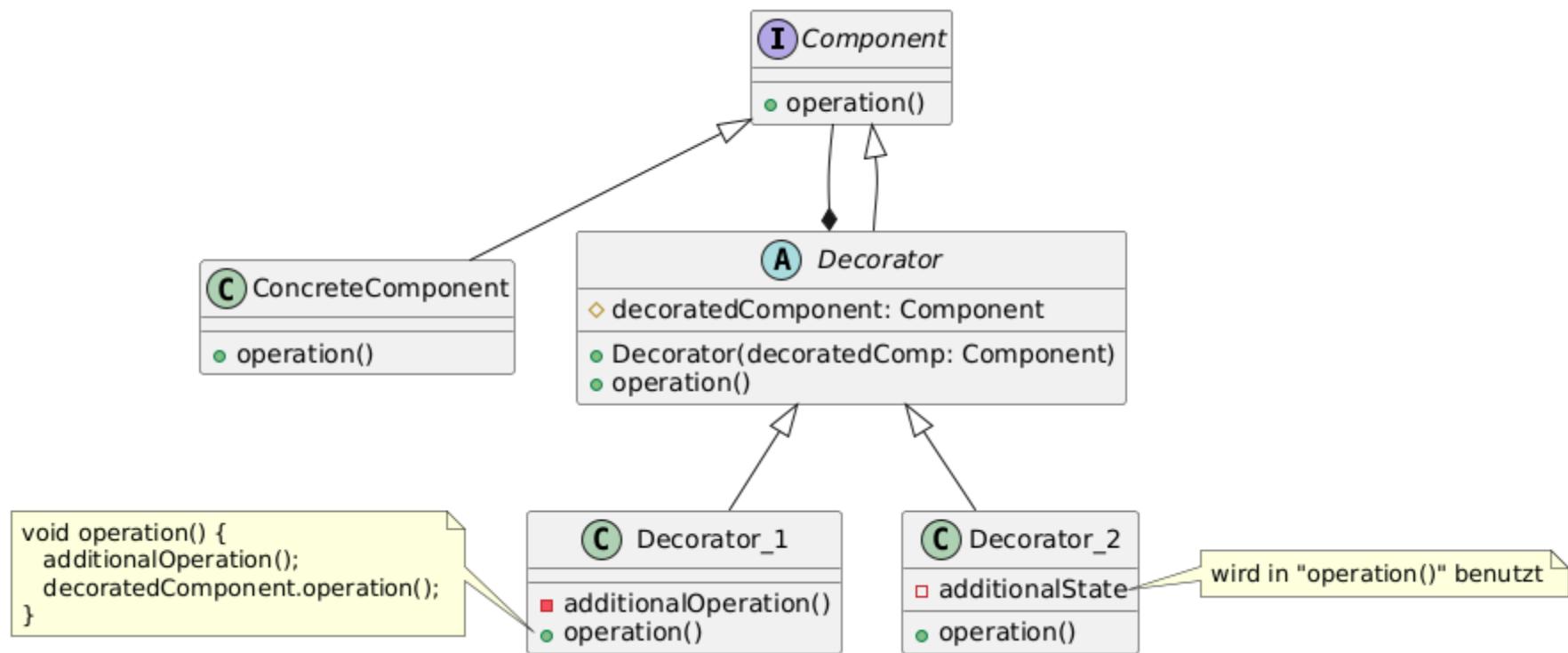
auch *Wrapper*



# 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;  
    }  
}
```

```
interface Text {  
    String toHtml();  
}  
  
class BasicText implements Text {  
    String unformattedText;  
  
