# Aufgabe 1 (Notation der Assoziation)

Zeichnen Sie zum nachstehenden Java-Programm das UML-Klassendiagramm.

```
public class Warenkorb
{
    private static Warenkorb derWarenkorb;
    private static float wert = 0f;

      private Warenkorb(){}

      public static Warenkorb getInstanceOfWarenkorb()
      {     if(derWarenkorb != null) return derWarenkorb;
            else {
                      derWarenkorb = new Warenkorb();
                      return derWarenkorb;
                 }
      }

      public boolean set(IArtikel einArtikel)
      {     wert += einArtikel.getPreis();
            return  true;
      }

      public float getWert()
      {     return wert;
      }
}
```

```
public class Kunde
{
      String name;
      Warenkorb meinWarenkorb;

      public Kunde(String name)
      { this.name =name;}
}
```

```
public class Einkauf
{     String kunde = null;

      public static void main(String[] args)
      {
            Kunde meinKunde = new Kunde("Meyer");
            meinKunde.meinWarenkorb = Warenkorb.getInstanceOfWarenkorb();

            Warenkorb aktWarenkorb = meinKunde.meinWarenkorb;

            Bildschirm derBildschirm = new Bildschirm();
            Festplatte dieFestplatte = new Festplatte();

            aktWarenkorb.set(derBildschirm);
            aktWarenkorb.set(dieFestplatte);
            float wert = aktWarenkorb.getWert();


            System.out.println(wert);
      }
}
```

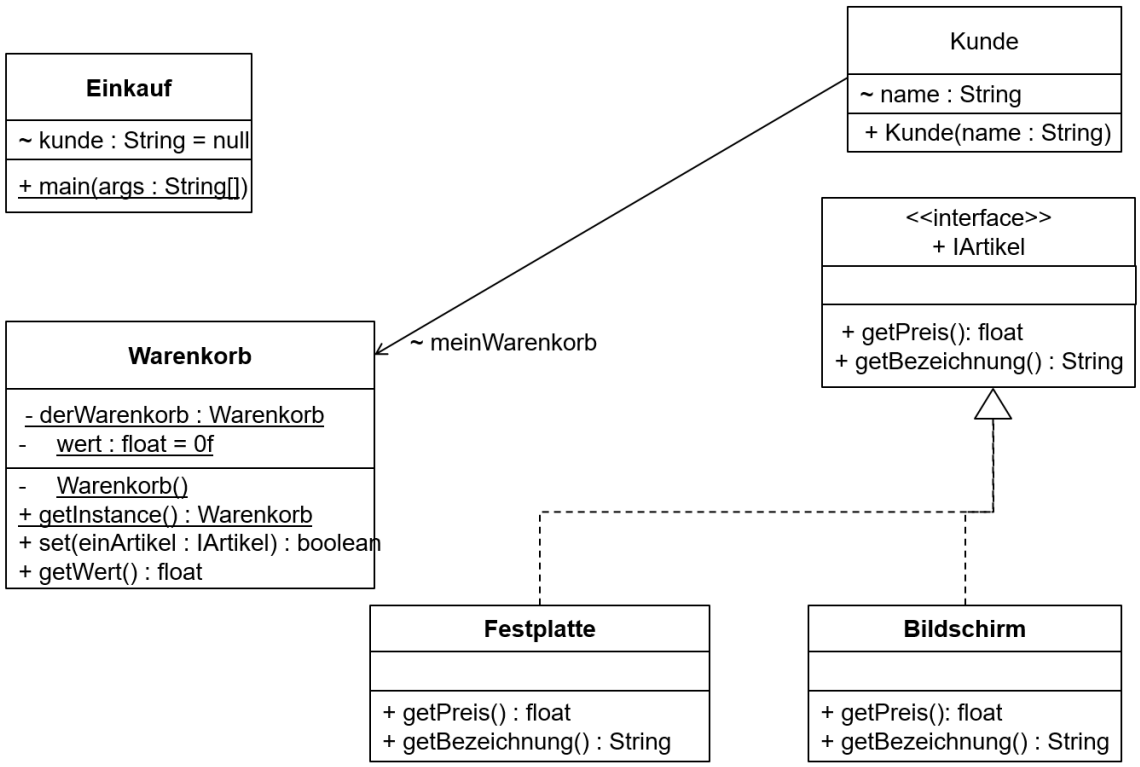
```
public interface IArtikel
{
      public String getBezeichnung();
      public float getPreis();
}
```

```
public class Bildschirm implements IArtikel
{
      public float getPreis(){
          return 71.0f ;}

      public String getBezeichnung() {
            return "FFT-Bildschrim";
      }
}
```

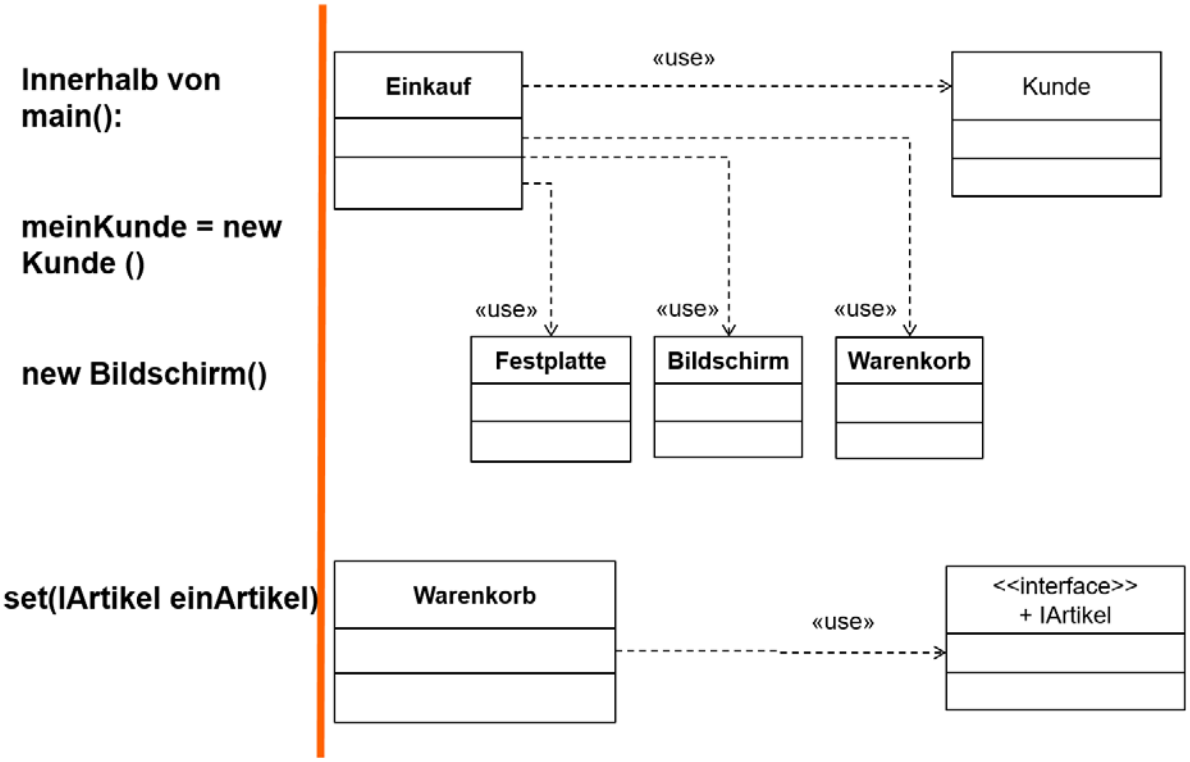
```
public class Festplatte implements IArtikel
{
      public String getBezeichnung() {
            return "2,5\"-Festplatte";
      }

      public float getPreis() {
            return 128.50f;
      }
}
```

# Aufgabe 1 (Notation der Assoziation)

Zeichnen Sie zum nachstehenden Java-Programm das UML-Klassendiagramm.



Folgende zusätzliche Abhängigkeiten sind im Java-Programm enthalten:



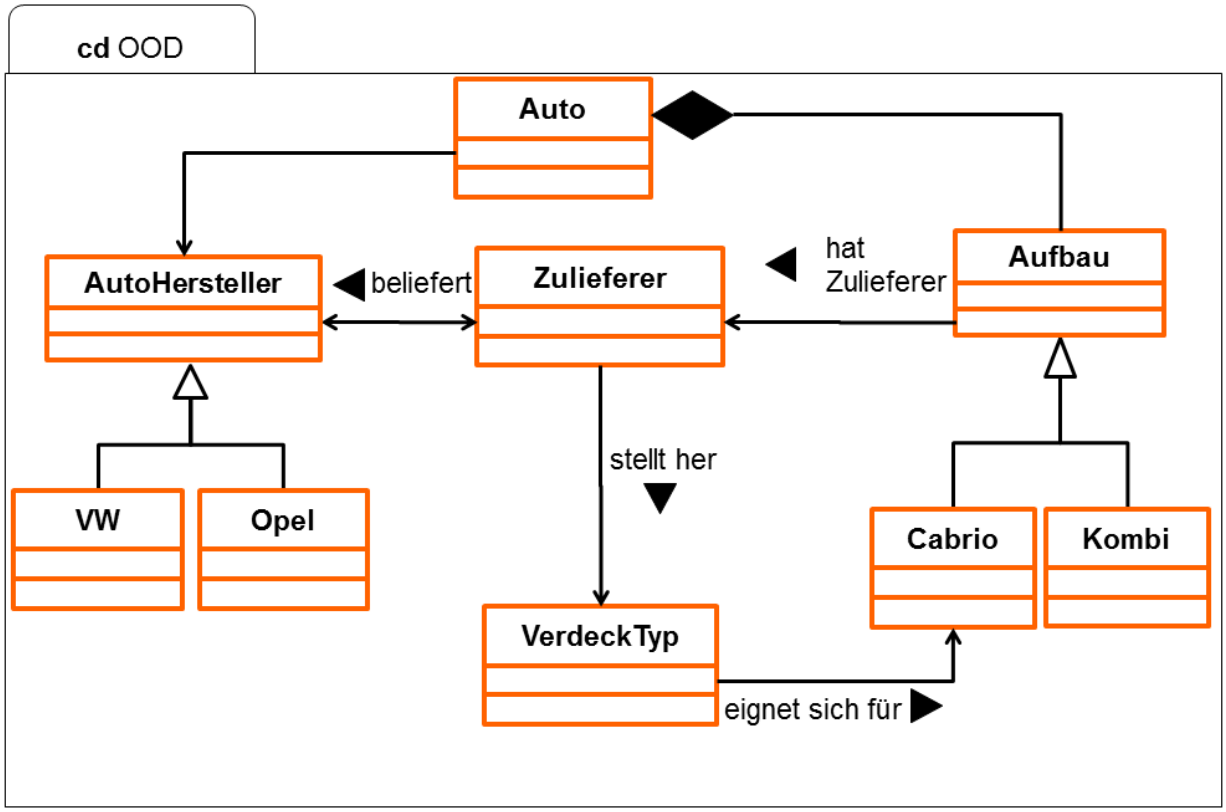
Hinweise

| | |
|---|---|
| 1. | Die main-Methode muss in der richtigen Signatur eingezeichnet werden.<br><br>`+ main(args: String[])` |
| 2. | Konstruktoren werden eingezeichnet, wenn<br>... sie nicht leer sind bzw.in einer Vererbung nur super() aufrufen<br>oder<br>... wenn sie eine andere Sichtbarkeit als `public` haben |
| 3. | Zu einem UML-Diagramm gehört immer eine zusätzliche Beschreibung. |
| 4. | Set-/get-Methoden werden nur bei Bedarf eingezeichnet.<br>Unter einem UML-Diagramm gehört ein Hinweis, wie das Diagramm in diesem Fall zu lesen ist:<br><br>Beispiele:<br>- Aus Gründen der besseren Lesbarkeit wurde auf die Angabe von get/set-Methoden verzichtet.<br>- Bei den Methoden, für die es get-/set-Methoden exisiteren, sind diese angegeben. |
| 5. | Referenzattribute auf Klassenebene müssen im Klassendiagramm als Assoziation eingetragen werden. |
| 6. | Referenzattribute und deren Sichtbarkeit müssen als Rollenbezeichnung einer Assoziation übernommen. |
| 7. | Aufgrund des Programmtextes ist zu untersuchen, ob es sich bei einer Assoziation um eine 'Einfache Assoziation‘, um eine 'Aggregation' oder um eine 'Komposition' handelt.<br>Im Zweifelsfall ist immer eine „einfache Assoziation“ zu wählen. |
| 8. | Attributdefinitionen innerhalb von Methoden erscheinen nicht im Klassendiagramm. |
| 9. | Private Methoden müssen nur dann im Klassendiagramm erscheinen, wenn dies dem Verständnis des Programmes dient. |
| 10. | Im Klassendiagramm können «`use`»-Beziehung eingetragen werden, wenn die Abhängigkeiten lokaler Natur sind:<br>- als lokale Variable innerhalb einer Methode,<br>- in der Parameterliste,<br>- als Rückgabewert. |

# Aufgabe 1 (Abhängigkeiten im Klassendiagramm)

Betrachten Sie für die nachstehende Zeichnung Abhängigkeiten im Sinne der Übersetzungsreihenfolge eines Compilers.

Wenn eine Klasse verändert wird, müssen alle direkt oder indirekt davon abhängigen Klassen auf korrekte Funktionsweise geprüft werden.



Welche Klassen müssen überprüft werden?

| **wenn verändert….** | **dann prüfen …** |
|---|---|
| Auto | |
| Autohersteller | |
| VW | |
| Opel | |
| Aufbau | |
| Kombi | |
| Cabrio | |
| Zulieferer | |
| VerdeckTyp | |

# Aufgabe 2 (Abhängigkeiten im Klassendiagramm)

Wie lassen sich die Abhängigkeitsstrukturen verbessern? Verbessern Sie das Klassendiagramm dahingehend, dass Zyklen aufgelöst und unnötige Abhängigkeiten vermieden werden.

a) Zeichnen sie das überarbeitete Klassendiagramm

b) Welche Klassen müssen nun überprüft werden?

| **wenn verändert….** | **dann prüfen …** |
|---|---|
| Auto | |
| Autohersteller | |
| VW | |
| Opel | |
| Aufbau | |
| Kombi | |
| Cabrio | |
| Zulieferer | |
| VerdeckTyp | |

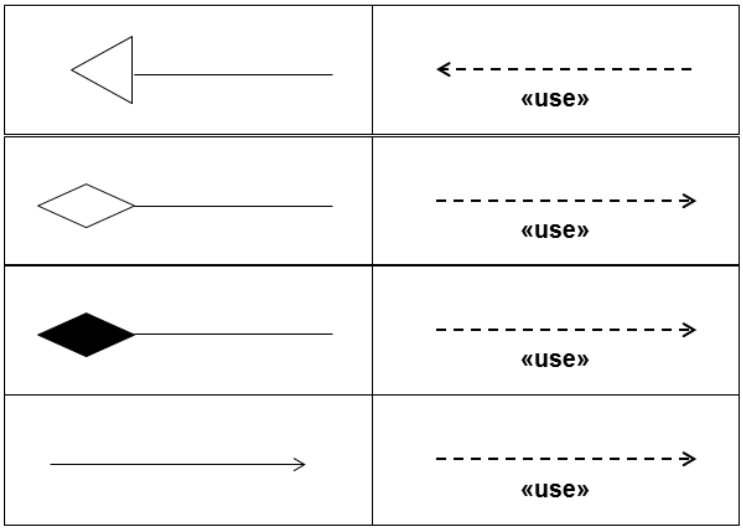
.

