Design Patterns

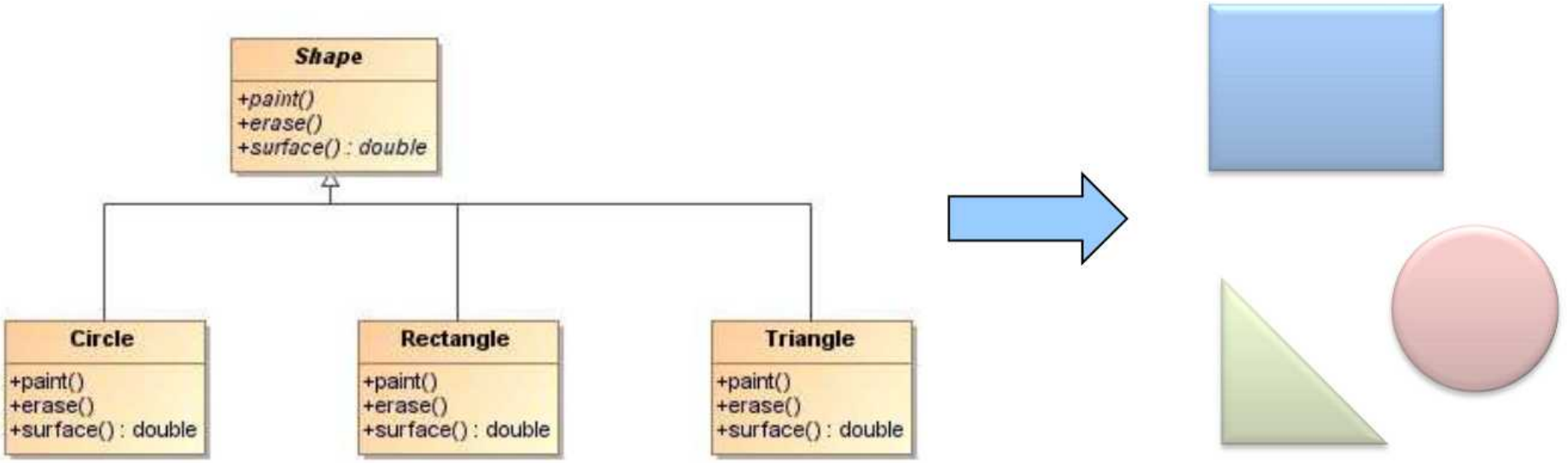
Entwurfsmuster

Tobias Mayr und Matthias Wallnöfer
(*) Einige Inhalte: Stefan Podlipnig

Wozu Patterns?

- Lösungen zu immer wieder auftretenden Situationen
  - *Das Rad nicht immer neu erfinden*
  - Daher bewährt und erprobt (im Englischen *"battle-proven"*)
- Bieten dem Entwickler Anhaltspunkte zur Umsetzung
  - Wie man es **nicht** machen sollte: Anti-Patterns
- Fördern wiederverwendbaren Code
- Hohe Lese-/Wartbarkeit
  - Einfacher Überblick für sich selbst und andere
  - Pattern erkannt → Code durchschaut! (..naja, fast)

