Grundlagen der Objektorientierung

Entwurfsmuster
(Design Patterns)

Was ist ein Entwurfsmuster?

- Bewährte Schablone für ein Entwurfsproblem
- wiederverwendbare Vorlage zur Problemlösung

Historie

- 1988: Erich Gamma überträgt Entwurfsmuster aus der Architektur in die Softwareentwicklung.

- 1995: Die sog. ***Gang of Four (GoF)*** (Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides) veröffentlicht ein Buch mit 23 Entwurfsmustern: *„Design Patterns - Elements of Reusable Object-Oriented Software“*

Klassifikation von Entwurfsmustern

Gof: klassifiziert Muster nach den Kriterien:

1. Zweck (*purpose*) und
2. Anwendungsbereich, auf den sie wirken (*scope*).

Kriterium *Zweck*:

- *Erzeugungsmuster* - Erzeugung von Objekten.
- *Strukturmuster* - sollen eine Vereinfachung der Struktur zwischen Klassen ermöglichen.
- *Verhaltensmuster* - das Verhalten der Klassen. Sie beziehen sich auf die Zusammenarbeit und den Nachrichten-austausch von Klassen.

Klassifikation von Entwurfsmustern

Kriterium *Anwendungsbereich*:

- *Klassenbasierte Muster*
  - ✓ beschreiben Beziehungen zwischen Klassen
  - ✓ bauen vorrangig Vererbungsstrukturen auf.
  - ✓ Die Strukturen sind damit zur Übersetzungszeit festgelegt

- *Objektbasierte Muster*
  - ✓ nutzen vorrangig Assoziationen und Aggregationen zur Beschreibung von Beziehungen zwischen Objekten.
  - ✓ Die durch sie beschriebenen Strukturen zwischen Objekten sind zur Laufzeit dynamisch änderbar.

Dokumentation von Entwurfsmustern (GoF)

1. Name und Klassifikation des Musters.
2. Zweck des Musters.
3. Synonyme: Andere bekannte Namen des Musters.
4. Motivation: (Hinter-)Gründe für den Einsatz des Musters.
5. Anwendbarkeit: Einsatzbereiche für das Muster.
6. Struktur: Beschreibung der allgemeinen Struktur des Musters.
7. Beteiligte Akteure: Klassen, die an dem Muster beteiligt sind.
8. Zusammenspiel der beteiligten Klassen.
9. Konsequenzen: Welche Vor- und Nachteile gibt es?
10. Implementierung: Praxisrelevante Tipps, Tricks und Techniken sowie Warnung vor Fehlern, die leicht passieren können.
11. Beispielcode: Quellcodefragment, das den Einsatz des Musters zeigt.
12. Praxiseinsatz: Wo wird das Muster bereits eingesetzt?
13. Querverweise: Wie spielt das Muster mit anderen Mustern zusammen?

Erzeugungsmuster (creational patterns)

helfen, ein System davon unabhängig zumachen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden.

- Ein klassenbasiertes Muster verwendet **Vererbung** zur Variation des zu erzeugenden Objekts
  - ✓ **Fabrikmethode** (***Factory Method, Virtual Constructor***)
- Ein objektbasiertes Muster **delegiert** die Erzeugung an ein anderes Objekt.
  - **Abstrakte Fabrik** ***(Abstract Factory, Kit)***
  - **Erbauer** ***(Builder)***
  - **Prototyp** ***(Prototype)***
  - ✓ **Einzelstück** ***(Singleton)***
  - ✓ **Multiton** ***(Verwaltung mehrere Singletons)***

Fabrikmethode (*factory method, Virtual Constructor*)

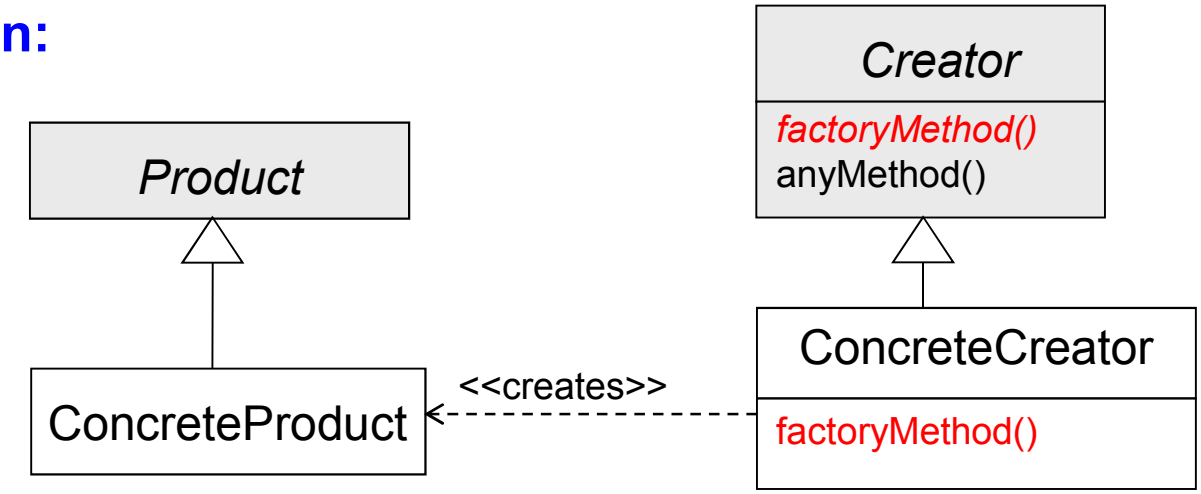
Es bietet eine Schnittstelle zum Erzeugen eines Objekts an, wobei die Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist.

Anwenden, wenn:

- eine Klasse die von ihr zu erzeugenden Objekte **nicht** im voraus kennen kann.
- eine Klasse benötigt wird, deren Unterklassen selbst festlegen, welche Objekte sie erzeugen.

Fabrikmethode (*factory method, Virtual Constructor*)

Allgemein:



Creator	definiert die Fabrikmethode, welche ein Produkt erzeugt
ConcreterCreator	überschreibt die Fabrikmethode zur Erzeugung konkreter Produkte
Product	definiert die Schnittstelle für das Produkt, welches die Fabrikmethode erzeugt
ConcreteProduct	definiert ein konkretes Produkt durch Implementierung der Schnittstelle und wird durch den korrespondierenden konkreten Erzeuger erzeugt

Fabrikmethode (*factory method, Virtual Constructor*)

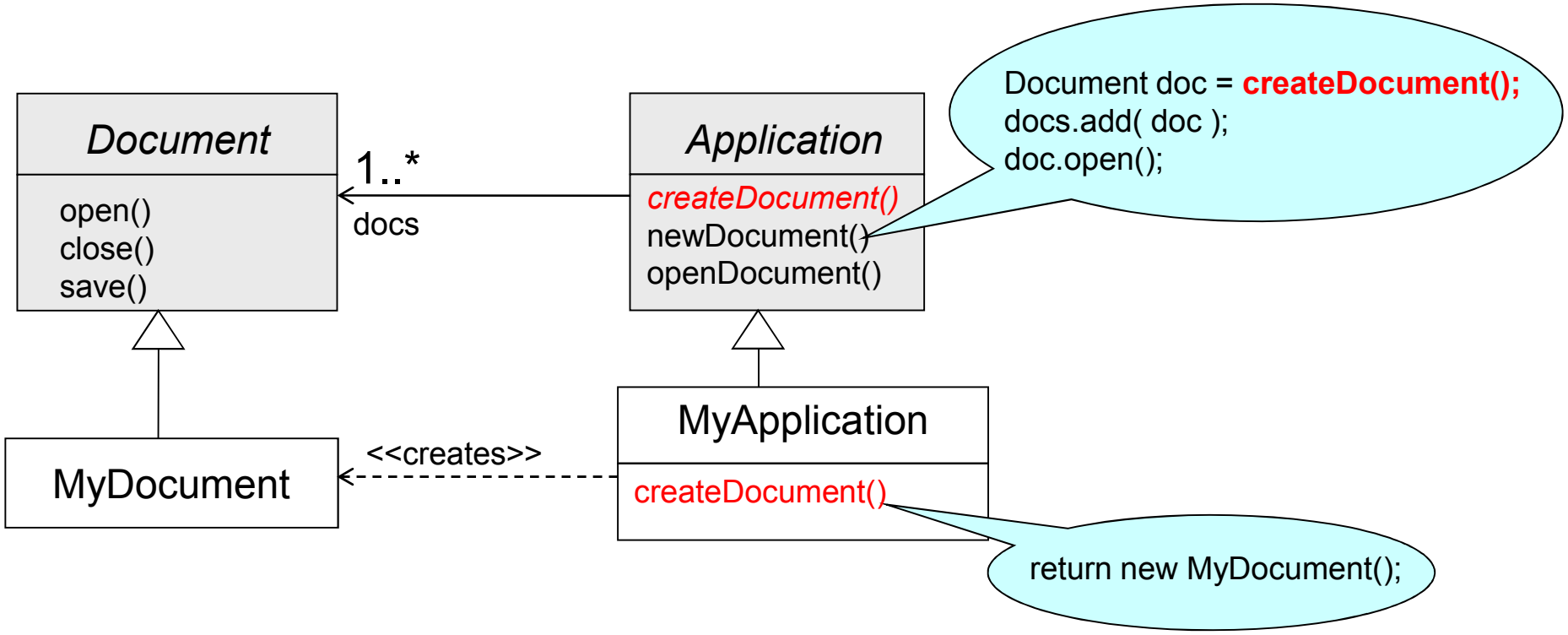
Beispiel:

- Framework für Anwendungen, die mehrere Dokumente gleichzeitig anzeigen können.
- Es verwendet die abstrakten Klassen *Document* und *Application* und modelliert eine Assoziation zwischen ihren Objekten.
- Die Klasse *Application* ist für die Erzeugung neuer Objekte zuständig.

Fabrikmethode (*factory method, Virtual Constructor*)

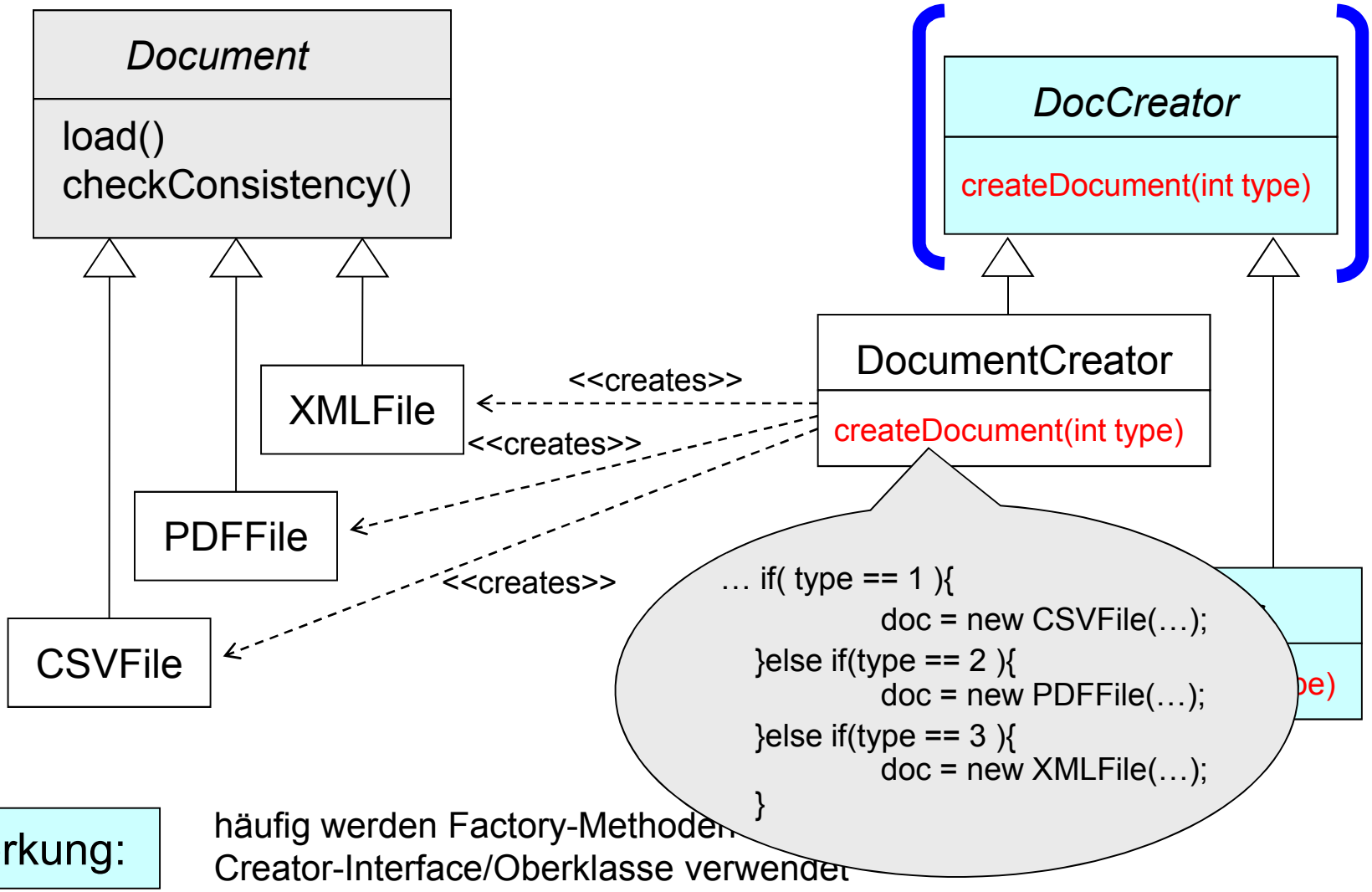
Beispiel:

Wenn *MyApplication* ein neues Objekt von *MyDocument* erzeugen soll, muss das Framework Objekte erzeugen, kennt aber nur die abstrakte Oberklasse (keine Instanzierung möglich) ➡ **Überschreiben der Fabrikmethode**



Fabrikmethode (*factory method, Virtual Constructor*)

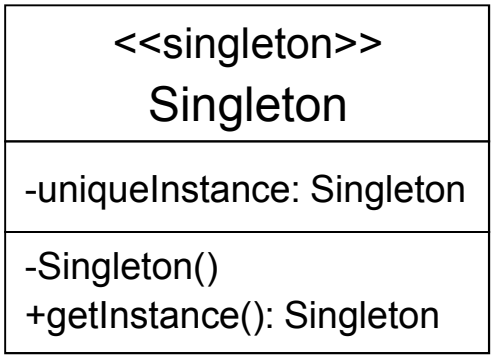
Beispiel:



Bemerkung: häufig werden Factory-Methoden in einer Creator-Interface/Oberklasse verwendet

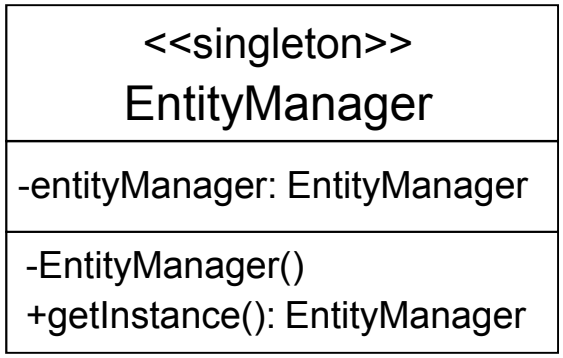
Singleton (Einzelstück)

Es stellt sicher, dass eine Klasse genau ein Objekt besitzt und ermöglicht den globalen Zugriff auf dieses Objekt



- erzeugt und verwaltet einziges Objekt zu einer Klasse
- globaler Zugriff auf dieses Objekt über Instanzoperation
- die Instanzoperation ist eine Klassenmethode, d.h. **statisch** gebunden

Beispiel:

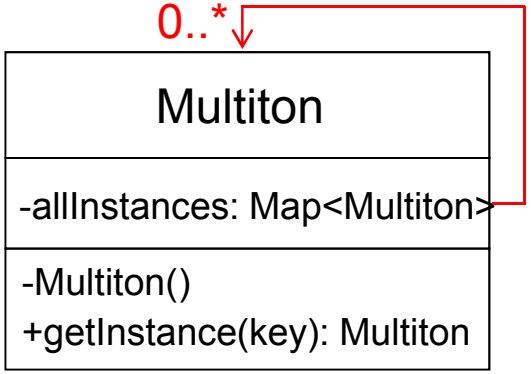


Realisierung in Java:

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

Multiton (Verwaltung mehrerer Einzelstücke)

Es stellt sicher, dass eine Klasse genau ein Objekt mit einem bestimmten Schlüssel besitzt und ermöglicht den globalen Zugriff auf dieses Objekt



- erzeugt und verwaltet Objekte zu einer Klasse
- bietet globalen Zugriff auf diese Objekte über eine Instanzoperation und zugehörigem Schlüssel
- die Instanzoperation ist eine Klassenmethode, d.h. **statisch** gebunden

Multiton (Verwaltung mehrerer Einzelstücke)

Realisierung in Java:

```
public class Multiton {
        private static Map<Object, Multiton> allInstances = new HashMap<Object, Multiton>();
        private Multiton() {}
        public synchronized static Multiton getInstance(Object key) {
            Multiton instance = allInstances.get(key);
            if (instance == null) {              // Lazy Creation, falls keine Instanz gefunden wurde
                    instance = new FooMultiton();
                    allInstances.put(key, instance);
            }
            return instance;
        }
}
```

Einschränkung: es wird jedes Mal eine Instanz erstellt, auch wenn der Schlüssel nicht existiert, d.h. es wird nie **null** geliefert.