# Aufgabe 3 (Allgemeine Fragen zu Abhängigkeiten)

a) Welche Bedeutung haben Zyklen in Klassendiagrammen?
b) Was versteht man unter implizite Abhängigkeiten?
Wie kann man geeignet mit impliziten Abhängigkeiten umgehen?

# Hinweise zu Abhängigkeiten im Klassendiagramm

Aus den in einem Klassendiagramm verwendeten Beziehungen lassen sich folgende Abhängigkeiten erkennen:




Dies entspricht der Reihenfolge der Übersetzung.

Weitere **«use»**-Beziehungen ergeben sich, wenn Beziehungen nur 'lokal' definiert sind, z.B. die Referenz in Parametern einer Methode.

Durch den Einbau von Schnittstellen können Abhängigkeitsbeziehungen gezielt aufgelöst werden!

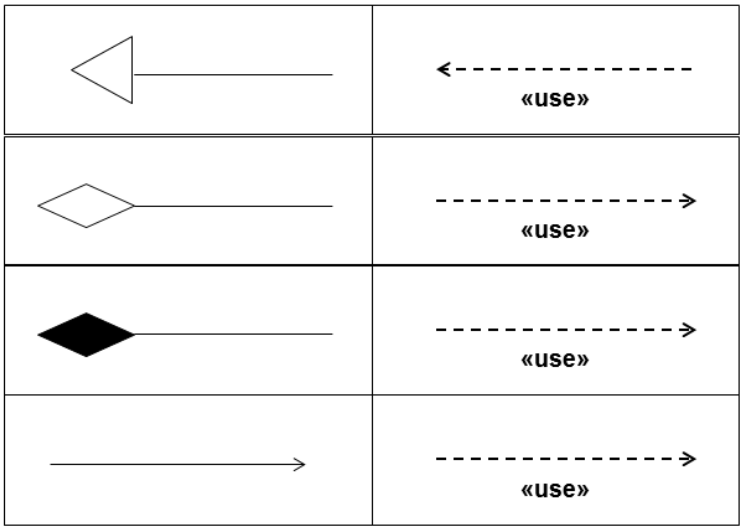
# Hinweise zum Auflösen von Abhängigkeiten

Es existieren mehrere Möglichkeiten Abhängigkeiten aufzulösen

- Einbau von Schnittstellen
  (gegenseitige Abhängigkeiten, zyklische Abhängigkeiten, Abhängigkeitsketten)
- Einbau einer assoziativen Klassen
- (bei gegenseitigen Abhängigkeiten)
- Einbau von Verwaltungsklassen
- (bei gegenseitigen Abhängigkeiten (zur Umkehrung)

# Aufgabe 1 (Abhängigkeiten im Klassendiagramm)

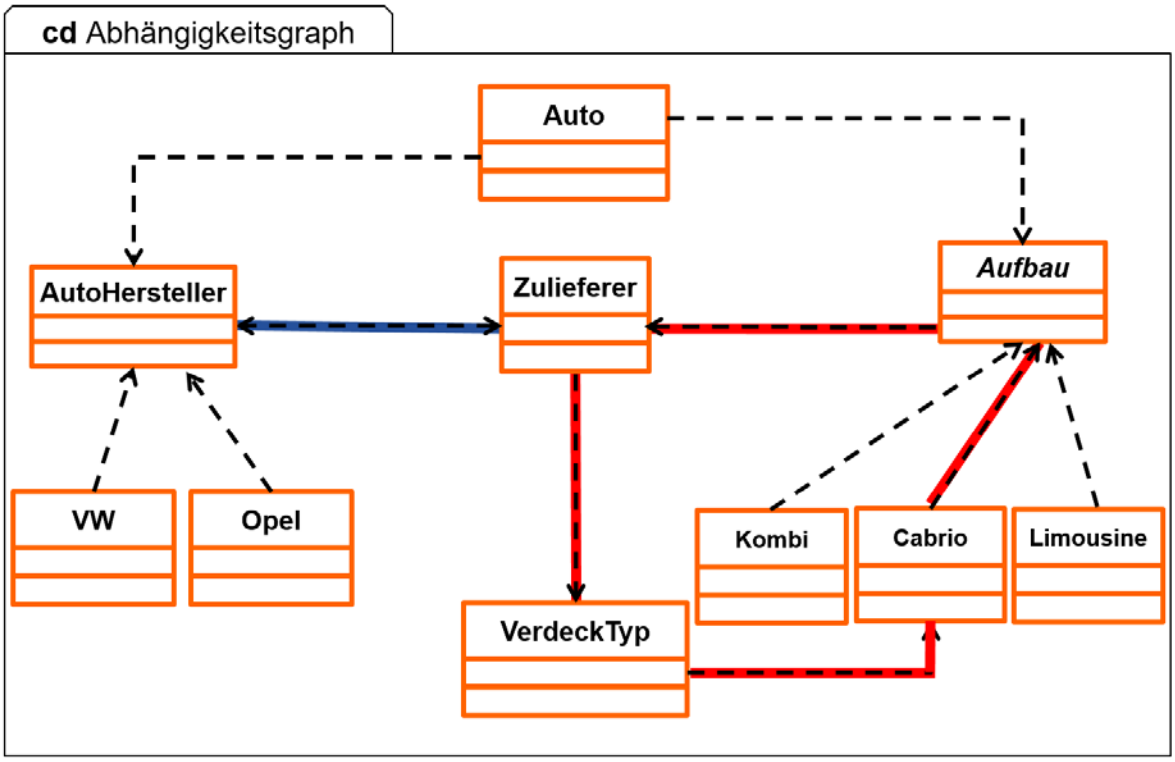
Aus den in einem Klassendiagramm verwendeten Beziehungen lassen sich folgende Abhängigkeiten erkennen:




Die Abhängigkeiten entsprechen der Reihenfolge der Übersetzung.

Betrachten Sie für die nachstehende Zeichnung Abhängigkeiten im Sinne der Übersetzungsreihenfolge eines Compilers.

Wenn eine Klasse verändert wird, müssen alle direkt oder indirekt davon abhängigen Klassen auf korrekte Funktionsweise geprüft werden.




Im Abhängigkeitsgraphen ist erkennbar:

- ein Zyklus (rot hinterlegt)
- eine gegenseitige Abhängigkeit (blau hinterlegt).
- Lange Abhängigkeitsketten:
  z.B. Auto → AutoHersteller → Zulieferer → Verdecktyp → Cabrio →Aufbau

a) Welche Klassen müssen überprüft werden?

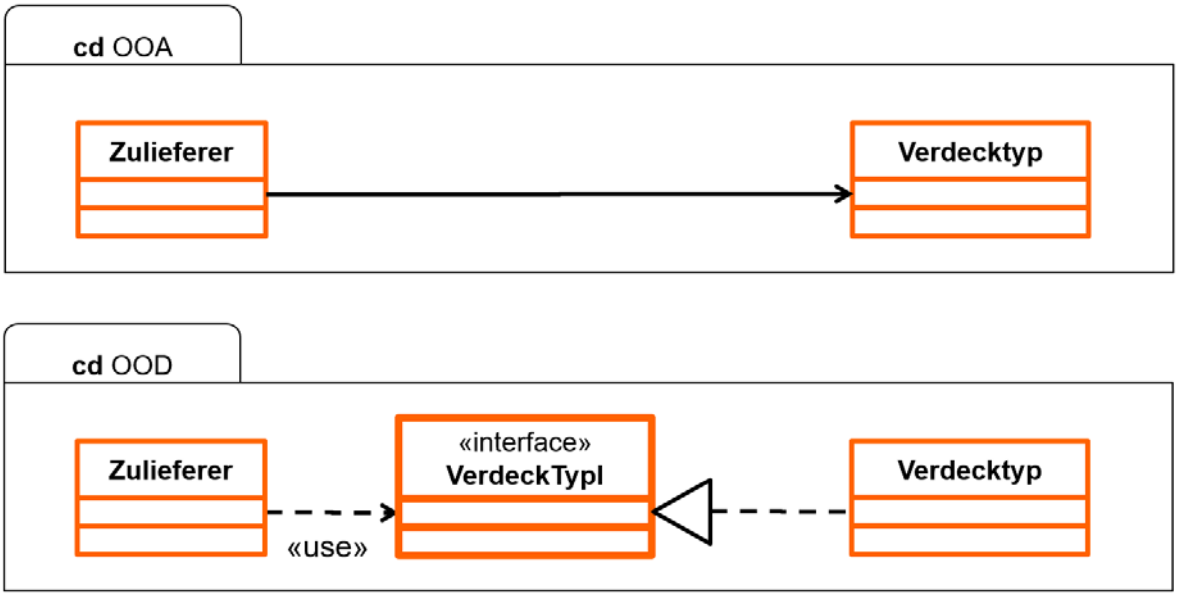
Es müssen alle Klassen überprüft werden, die direkt oder indirekt von der veränderten Klasse abhängen.

| **wenn verändert….** | **dann prüfen …** |
|---|---|
| Auto | Keine |
| Aufbau | Alle |
| VW | Keine |
| Opel | Keine |
| Kombi | Keine |
| Cabrio | Alle |
| Verdeck | Alle |
| AutoHersteller | Alle |
| VerdeckHersteller | Alle |

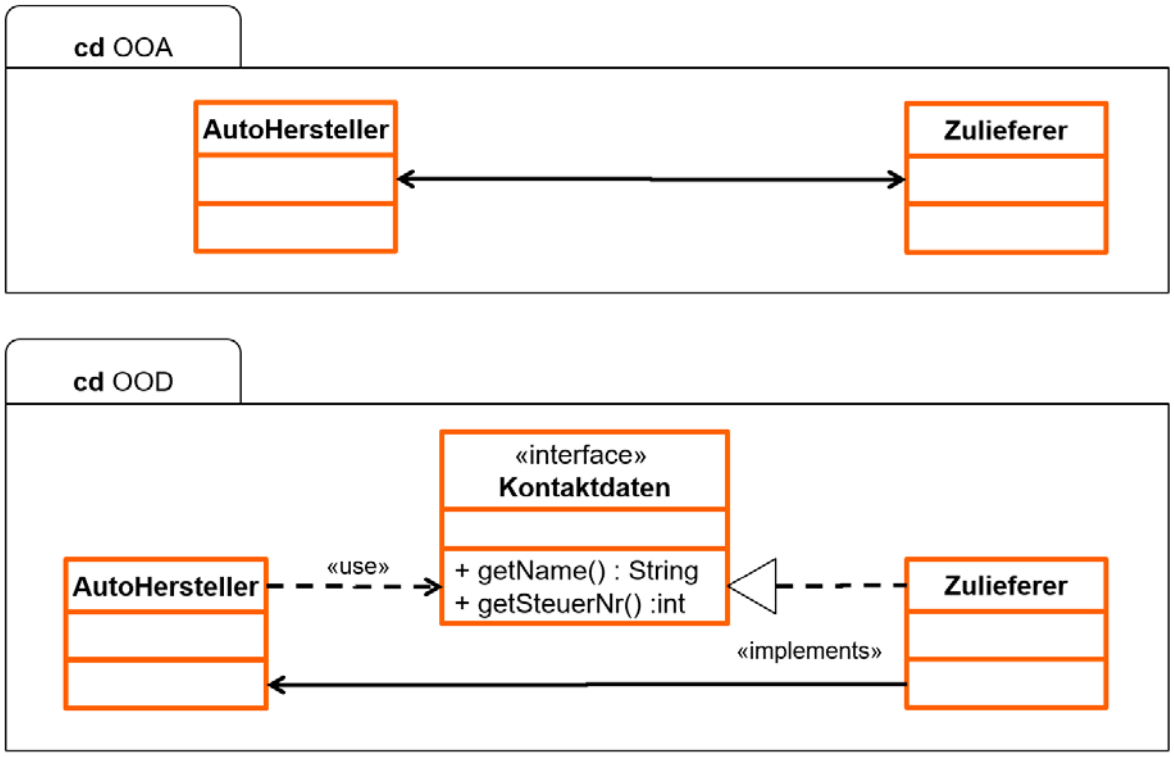
b) Wie lassen sich die Abhängigkeitsstrukturen verbessern?
Zeichnen sie das überarbeitete Klassendiagramm.

**Allgemeines Prinzip:**

Um (einfache) Abhängigkeiten von der konkreten Implementierung einer Klasse zu vermeiden hilft der Einbau von Schnittstellen.



Auch gegenseitige Abhängigkeiten können durch den geschickten Einbau einer Schnittstelle aufgelöst werden::




Mit der Einführung von Schnittstellen lassen sich auch Zyklische Abhängigkeiten vermeiden.

Anmerkungen:

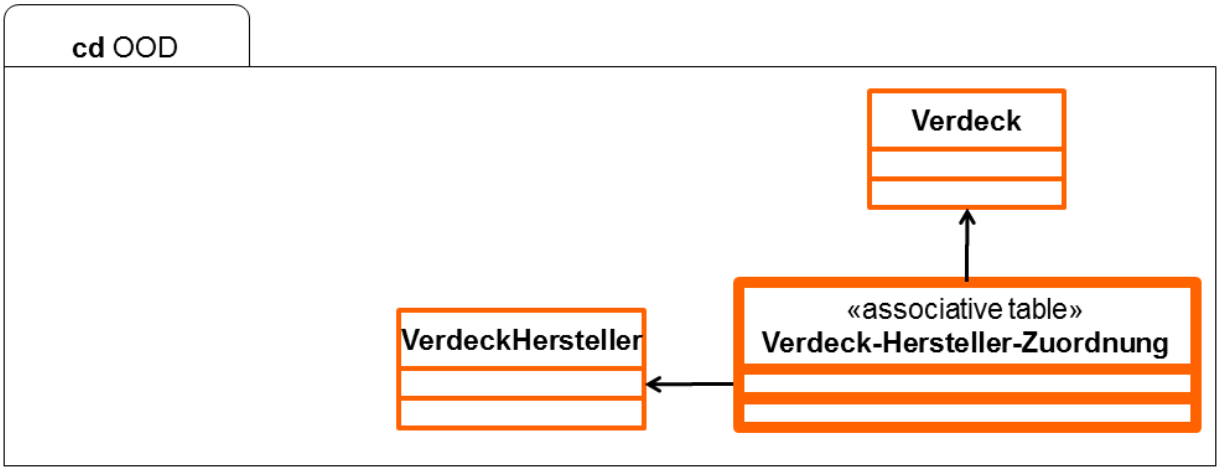
- Es reicht eine Schnittstelle einzuführen, um einen Zyklus aufzubrechen.
- Das weitere Einführen zusätzlicher Schnittstellen führt ggf. zu einer weiteren Reduktion der zu überprüfenden Klassen.
- Hat man mehrere Alternativen, Schnittstellen einzusetzen, so können unterschiedliche Kriterien herangezogen werden:
  Wodurch werden die größten Verstrickungen aufgehoben?
  Welche Schnittstelle ist 'stabil' und wird voraussichtlich nicht mehr geändert?

## Alternativen zu Schnittstellen

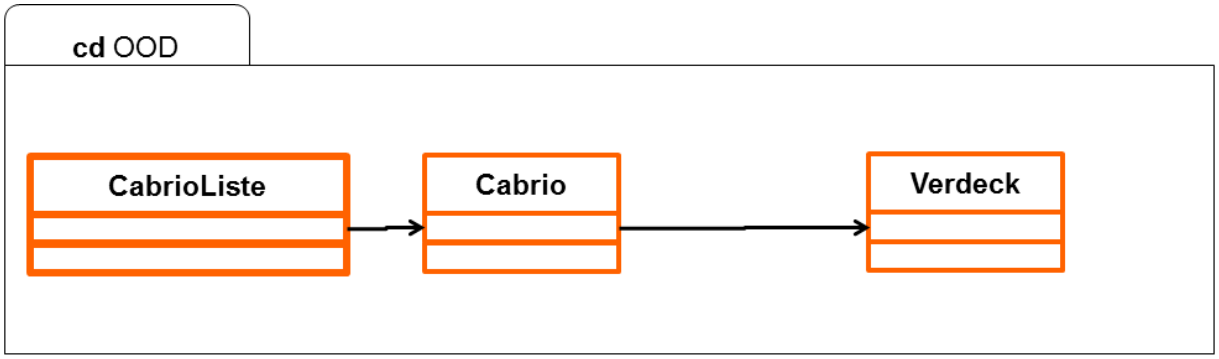
Gegenseitige Abhängigkeiten im OOA-Diagramm dürfen nicht einfach im OOD gestrichen werden.

