
UNIVERSITÄT
LEIPZIG

# Softwaretechnik 2024/25 – Übung 06

Prof. Dr. Norbert Siegmund
M.Sc. Stefan Jahns

# Aufgabe 1: Begriffe

## a) Warum sind Design Patterns wichtig?

- Gemeinsame Sprache für Entwicklerinnen
  - Verbessert Kommunikation
  - Beugt Missverständnisse vor
- Lernen aus Erfahrung
  - Eine gute Designerin / Entwicklerin zu werden, ist schwer
    - Gute Designs kennen / verstehen ist der erste Schritt
  - Erprobte Lösungen für wiederkehrende Probleme
    - Durch Wiederverwendung wird man produktiver
    - Eigene Software wird selbst flexibler und wiederverwendbarer

# Aufgabe 1: Begriffe

b) Nennen Sie 3 GoF-Design-Patterns und ordnen Sie jeweils die Zweck-Klassifikation zu.

- Structural Patterns - Helfen bei der Komposition von Klassen und Objekten
  - Adapter
  - Decorator
- Behavioral Patterns - Helfen bei der Interaktion von Klassen und Objekten (Verhalten kapseln)
  - Observer
  - Visitor
  - Iterator
- Creational Patterns - Helfen bei der Objekterstellung
  - Factory
  - Singleton

# Aufgabe 1: Begriffe

c) Beschreiben Sie die Funktion eines Interfaces und geben Sie an in wie weit sich Interfaces von Klassen-Vererbung unterscheiden.

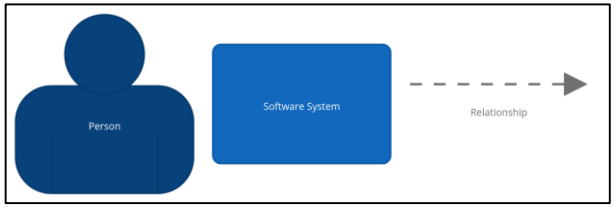
- „Vertrag“, der definiert, welche Methoden eine Klasse implementieren muss
  - In der Regel nur Methodensignaturen
  - Kann auch Konstanten enthalten oder default-Methoden (seit Java 8)
- Konzept zur Ermöglichung von Polymorphie und loser Kopplung
  - Polymorphie := Einheitliche Behandlung unterschiedlicher Objekte
- Unterschied zur Vererbung:
  - Eine Klasse kann mehrere Interfaces implementieren, aber nur von einer Klasse erben