    String toHtml() {  
        return unformattedText;  
    }  
}
```

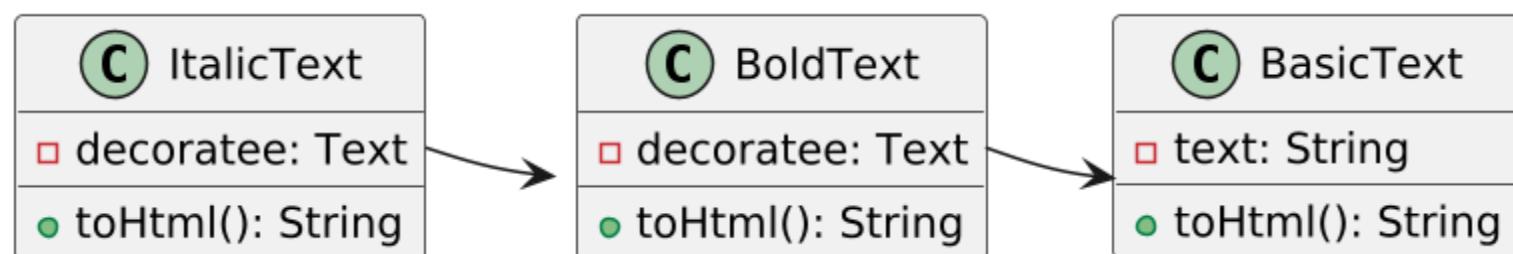
```
abstract class TextDecorator implements Text {  
  
    protected final Text decoratee;  
  
    protected TextDecorator(Text decoratee) {  
        this.decoratee = decoratee;  
    }  
}  
  
class BoldText extends TextDecorator {  
  
    BoldText(Text decoratee) {  
        super(decoratee);  
    }  
  
    String toHtml() {  
        return "<b>" + decoratee.toHtml() + "</b>";  
    }  
}
```

```
class ItalicText implements TextDecorator {  
  
    ItalicText(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);
```

# Interaktion

- jeder Dekorierer leitete Anfragen an sein Komponentenobjekt weiter
  - vor und/oder nach dem Weiterleiten kann Funktionalität erweitert werden



# Vorteile

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

# NACHTEILE

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

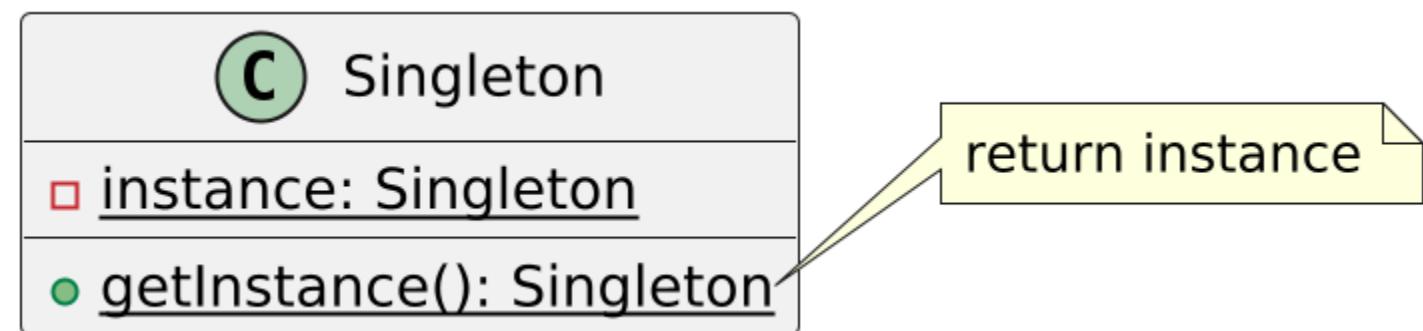
# **Singleton**

dt: Einzelstück

# 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.
- *Enums* sind ein Sprachfeature, dass die einzigartige Erzeugung garantiert.

# Vergleich der Varianten

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

# Singleton: Einsatzgebiete

- Zugriff auf eingeschränkte Ressourcen
  - z.B. Drucker, Netzwerkinterface, etc. → hier ist ein gleichzeitiger Zugriff nicht sinnvoll ⇒ alles sollte über einen Aufrufer gehen
- **Antipattern:** aus Bequemlichkeit Singleton für einfachen Zugriff missbrauchen

# 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 Architektsicht 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.

# 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.
  - Lösung: *Sealed Interfaces*

# 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



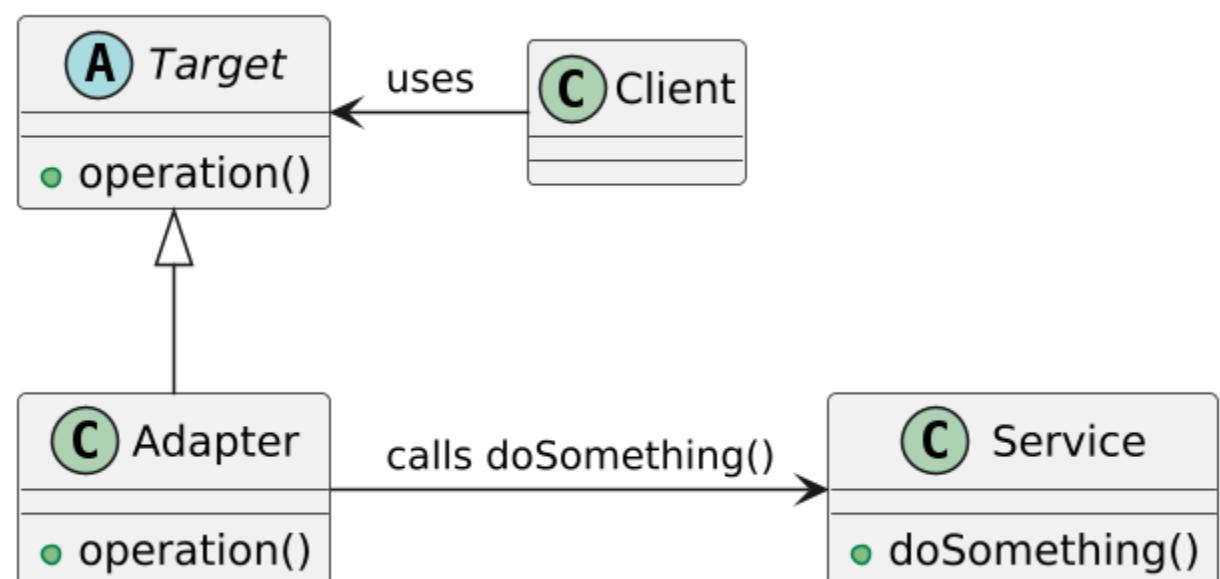
# **Adapter**

auch: Hüll-Klasse oder Wrapper

# Steckbrief

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

# Adapter: UML



# Adapter: Beispiel auf Klassenebene