Wichtig ist, dass man auch weiter seine Informationen über eine geeignete Navigation erreichen kann.

Eine Alternative zu Schnittstellen ist die Verwendung Assoziativer Klassen.




Eine weitere Alternative ist die Verwendung einer Liste:




- Es gibt keine Beziehung zwischen VerdeckHersteller und Cabrio.
  Dennoch kann man indirekt von VerdeckHersteller auf Cabrio schließen.
  Beispiel:
  Suche für VerdeckHersteller1 welche Cabrios er ausstattet.
  Man gehe mit dem Iterator durch die Cabrio-liste, findet dann über Cabrio den jeweiligen Verdeckhersteller. Man merkt sich dann die Cabrios, die auf VerdeckHersteller1 verweisen.
- Es gibt keine beidseitige Beziehung zwischen VerdeckHersteller und Verdeck.
  Diese Information wird über die Verdeck-Hersteller-Zuordnung erbracht.

# Aufgabe 2 (Allgemeine Fragen zu Abhängigkeiten)

a) Welche Bedeutung haben Zyklen in Klassendiagrammen?
Zyklen in Klassendiagrammen sind grundsätzlich ungünstig:

Alle Klassen, die von den betroffenen Klassen direkt oder indirekt abhängen müssen überprüft werden.

Bei einem Zyklus:
Wenn sich eine Klasse ändert von der der Zyklus (direkt oder indirekt) abhängig ist
dann müssen alle Klassen im Zyklus überprüft werden
und alle Klassen, die (direkt oder indirekt) vom Zyklus abhängig sind.

b) Was versteht man unter 'Implizite Abhängigkeiten'?
Wie kann man geeignet mit impliziten Abhängigkeiten umgehen?

Implizite Abhängigkeiten sind solche, die aus dem Entwurfsmodell (UML-Klassendiagramm) nicht ersichtlich sind.

| **Beispiele** | **Umgang/Vermeidung:** |
|---|---|
| Mehrfach definierte Konstanten | Zentrale Definition von Konstanten:<br>public static int anzSpalten = 8<br><br>Dies ist mit dem Singleton-Muster vermeidbar (siehe später) |
| Verwendung von Bezeichnern als Textkonstanten:<br>case 'Auto' : vehicle = new Auto(); | Dokumentation impliziter Abhängigkeiten |
| | |
| Klassen besitzen ein identisches Verhalten, unterscheiden sich nur in den Attributen | Java: Parametrisierte Klassen |
| Identische Abläufe mit unterschiedlichen Details in der Ausführung | Die ist mit dem Schablonenmuster vermeidbar<br>(siehe später) |

# Aufgabe 1 (vom OOA- zum OOD-Klassendiagramm)

Was sind die Gemeinsamkeiten und Unterschiede von OOA und OOD in Bezug auf den Zweck (der Diagramme und der ergänzenden Dokumentation)?

| **Objektorientierte Analyse (OOA)** | **Objektorientierter Entwurf (OOD)** |
|---|---|
| Gemeinsamkeiten | |
| | |
| Unterschiede | |
| OOA | OOD |
| | |

# Aufgabe 2 (vom OOA- zum OOD-Klassendiagramm)

| | | | |
|---|---|---|---|
| b) | Es gibt <u>eine</u> UML-Notation, die sowohl für die Analyse, als auch für den Entwurf verwendet wird.<br>Was sind hier die Unterschiede. | | |
| | | OOA | OOD |
| **Allgemeines** | Bezeichner | | |
| | Eigenschaftswerte<br>z.B. {ordered} | | |
| | Stereotype | | |
| **Klassen und Schnittstellen** | Parametrisierte Klassen<br>List<T> | | |
| | Collection Containerklassen | | |
| | Verwaltungsklassen | | |
| | Abstrakte Klassen, Schnittstellen | | |
| **Attribute** | Attribute | | |
| | Datentypen | | |
| | Klassenattribute | | |
| | Sichtbarkeit von Attributen | | |
| | Multiplizitäten bei | | |

| | | | |
|---|---|---|---|
| | Attributen | | |
| **Methoden** | | | |
| | In/out | | |
| | Sichtbarkeiten von Methoden | | |
| **Assoziationen** | (einfache ) **Assoziationen** | | |
| | Aggregation | | |
| | Komposition | | |
| | Assoziationsnamen | | |
| | Navigationsrichtung | | |
| | Rollen | | |
| | Sichtbarkeit bei Rollen | | |
| | Multiplizitäten bei Assoziationen | | |
| | Navigierbarkeit | | |
| **Vererbng** | Einfache Vererbung | | |
| | Mehrfachvererbung | • | • |
| | | | |
| **Sonstiges** | Pakete, Paketstrukturen | | |
| | | | |
| | Ausnahmen | | |
| | Polymorphismus | | |

# Aufgabe 1 (vom OOA- zum OOD-Klassendiagramm)

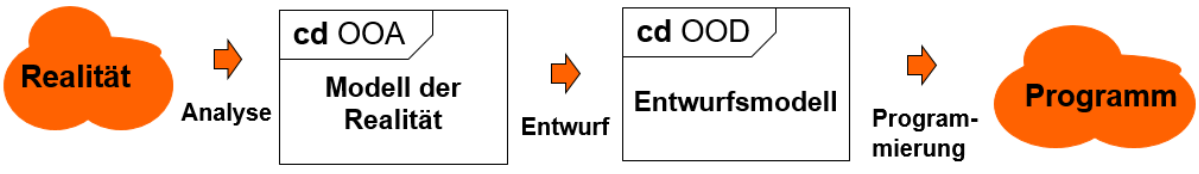
a) Was sind die Gemeinsamkeiten und Unterschiede von OOA und OOD in Bezug auf Zweck?

**Gemeinsamkeiten bzgl. des Zwecks:**

- OOA und OOD dienen zur **Modellbildung**:
  Darstellung von inneren Strukturen und von Beziehungen.
  Modell der Realität bzw. Modell des Programmes
- Beide **abstrahieren** von Details:
  OOA: von unwesentlichen Aspekten der Realität
  statt konkreter Objekte (Meier, Müller) nur die Klassen (Arzt, Patient)
  statt konkreter Werte (Schuhgröße des Arztes,…) nur die relevanten Attrbute
  bzw.
  OOD: von Details der Implementierung.
- Die UML-Diagramme dienen
  (1) zum Erfassen/Zusammentragen,
  (2) zur Konsolidierung
  (3) zur Dokumentation
  (4) zur Verbesserung/Optimierung.

Sowohl für die Objektorientierte Analyse (OOA), als auch für das Objektorientierte Design (OOD) wird die UML als Notation verwendet.

| **Objektorientierte Analyse (OOA)** | **Objektorientierter Entwurf (OOD)** |
|---|---|
| Modellierung der Realität<br>Wie ist die Anwendungsdomäne strukturiert?<br>• Beschreibung der Anwendungsdomäne<br>• Weitgehend ungeachtet der späteren Realisierung<br>• Programmiersprachen-unabhängig<br>• Geeignet auch für eine Papier-und-Bleistift-Lösung | Abstrakte Modellierung der Realisierung<br>'Wie' ist die Software strukturiert?<br>• Orientierung im Programm finden<br>• Berücksichtigung von Programmiersprechen-Spezifika (z.B. GUI und DB) |




| | |
|---|---|
| Modellbildung, Abstraktion:<br>• Begriffe und Fakten müssen modelliert werden.<br>• Sortierung nach Objekten<br>• Abstraktion:<br>Klassen statt Objekte | Modellbildung, Abstraktion:<br>• Die (Programm)-Strukturen müssen korrekt dargestellt werden<br>• Nicht jedes Attribut und jede Methode muss angeführt werden |

| | | |
|---|---|---|
| | • Weitere Abstraktion: Vererbung | |

| b) | Es gibt <u>eine</u> UML-Notation, die sowohl für die Analyse, als auch für den Entwurf verwendet wird.<br>Was sind hier die Unterschiede. | | |
|---|---|---|---|
| | | OOA | OOD |
| **Allgemeines** | Bezeichner | ... aus der Domäne des Anwenders<br><br><u>Beispiele:</u><br>Rezept, Patienten, Anamnese (Vorgeschichte,....) | ... in der Notation der Programmiersprache<br><br><u>Beispiele:</u><br>• Java-konform<br>• Wie die Implementierung in englischer Sprache |
| | Eigenschaftswerte z.B. {ordered} | Zur Dokumentation ergänzender <u>fachlicher</u> Eigenschaften<br><br><u>Beispiel:</u><br>{behandlungsdatum>aktuelles Datum}<br><br>vornamen : String[] [ordered} | Zur Dokumentation ergänzender <u>technischer</u> Eigenschaften<br><br><u>Beispiel:</u><br>{0<x<255} |
| | Stereotype | Zur Kategorisierung nach <u>fachlichen</u> Gesichtspunkten<br><br><u>Beispiel</u>:<br>• «Vertrieb», «Lager»,..<br>• «Labor», «Empfang», «Behandlungszimmer »<br><br>(Trennung nach Aufgabenbereichen, Sub-Domänen) | Zur Kategorisierung nach <u>technischen</u> Gesichtspunkten:<br><br><u>Beispiele:</u><br>• z.B. zur Darstellung von Zugehörigkeiten zu Subsystemen<br>beispielsweise: «GUI», «Fachklassen», «Peripherie»<br>•<br>• zur Unterscheidung von «intern» «extern»<br><br>Die Aufteilung muss disjunkt sein, beispielsweise. in<br>«Client»<br>«Server»<br>«Common» |
| **nd Schnittstellen** | Parametrisierte Klassen<br>Generische Klassen<br><br>`List<T>` | nein, es gibt die separaten Klassen<br><br><u>Beispiele:</u><br>• Behandlungsliste<br>• Adressverzeichnis | • Methodische Gemeinsamkeiten erkennen<br>• Redundanzen vermeiden ('Duplicate Code')<br>• Implizite Abhängigkeit vermeiden<br>•<br><u>Beispiele:</u><br>• add/remove/sort/search |
| | Collection Containerklassen | eher selten, nur wenn fachlich (exakt) so gefordert | Ja<br><u>Beispiele:</u><br>• Arrays, Listen, Tree, |

| | | | |
|---|---|---|---|
| | | Beispiel:<br>Rezept (Kopfdaten und bis zu 3 Verschreibungen)<br><br>Beim Student wird `getAnzahlMitstudenten()` als statische Methode notiert.<br><br>Gegenbeispiel: Inventurlisten | Vector, Tree, Matrix<br>• Prüfungsleistungen (fachlich/technisch) |
| | Verwaltungsklassen | Nein, diese Informationen sind im Use-Case-Diagramm notiert<br>Nein, hier wird nur das dem zugrunde liegende Informationsmodell notiert. | Subsysteme im Entwurf:<br>Beispiele:<br>• Studentenverwaltung, Raumverwaltung, Prüfungsverwaltung<br>• Patientenverwaltung<br><br>Verwaltungsklassen (Prüfungsverwaltung) nutzen Containerklassen (Prüfungsliste)und die wiederum die Basisklassen (Student, Prüfungsleistung) |
| | Abstrakte Klassen | • eher selten, eher nicht<br>• oftmals Vererbung<br>Arzt = Chefarzt, Oberarzt, Assistenzarzt<br>•<br><br>seltene Beispiele:<br>• Rezept = Privatrezept oder gesetzlich<br>• | Zweck:<br>• Synergieeffekte<br>• Redundanzen vermeiden<br>• Implizite Abhängigkeiten vermeiden<br><br>Beispiel:<br>Person = Arzt, Patient oder Mitarbeiter<br>• |
| | Schnittstellen | • eher unüblich,<br>• zur Darstellung einer gleichartigen Verwendung<br>• überwiegend bei technischen Systemen (Drucker)<br>• häufiger Vererbung | aus OOA übernommen<br>zusätzlich:<br>• Als Alternative zur Mehrfachvererbung<br>• Zum Hervorheben der gleichartigen Verwendung<br>• Zum Auflösen von Abhängigkeiten (getrennte Entwickelbarkeit, Testbarkeit)<br><br>Sowohl Ärzte als auch Pfleger dürfen spritzen:<br>... `implements inject` |
| **Attribute** | Attribute | Attribute der Fachobjekte | Zusätzliche Attribute<br>(fachlich wie technisch)<br>Beispiele:<br>flags, Hilfsattribute, Statusangaben,… |
| | Datentypen | Ggf. sehr grob:<br>ggf. Unterscheidung zwischen einfacher Genauigkeit und hoher Genauigkeit<br><br>Beispiele: | Datentypen der Programmiersprache, konkretisiert<br>aus '*' wird z.B. Byte (0..255'<br><br>int, short, long<br>int {0..255} |

| | | | |
|---|---|---|---|
| | | • ganzzahlig<br>• numerisch mit 2 Nachkommastellen | double, float |
| | Klassenattribute | Attribute, die einer Klasse zugeordnet werden, aber vom einzelnen Objekt unabhängig sind. | Wird in anderen Klassen umgesetzt:<br>Patientenliste<br>Parameter |
| | Sichtbarkeit von Attributen | Die Sichtbarkeit kann auch weggelassen werden. | Wichtig, Geheimnisprinzip:<br>+/-/#/~<br>Strenge Beschränkung nur über einen Methodenzugriff |
| | Multiplizitäten bei Attributen | Ja! | Ja! |
| **Methoden** | | | |
| | In/out | Nein! | • In Java gibt es keine direkte Umsetzung<br>Präzisierung:<br>Attribut dient nur der Eingabe, dient nur der Ausgabe (Geheimnisprinzip) |
| | Sichtbarkeiten von Methoden | Private Methoden können weggelassen werden | Wichtig zur Durchsetzung des Geheimnisprinzips |
| **Assoziationen** | (einfache ) **Assoziationen** | | |
| | Aggregation | | • |
| | Komposition | | |
| | Assoziationsnamen | Arzt behandelt Patient | Assoziationsnamen spielen bei der Implementierung keine Rolle mehr. Dort wird vielmehr der Rollenname verwendet. |
| | Navigationsrichtung | Dokumentation der Navigationsmöglichkeiten, die auch für die Anwendung benötigt werden.<br><br>z.B. Student →SBahn | Volle fachliche Navigierbarkeit aufrechterhalten<br>mit<br>• Einführung abgeleiteter Assoziationen, Schnittstellen etc.<br>Zweck:<br>Vermeidung<br>• ... gegenseitiger Abhängigkeiten<br>• ... Zyklen<br>• ... von Spinnennetzen<br>• .. von lange Abhängigkeitsketten |
| | Rollen | Bei der Analyse sind eher die Bezeichnungen der Assoziationen von Bedeutung. | Aus Rollen werden Referenzattribute<br>Beispiel:<br>`behandelndeAerzte` |
| | Sichtbarkeit bei | | Die Sichtbarkeiten von Rollen |

| | | | |
|---|---|---|---|
| | Rollen | | werden wichtig<br><br>«use»-Beziehungen werden eingeschränkt |
| | Multiplizitäten bei Assoziationen | | Entscheiden, über<br>One-to-one, one-to-may, many-to-many |
| | Navigierbarkeit | • Entsprechend der fachlichen Erfordernisse | • Spinnennetze vermeiden (Abgeleitete Asoziationen)<br>• Gegenseitige Abhängigkeiten vermeiden<br>• Zyklen vermeiden<br>• Lange Abgängigkeitsketten vermeiden |
| **Vererbng** | Einfache Vererbung | Fachliche motivierte ist_ein-Beziehung | |
| | Mehrfachvererbung | • ist erlaubt<br>• nach fachlichen Gesichtspunkten<br>• kommt in der Realität häufig vor:<br>ein Student ist auch ein Autofahrer, Mieter | Mehrfachvererbung in Java <u>nicht</u> möglich.<br>Es gibt Alternativen zur Mehrfachvererbung<br>• Verzicht auf Vererbung (z.B. nur Oberklasse oder nur Unterklassen)<br>• Schnittstellen<br>• Delegation |
| | | | |
| **Sonstiges** | Pakete, Paketstrukturen | Klassen können einzelnen Domänen zugeordnet werden | Pakete ordnen Implementierungsklassen, Sichtbarkeiten werden eingeschränkt, Namensräume |
| | | | • |
| | Ausnahmen | | Kontrollfluss-Ausnahmebehandlung<br>Exception-Handling |
| | Polymorphismus | | |

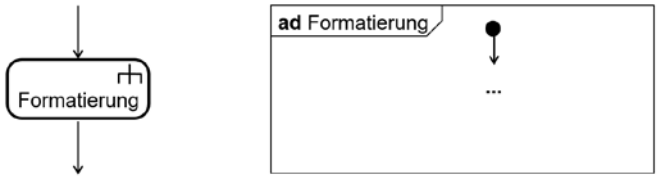
# Aufgabe 1
## (Aktivitätsdiagramm zur Strukturierung umgangssprachlicher Beschreibungen)

Erstellen Sie ein Aktivitätsdiagramm (ohne strukturierte Knoten).