Konkretes Beispiel

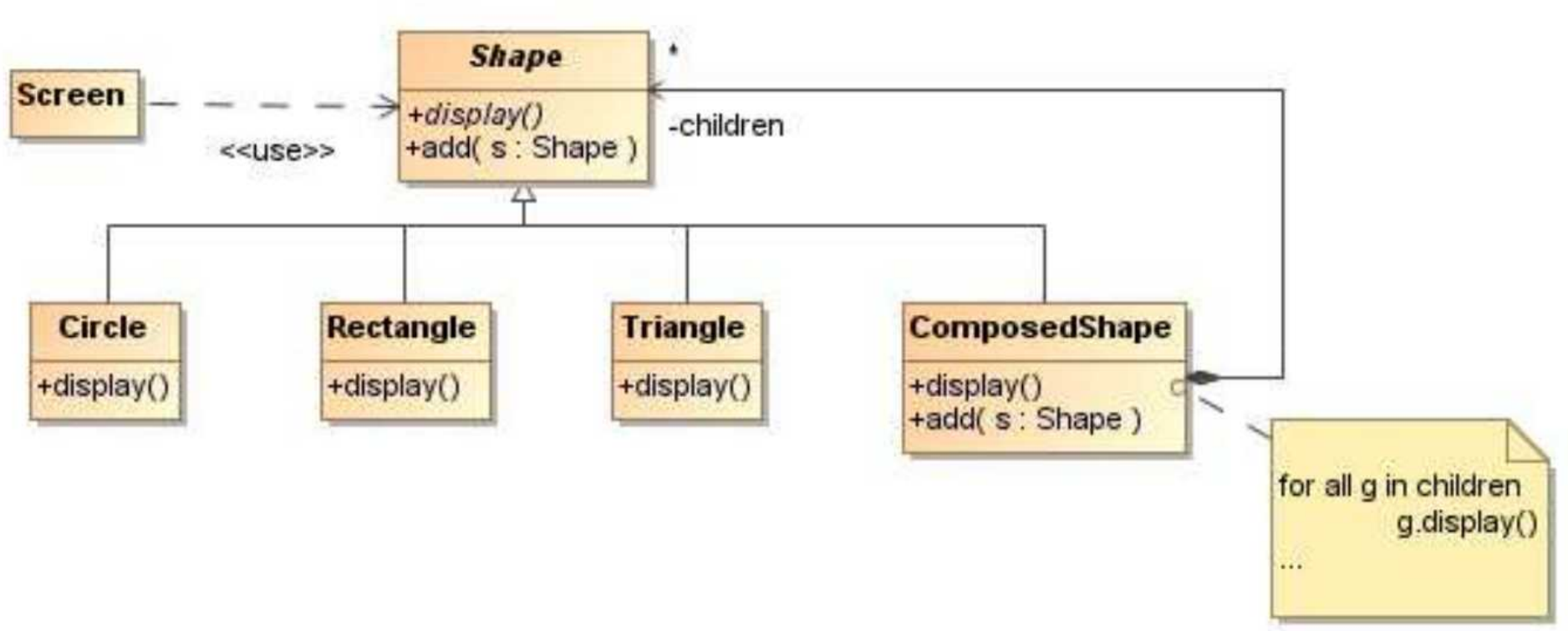


Und wie modelliert man das?

*

Mögliche Lösung

- Heißt **Compositum**

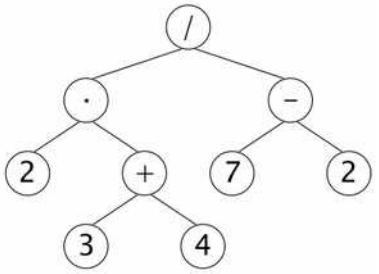


*

Das hatten wir doch schon?

Beispiel: Mathematische Ausdrücke

$$\frac{2 \cdot (3+4)}{7-2}$$

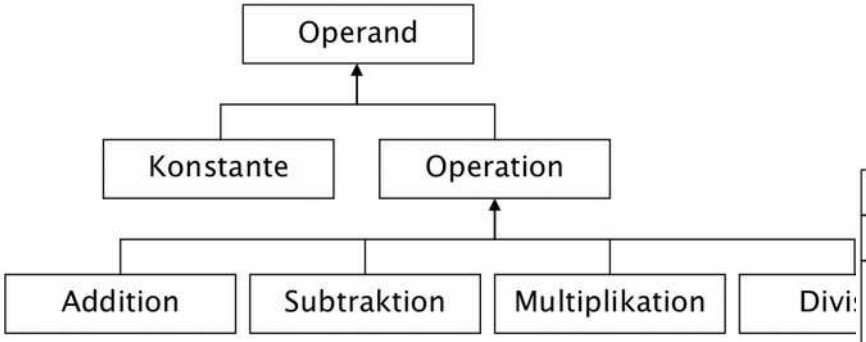


Konstante z. B. 2, 3, 4

Operand kann Konstante oder Operation sein

Operation besteht aus zwei Operanden, hat Ergebnis
kann Addition, Subtraktion, Multiplikation oder Division sein

Konstanten, Operanden und Operationen liefern ein Ergebnis



Operand
getErgebnis()

Konstante
double ergebnis
Konstante(**double**) Konstante() setErgebnis(**double**) **double** getErgebnis() String toString()

Operation
Operand[] operand
Operation(Operand, Operand) Operation() setOperand(Operand) Operand getOperand(**int**) vertausche() loescheOperand(**int**)

Addition
Addition(Operand, Operand) Addition() **double** getErgebnis() String toString()

Wiederverwendung

(aus "Entwurfsmuster von Kopf bis Fuß")

Nutzung von Schnittstellen/Oberklassen

- Objekte so weit es geht mit generischen Referenzen ansprechen → Polymorphismus
- Bsp. *Mathem. Ausdrücke*: Operation und Konstante **Operand**, von dem sich *Ergebnis* ermitteln lässt.
- Bsp. *Listen*: Durch Nutzung von SS **List** einfacher Wechsel zwischen **ArrayList** und **LinkedList** möglich:

```
List<String> l = new ArrayList<String>(10);
List<String> l = new LinkedList<String>();
```