Alternative: *getInstance(Object key, boolean create)* o.ä.
(=> Objekt nur erzeugen, wenn create=true)

Strukturmuster (structural patterns)

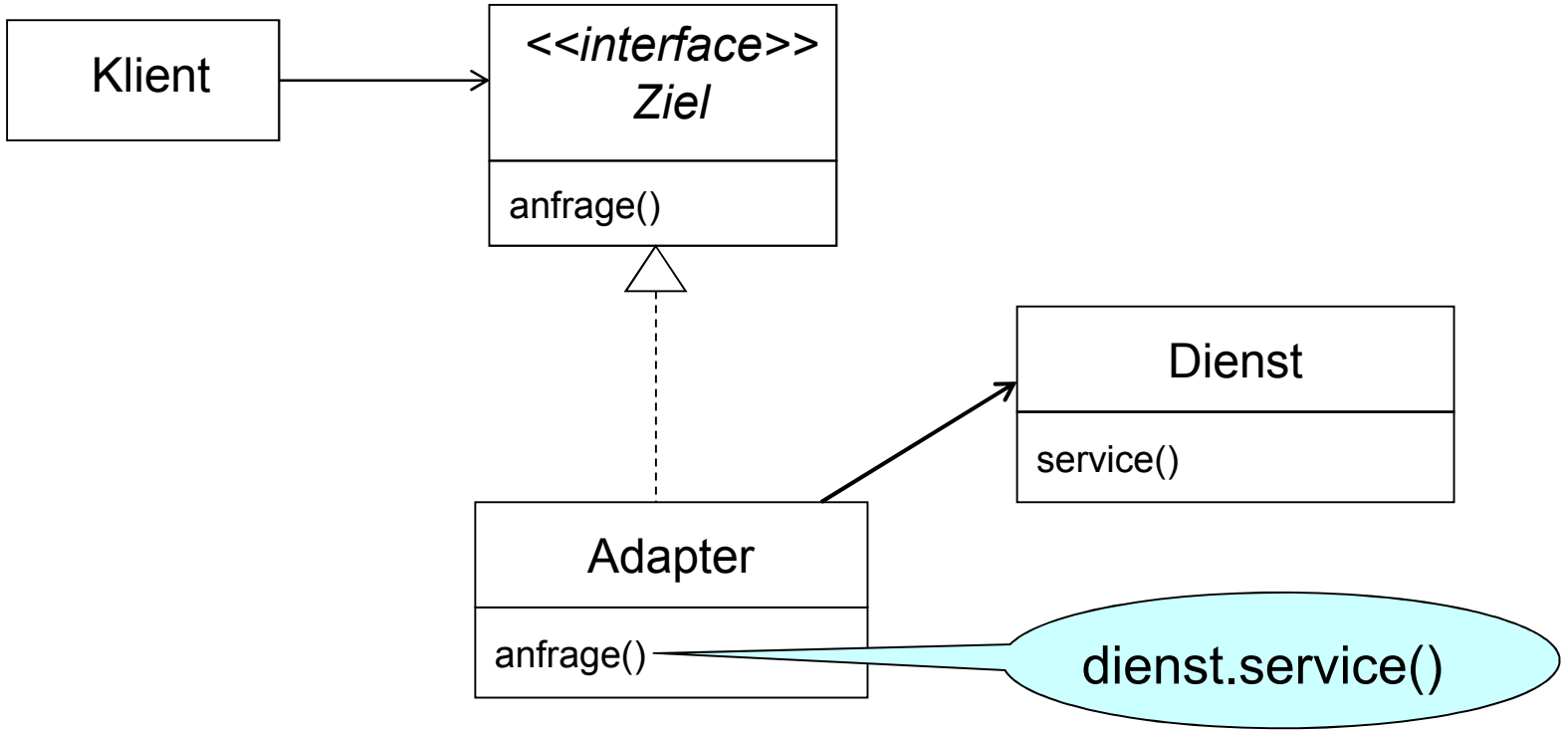
befassen sich damit, wie Klassen und Objekte zu größeren Strukturen zusammengesetzt werden.

- Ein klassenbasiertes Muster verwendet **Vererbung**, um Schnittstellen und Implementierungen zusammenzuführen. (Bsp. Mehrfachvererbung)
  - ✓ **Adapter** (***Adapter, Wrapper***)(***Ad. mit Vererbung oder Klassenadapter***)
- Ein objektbasiertes Muster beschreibt Wege, Objekte zusammenzuführen, um neue Funktionalität zu gewinnen.
  - ✓ **Adapter** ***(Adapter, Wrapper) (Ad. mit Assoziation oder Objektadapter).***
  - • **Brücke** ***(Bridge, Handle/Body)***
  - ✓ **Kompositum** ***(Composite)***
  - • **Dekorierer** ***(Decorator, Wrapper)***
  - ✓ **Fassade** ***(Facade)***
  - • **Fliegengewicht** ***(Flyweight)***
  - • **Stellvertreter** ***(Proxy, Surrogate)***

Adapter (*Adapter, Wrapper*)

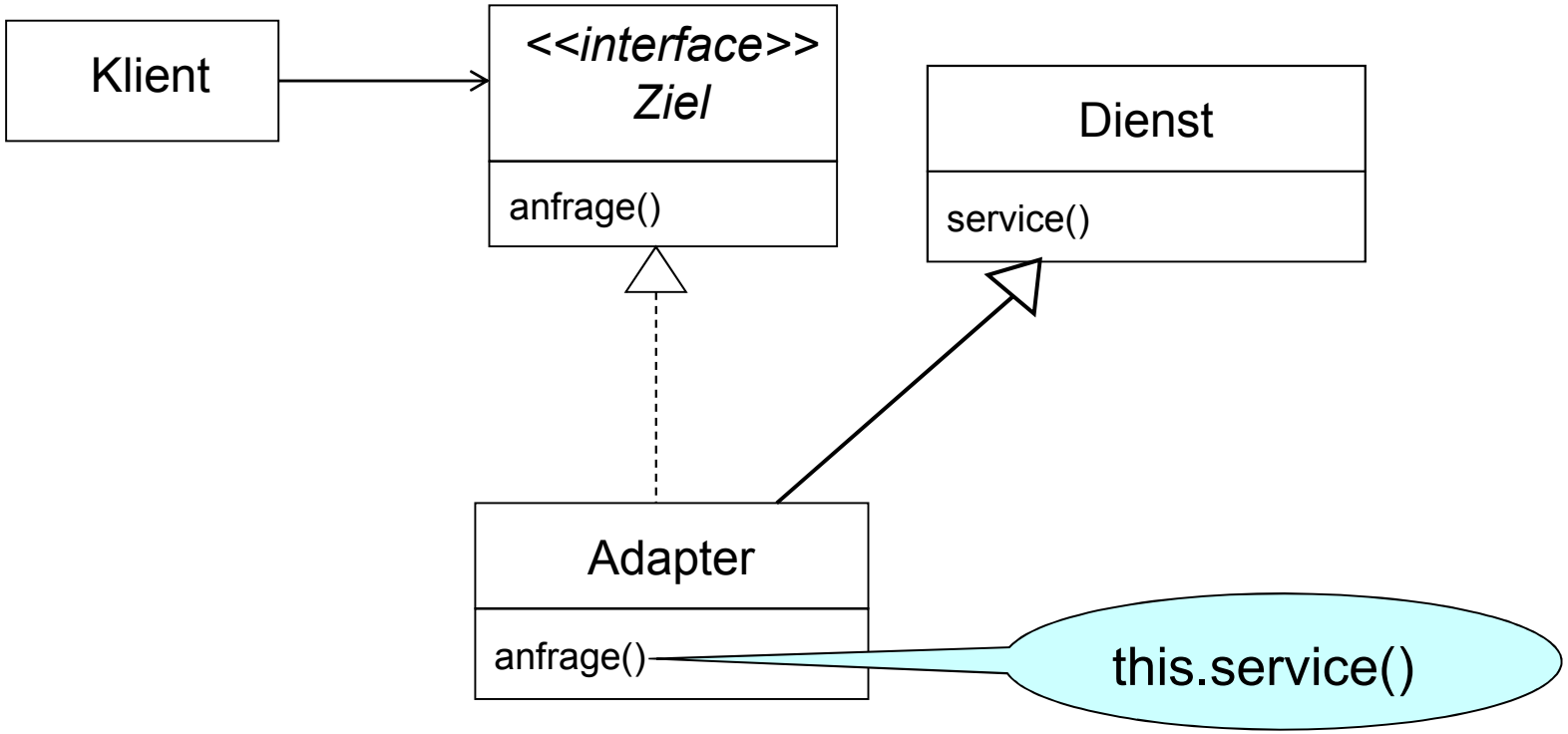
Ziel: Wiederverwendbarkeit einer existierenden Klasse (Dienst), deren Schnittstelle nicht der benötigten Schnittstelle (Ziel) entspricht