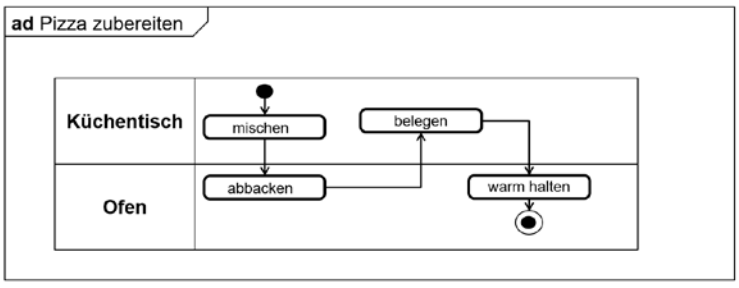
- Finden Sie geeignete Aktivitäten-Bezeichnungen
- Nutzen Sie geeignete Schwimmbahnen/Partitionen.
- Nutzen Sie– bei Bedarf - Aktionen mit Referenz

| | |
|---|---|
| /01/ | Auf dem Client wird eine Eingabemaske mit Auftragsdaten gefüllt. |
| /02/ | Nach Auslösen mit der 'Senden'-Taste werden die Angaben auf Plausibilität geprüft. |
| /03/ | Ist die Plausibilität nicht gegeben, so gelangt man zurück in der Eingabemaske. |
| /04/ | Ansonsten werden die Daten zum Senden vorbereitet:<br>• die Inhalte werden zunächst in ein Nachrichtenformat überführt<br>• die Nachricht wird danach komprimiert<br>• Danach werden die Konfigurationsdaten gelesen.<br>sofern in den Konfigurationsdaten vermerkt ist 'mit Verschlüsselung=ja' werden die Nachrichteninhalte zusätzlich verschlüsselt.<br>In dem Header der Nachricht wird diese Verschlüsselung vermerkt. |
| /05/ | Im Erfolgsfall wird die Nachricht zum Server gesendet. Ansonsten erfolgt eine spezifische Fehlermeldung auf dem Bildschirm. |
| /06/ | Nachrichten, die zum Server gelangen werden – sofern die Verschlüsselung vermerkt wurde – zunächst entschlüsselt. Danach werden sie dekomprimiert. |
| /07/ | Die Auftragsdaten werden aus dem Nachrichtenformat extrahiert und die entsprechende Buchung ausgeführt. |
| /08/ | Tritt auf dem Server ein Fehler (Entschlüsselung, Dekomprimierung, Datenextraktion, Buchung), so wird die spezifische Fehlermeldung auf dem Server protokolliert. |

**Modellierung mit Referenzen**




**Modellierung mit Schwimmbahnen**



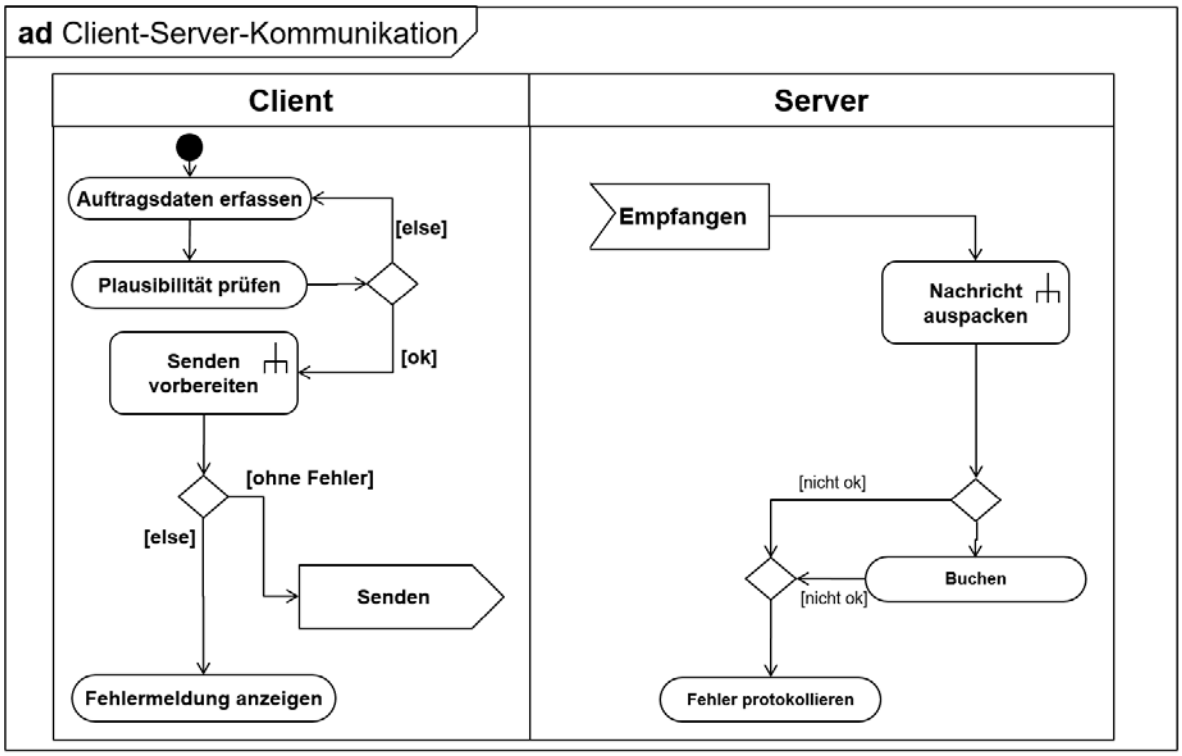
# Aufgabe 2
## (Aktivitätsdiagramm zum Erkennen von Spezifikationsmängeln)

Das Ergebnis der Aufgabe 1 zeigt, dass Aktivitätsdiagramme mehr Übersicht bieten, als beschreibende Texte.

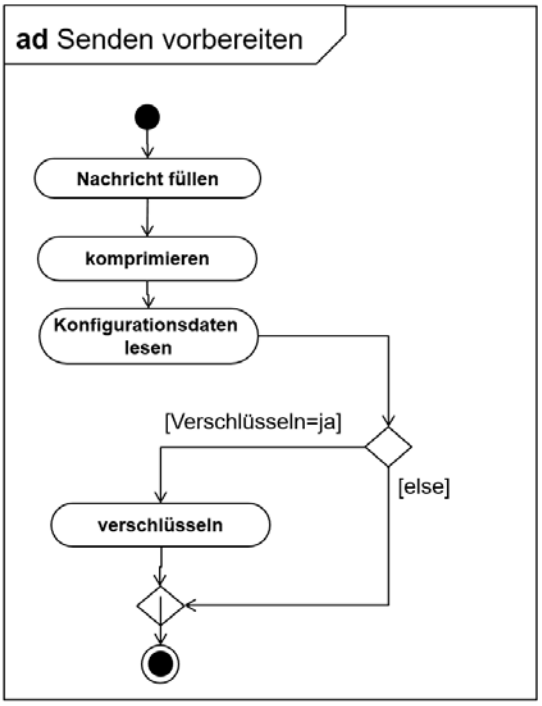
Anhand der Aktivitätsdiagramme lassen sich zudem auch Spezifikationsmängel erkennen: Fehlendes/Undefiniertes, Unvollständiges, Falsches, Überflüssiges, Unpräzises, Redundantes u.a.m.

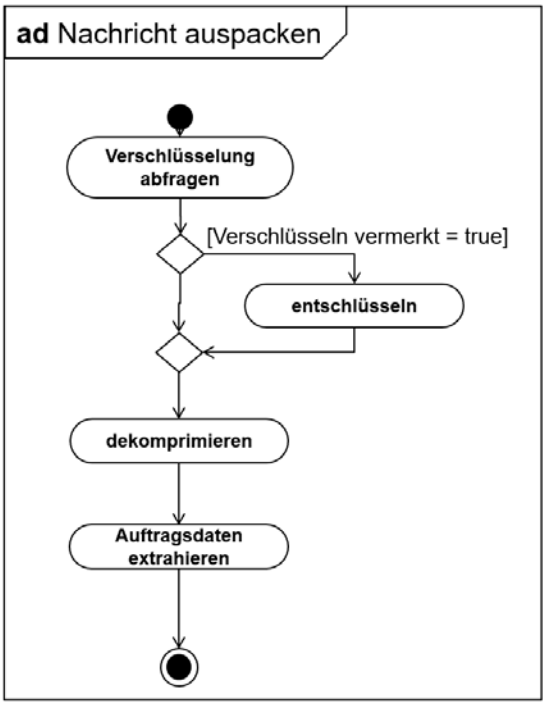
Welche Mängel sind in der Lösung erkennbar?

# Aufgabe 1 (Aktivitätsdiagramm ohne strukturierte Knoten)




**Referenzierte Aktivitäten:**



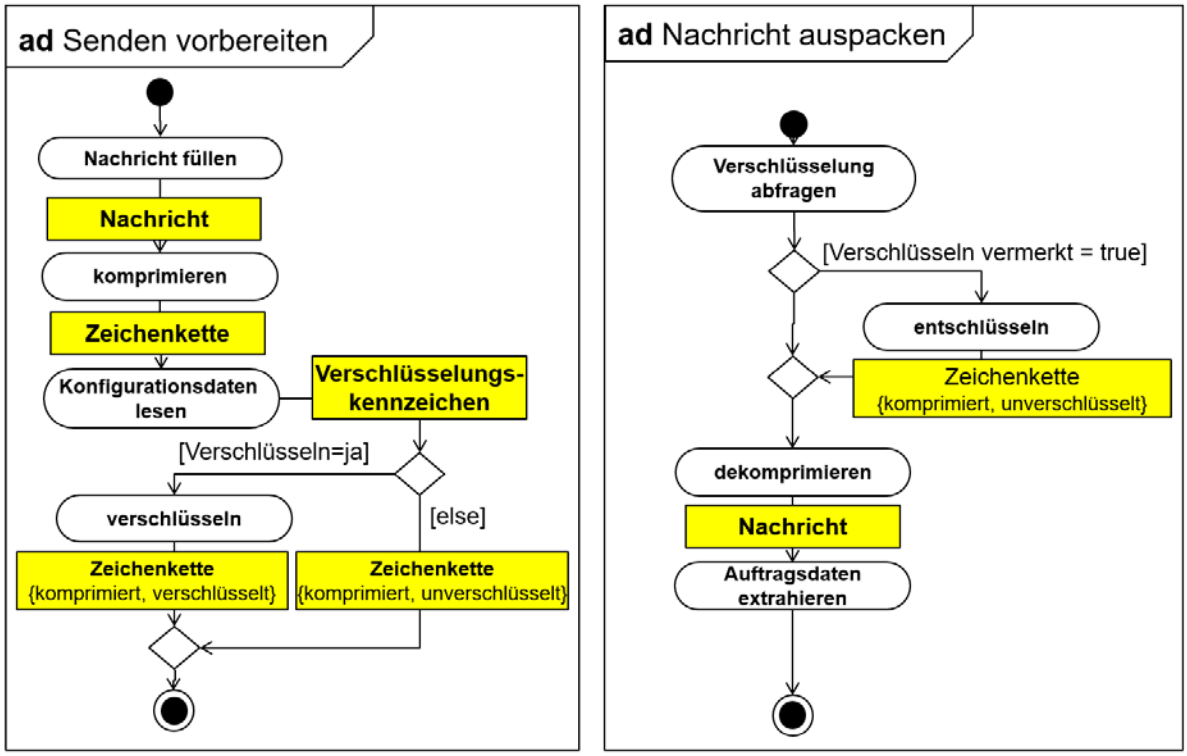
**Checkliste zum Aktivitätsdiagramm(unvollständig)**

| **UML-Syntax** | |
|---|---|
| | Hat das Aktivitätsdiagramm einen Anfangs- und einen Endknoten in UML-Syntax. |
| | Sind Entscheidungspunkte als Rauten dargestellt? |
| | Sind alle Ausgänge aus Entscheidungsknoten mit der Bedingung in eckigen Klammern bezeichnet? |
| | Sind alle Knoten mit gerichteten Pfeilen verbunden? |
| | Sind Ausnahmen (exceptions) – sofern vorhanden - korrekt notiert? |
| | … |

| **Fachliche Umsetzung** | |
|---|---|
| | Sind alle Kontrollflüsse korrekt umgesetzt? |
| | Ist der Bezug von den (Programm-) Anweisungen zu den Aktivitäten im UML-Aktivitätsdiagramm erkennbar?<br><br>Es braucht keine exakte 1:1-Äquivalenz zwischen Programmierzeile und Aktion zu geben,<br>aber<br>einen klar erkennbaren Bezug |

Gefordert war nur den Kontrollfluss durch das Aktivitätsdiagramm aufzuzeigen.

Es ist aber auch sinnvoll, sich Gedanken über den Datenfluss/Informationsfluss/Objektfluss zu machen und dies in den Aktivitätsdiagrammen zu ergänzen.



# Aufgabe 2
## (Aktivitätsdiagramm zum Erkennen von Spezifikationsmängeln)

Das Ergebnis der Aufgabe 1 zeigt, dass Aktivitätsdiagramme mehr Übersicht bieten, als beschreibende Texte.