Wiederverwendung

(aus "Entwurfsmuster von Kopf bis Fuß")

Vererbung (**White-Box-Reuse**)

- Problem der instabilen Basisklasse
- Bsp. *Java*: Man möchte sich eigene *ArrayList* programmieren
  - Ableitung von **ArrayList**
  - Überschreiben der relevanten Methoden
  - Neuere Java-Versionen erweitern Oberklasse, plötzlich neue Methoden nach außen hin sichtbar → Hinterher-Programmieren?

```
public class MyArrayList<T> extends ArrayList<T> {
    public boolean add(T o) {
        return super.add(o);
    }
}
```

Wiederverwendung

(aus "Entwurfsmuster von Kopf bis Fuß")

Komposition/Delegation (**Black-Box-Reuse**)

- Implementierung stabiler Schnittstelle der Basisklasse
- Objekt der Basisklasse als gekapseltes Attribut, jederzeit änderbar und künftige Erweiterungen **nicht** nach außen hin sichtbar
- Methoden können bei Bedarf an Objekt weiter delegieren
- Bsp. *Java*: Implementierung eigener *ArrayList*

```
public class MyArrayList<T> implements List<T> {
    private List<T> l = new ArrayList<T>();
    public boolean add(T o) {
        return l.add(o);
    }
}
```

Better Code!

- Entkopplung
  - Komponenten und Zuständigkeiten klar erkennbar
  - Testen und Debuggen einfacher
  - Fördert Durchblick und Dokumentation des Codes
- Effizienz
  - Oft besser als Eigenproduktionen
  - Grundlage der *strukturierten Softwareentwicklung*

Schwierigkeiten

- Einhaltung erfordert Planung und Disziplin
- Nicht alles vorhersehbar
  - besser **zu** gut strukturieren, später optimieren
  - Refactoring ist wichtig
- Performance kann schlechter werden
  - es gibt auch Performance-Patterns

Gang of Four

Autoren des Buches *„Design Patterns – Elements of Reusable Object-Oriented Software“* (1994)

Erich Gamma*

Richard Helm

Ralph Johnson

John Vlissides († 2005)

* Autor von JUnit

Arten von GoF-Patterns

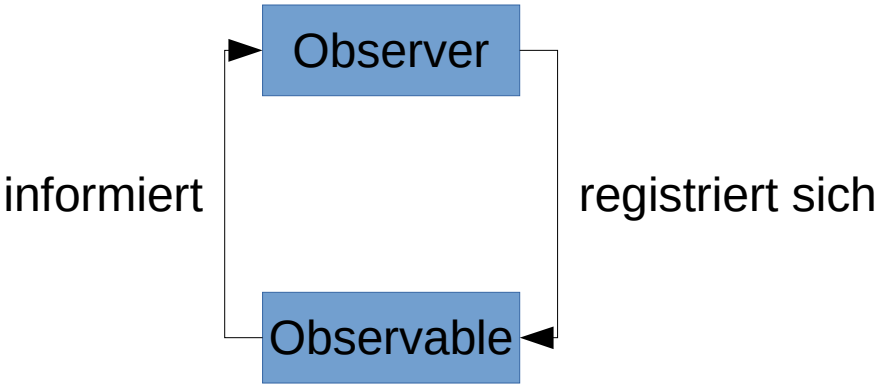
- **Verhaltensmuster** - behavioral patterns
  - Observer, State, Template Method, Iterator
- **Strukturmuster** - structural patterns
  - Adapter, Proxy, Facade, Compositum
- **Erzeugungsmuster** - creational patterns
  - (Abstract) Factory, Singleton, Prototype (and clones)

Observer-Pattern

&

Publish-Subscribe-Pattern

Observer Pattern



- Müsste eigentlich "Informer-Pattern" heißen
  - *Push*: Änderungen werden sofort mitgegeben
  - *Pull*: Lediglich Benachrichtigung, Änderungen auf Anfrage
- Observer registrieren sich bei Observable (= Subject)
- Observable informiert alle Observer über Neuigkeiten

Observable & Observer

Observable je nach Implementierung als Basisklasse oder Schnittstelle, Observer eigentlich immer Schnittstelle

- Observable (von der *Quelle* implementiert oder Ableitung):
  - add/remove Observer(Observer o)
  - notifyObservers
- Observer (von allen *konkreten Abhörern* implementiert):
  - onNotify(Observable subject, params)