1. Adapter mit Delegation (Objektadapter):



Adapter (*Adapter, Wrapper*)

2. Adapter mit Vererbung (Klassenadapter):



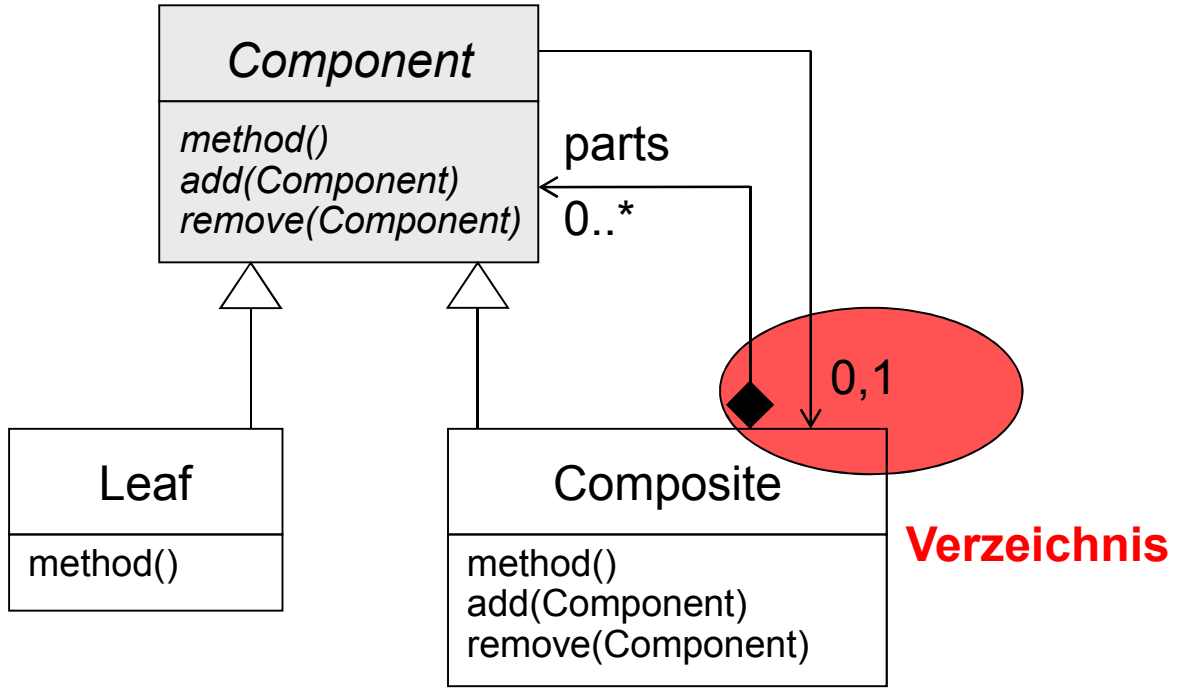
Kompositum (*Composite*)

- Es setzt Objekte zu Baumstrukturen zusammen, um *Teil-Ganzes*-Hierarchien darzustellen.
- Es ermöglicht es, sowohl einzelne Objekte als auch einen Baum von Objekten einheitlich zu behandeln..

Verzeichniseintrag

Struktur:

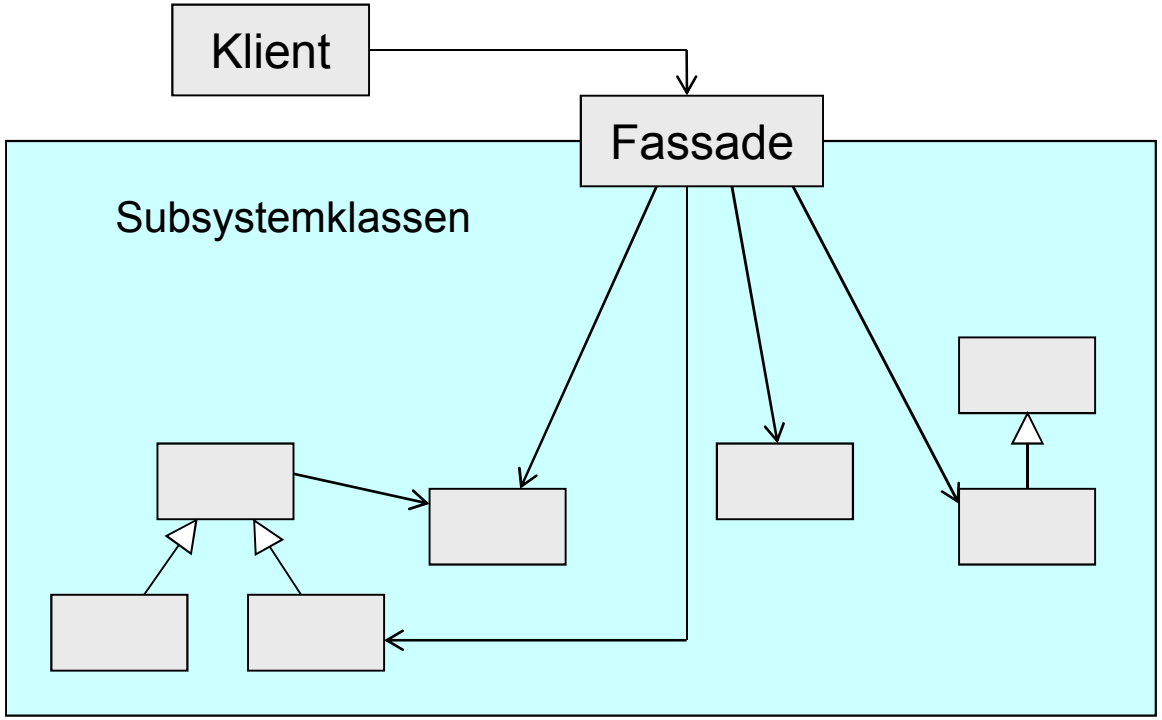
Beispiel:
Dateien und Verzeichnisse



Fassade

- bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems.
- Auf Subsystemklassen kann trotzdem noch zugegriffen werden

Struktur:



Verhaltensmuster (behavioral patterns)

- Ein objektbasiertes Verhaltensmuster verwendet Aggregation bzw. Komposition anstelle von Vererbung, um das Verhalten unter den Klassen zu verteilen.
  - **Zuständigkeitskette *(Chain of Responsibility)***
  - **Kommando *(Command, Action, Transaction)***
  - **Iterator *(Iterator, Cursor)***
  - **Vermittler *(Mediator)***
  - ✓ **Beobachter *(Observer, Dependents, Publish-Subscribe, Listener)***
  - ✓ **Memento *(Memento, Token)***
  - **Zustand *(State, Objects for States)***
  - **Strategie *(Strategy, Policy)***
  - **Besucher *(Visitor)***
  - **Plugin *(Plugin)***

Beobachter (*Listener, Observer*)

- Es sorgt dafür, dass bei der Änderung eines Objekts alle davon abhängigen Objekte benachrichtigt und automatisch aktualisiert werden bzw. auf diese Benachrichtigung reagieren.