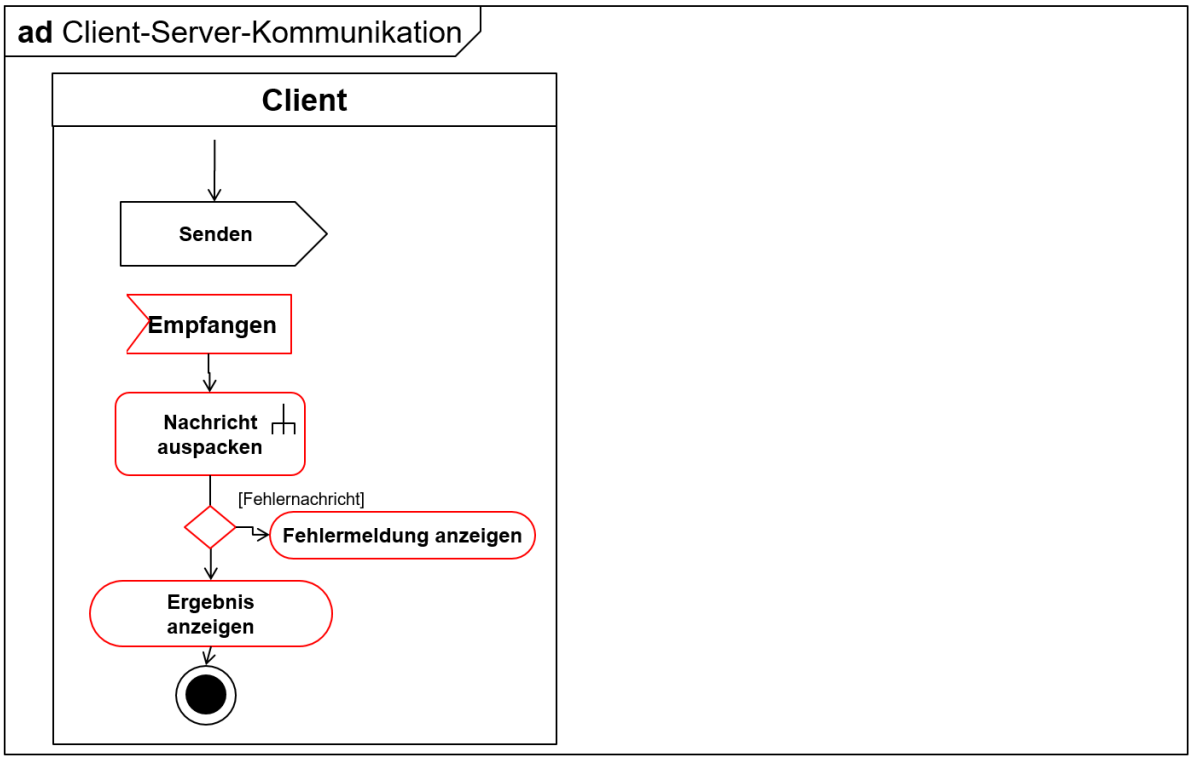
Anhand der Aktivitätsdiagramme lassen sich zudem auch Spezifikationsmängel erkennen: Fehlendes/Undefiniertes, Unvollständiges, Falsches, Überflüssiges, Unpräzises, Redundantes u.a.m.

Es ist aus den Formulierungen nicht erkennbar, wie der Kontrollfluss nach den Aktionen 'Fehlermeldung anzeigen', 'Fehlermeldung protokollieren' bzw 'Buchen' aussieht.

Welche Mängel sind in der Lösung erkennbar?

| | |
|---|---|
| Client-seitig | Was passiert Client-seitig nach dem Senden der Nachricht?<br>Was passiert Client-seitig nach „Fehlermeldung anzeigen“ |
| Server-seitig | Fehler protokollieren, auch wenn kein Fehler aufgetreten ist |

Einige der erforderlichen Ergänzungen sind im nachstehenden Diagramm notiert:



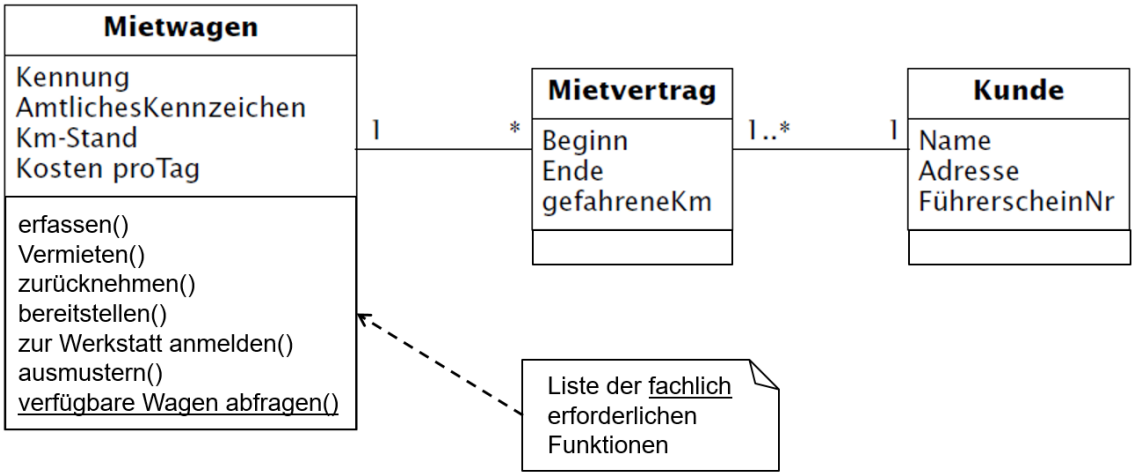
**Aktivitätsdiagramm im Vergleich von OOA und OOD**

| OOA-Aktivitätsdiagramm | OOD-Aktivitätsdiagramm |
|---|---|
| Spiegeln fachliche Abläufe wieder | Geben eine Abstraktes Modell der Kontrollflüsse (ggf. der Datenflüsse) im Programm wieder |
| | Detailebene mit GUI |
| | |
| | Zusätzliche Elemente:<br>Exception |
| | |
| Reihenfolgen rein fachlich! | Zusätzliche Festlegungen über die zu programmierende Verarbeitungsreihenfolge |
| | Es braucht keine exakte 1:1-Äquivalenz zwischen Programmierzeile und Aktion zu geben,<br>aber<br>einen klar erkennbaren Bezug |
| | Es kann zwischen Kontrollfluss und Datenfluss unterschieden werden. |

v

# Aufgabe 1 (Zustandsautomat, 'State Pattern‘)

Gegeben ist aus der Objektorientieren Analyse folgendes Klassendiagramm:




Aus der textuellen Beschreibung des OOA-Klassendiagramms geht die Spezifikation der fachlichen Methoden hervor:

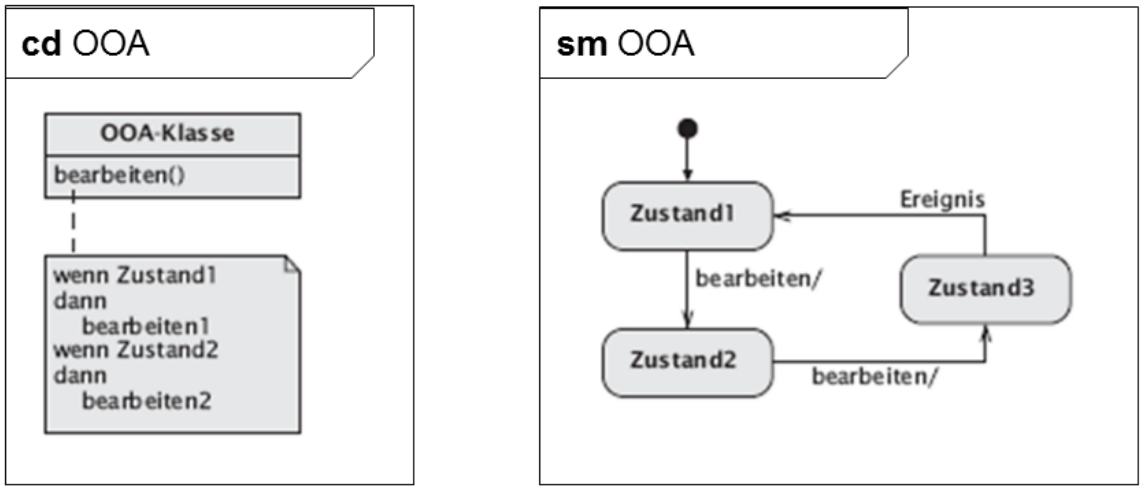
| **Fachliche Methoden (Ereigniss)** | **Funktionalität** |
|---|---|
| vermieten() | Ein neues Objekt von Mietvertrag erzeugen und entweder vorhandenem Kunden zuordnen oder neuen Kunden erfassen |
| zurücknehmen() | Gibt der Kunde den Mietwagen vertragsgemäß zurück, so wird das Auto gewaschen und überprüft. |
| bereitstellen() | Ergibt sich bei der Prüfung die einwandfreie Funktion, so wird der Wagen wieder in den Fuhrpark überstellt. |
| zur Werkstatt anmelden () | Ein defekter Mietwagen wird zur Reparatur angemeldet. Sobald ein Mietwagen repariert ist, wird er wieder bereitgestellt. |
| ausmustern() | Ergibt sich bei der Prüfung allerdings, dass der Kilometerstand des Mietwagens größer als 80000 km ist, dann wird er aus dem Fuhrpark des Mietwagen-Verleihs entfernt und zum Verkauf bereitgestellt. |

(1) Spezifizieren Sie für die Klasse Mietwagen den Lebenszyklus in Form eines Zustandsdiagrammes.

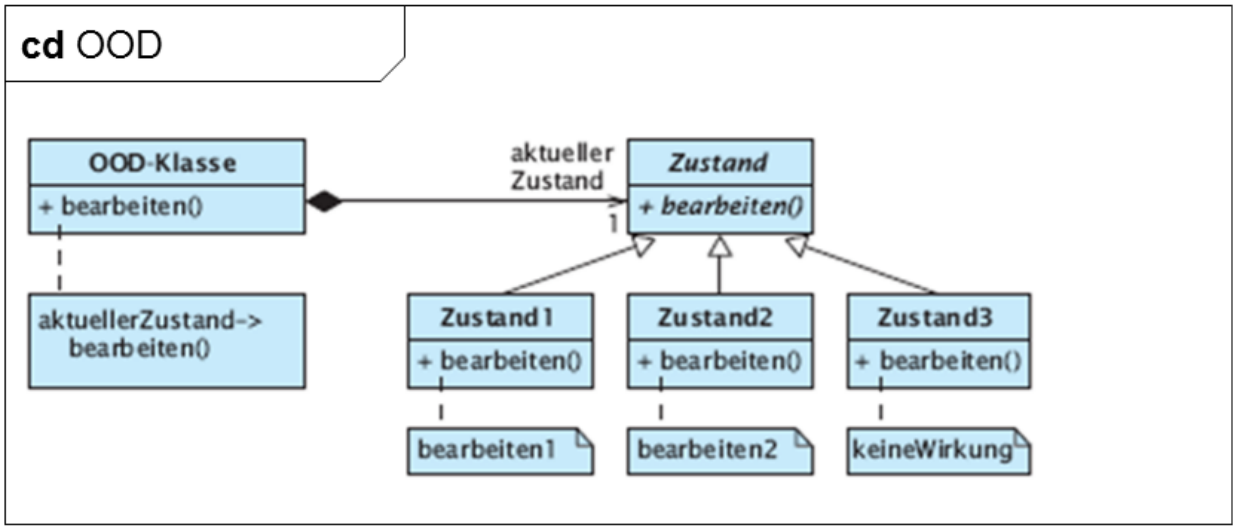
(2) Überführen Sie den Zustandsautomaten in ein OOD-Klassendiagramm nach dem Entwurfsmuster ‚State Pattern‘

Hinweis

(1) Gegeben ist die fachliche Fragestellung




(2) Gegeben ist das Entwufsmuster, in das die fachliche Fragestellung zu überführen ist:



# Aufgabe 1 (Zustandsautomat, 'State Pattern')

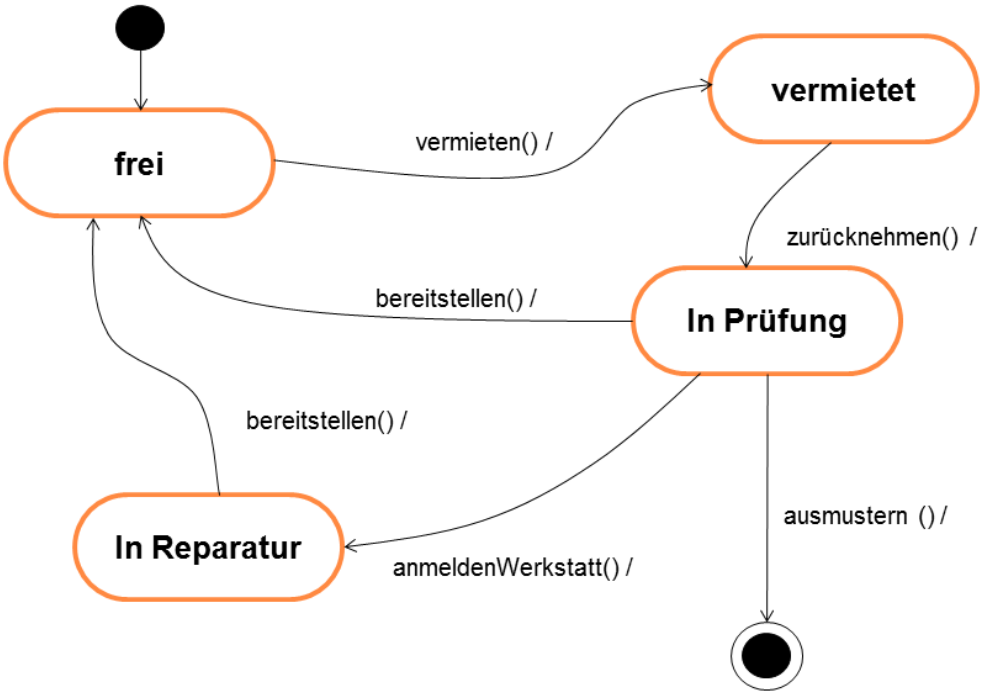
(a) Spezifizieren Sie für die Klasse Mietwagen den Lebenszyklus in Form eines Zustandsdiagrammes.

Mit den nachstehenden Zuständen kann der gesamte Lebenszyklus (d.h. sämtliche Zeitabschnitte) überdeckt werden:

| frei | vermietet | in Prüfung | In Reparatur | |
|---|---|---|---|---|
| **frei** | **vermietet** | **in Prüfung** | **in Reparatur** | **ausgemustert** |

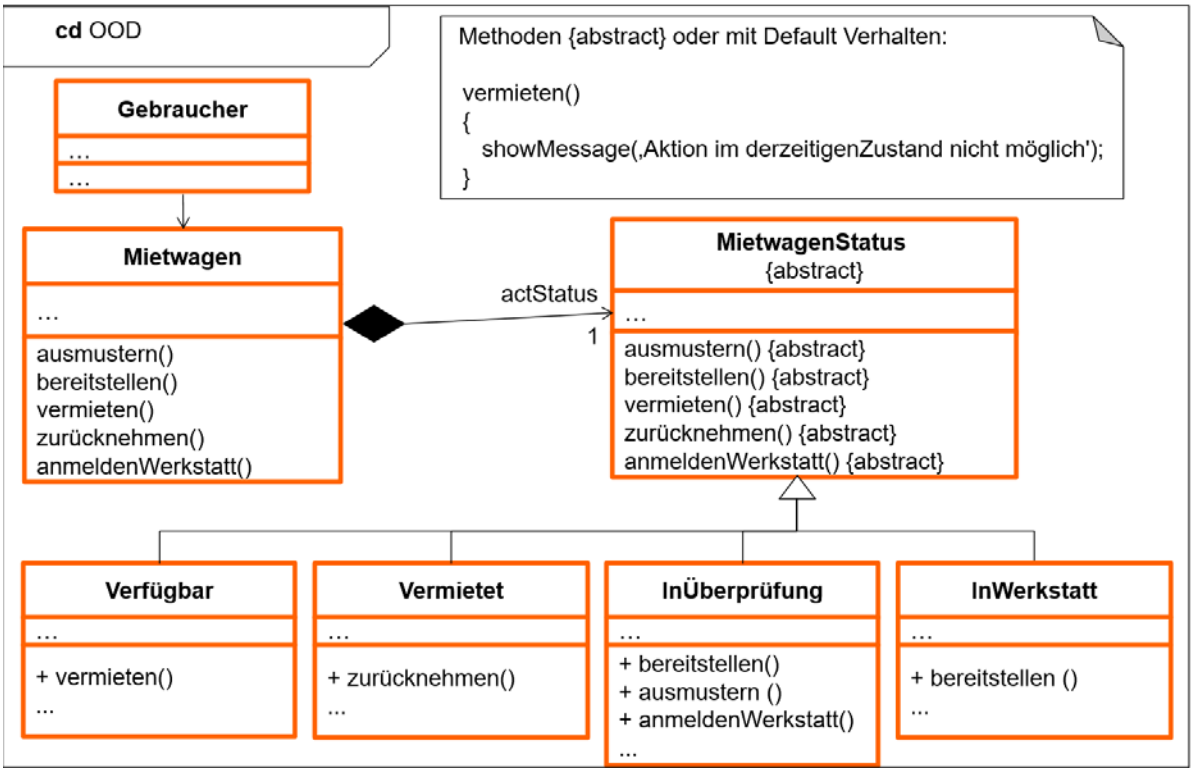
'Ausgemustert' kann – in Bezug auf die Vermietung - als Endpunkt betrachtet werden.