Publish-Subscribe

- Vom Prinzip her ein erweitertes Observer-Pattern

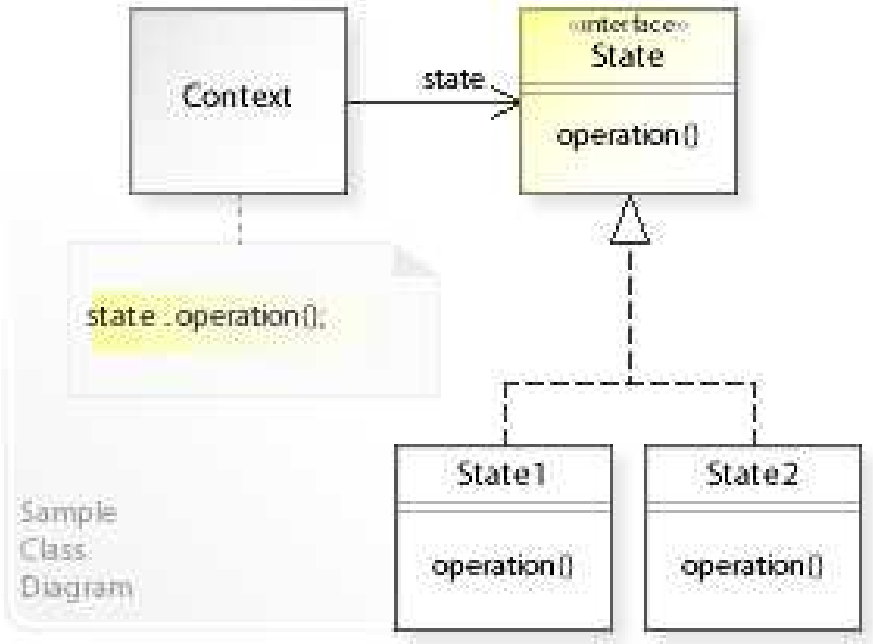
Unterschiede:

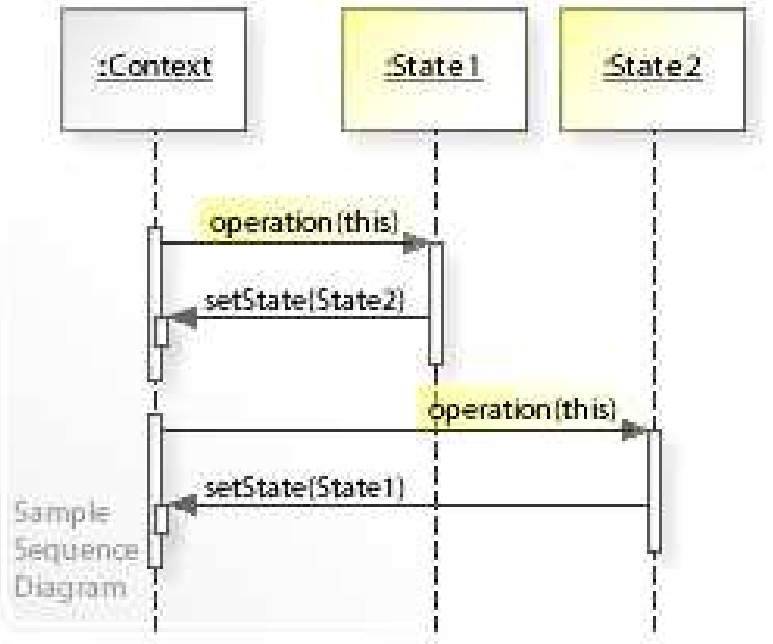
- Es gibt einen Broker (Vermittlungsstelle), welcher Kontakte zwischen den *Observables/Subjects* und den *Observers* herstellt
- Oft asynchron implementiert, d.h. nicht beide müssen gleichzeitig aktiv werden
- Beispiele: Messaging-Systeme, E-Mail
- Eigenes Kapitel „Message Queues“ (Ende des SJahres)

States

State

- Eine saubere Zustandsverwaltung





https://commons.wikimedia.org/wiki/File:W3sDesign_State_Design_Pattern_UML.jpg

State

- Klasse wird in eine Unterklasse pro Zustand unterteilt → Gleiche Schnittstelle, aber verschiedene Implementationen
- Der Methode *operation()* wird der (Aufrufer-) Kontext übergeben
- Die Operation wird ausgeführt, bei Zustandsänderung ein neues Objekt instanziiert und dessen Referenz im Kontext getauscht
- Bsp. *Bluetooth-Gadget* mit Zuständen „*verbunden*“ und „*nicht verbunden*“.