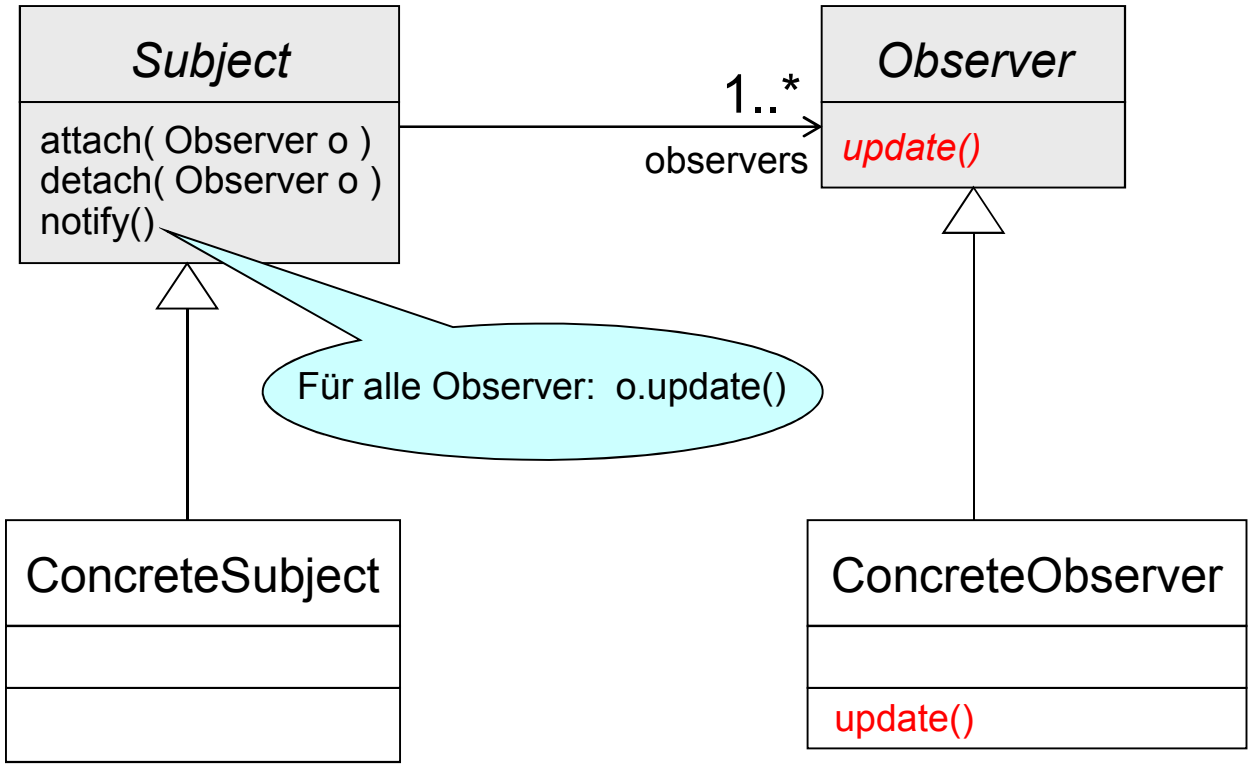
Anwenden, wenn:

- eine Abstraktion zwei Objekte besitzt, bei der eines vom anderen abhängt. Die Kapselung in zwei Objekte ermöglicht es, sie unabhängig voneinander wieder zu verwenden oder zu modifizieren.
- eine Abstraktion zwei Objekte besitzt, die wechselseitig voneinander abhängen.
- die Änderung eines Objekts die Änderung anderer Objekte impliziert und es nicht bekannt ist, wie viele Objekte geändert werden müssen.
- ein Objekt andere benachrichtigen soll und diese Objekte nur lose miteinander gekoppelt sind.

Beobachter (*Listener, Observer*)

Struktur mit abstrakten Oberklassen (1):

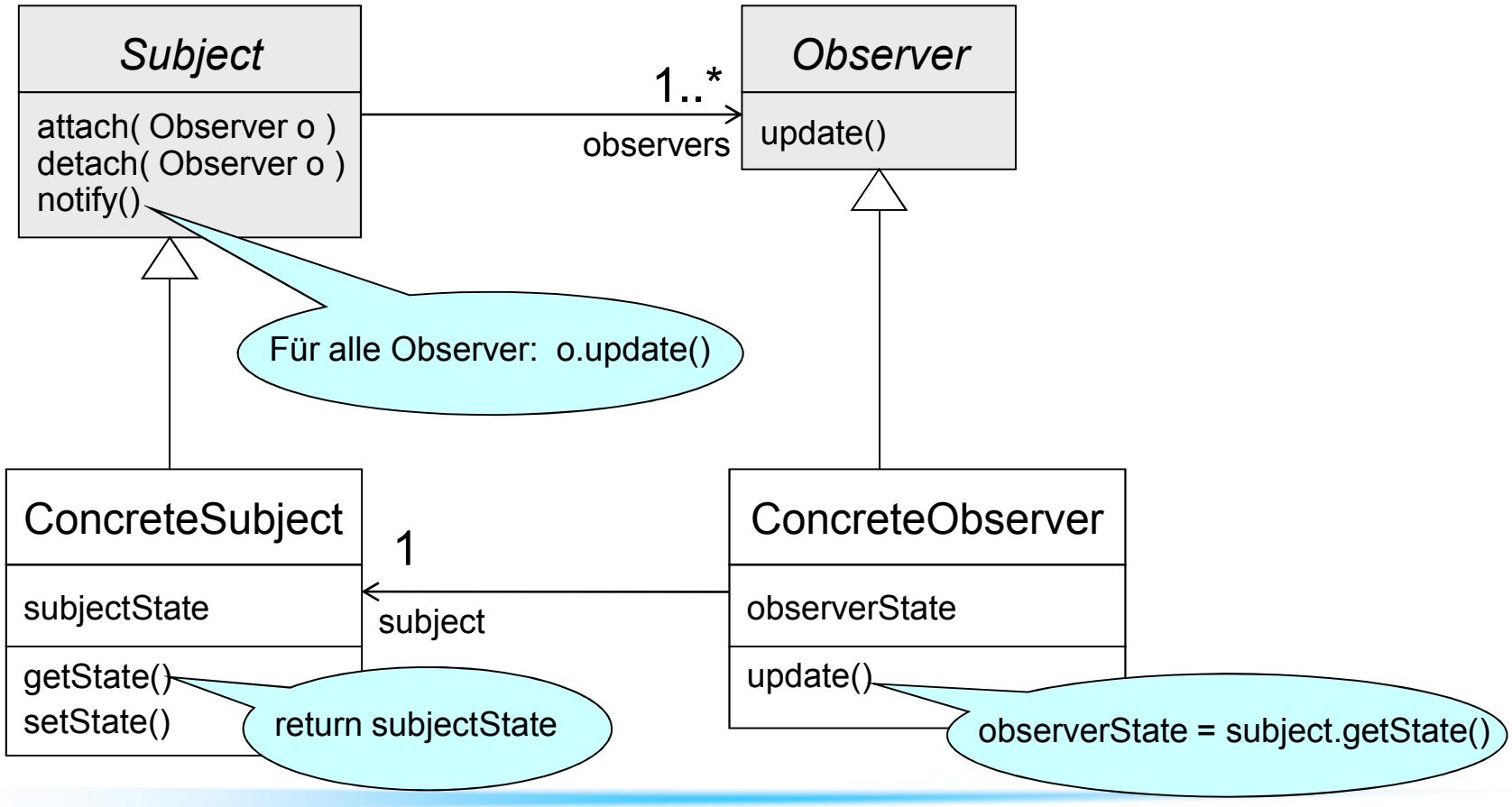
- „Passiver“ Observer: Observer reagiert nur, d.h. wird nur benachrichtigt
- Beide konkrete Unterklassen wissen nichts voneinander