```
class User {  
    private Birthday birthday;  
  
    void setBirthday(Birthday date) {  
        birthday = date;  
    }  
}
```

```
interface Birthday {  
    int getCurrentAge();  
    boolean isToday();  
}
```

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

```
class AdaptedLocalDate implements Birthday {  
    private final LocalDate birthday;  
  
    AdaptedLocalDate(LocalDate date) {  
        birthday = date;  
    }  
  
    int getCurrentAge() {  
        return Period  
            .between(birthday, LocalDate.now())  
            .getYears();  
    }  
  
    boolean isToday() {  
        LocalDate today = LocalDate.now();  
        return today.getDayOfMonth() == birthday.getDayOfMonth() &&  
               today.getMonth() == birthday.getMonth();  
    }  
}
```

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

# Adapter: Beispiel auf Serviceebene

```
class CreditService {  
  
    @Inject  
    SalaryService salaryService;  
  
    Credit calculateMaximumCredit(User user) {  
        Salary salary = salaryService.getFor(user);  
        if(salary.annual() >= 100_000) {  
            return Credit.max(100_000);  
        } else if(salary.annual() <= 15_000){  
            return Credit.noCredit();  
        } else {  
            return Credit.max(15_000);  
        }  
    }  
}
```

Jetzt soll der Service erweitert werden mit einem intern berechneten CreditScore.

```

class CreditService {

    @Inject
    SalaryService salaryService;
    @Inject
    CreditScoreService creditScoreService;

    Credit calculateMaximumCredit(User user) {
        Salary salary = salaryService.getFor(user);
        CreditScore creditScore = creditScoreService.getFor(user);
        if(salary.annual() >= 100_000) {
            return creditScore.isVeryGood() ? Credit.max(200_000) : Credit.
        } else if(salary.annual() <= 15_000){
            return Credit.noCredit();
        } else {
            return Credit.max(15_000);
        }
    }
}

```

Jetzt soll der CreditScore aus einem externen System eingelesen werden für den es sogar eine Library gibt. Problem: **CreditScore**, **CreditService** und **User** der Library passen nicht zu unserem Datenmodell.