Template Methods

Template Methods

- **Basisklasse** definiert
  - Leere bzw. abstrakte Methoden als Platzhalter
    - noch nicht konkretisierter Code
    - auch als "*hooks*" (= Haken) bezeichnet
  - Skeleton = Bekannte Grundstruktur des Algorithmus
    - ruft Hooks in bestimmter Reihenfolge auf
    - Code aber eigentlich noch unvollständig

- **Kindklassen**
  - Implementieren die fehlenden Methoden (Stichwort Polymorphismus)
  - Nun ist Programm lauffähig, vom Skelett zum vollständigen Organismus

Vorgangsweise

1)Planung Skelett vs. Hooks

2)Basisklasse mit dem Skelett implementieren (= unveränderlicher Code d.h. *standard* steps)

3)Dabei Hooks definieren und Aufrufe einbauen

4)Kindklassen implementieren die Hooks (= veränderlicher Code d.h. *variable* steps)

- Beispiele: Methoden equals(), toString(), Übung zu den *Mathem. Ausdrücken* (siehe Anfang)

Inversion Of Control

- Traditionell:
  - Spezifischer Code ruft vordefinierten Code auf
  - z.B. Verwendung einer Bibliothek
- IoC:
  - Vordefinierter Code ruft spezifischen Code auf
  - Don't call us, we call you!

- Beispiel: *JUnit-Testklassen*

Function-Level IoC

- Auch mit einer Funktion allein lässt sich IoC implementieren
- Template Methods als IoC/Callbacks
- Callbacks übergeben, die bei Eintreten bestimmter Ereignisse aufgerufen werden:

  Zwei Beispiele in Java:

  - btnOk.addActionListener(**ActionListener l**)
  - Collections.sort(List<T> list,
    Comparator<? super T> c)

Iterator

- Sequentieller Zugriff auf Sammlung, ohne darunter liegende Struktur offen zu legen
- Motivation
  - Benutzer sollte sich nicht um interne Struktur kümmern
  - Unterschiedliche Traversierung (vorwärts, rückwärts)
  - Robustere Iteratoren erlauben auch das Ändern der Daten (invalidieren dann oft andere aktive Sammlungs-Iteratoren)
  - In Java foreach-Schleife mittels Interface **Iterable**