Beobachter (*Listener, Observer*)

Struktur mit abstrakten Oberklassen (2):

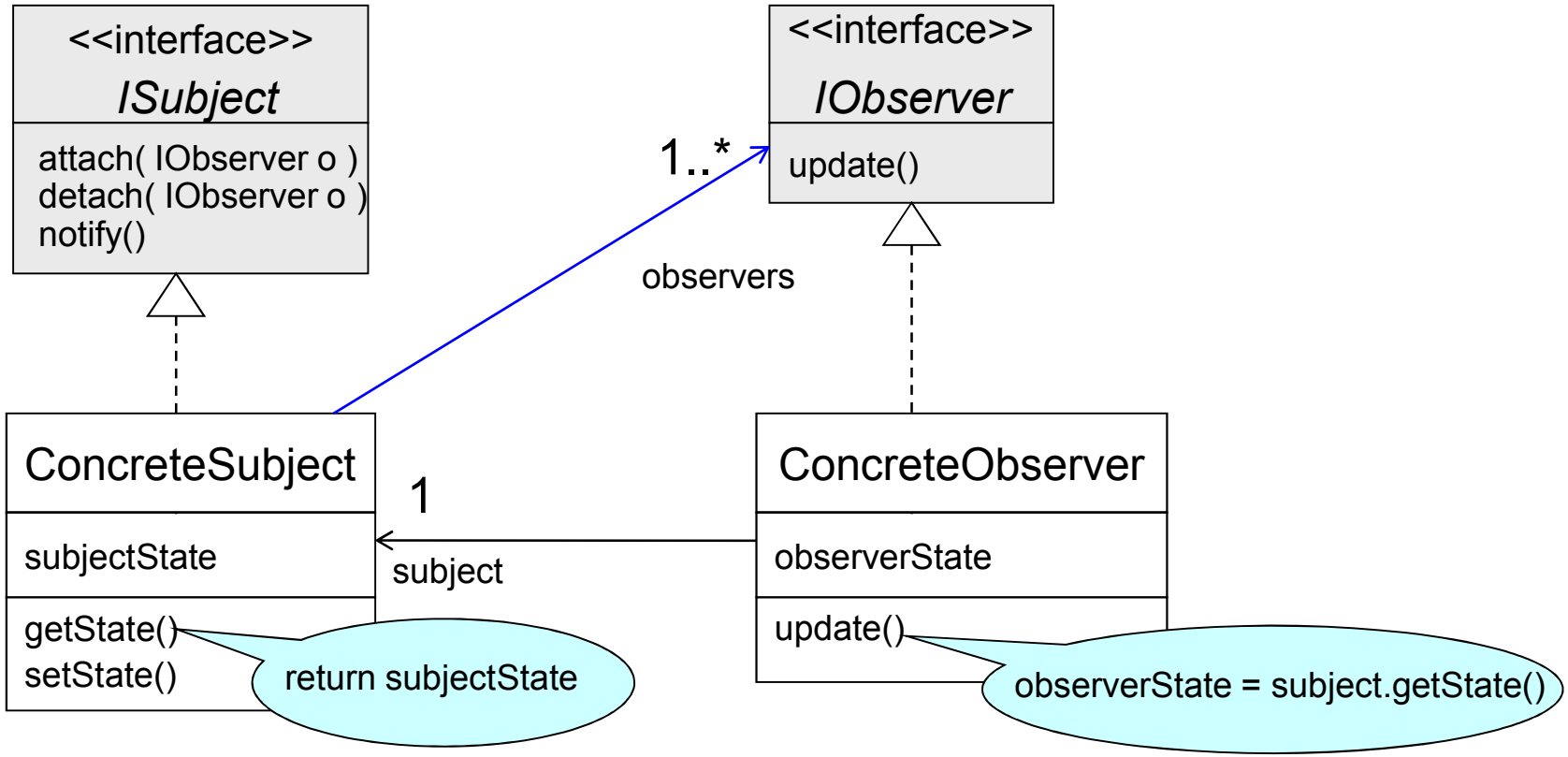
- „Aktiver“ Observer: Observer fragt zusätzlich den Zustand des Subjects ab
- Observer kennt das konkrete *Subject*



Beobachter (*Listener, Observer*)

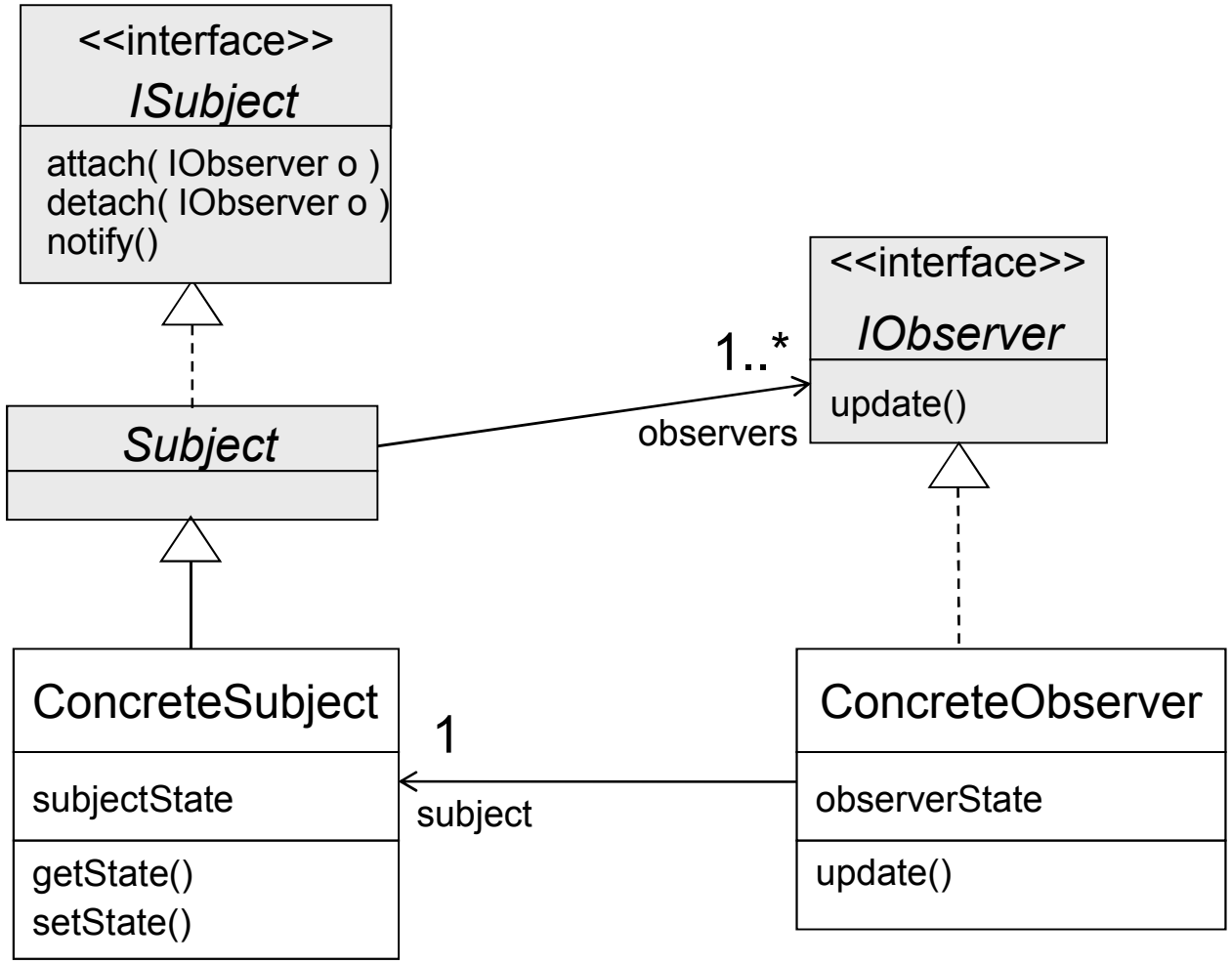
Struktur mit Interfaces:

- ohne Oberklasse muss jedes *Subject* eigene Referenzen halten
- Jeder aktive Observer muss das konkrete Subject kennen