```

class ExternalCreditScoreServiceAdapter implements CreditScoreService {

    @Inject
    ExternalCreditScoreService service;

    @Override
    CreditScore getFor(User user) {
        Result externalScore = service.fetchScore(user.taxId());
        return new CreditScore(externalScore.getDouble("ratingXY"),
                               externalScore.getDouble("homeLoanRating"));
    }

}

```

- externer Dienst kann nun problemlos intern genutzt werden → Aufruf und Rückgabe wurden adaptiert

## 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 Birthday {  
  
    public static Birthday of(LocalDate date) {  
        return new Birthday(date);  
    }  
  
    public static Birthday of(Date date) {  
        return new Birthday(toLocalDate(date));  
    }  
  
    private final LocalDate birthday;  
  
    private Birthday(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
- SDK-Klassen und Primitive

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

Z.B.: besser LocalDate → Birthday anstatt LocalDate → MyLocalDate

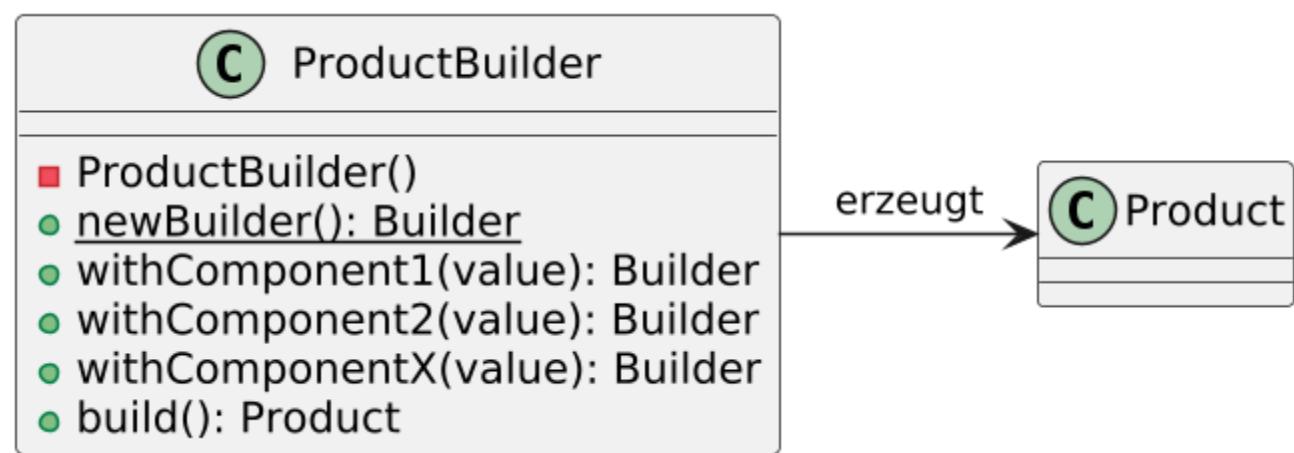
# **Builder**

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: Beispiel

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

## Aufruf

```
new User("Mustermann", "Max", "0123456789");  
new User("Erika", "Mustermann", "erika.mustermann@example.com");
```

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

```
class UserBuilder {  
    // ...  
  
    public UserBuilder() {  
    }  
  
    public Builder withFirstName(String firstName) {  
        this.firstName = firstName;  
        return this;  
    }  
  
    public Builder withLastName(String lastName) {  
        this.lastName = lastName;  
        return this;  
    }  
  
    public Builder withEmail(String email) {  
        this.email = email;  
        return this;  
    }  
  
    // ...  
  
    public User build() {  
        return new User(firstName, lastName, ...);  
    }  
}
```

## Aufrufer des Builder-Patterns

```
var user = new UserBuilder()  
    .withFirstName("Terry")  
    .withLastName("Pratchett")
```

# Builder: Fortgeschritten

- der große Konstruktor ist weiterhin notwendig
  - Zugriff einschränken (z.B. *package private*)
- Validierungen direkt in den Builder implementieren
- *Defaults* festlegen, die angepasst werden können
  - new  
CarBuilder().sportLine().withColor(Color.Red)

# Builder: Vorteile

- Erzeugung wird vereinfacht
- sprechende Methodennamen
- schwieriger falsch zu benutzen
- *Separation 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: Alternative

- Named-Parameter
- Bonus: Default-Parameter