Adapter & Proxy

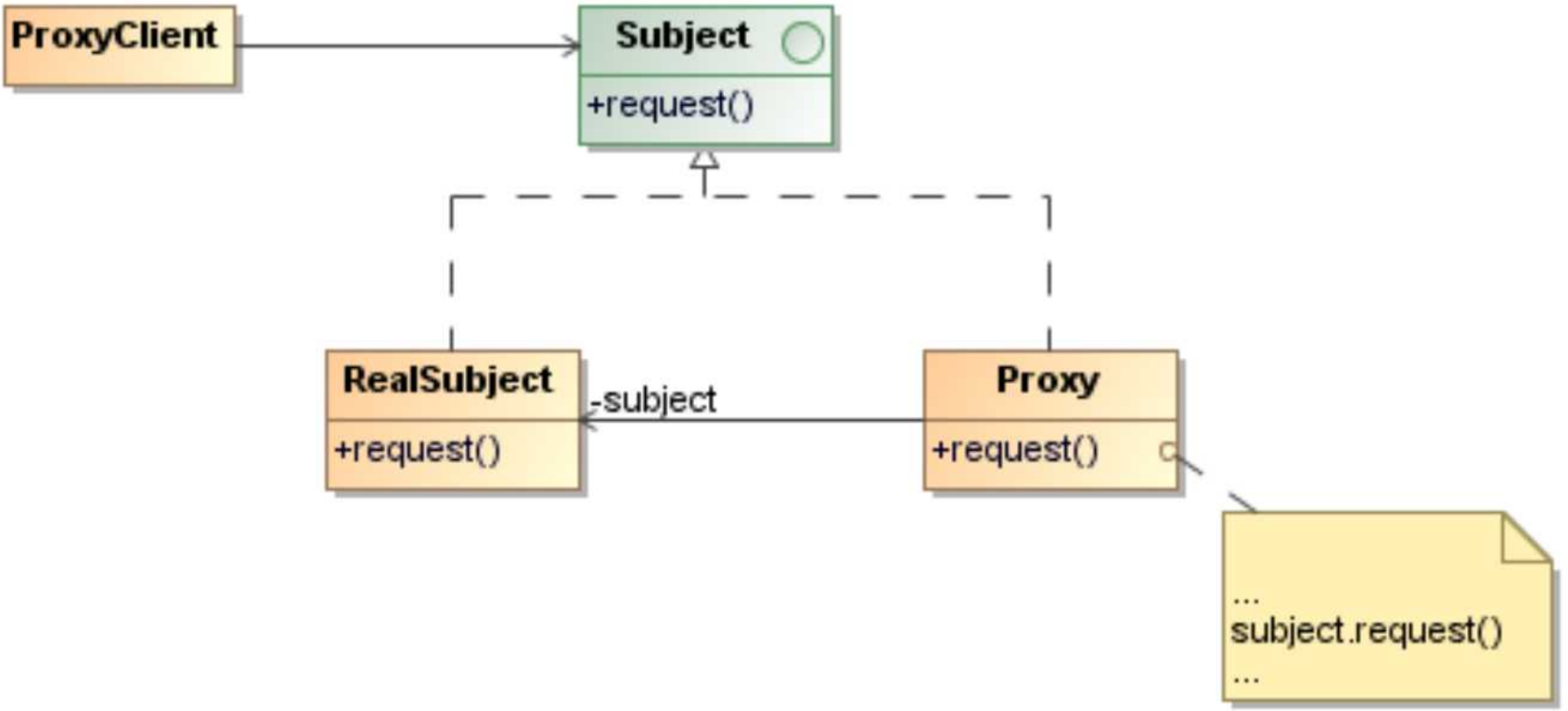
Adapter

- Auch *Wrapper* genannt
- Man möchte eine Bibliothek/Komponente mit unpassender Schnittstelle verwenden:
  - Oftmals Drittsystem außerhalb eigener Kontrolle
  - Quellcode oft nicht verfügbar/Änderungen aufwendig
- **Lösung**: Neue Klasse mit passender Schnittstelle, nutzt intern aber jene der Bibliothek (evtl. auch in anderer Technologie)

Proxy

- Man will nicht direkt eine gewisse Komponente benutzen, da
  - Zusätzliche Aufgaben erfolgen oder Vorbedingungen zu erfüllen sind
  - Das reale Objekt an einem anderen Ort existiert
- **Lösung**: Gemeinsame Schnittstelle zwischen Stellvertreter (Proxy) und Komponente, alle Verweise auf die Komponente durch Proxy zu ersetzen