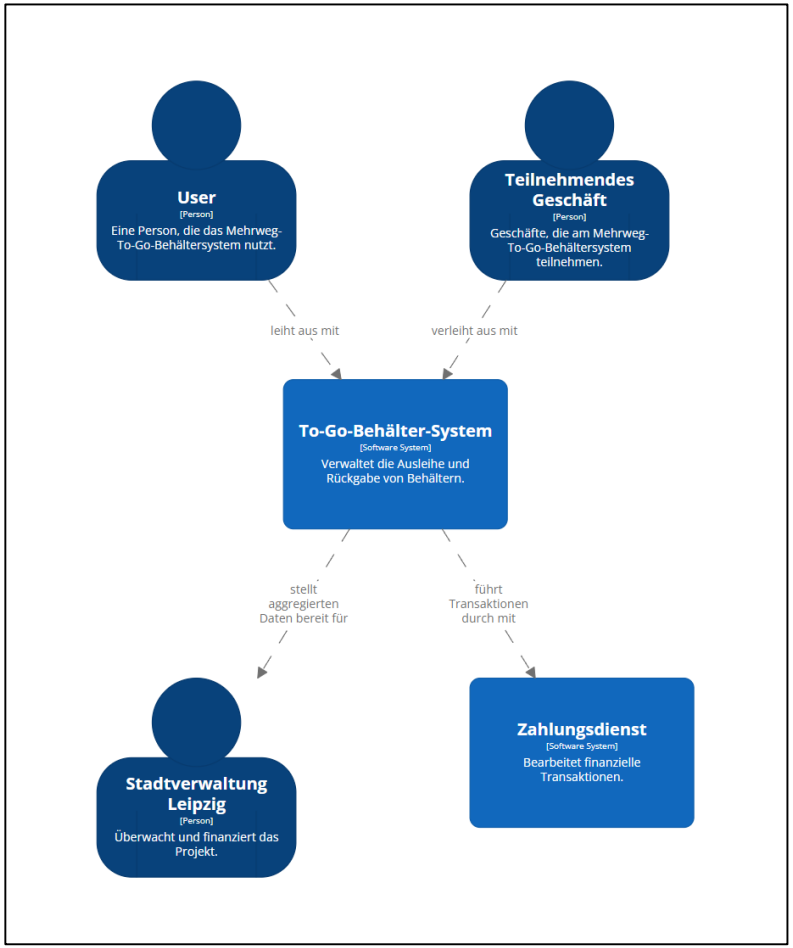
# Aufgabe 2: C4-Diagramm

Sie sollen eine Software-Anwendung für ein Leipzig-weite Mehrweg-To-Go-Behälter für Getränke und Speisen entwickeln. Statt Pfand zu bezahlen, ist jeder Behälter mit einem QR- Code ausgestattet, welcher bei der Ausleihe von Usern mit der Anwendung gescannt und damit mit ihrem Konto verknüpft wird. Hierfür registrieren sich User vorab. User können teilnehmende Geschäfte in einer übersichtlichen Kartenansicht suchen. Der Scan-Vorgang muss schnell funktionieren. Überschreiten User die Rückgabefrist, wird automatisch eine Gebühr von 1€ pro Woche über einen externen Zahlungsdienstleister eingezogen. Geschäfte können Behälter über eine eigene Ansicht in der Anwendung bestellen. User-Daten sind sicher zu speichern, damit keine Nutzungsmuster erbeutet werden können. Die Stadt Leipzig finanziert das Projekt und wünscht Zugang zu aggregierten Statistiken, etwa zur Anzahl der ausgegebenen Behälter.

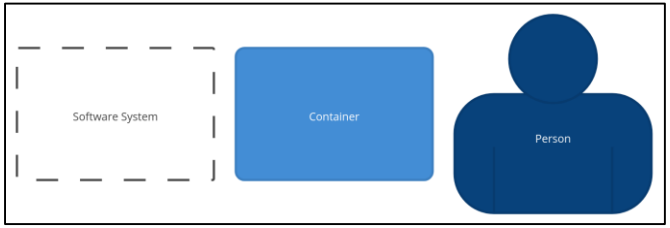
→ Stufen Context, Container, Component

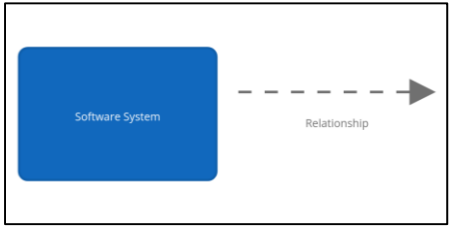
# Aufgabe 2: C4-Diagramm
# Level 1: System context diagram