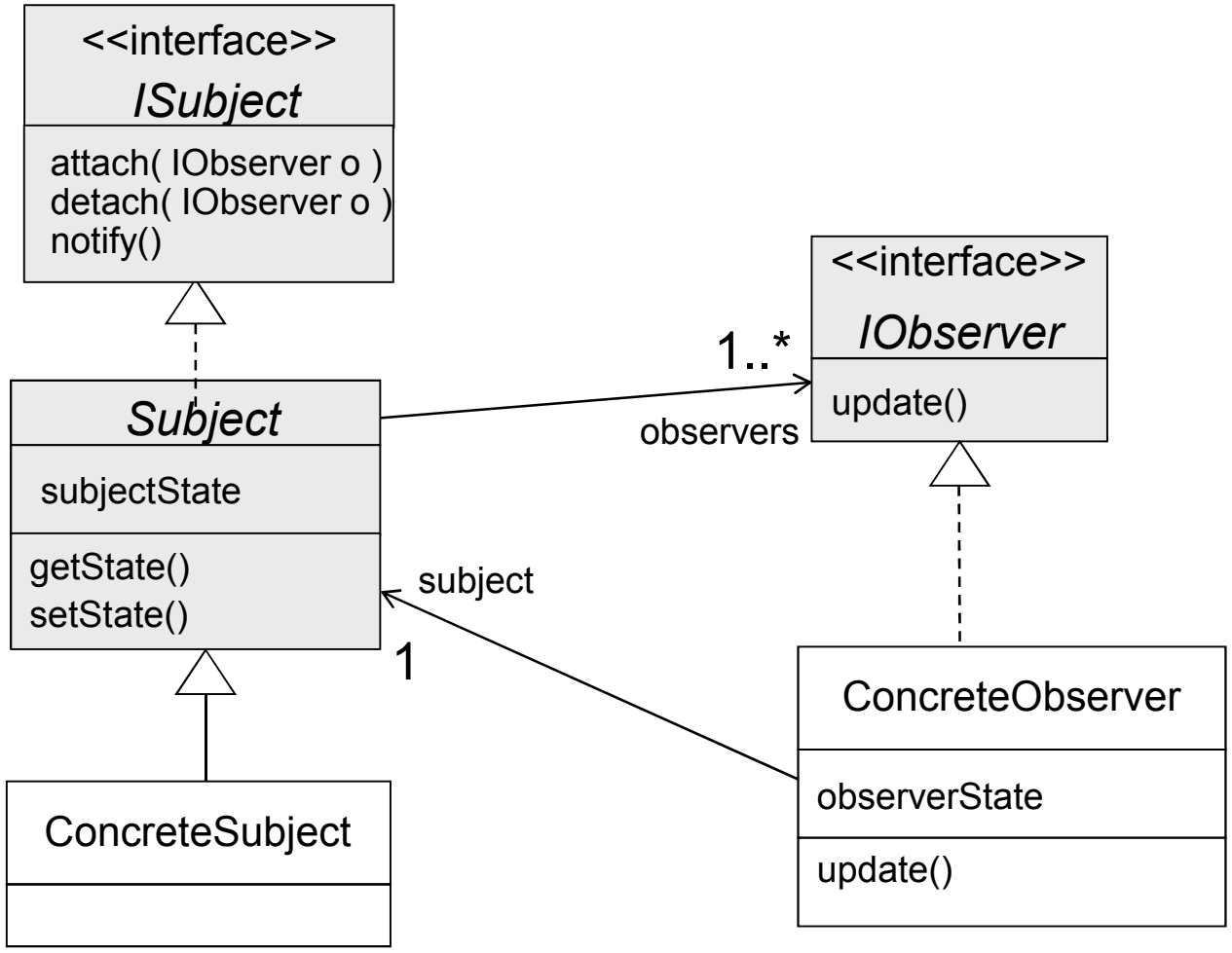
Beobachter (*Listener, Observer*)

Struktur mit Interfaces und Oberklasse:



Beobachter (*Listener, Observer*)

Noch bessere Struktur mit Interfaces und Oberklasse:



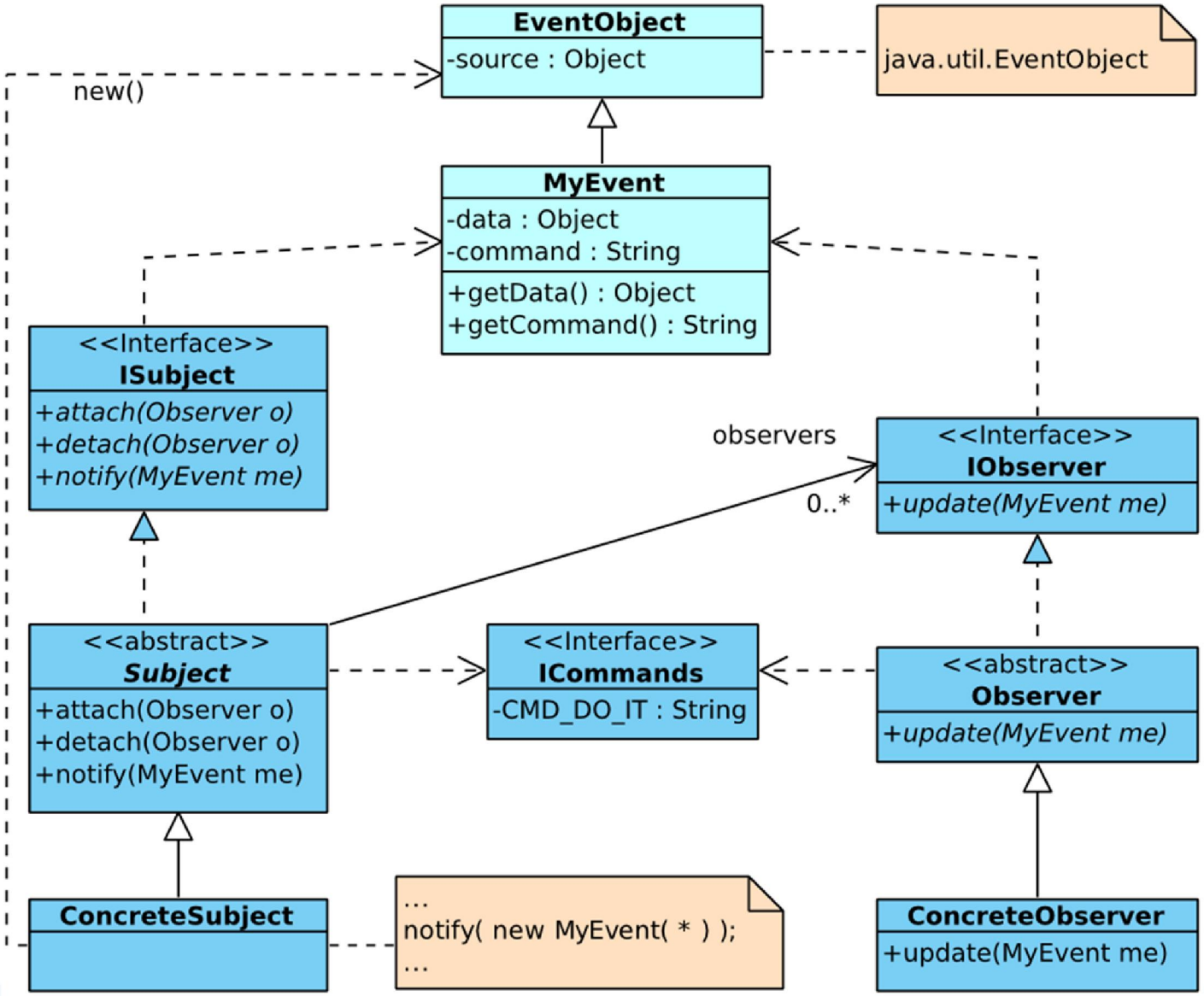
Einschränkungen aktiver Observer

- Aktiver Observer muss das *Subject* kennen (referenzieren)
- Es kann jeweils nur ein *Subject* beobachtet werden

Verbesserungen / Erweiterungen:

- Kommunikation über Events (*Subject* als Quelle referenzieren) plus Möglichkeit, Daten anzuhängen
  - Über Event kann Quell-*Subject* erkannt werden
  - Es können viele *Subjects* beobachtet werden
  - Über „neutrale“ *Commands* festlegen, was vom Observer getan werden soll

Beobachter (*Listener, Observer mit Events*)