Proxy

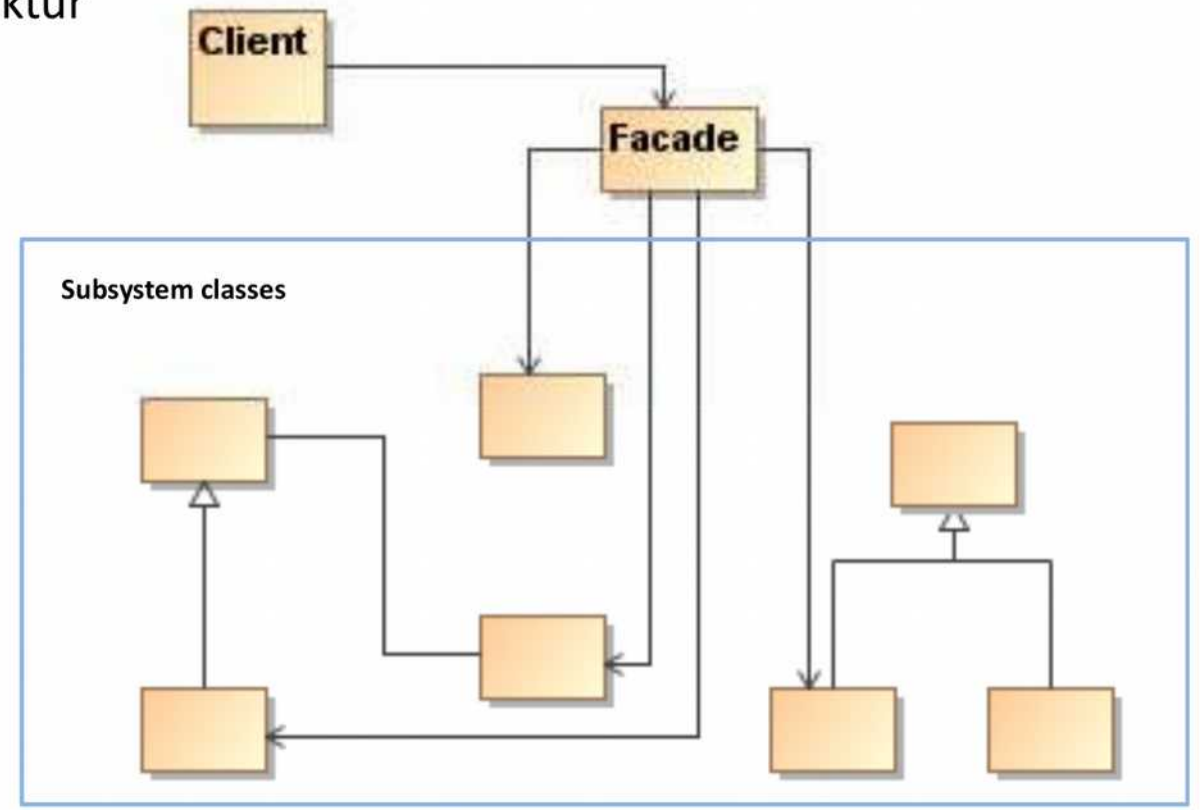

ProxyClient
Subject
+request()
RealSubject
+request()
-subject
Proxy
+request()
...
subject.request()
...