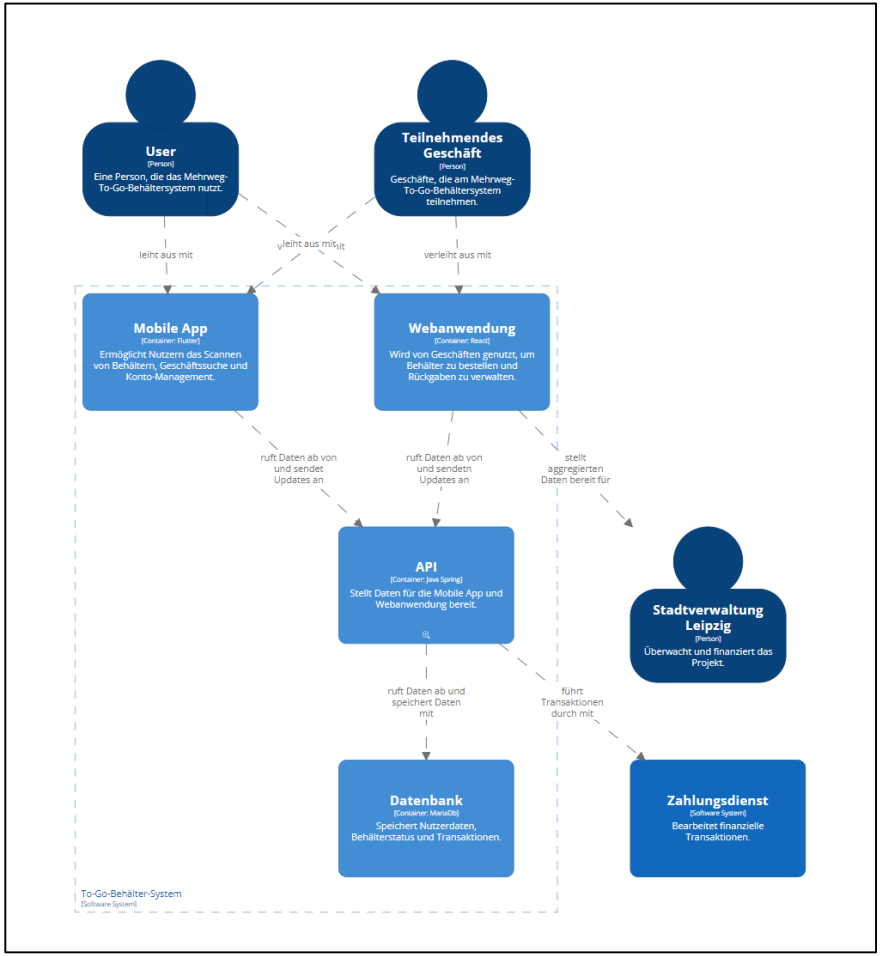


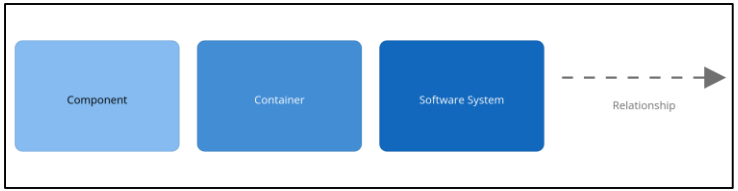
# Aufgabe 2: C4-Diagramm
# Level 2: Container diagram