```
// kotlin
fun greet(greeting: String = "Hello", name: String): String {
    return greeting + " " + name
}

println(greet(greeting = "Dear", name = "Erika"))
// Expected output: Dear Erika
println(greet(name = "Max"))
// Expected output: Hello Max
```

# Strategy

## *Strategie*

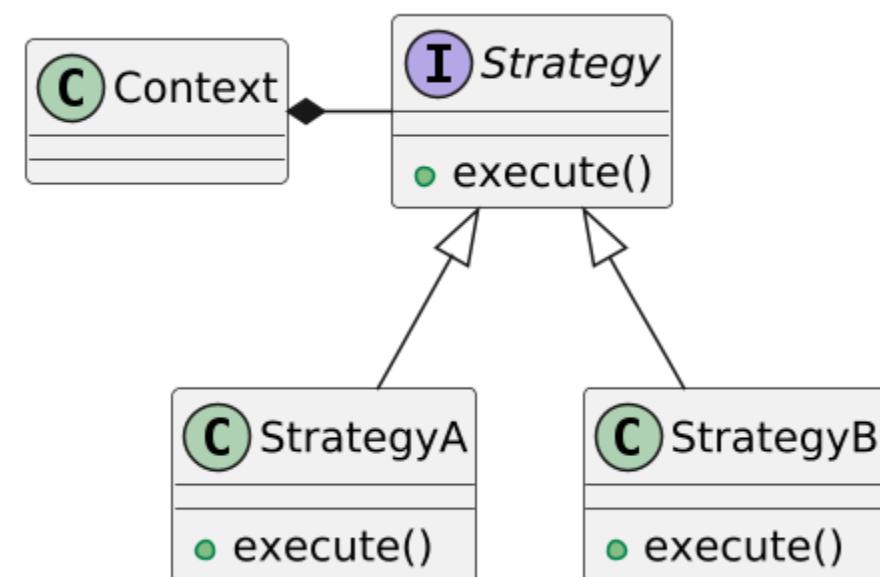


# Steckbrief

- Art: Verhaltensmuster (behavioral pattern)
- Zweck
  - Definition/Sammlung von austauschbaren Algorithmen
  - einfaches Austauschen von Algorithmen
    - sowohl zur Laufzeit als auch zur Compile-Zeit

*'fest' implementierte Algorithmen/Logik ist schwer austauschbar*

# Strategy: UML



# Strategy: Beispiel

- Empfehlung zum Kauf von Aktien

```
class ShareRecommender {  
    private Money investmentAmount;  
    private ShareStrategy strategy;  
  
    ShareRecommender(Money amount, ShareStrategy strategy) {  
        this.investmentAmount = amount;  
        this.strategy = strategy;  
    }  
  
    public List<Share> getRecommendations() {  
        return getAllShares().stream()  
            .filter(it -> it.inReasonablePriceRange(investmentAmount))  
            .filter(it -> strategy.fits(it))  
            .toList();  
    }  
  
    // [...]  
}
```

```
interface ShareStrategy {  
    boolean fits(Share share);  
}  
  
class LowRisk implements ShareStrategy {  
    boolean fits(Share share) {  
        return share.risk <= 0.5;  
    }  
}  
  
class HighProfit implements ShareStrategy {  
    boolean fits(Share share) {  
        return share.profitForecast5Years >= 1.5;  
    }  
}  
  
class Mixed implements ShareStrategy {  
    boolean fits(Share share) {  
        return share.profitForecast5Years >= 0.7 && share.risk <= 0.75;  
    }  
}
```

# Strategy: Vorteile

- einfache Wiederverwendung von Algorithmen
- einfache Erweiterbarkeit für neue Algorithmen
  - unterstützt OCP
- Alternative für Konditionalstruktur zur Auswahl von Verhalten
- dynamische Änderung des Verhaltens zur Laufzeit

# Strategy: Nachteile

- Aufrufer muss unterschiedliche Strategien kennen
  - → Standard sollte vorgegeben sein
- zusätzliche Kommunikation zwischen Kontext und Strategie
- erhöhte Anzahl von Objekten

# Kompositum

*engl. Composite*

# Steckbrief

- Art: Strukturmuster (structural pattern)
- Zweck
  - Definition einer Teil-Ganzes-Hierarchie (Baumstruktur)
  - ermöglicht transparenter Aufruf zwischen individuellen Objekt und Komposition

# Kompositum: Beispiel Java