*

Facade

- Einheitliche Schnittstelle verschiedener Teile

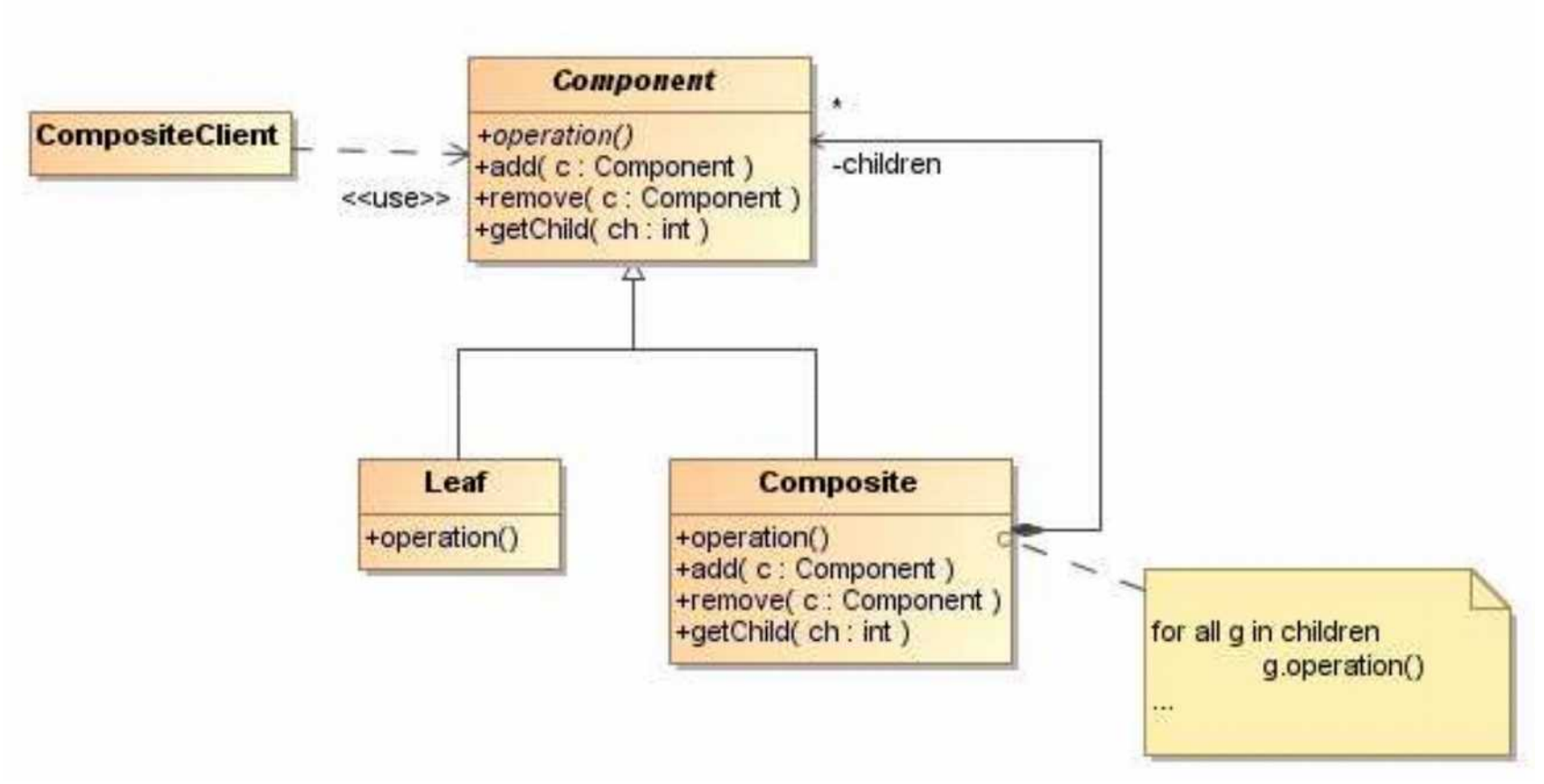
Struktur



*

Compositum

- Modellierung von Baumstrukturen, z.B. UIs



Factory

Static Factory Method

- Client soll nicht wissen
  - Woher ein Objekt stammt
  - Welchen genauen Typ es aufweist
- Daher **keine** direkte Nutzung der Konstruktoren
- Klasse besitzt statische Methode, welche eine Referenz zurückliefert
  - Auch *Instanziierung einer Kindklasse* möglich
  - Über *Parameter der Methode* beeinflussbar, aber auch über Komponenten, Plattform, Dienste, externe Konfigurationen etc.

Beispiel

- Basisklasse: Logger
- Unterklassen:
  - ConsoleLogger
  - FileLogger
  - UdpLogger

- Ohne Factory
  - der Client muss die richtige Logger-Klasse auswählen und mittels Konstruktor initialisieren (new ConsoleLogger(), new FileLogger() etc.)
- Mit Factory
  - der Client ruft Logger.create() auf und je nach Argumente (*void*, *path*, *host:port*) wird die gewünschte Unterklasse instantiiert

Singleton

Globals, Single Instances

- Garantiert, dass es nur eine Instanz einer Klasse gibt
  - Konstruktoren sind protected/privat
- Instanz über öffentliche statische Methode global verfügbar
  - z.B. Class.*getInstance*( )
- *Bsp.*: Datenbank-Komponenten wie JPA (Stichwort *Entity-Mapper*)

Globals, Single Instances

- Aber die Instanz ist überall verfügbar (… can be evil!)
  - Schwer nachzuvollziehen, wer sie verwendet da überall sichtbar
  - Genau eine Instanz, diese ist immer die selbe → nicht auf Variation ausgelegt
  - Nachträgliches Refactoring **schwierig!**
- Erweiterung: Lazy Initialization
  - Objekterzeugung erst bei erstem Abruf, nicht schon bei Start des Programms
  - Threadsicherheit beachten!

Singleton in Java

```
public class Singleton {
  private Singleton() {}
  private static class SingletonHolder {
    private final static Singleton INSTANCE = new
Singleton();
  }
  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

Voller Überblick

		Zweck		
		Erzeugungsmuster	**Strukturmuster**	**Verhaltensmuster**
Bereich	**Klasse**	Fabrikmethode (*Factory Method*)	Adapter (klassenbasiert) (*Adapter(class)*)	Interpreter (*Interpreter*) Schablonenmethode (*Template Method*)
	Objekt	Abstrakte Fabrik (*Abstract Factory*) Erbauer (*Builder*) Prototyp (*Prototype*) Singleton (*Singleton*)	Adapter (objektbasiert) (*Adapter (object)*) Brücke (*Bridge*) Dekorierer (*Decorator*) Fassade (*Facade*) Fliegengewicht (*Flyweight*) Kompositum (*Composite*) Stellvertreter (*Proxy*)	Befehl (*Command*) Beobachter (*Observer*) Besucher (*Visitor*) Iterator (*Iterator*) Memento (*Memento*) Strategie (*Strategy*) Vermittler (*Mediator*) Zustand (*State*) Zuständigkeitskette (*Chain of Responsibility*)

*

Antimuster/
Anti-Patterns

Design-Antimuster

- **Außerirdische Spinnen** (engl. *Alien Spiders*)
  - Zu gesprächige Objekte die sich gegenseitig kennen
- **Gottobjekt** (engl. *God Object*)
  - Allumfassend, für alles zuständig
  - Auch „**Blob“** genannt
- **Innerer Plattform-Effekt** (engl. *Inner platform effect*)
  - System, das zuviel macht (Plattform in Plattform)
- **Spaghetti-Code**

Programmier-Antimuster

- **Zwiebel** (engl. *Onion*)
  - Ewiges Drumherumprogrammieren
- **Copy&Paste**
- **Lavafluss** (engl. *Lava Flow*)
  - Zuviel toter oder auskommentierter Code, Folge davon sind viele Verzweigungen
- **Switch-Statements**
  - In vielen Fällen State-Pattern die bessere Wahl
- **Magic Values**
  - Hartkodierte Werte ohne Konstanten