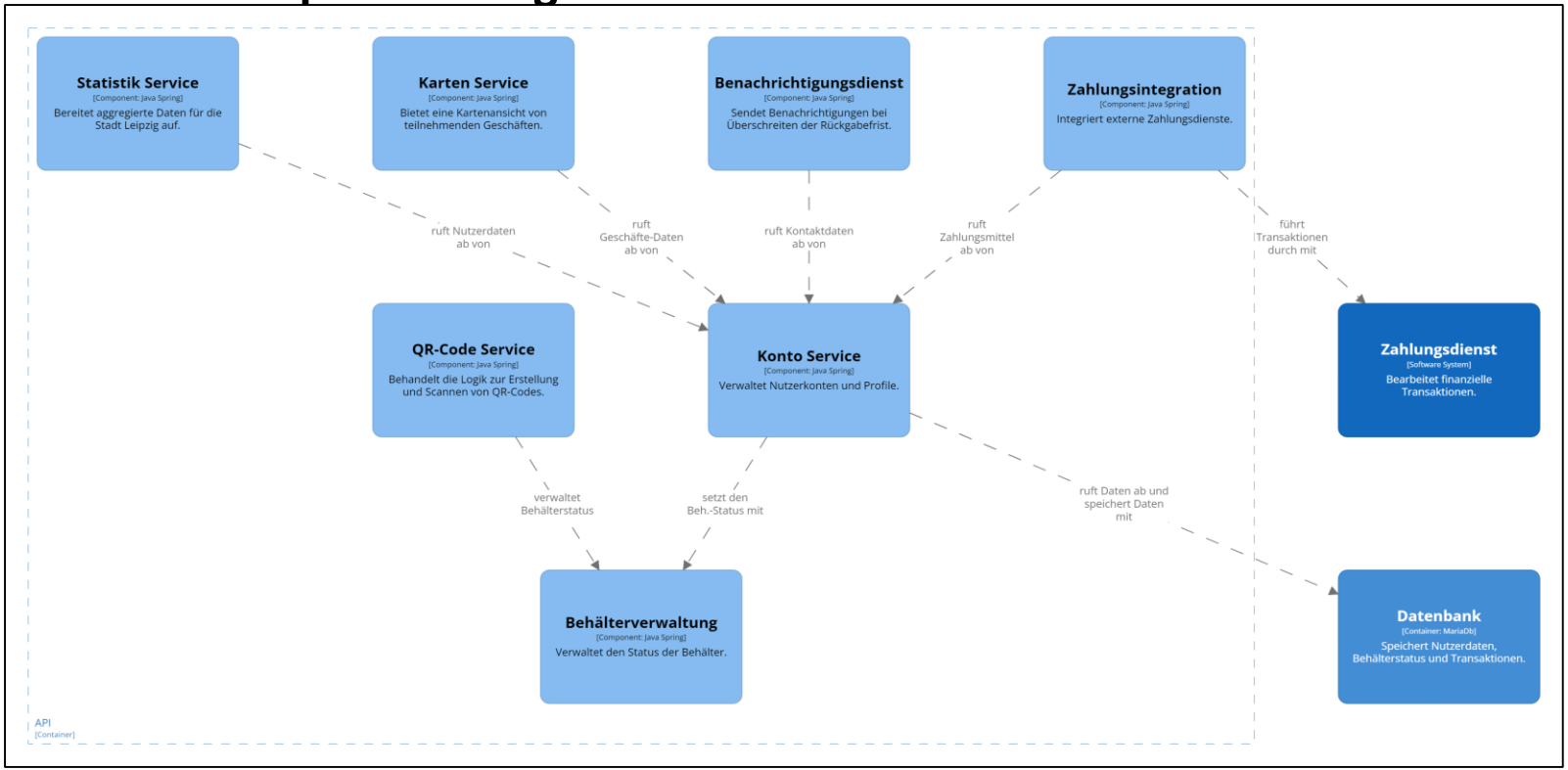


# Aufgabe 2: C4-Diagramm
# Level 3: Component diagram



# Aufgabe 2: C4-Diagramm — Definition Level 1 - 3

https://structurizr.com/dsl

(Source wird als .txt in moodle hochgeladen)

```
workspace {
  model {
      nutzer = person "User" "Eine Person, die das Mehrweg-To-Go-Behältersystem  nutzt."
      geschaeft = person "Teilnehmendes  Geschäft" "Geschäfte, die am Mehrweg-To-Go-Behältersystem  teilnehmen."
      stadtLeipzig = person "Stadtverwaltung Leipzig" "Überwacht und finanziert das Projekt."
      zahlungsdienst = softwaresystem "Zahlungsdienst" "Bearbeitet finanzielle Transaktionen."
      toGoBehaelterSystem = softwaresystem "To-Go-Behälter-System" "Verwaltet die Ausleihe und Rückgabe von Behältern." {
        mobileApp = container "Mobile App" "Ermöglicht Nutzern das Scannen von Behältern, Geschäftssuche und Konto-Management." {
          technology "Flutter"
        }
        webApp = container "Webanwendung" "Wird von Geschäften genutzt, um Behälter zu bestellen und Rückgaben zu verwalten." {
          technology "React"
        }
        datenbank = container "Datenbank" "Speichert Nutzerdaten, Behälterstatus und Transaktionen." {
          technology "MariaDb"
        }
        api = container "API" "Stellt Daten für die Mobile App und Webanwendung bereit." {
          technology "Java Spring"
          kontoService = component "Konto Service" "Verwaltet Nutzerkonten und Profile."{
            technology "Java Spring"
          }
          qrCodeService = component "QR-Code Service" "Behandelt die Logik zur Erstellung und Scannen von QR-Codes."{
            technology "Java Spring"
          }
          zahlungsIntegration  = component "Zahlungsintegration"  "Integriert externe Zahlungsdienste."{
            technology "Java Spring"
          }
          benachrichtigungsService = component "Benachrichtigungsdienst"  "Sendet Benachrichtigungen bei Überschreiten der Rückgabefrist."{
            technology "Java Spring"
          }
          behaelterVerwaltung  = component "Behälterverwaltung"  "Verwaltet den Status der Behälter."{
            technology "Java Spring"
          }
          kartenService = component "Karten Service" "Bietet eine Kartenansicht von teilnehmenden Geschäften."{
            technology "Java Spring"
          }
          statistikService = component "Statistik Service" "Bereitet aggregierte Daten für die Stadt Leipzig auf."{
            technology "Java Spring"
          }
        }
        statistikService -> kontoService "ruft Nutzerdaten  ab von"
        benachrichtigungsService -> kontoService "ruft Kontaktdaten  ab von"
        kontoService -> datenbank "ruft Daten ab und speichert Daten mit"
        kartenService -> kontoService "ruft Geschäfte-Daten  ab von"
        zahlungsintegration  -> kontoService "ruft Zahlungsmittel ab von"
        kontoService -> behaelterVerwaltung  "setzt den Beh.-Status mit"
      }
      webApp -> stadtLeipzig "stellt aggregierten Daten bereit für"
      nutzer -> mobileApp "leiht aus mit"
      geschaeft -> mobileApp "verleiht aus mit"
      nutzer -> webApp "leiht aus mit"
      geschaeft -> webApp "verleiht aus mit"
      mobileApp -> api "ruft Daten ab von und sendet Updates an"
      webApp -> api "ruft Daten ab von und sendetn Updates an"
      qrCodeService  -> behaelterVerwaltung  "verwaltet Behälterstatus"
      zahlungsintegration  -> zahlungsdienst "führt Transaktionen  durch mit"
  }
  views {
      systemLandscape {
        include *
        autoLayout
      }
      container toGoBehaelterSystem {
        include *
        autoLayout
      }
      component api {
        include *
        autoLayout
      }
      theme default
  }
}
```