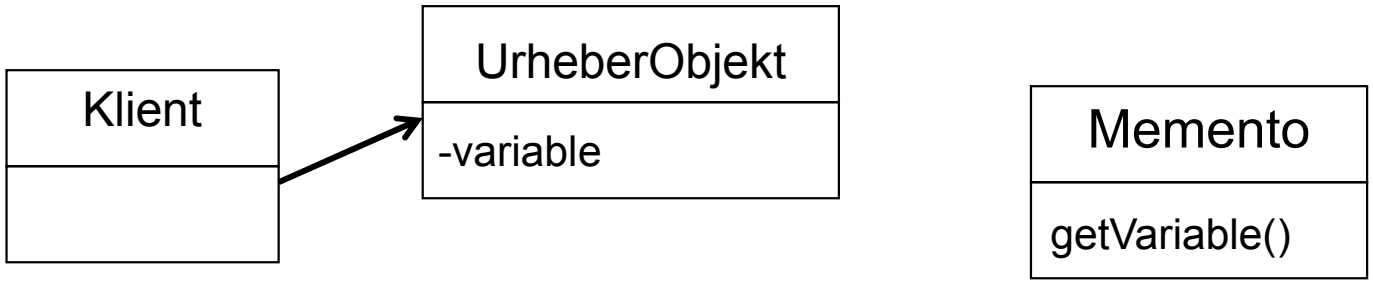
Memento (*Memento, Token*)

- Es dient dazu, den internen Zustand (=Variablenbelegung) eines Objektes festzuhalten und ggf. das Objekt später mit den festgehaltenen Daten zu belegen.
- Verwendung beispielsweise für die Realisierung von UNDO-Operationen.
- Die Gestaltung der Schnittstellen ist hier sehr wichtig!

Beispiel:

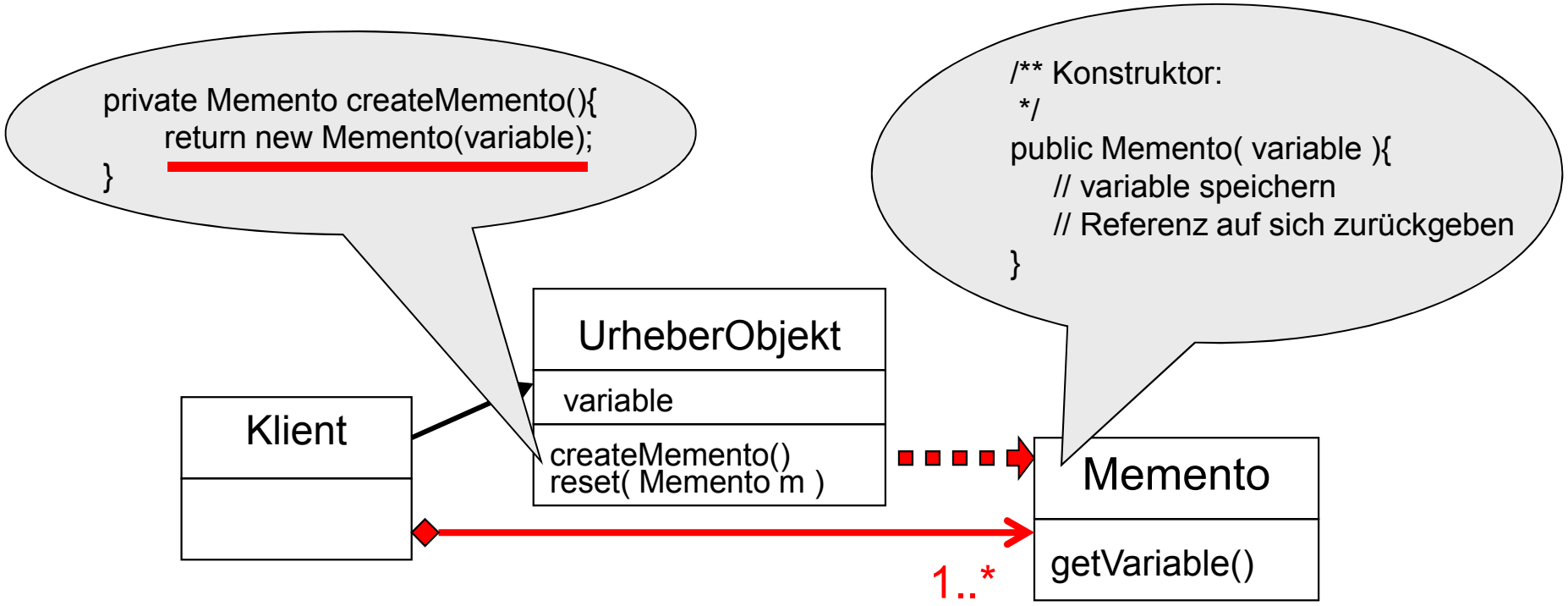
Der Klient möchte den Zustand des *UrheberObjektes* speichern, hat aber keinen Zugriff auf die Variablen, da er lediglich eine Referenz auf dieses Objekt besitzt.

Um dem Klienten jedoch die Verwaltung des gespeicherten Zustandes zu ermöglichen, ohne die Kapselung zu zerstören, erweitert man die Klassenhierarchie um eine zusätzliche Klasse, dem *Memento*:



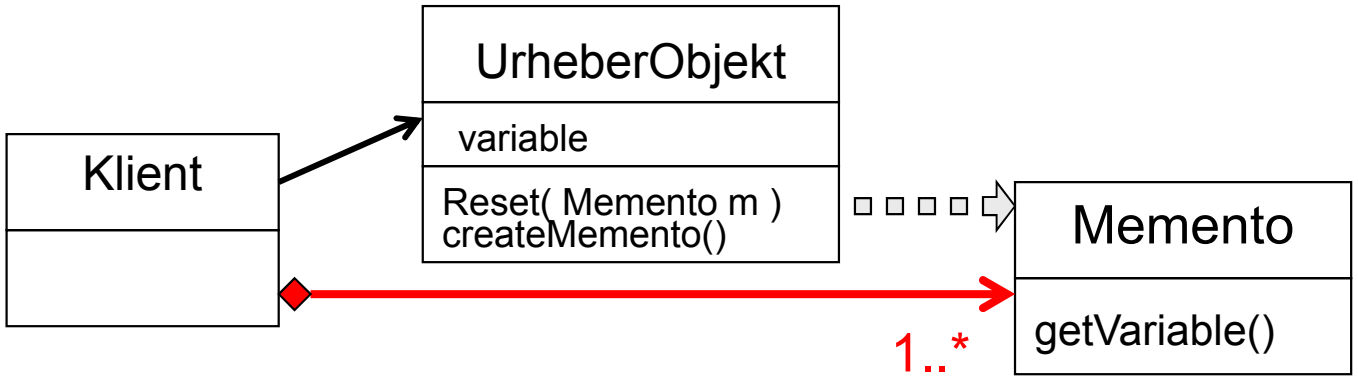
Memento (*Memento, Token*) (2)

1. Der Klient fordert vom *UrheberObjekt* ein Memento an (*createMemento()*).
2. *createMemento()* ruft den Konstruktor des Mementos auf und übergibt diesem seinen Zustand (=Variablenbelegung).
3. So erhält der Klient über das *UrheberObjekt* eine Referenz auf das fertige Memento und verwaltet dies nach Belieben (evtl. auch mehrere)



Memento (*Memento, Token*) (2)

4. Der Klient kann das Objekt (auch öfter) anweisen, den Zustand eines Mementos (das der Klient nennen muss) anzunehmen, indem er *reset(m)* aufruft.
5. Der Klient hat jederzeit die Möglichkeit, das Memento zu verwerfen, da dem *UrheberObjekt* das Memento nicht bekannt ist, es ist lediglich sein Erzeuger.
6. Die Kapselung wird somit beibehalten!



Verhaltensmuster (behavioral patterns)

- Beziehen sich auf die **Interaktion** zwischen Klassen und Objekten.
- beschreiben Kontrollflüsse, die zur Laufzeit schwer nachvollziehbar sind.
- lenken die Aufmerksamkeit weg vom Kontrollfluss hin zu der Art und Weise, **wie** die Objekte interagieren.

- Ein klassenbasiertes Verhaltensmuster verwendet **Vererbung**, um das Verhalten unter den Klassen zu verteilen.
  - **Interpreter *(Interpreter)***
  - **Schablonenmethode *(Template Method)***