So sieht dann das Zustandsdiagramm aus:



In der Aufgabenstellung waren Angaben gemacht, die einem OOA-Zustandsdiagramm hinzuzurechnen sind: Diese Angaben wurden hier nicht übernommen (reparieren, waschen, prüfen).

(b) Auslöser für einen Zustandswechsel sind jeweils die angeführten Methodenaufrufe.
Überführen Sie den Zustandsautomaten in ein OOD-Klassendiagramm nach dem Entwurfsmuster ‚State Pattern'




Im vorliegenden UML-Diagramm wurde auf die Darstellung der Aspekte verzichtet (...), die im Zustandsautomaten nicht erlaubt sind.

•

Hinweise:

(1) Es gibt alternative Realisierungsmöglichkeiten:

- Alle Methoden in der Oberklasse werden {abstract}.
  In diesem Fall müssen in jedem konkreten Zustand sämtliche Methoden implementiert werden.

- In der Oberklasse werden die Methoden mit einem Default-Verhalten implementiert.
  In diesem Fall müssen im jedem konkreten Zustand nur die Methoden implementiert werden,
  bei denen das Verhalten vom Default abweicht (siehe UML-Diagramm).

(2) Zum endgültigen OOD-Klassendiagramm gehören natürlich weiterhin die Klassen 'Mietvertrag' und 'Kunde'.

(3) Es gibt in den Überlegungen oftmals alternative Begriffe:

| | | |
|---|---|---|
| Gleichwertige Begriff | z.B. frei“ oder „verfügbar“ | |
| | | |
| Falsche Begriffe: | z.B.<br>Prüfung<br>Prüfen | <br>Substantiviertes Verb<br>Verb |
| | prüfend | Dies ist eine Eigenschaft des Prüfers und nicht des Autos. |
| ungünstig | z.B. „zurückgenommen“ | Beschreibt ein Ereignis, welches verstrichen ist, weniger den Zeitraum.- |

# Aufgabe 1 (Vorteile Fabrikmuster)

Betrachten Sie das Factory-Entwurfsmuster.

(a) Mit welchen Klassen muss man sich auseinandersetzen, um das Fabrikmuster zu verwenden?

(b) Was muss programmiert werden, um die Anwendung um ein konkretes Produkt zu erweitern.

# Aufgabe 2 (Anwendbarkeit Fabrikmuster)

Das Factory-Muster kann in folgenden Fällen verwendet werden:

- Man hat eine Oberklasse und kennt im Voraus nicht die Klasse, von der ein Objekt zu erzeugen ist.
- Die unterschiedlichen Unterklassen sollen festlegen, von welcher Klasse ein Objekt erzeugt wird.

Benennen Sie hierfür konkrete Beispiele.

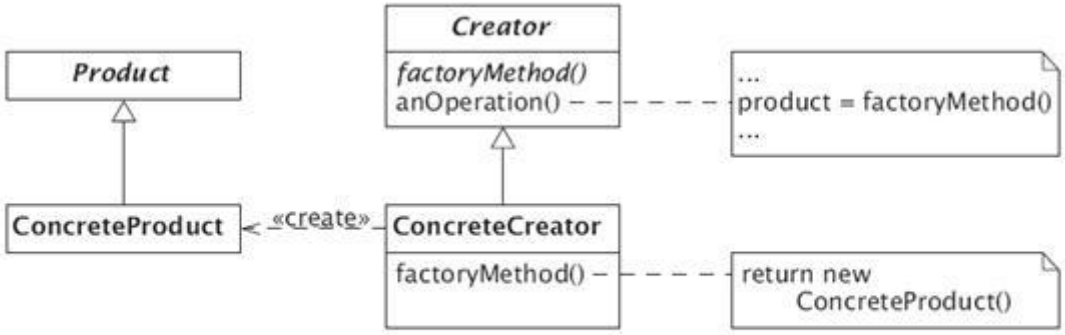
# Aufgabe 3 (Anwendung Fabrikmuster)

Sie haben in Ihrem Programm zu dem Namen und den Vornamen einer Person die Personalien gespeichert. Dabei wird unterschiedene zwischen: Masterstudenten und Bachelorstudenten, Hausmeister, Professoren und Gäste

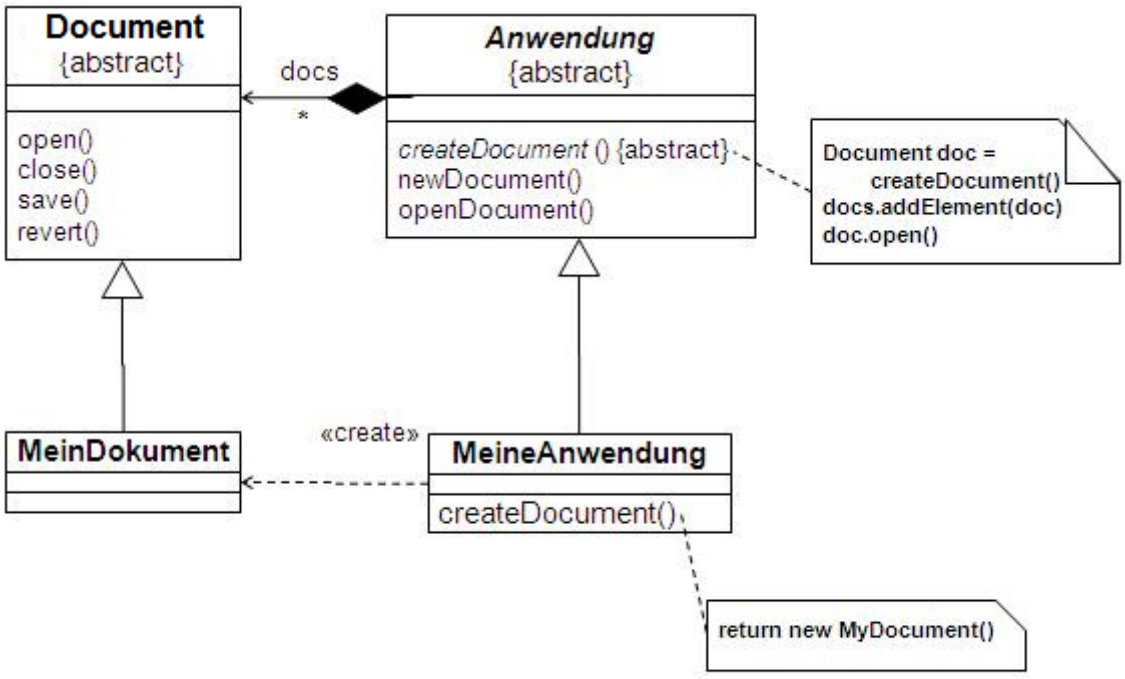
Sie möchten nun in einem Programm aufgrund von Name und Vorname ein Objekt der entsprechenden Klasse erzeugen um dann die Methode `showDetails()` aufrufen zu können, die je nach Berufsgruppe unterschiedlich implementiert ist.

Zeichnen Sie das entsprechende Klassendiagramm. Verwenden Sie zur Lösung das Factory-Entwurfsmuster!

# Factory-Muster




Quelle: Lehrbuch der Objektmodellierung, Balzert




Quelle: Lehrbuch der Objektmodellierung, Balzert

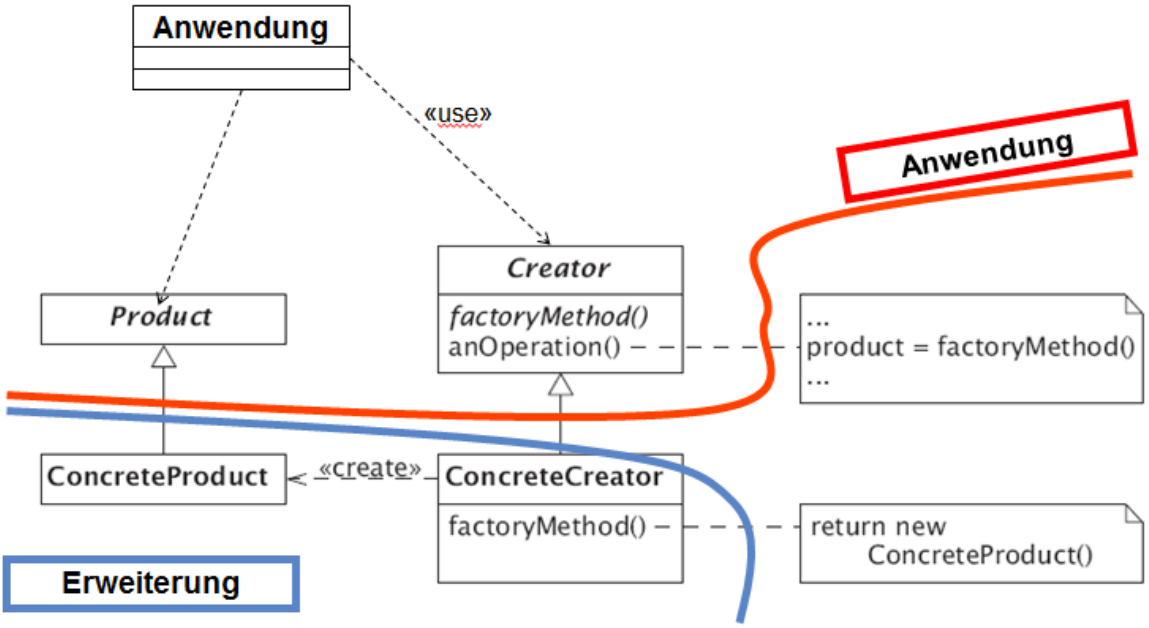
# Aufgabe 1 (Vorteile Fabrikmuster)

Betrachten Sie das Factory-Entwurfsmuster.

(a) Wenn man in einer Anwendung diese dieses Entwurfsmuster verwendet, so muss man sich nur mit den abstrakten Klassen `Product` und `Creator` auseinandersetzen (siehe unten).

(b) Um die Anwendung um ein konkretes Produkt zu erweitern ist nur Folgendes zu tun:

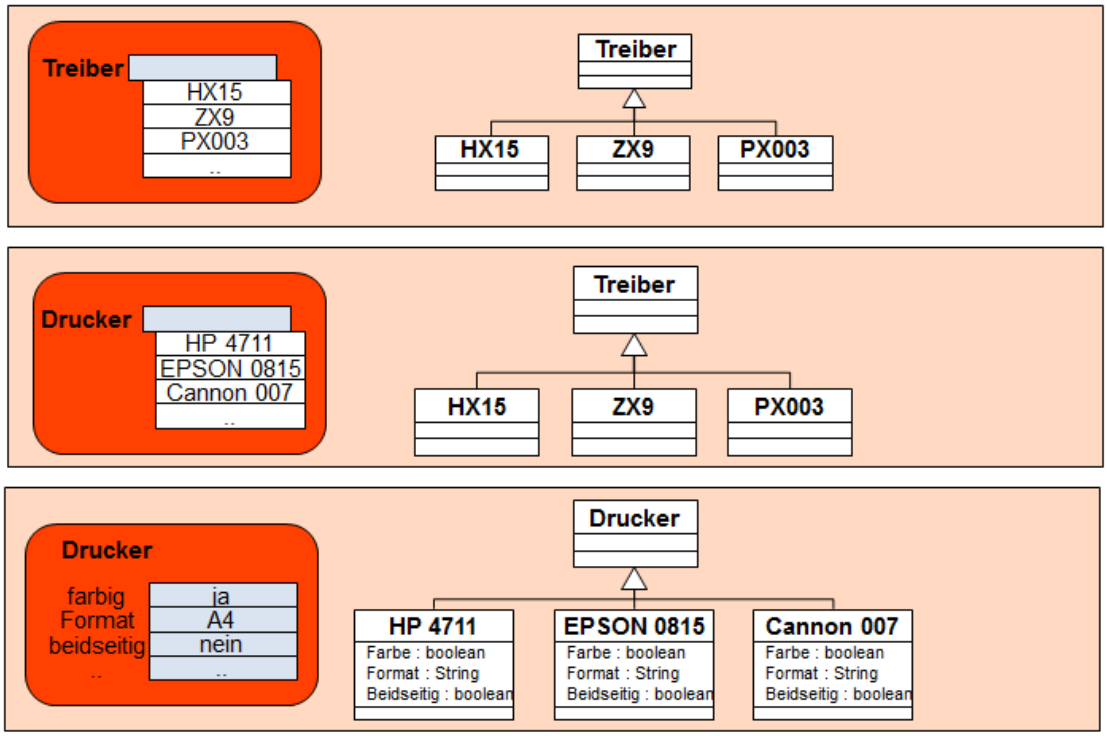
- Es muss nur ein konkretes Produkt implementiert werden.
- In der Klasse `ConcreteCreator` muss die `factoryMethod` nur um das konkrete Produkt erweitert werden.
  Bei Verwendung des Reflection-API's (siehe unten) entfällt auch diese Aufgabe.



## Aufgabe 2 (Anwendbarkeit Fabrikmuster)

Das Fabrikmuster kann in sehr unterschiedlichen Situationen verwendet werden:

1. Zum gegebenen Klassennamen (als Zeichenkette) wird eine Klasse gleichen Namens erzeugt
2. Aufgrund einer Bezeichnung (z.B. ein logischer Name) wird eine Klasse die entsprechende Instanz erzeugt.
3. Aufgrund verschiedener Attribute wird eine Klasse ermittelt und die entsprechende Instanz erzeugt.