```
final HTMLTable sites = new HTMLTable(  
    new TableRow(  
        new HeaderCell(new PlainText("Karlsruhe")),  
        new HeaderCell(new PlainText("Stuttgart")),  
        new HeaderCell(new PlainText("Mannheim"))),  
    new TableRow(  
        new DataCell(new PlainText("Marktplatz")),  
        new DataCell(new PlainText("Marienplatz")),  
        new DataCell()));  
  
System.out.println(sites.asString());
```

Karlsruhe

Stuttgart

Mannheim

Marktplatz

Marienplatz

```

final HTMLTable events = new HTMLTable(
    new TableRow(
        new HeaderCell(new PlainText("Veranstaltungen")),
        new HeaderCell(new PlainText("Orte"))),
    new TableRow(
        new DataCell(new PlainText("Oktoberfest")),
        new DataCell(sights)));
System.out.println(events.asString());

```

**Veranstaltung**

**Orte**

Oktoberfest

Karlsruhe

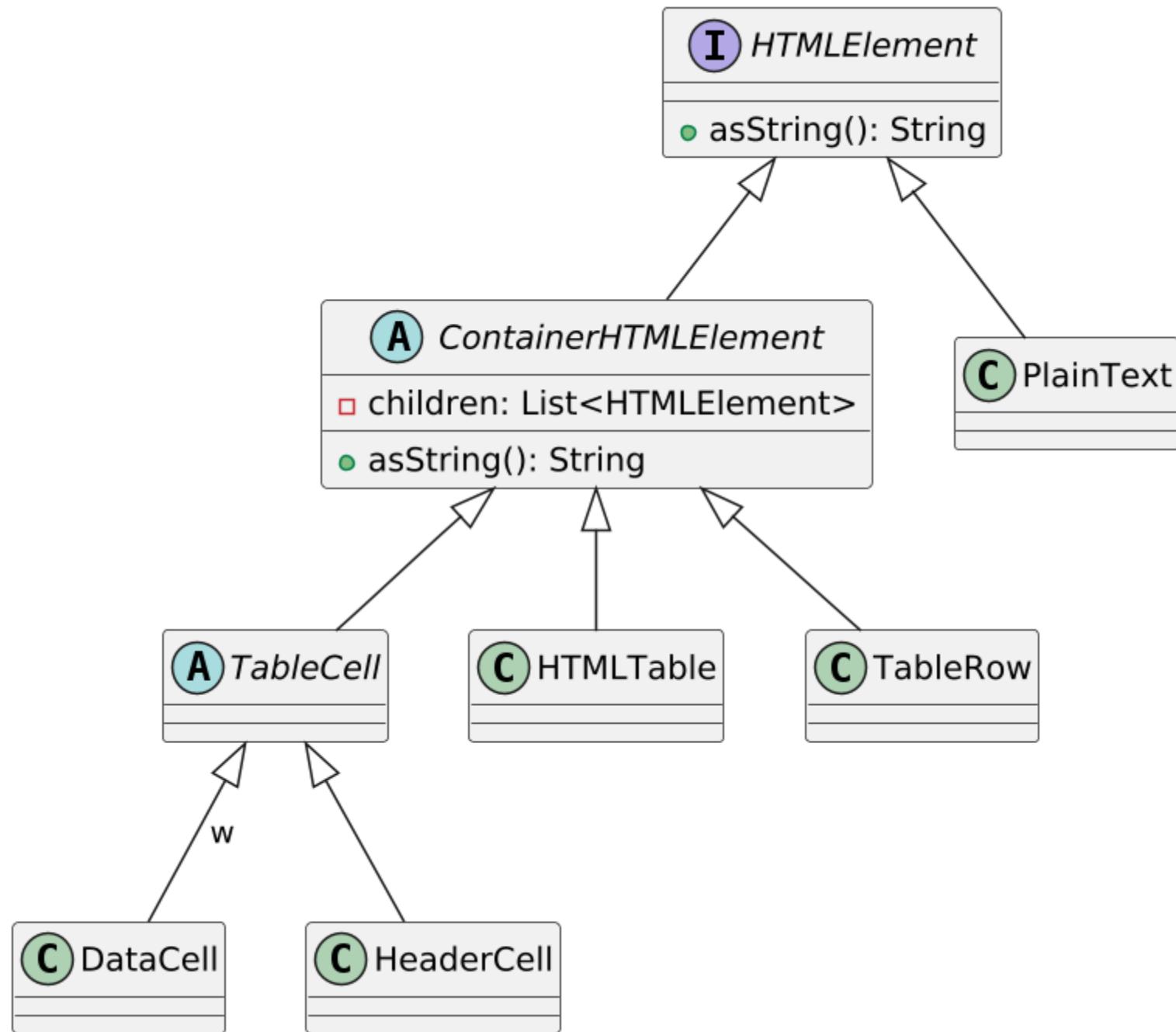
Stuttgart

Mannheim

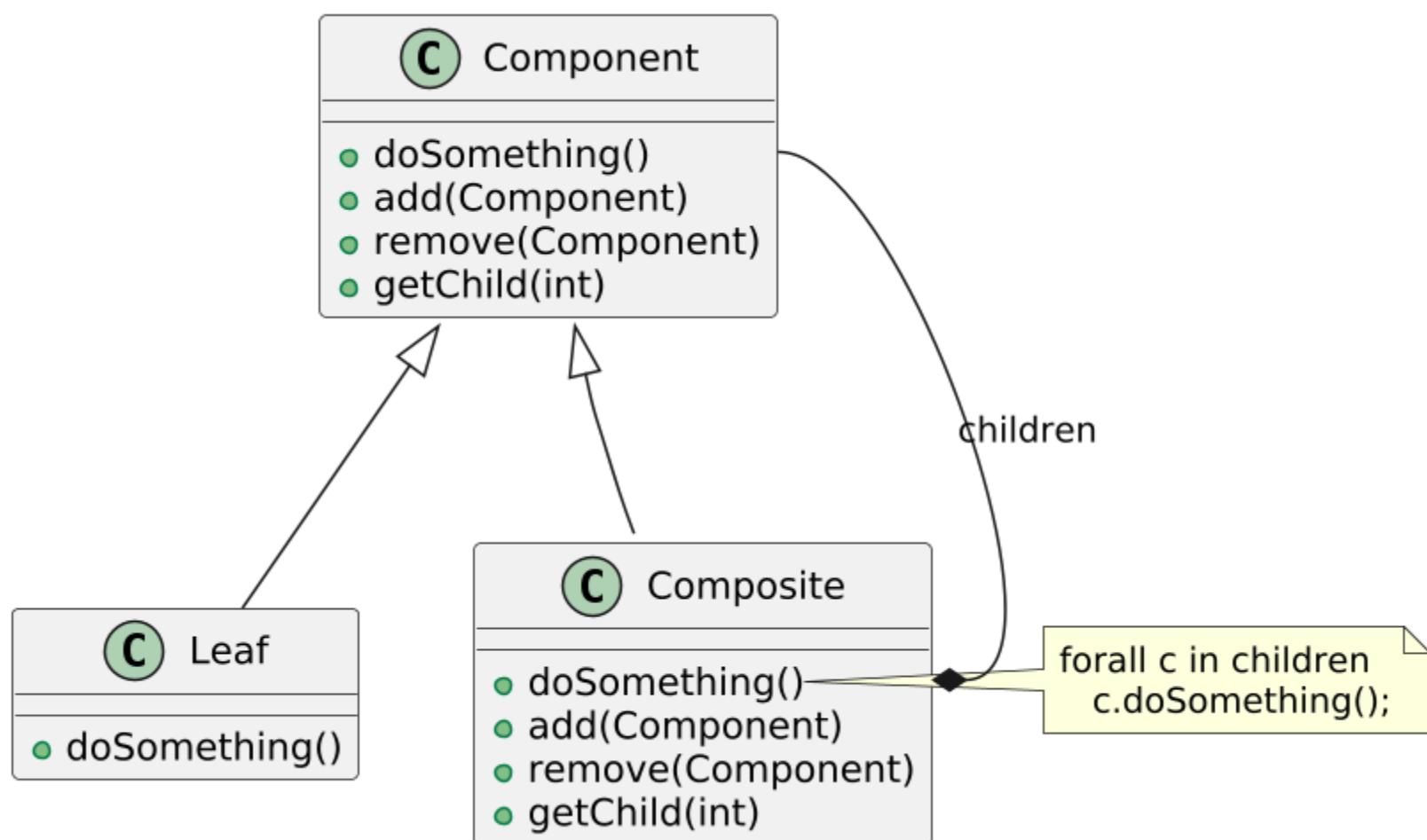
Marktplatz

Marienplatz

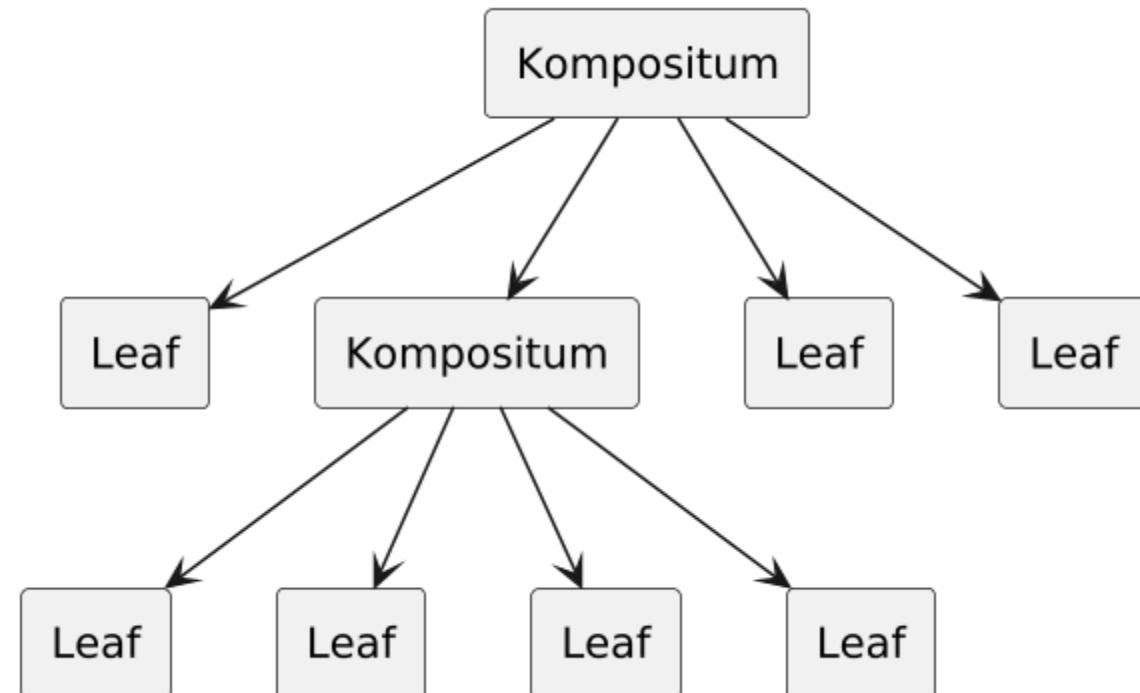
# Kompositum: Beispiel UML



# Kompositum: UML (allgemein)



# Kompositum: Beziehungen der Instanzen



# Kompositum: Vorteile

- einheitliche Schnittstelle für komplexe Strukturen
  - höhere Abstraktion für Datenstruktur
- leichte Erweiterbarkeit neuer Klassen
  - insbesondere Blatt-Klassen
- sehr gute Anwendbarkeit von rekursiven Algorithmen

# Kompositum: Nachteile

- verfügbare Schnittstelle entspricht dem kleinsten gemeinsamen Nenner aller beteiligten Klassen
- zu allgemeine Struktur erschwert den Umgang
  - notwendige Downcasts sind ein Hinweis hierauf
- folgenreiche Operationen sind sehr einfach
  - `rm -rf /`

# Weiterführende Bücher