## Allgemeine Hinweise:

- Bestenfalls keine Hin- und Rückpfeile
- Generische Beziehungen wie „nutzt“ oder „interagiert mit“ vermeiden
- Detailgrad flexibel anpassen

## Aufgabe 3: Composite-Pattern

Für diese Aufgabe ist das folgende Szenario gegeben:

Sie entwickeln ein Betriebssystem für Netzwerk-Datenspeicher (Network-Attached Storage, NAS). Sowohl für einzelne Dateien als auch für ganze Ordner sollen Aktionen, wie etwa das Berchnen der Größe, oder das Ändern von Zugriffsrechten ausgeführt werden können. Zusätzlich sollen sich verschiedene externe Datenquellen als spezielle Ordner in der Dateistruktur einbinden lassen.

# Aufgabe 3: Composite-Pattern

## a)

| Beschreibung | Erläuterung |
|---|---|
| Pattern Name und Klassifikation | Composite-Pattern (*Structural Pattern*) |
| Zweck | Erlaubt die Behandlung einzelner Objekte und deren Kompositionen (z. B. Hierarchien) auf gleiche Weise. |
| Auch bekannt als | Part-Whole Pattern |
| Motivation | In Szenarien mit Baumstrukturen (z. B. Dateisysteme), bei denen Clients nicht zwischen Blättern und Knoten unterscheiden sollen. |
| Anwendbarkeit | Wenn Objekte in einer Baumstruktur organisiert sind und Operationen auf der gesamten Struktur einheitlich durchgeführt werden sollen. |
| Struktur/Teilnehmer | Siehe nächste Folie |

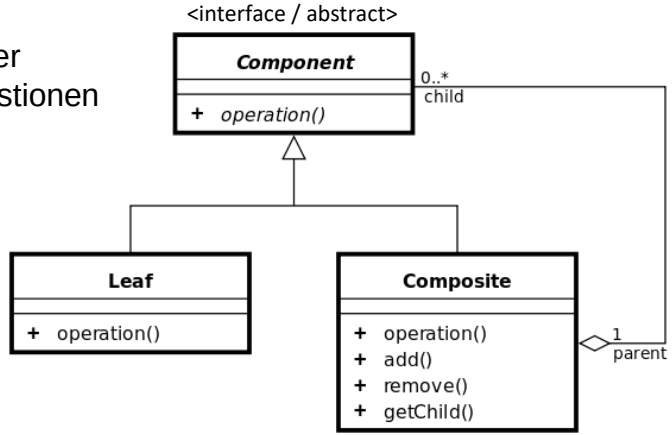
# Aufgabe 3: Composite-Pattern

## a)

| Beschreibung | Erläuterung |
|---|---|
| Kollaborationen | – Composite leitet Operationen an seine Kinder weiter<br>– Leaf führt die Operationen direkt aus |