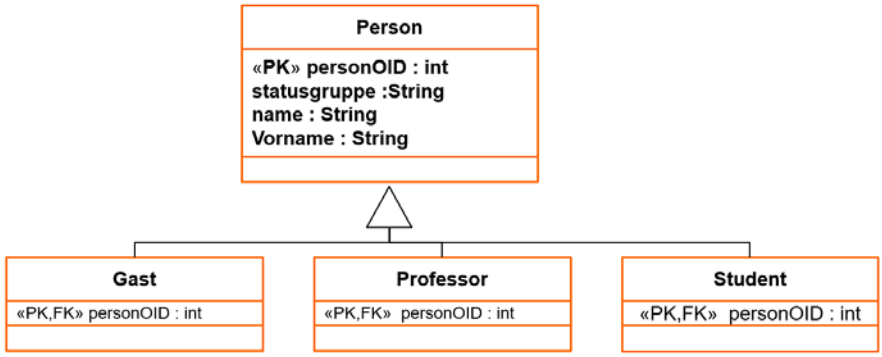

4. Aufgrund der Dateiendung (.pdf, doc) wird die 'Bearbeiterklasse' ausgewählt.
5. In einer Client-Server-Kommunikation wird der Klassenname über das Netzwerk übertragen.
6. Bei einem Bild wird das Motiv erkannt als 'Gruppenfoto', 'Portrait', 'Landschaftsaufnahme'.

7. Zur Laufzeit wird ein Logger ausgewählt, der die Logergebnisse wahlweise auf dem Bildschirm darstellt, in eine Datei speichert oder über Netz sendet.
8. Bei einem Entity-Manager wird aufgrund der ID das entsprechende Objekt der entsprechenden Klasse geladen.
9. Bei einem Shopsystem wird nach Auswahl der Kategorie 'Schuhe', 'Bücher' das entsprechende Suchobjekt geladen (mit Schuhgröße oder Erscheinungsjahr)
10. Aufgrund von Belegtypen (Bestellung, Reservierung, Stornierung) wird ein entsprechender Workflow etabliert.
11. Konten → Nummernkreise → Sparkonto, Girokonto Darlehenskonto

## Hinweis:

Mit dem Reflection-API von Java läßt sich das Fabrikmethode-Muster noch vereinfachen:

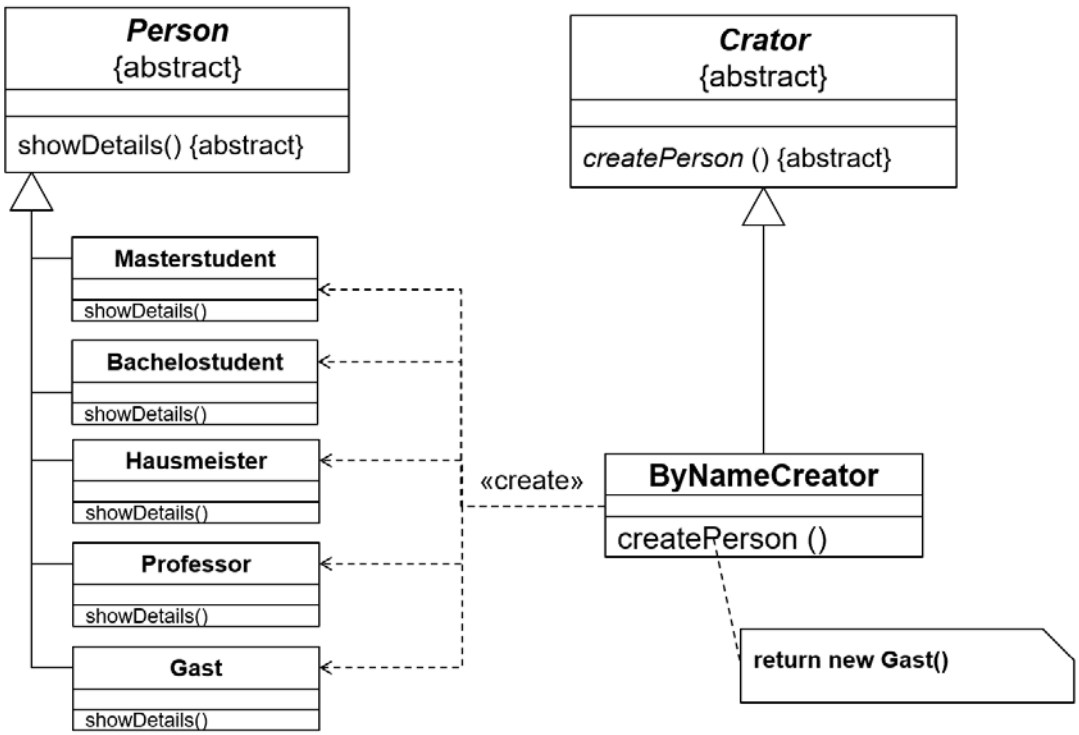
| | |
|---|---|
| Fabrikmethode-Muster mit Kontrollstrukturen | `public static Figur getFactoryFigur(String s)`<br>`{`<br>`  Figur temp = null;`<br>`  if (s.equals("Kreis")) temp = new Kreis ();`<br>`  else if (s.equals("Quadrat")) temp = new Quadrat();`<br>`  else if (s.equals("Dreieck")) temp = new Dreieck();`<br>`  else`<br>`  // … weitere Einträge für weitere Figuren`<br>`  return temp;`<br>`}` |
| Fabrikmethode-Mustermit Reflection | `public static Figur getFactoryFigur(String s)`<br>`{`<br>`  Figur temp = null;`<br>`  try`<br>`  {`<br>`  temp = (Figur) Class.forName(s).newInstance();`<br>`  }`<br>`  catch (Exception e)`<br>`  {`<br><br>`  }`<br>`  return temp;`<br>`}` |

| | |
|---|---|
| | |
| Fabrikmethode-Muster mit Datenbank | Datenbank<br>  <br><br>Mögliche Signatur der Fabrikmethode (in UML-Syntax):<br>`newInstance(name: String, vorname : String) : Person`<br><br>Die Fabrikmethode ermittelt aufgrund von Name und Vorname.aus der Tabelle Person die OID und die Statusgruppe.<br><br>Der Eintrag mit der entsprechenden OID wird dann als Objekt der spezifischen Statusgruppen-Klasse zurückgegeben. |

Wann hat man mehrere 'Concrete Creator'?

- Ich kann einen Druckeradapter erzeugen, in dem ich den Standort angebe.
- Ich kann auch einen Druckeradapter erzeugen, indem ich meine Druckanforderungen präzisiere (DIN A4, Farbe, beidseitig…)

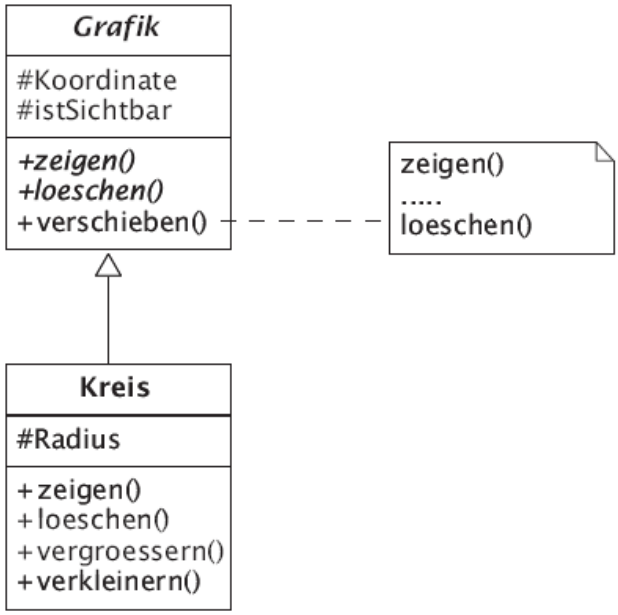
# Aufgabe 3 (Anwendung Fabrikmuster)

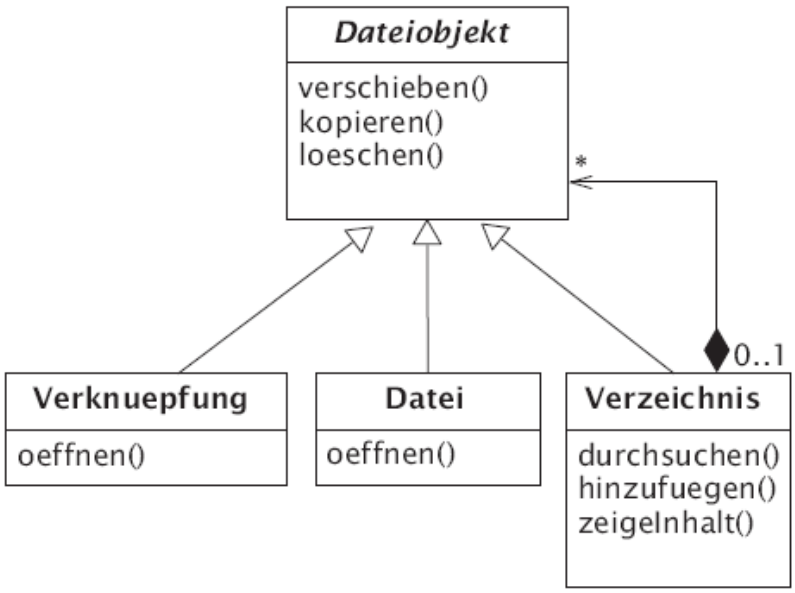


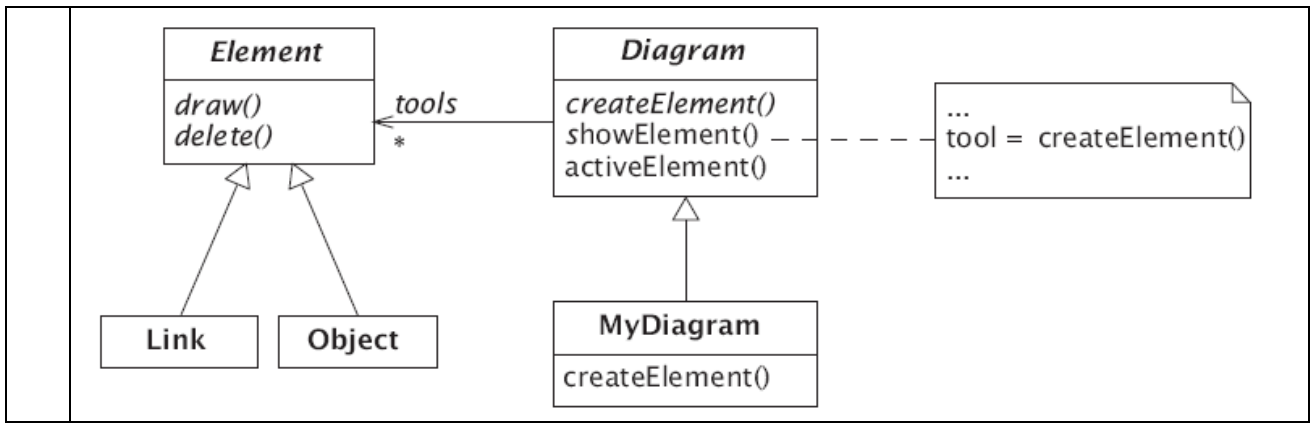
# Aufgabe 1 (Erkennen von Entwurfsmustern)

Geben Sie an, ob und gegebenenfalls welche Muster in den folgenden Klassendiagrammen beschrieben sind:

(a)



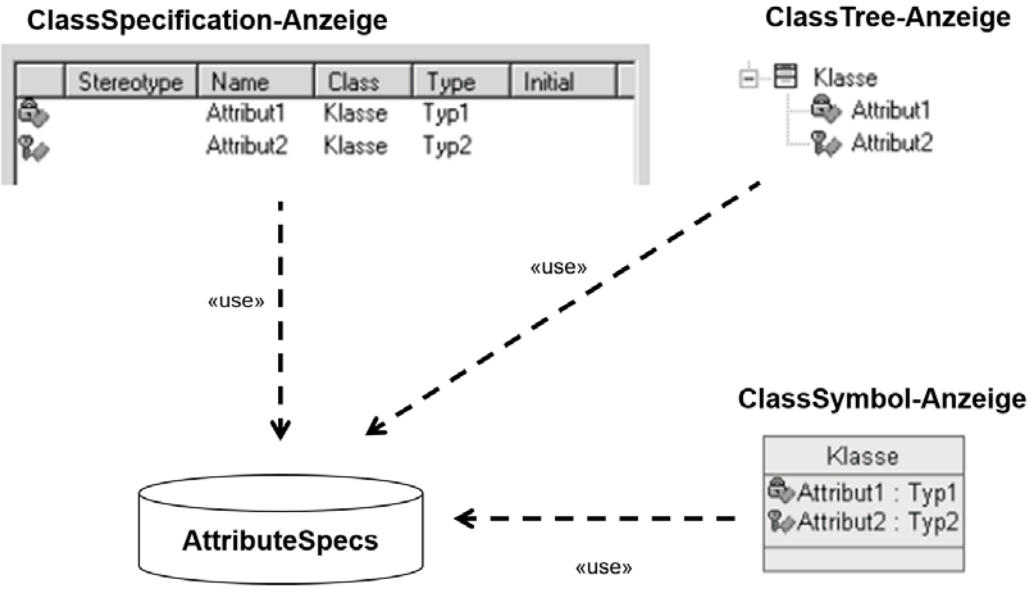



## Aufgabe 2 (Entscheidung zum Entwurfsmuster)

Sie sollen in einer Anwendung, aus allen Programmteilen heraus, Logeinträge schreiben. Dazu bieten Sie eine Logger-Klasse an, die die Konfiguration und das Herausschreiben der Datensätze übernimmt.
Welches Entwurfsmuster kann hier sinnvoll eingesetzt werden?
Begründen Sie ihre Antwort.

## Aufgabe 3 (Modellierung mit Entwurfsmustern)

Erstellen Sie für folgende Problemstellung ein Klassendiagramm. Prüfen Sie, welches der beschriebenen Muster vorliegt und wenden Sie es bei der Modellierung an.




Die Abbildung zeigt Ausschnitte aus der Benutzeroberfläche eines Werkzeugs für die objektorientierte Modellierung. Attribute werden in einem Spezifikationsfenster eingegeben, das durch die Klasse ClassSpecification realisiert wird. Die eingegebenen Daten sind ebenfalls im Klassensymbol (Klasse ClassSymbol) und in der Baumstruktur (Klasse ClassTree) sichtbar. Während im Spezifikationsfenster und im Klassensymbol die gleichen Informationen (AttributeSpecs) dargestellt werden, enthält der Baum nur die AttributeList.

# Aufgabe 1 (Erkennen von Entwurfsmustern)

Geben Sie an, ob und gegebenenfalls welche Muster in den folgenden Klassendiagrammen beschrieben sind:

(a) Schablonenmuster (*template method*)
Die Methode `verschieben`() ist eine Schablonenmethode.
Sie ruft die abstrakten Methoden `zeigen`() und `loeschen`() auf, die in der Unterklasse `Kreis` implementiert werden.

(b) Kompositum-Muster (*composite*)
Ein Verzeichnis-Objekt kann Objekte der Klassen `Verknüpfung`, `Datei` und `Verzeichnis` enthalten.
Zusammengesetzte und elementare Objekte werden gleich behandelt.

(c) Fabrikmethode (*factory method*)
In einem Diagramm sollen verschiedene Elemente dargestellt werden, wobei jede Diagrammart andere Elemente enthalten kann.
In der Klasse MyDiagram wird durch die Fabrikmethode createElement() konkret festgelegt, welche Objekte sie erzeugen soll.