**Component** → Basisklasse oder Interface für Blätter und Kompostionen

**Leaf** → Einzelobjekte



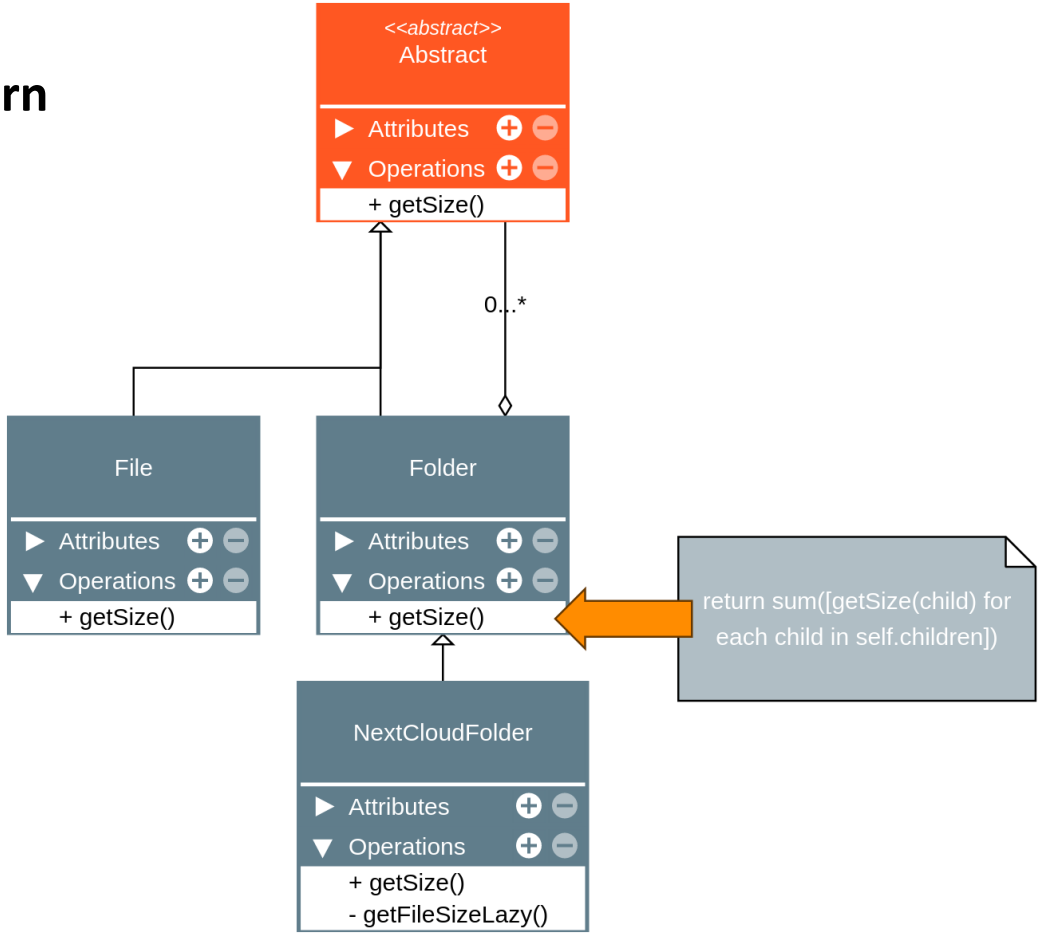

**Composite** → Kompositionen, die aus mehreren Component-Objekten besteht

# Aufgabe 3: Composite-Pattern

## a)

| Beschreibung | Erläuterung |
|---|---|
| Konsequenzen | **Vorteile:**<br>– Einfache Erweiterbarkeit durch Hinzufügen neuer Komponenten<br>– Einheitliche Schnittstelle für Blätter und Kompositionen<br>**Nachteile:**<br>– Kann komplex werden bei sehr großen Hierarchien oder spezifischen Anforderungen |
| Implementierung | Rekursive Datenstruktur für Composite, Liste für Kind-Objekte.<br>Basis-Operationen in der Component definieren. |
| Bekannte Verwendungen | – GUI-Frameworks (z. B. Swing, HTML-DOM-Struktur)<br>– Dateisysteme<br>– Unternehmenshierarchien |
| Verwandte Pattern | – Decorator<br>– Visitor |

# Aufgabe 3: Composite-Pattern
# b) Filesystem UML





https://www.yworks.com/yed-live/

# Aufgabe 3: Composite-Pattern
# c)

## Versus Visitor

Gemeinsam:

- Umgang mit Datenstrukturen
- Erleichtern rekursive Operationen

Unterschiede:

- Visitor: Behavioral
- Visitor: gleiche Struktur
  Composite: gleiche Aktionen

Use Case:

- Datenstruktur mit Composite
- Mit Visitor neue Funktionen hinzufügen
  - Jedes Element braucht accept()

## Versus Decorator

Gemeinsam:

- Structural Pattern
- Decorator fast Spezialfall von Composite mit 1 Kind

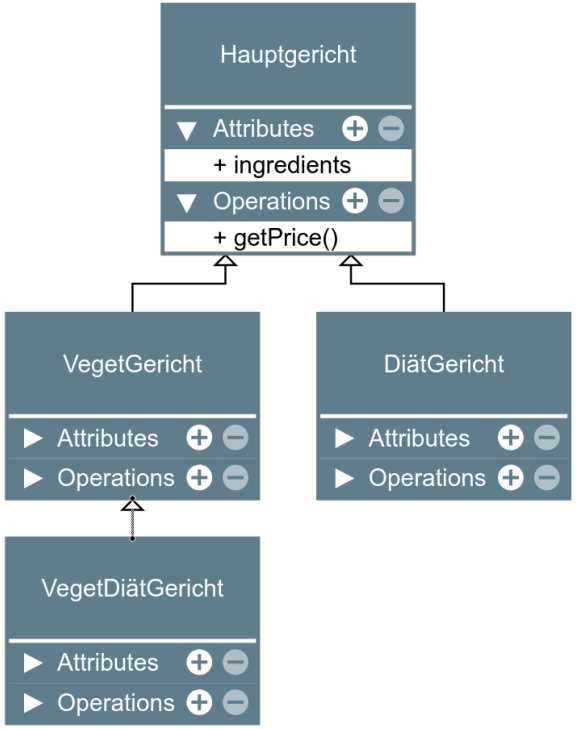
Unterschiede:

- Decorator: Keine Baumstruktur möglich

# Aufgabe 4: Entwurfsmuster und Konfigurierbarkeit

Gegeben ist folgendes Szenario und Diagramm:

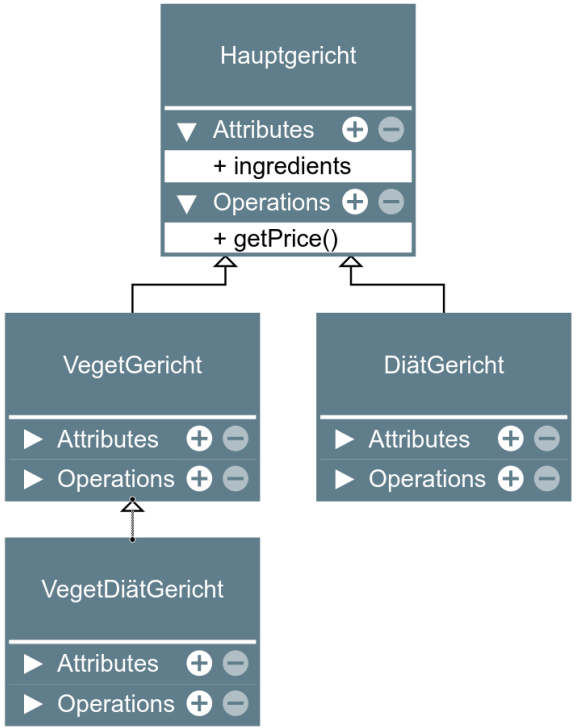
Für die Menüplanungs-Software einer Kantine sollen einzelne Hauptgerichte modelliert werden. Jedes Hauptgericht kann normales Gericht, oder aber vegetarisch, oder kalorienreduziert sein, wobei mehrere Kategorien gleichzeitig zutreffen können; es kann also beispielsweise ein kalorienreduziertes vegetarisches Gericht geben. Der initiale Klassenentwurf (UML-Diagramm unten) berücksichtigt diese Eigenschaften.



# Aufgabe 4: Entwurfsmuster und Konfigurierbarkeit

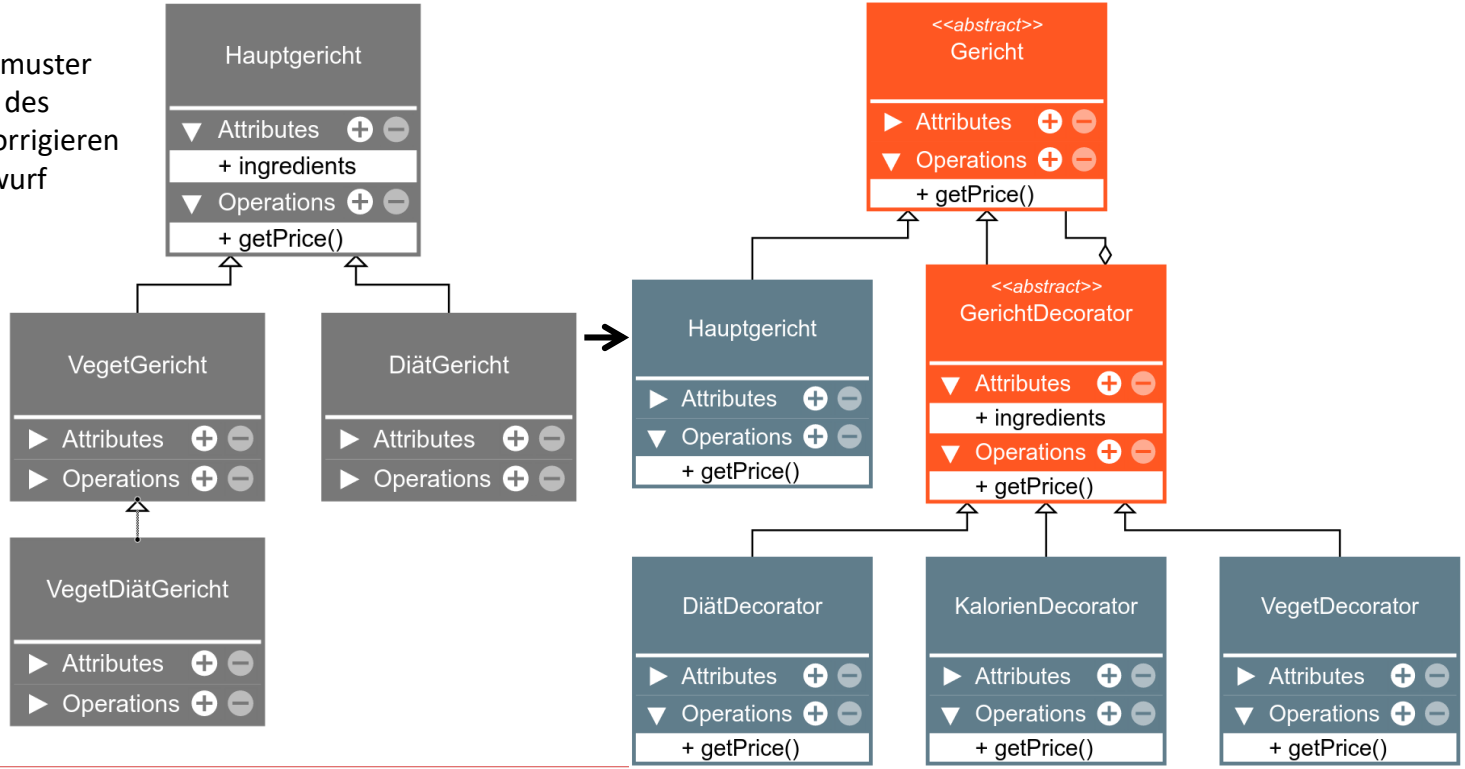
a) Die Kantine möchte in Zukunft auch glutenfreie Hauptgerichte anbieten. Welche Aspekte des Klassenentwurfs erschweren diese Erweiterung?

- Code Duplication DiätGericht <-> VegetDiätGericht
- Erweiterung mittels Vererbung ist unflexibel
  - um alle nun möglichen Hauptgerichte zu modellieren muss jede Klasse um eine Subklasse erweitert werden
- Dadurch schlecht wartbar



# Aufgabe 4: Entwurfsmuster und Konfigurierbarkeit

b) Mit welchem Entwurfsmuster lässt sich die Modularität des Entwurfes verbessern? Korrigieren Sie den UML-Klassenentwurf entsprechend.



# Aufgabe 4: Entwurfsmuster und Konfigurierbarkeit

c) Implementieren Sie Ihre neue Lösung für mindestens eine Eigenschaft eines Hauptgerichts in Java.

```java
public abstract class Gericht {
    abstract float getPrice();
}

public class Hauptgericht extends Gericht {

    // Standardpreis
    float getPrice() {
        return 2;
    }
}

public abstract class GerichtDecorator extends Gericht {
    protected Gericht decoratedGericht;
    public GerichtDecorator(Gericht decoratedGericht) {
        this.decoratedGericht = decoratedGericht;
    }

    // Delegate invocation to decorated component
    float getPrice() {
        this.decoratedGericht.getPrice();
    }

}

public class KalorienDecorator extends GerichtDecorator {

    public KalorienDecorator(Gericht decoratedGericht) {
        super(decoratedGericht);
    }

    // Preisänderung, mehr Kalorien sind teurer?
    @Override
    float getPrice() {
        return this.decoratedGericht.getPrice() + 1;
    }
}
```