# Aufgabe 2 (Entscheidung zum Entwurfsmuster)

| | |
|---|---|
| State-Pattern | • Kein Hinweis auf 'States'. |
| DAO/DTO | • Kein Hinweis auf eine gewünscht schmale Schnittstelle |
| Schablonenmuster | • Kein Hinweis auf einen generischen Algorithmus. |
| Kompositum-Muster | • Kein Hinweis auf Elemente und Zusammengesetztes |
| Fabrikmethoden-Muster | • Kein Hinweis auf Objekterzeugung |
| Proxy-Muster | Logger kann die Zusatzleistung vor Erbringung der ursprünglichen Funktion liefern. |
| Beobachter-Muster | Kein Hinweis auf Ereignis-Orientierung |
| Singleton | Ja!<br>• Zugriff von überall aus<br>• genau eine Instanz des Loggers |
| Fassade | • Kein Hinweis auf ungünstige Abhängigkeiten zu komplexen Paketstrukturen |
| Adapter | • Kein Hinweis auf unterschiedliche Schnittstellen, für die ein Übergang geschaffen werden soll. |

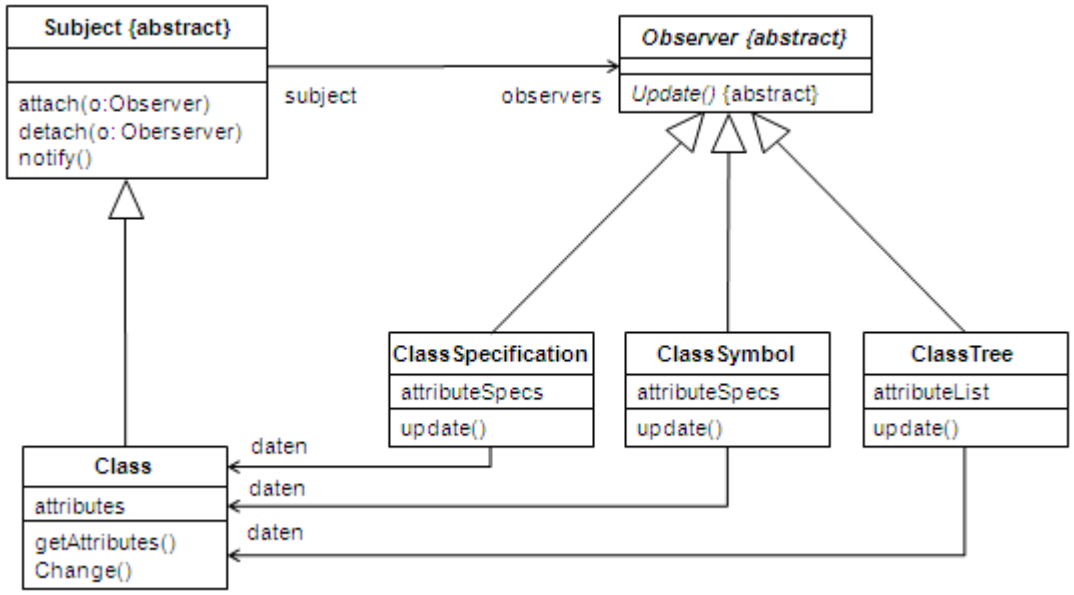
# Aufgabe 3 (Modellierung mit Entwurfsmustern)

Erstellen Sie für folgende Problemstellung ein Klassendiagramm. Prüfen Sie, welches der beschriebenen Muster vorliegt und wenden Sie es bei der Modellierung an.

Die Problemstellung läßt sich mit dem **Beobachter-Muster** realisieren.

Die abstrakten Klasse `Subject` als `'Observable'` und die abstrakte Klasse `Observer` bilden das Grundgerüst.

Für die Fachkonzeptklasse Class gibt es drei Sichten:
`ClassSpecification`, `ClassSymbol` und `ClassTree`



# Hinweis

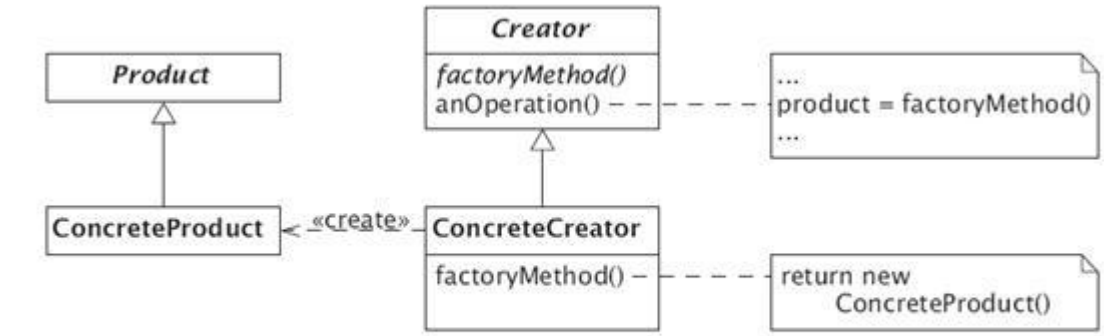
Erkennen von Entwurfsmustern:

| | |
|---|---|
| State | • Verschiedene Zustände<br>• Unterschiedliches Verhalten (zustandsabhängig) |
| Schablonenmuster | • Generischer Algorithmus<br>• Spezifische Aktionen |
| Kompositum | • Elemente und Zusammengesetztes<br>• ... und deren Gleichbehandlung |
| Fabrikmethodenmuster | • Fragestellung zur Erzeugung<br>• Klasse des Objektes von vornherein nicht bekannt |
| Proxy | • Trennung von Gebraucher, Kern Funktionalität und Zusatzfunktionalität |
| Beobachter | • Ereignisorientierung<br>• Observables (mit Änderungen)<br>• Mehrere Oberserver von Änderungen abhängig)<br>1 → * |
| Singleton | • Genau eine Instanz<br>• Aufruf von überall aus |
| Fassade | • Kraut und Rüben verstecken: Klasse(n) als Fassade |
| Adapter | • Zwei inkompatible Schnittstellen |
| | |

# Aufgabe 1

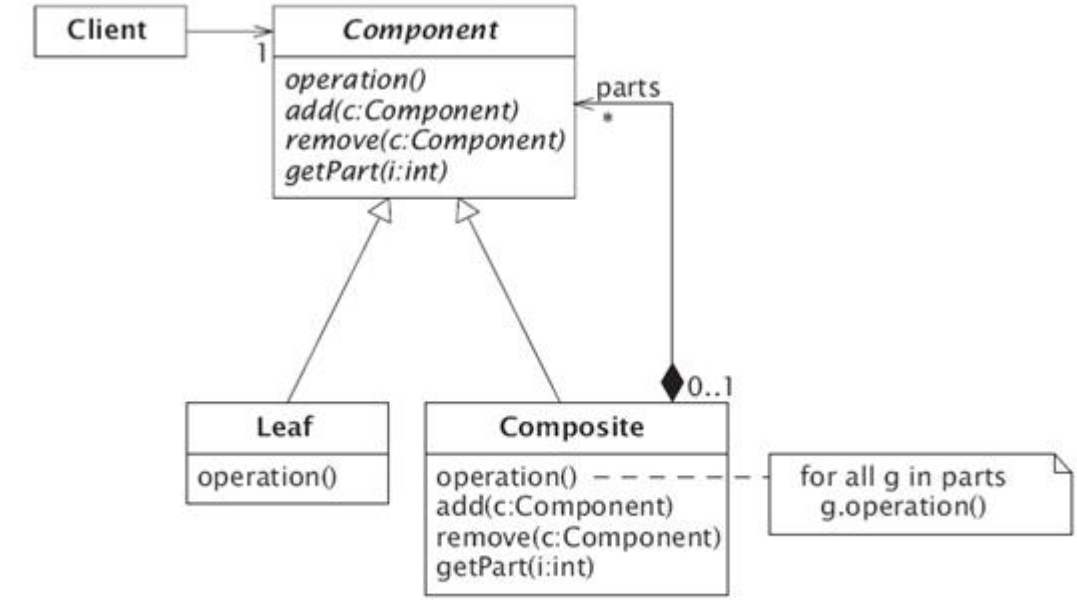
Untersuchen Sie für die beiden nachstehenden Entwurfsmuster, welche Architektur- und Designprinzipen dort berücksichtigt werden.
Füllen Sie hierzu die beigefügten Tabellen aus.

### Fabrikmethode-Muster




Quelle: Lehrbuch der Objektmodellierung, Balzert

### Composite-Muster




Quelle: Lehrbuch der Objektmodellierung, Balzert

## Fabrikmethode-Muster

| **Prinzip:** | **Begründung:** |
|---|---|
| Separation of concern | |
| Prinzip der Abstraktion | |
| Information Hiding | |
| Trennung von Interface und Implementierung | |
| Rigor and Formality | |
| Prinzip der Allgemeinheit | |
| Prinzip der Evolution/ Inkrementalität | |

## Composite-Muster

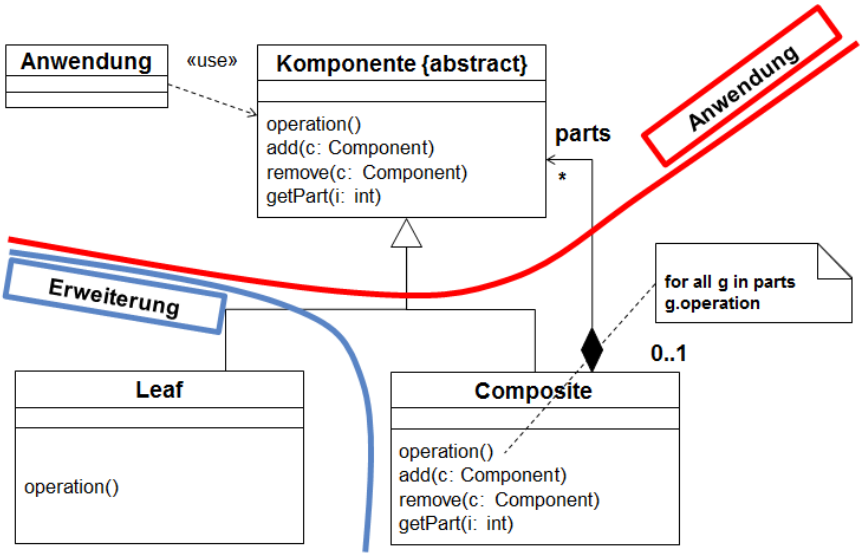
| **Prinzip:** | **Begründung:** |
|---|---|
| Separation of concern | |
| Prinzip der Abstraktion | |
| Information Hiding | |
| Trennung von Interface und Implementierung | |
| Rigor and Formality | |
| Prinzip der Allgemeinheit | |
| Prinzip der Evolution/ Inkrementalität | |

# Aufgabe 1

## Fabrikmethode-Muster

| Prinzip: | Begründung: |
|---|---|
| **Prinzip der Strukturierung** | Die Aufteilung der Verantwortlichkeiten trägt wesentlich zur Strukturierung bei. |
| **Separation of concern** | Die Verantwortlichkeiten der einzelnen Klassen sind exakt voneinander abgegrenzt:<br>• Die Frage der Objekterzeugung ist in einer einzigen Methode konzentriert.<br>In dieser Klasse passiert nichts Anderes.<br><br>(ungeachtet, was mit den Objekten passiert) |
| **Prinzip der Abstraktion** | Die Anwendungsklassen benutzen nur die Methoden der ‚abstrakten' Klassen.<br><br>Dies gilt nicht nur für Methoden der abstrakten Klasse 'Product' (Vorteil der Vererbung), sondern jetzt auch für die Objekterzeugung! |
| **Information Hiding** | Den Gebraucherklassen bleiben die Details der Objekterzeugung verborgen. |
| **Trennung von Interface und Implementierung** | Die Trennung von Interface und Implementierung wird hier über die abstrakte Klasse realisiert. |
| **Rigor and Formality** | Das Muster macht weitreichende Vorgaben bzgl. Struktur und Verarbeitung.<br>Die Vorteile hat man (nur) bei der 1:1-Umsetzung.<br>Verwendung braucht nicht gesondert dokumentiert zu werden. |
| **Prinzip der Allgemeinheit** | |
| **Prinzip der Evolution/ Inkrementalität** | Anwendung lässt sich sehr leicht erweitern um neue Komponenten und neue Fabrik-Instanzen. |

# Kompositum-Muster

| Prinzip: | Begründung: |
|---|---|
| **Prinzip der Strukturierung** | Die Aufteilung der Verantwortlichkeiten trägt wesentlich zur Strukturierung bei. |
| **Separation of concern** | Die Verantwortlichkeiten der einzelnen Klassen sind exakt voneinander abgegrenzt. |
| **Prinzip der Abstraktion** | Die Gebraucherklassen (Anwendung) greifen nur auf die abstrakte Komponente zu. |
| **Information Hiding** |  |
| **Trennung von Interface und Implementierung** | Die Schnittstelle (Oberklasse) dient zur Erstellung weiterer Objekte über die Komposition. |
| **Rigor and Formality** | Durch die Komposition und die Oberklasse besteht eine vorgegebene UML-Struktur, welche immer eingehalten werden muss. |
| **Prinzip der Allgemeinheit** | Das Entwurfsmuster lässt sich immer dann einsetzen, wenn in Teil-/Ganzes-Beziehungen dem Gebraucher Teile und Zusammengesetzten |
| **Prinzip der Evolution/ Inkrementalität** | Die Struktur kann durch weitere Blatt-Unterklassen leicht erweitert werden. |

# Singleton-Muster

| Prinzip: | Begründung: |
|---|---|
| **Prinzip der Strukturierung** | |
| **Separation of concern** | |
| **Prinzip der Abstraktion** | |
| **Information Hiding** | Die Art und Weise, wie sichergestellt wird, dass nur ein Objekt erzeugt wird und immer dasselbe zurückgegeben wird, bleibt den Gebraucher-Klassen verborgen. |
| **Trennung von Interface und Implementierung** | |
| **Rigor and Formality** | |
| **Prinzip der Allgemeinheit** | Das Muster lässt sich allgemein einsetzen. |
| **Prinzip der Evolution/ Inkrementalität** | Die Struktur kann durch beliebig viele Unterklassen erweitert werden. |

## Proxy-Muster

| Prinzip: | Begründung: |
|---|---|
| | |
| **Prinzip der Strukturierung** | |
| **Separation of concern** | Die Proxy-Klasse hat die Aufgabe, den Zugriff auf ein anderes Objekt zu steuern. Sie hat sowohl eine eigene Verantwortung, als auch die Aufgabe, die Methoden an ein anderes Objekt weiterzuleiten. |
| **Prinzip der Abstraktion** | |
| **Information Hiding** | Durch die Weiterleitung an ein anderes Objekt ist nicht bekannt, welche Funktionen in diesem anderen Objekt ausgeführt werden. |
| **Trennung von Interface und Implementierung** | Die Proxy-Klasse dient als Schnittstelle um die Methodenausführung an das andere Objekt weiterzuleiten. |
| | |
| **Rigor and Formality** | Es gibt eine festgelegte UML-Darstellung für dieses Muster, durch die Verbindung der Proxy-Klasse und der Klasse, an welche die Methoden weitergeleitet werden. |
| **Prinzip der Allgemeinheit** | |
| **Prinzip der Evolution/ Inkrementalität** | |

## Fassaden-Muster

| Prinzip: | Begründung: |
|---|---|
| **Prinzip der Strukturierung** | |
| **Separation of concern** | Die Fassade weiß, welche Klassen eines Paketes für die Bearbeitung eines Methodenaufrufes zuständig sind und delegiert Methodenaufrufe vom Klienten an die zuständige Klasse. |
| **Prinzip der Abstraktion** | |
| **Information Hiding** | Es wird nach außen verborgen, auf welche Klassen die Fassade zugreift. |
| **Trennung von Interface und Implementierung** | Die Fassade dient als Schnittstelle für die Verwendung der Klassen innerhalb des Pakets. |
| **Rigor and Formality** | Durch die Fassade wird die Struktur im UML vorgegeben. |
| **Prinzip der Allgemeinheit** | Die Klassen innerhalb der Pakete können beliebig erweitert werden. Zudem können auch weitere Pakete mit dem Fassaden-Muster erstellt werden. |
| **Prinzip der Evolution/ Inkrementalität** | |

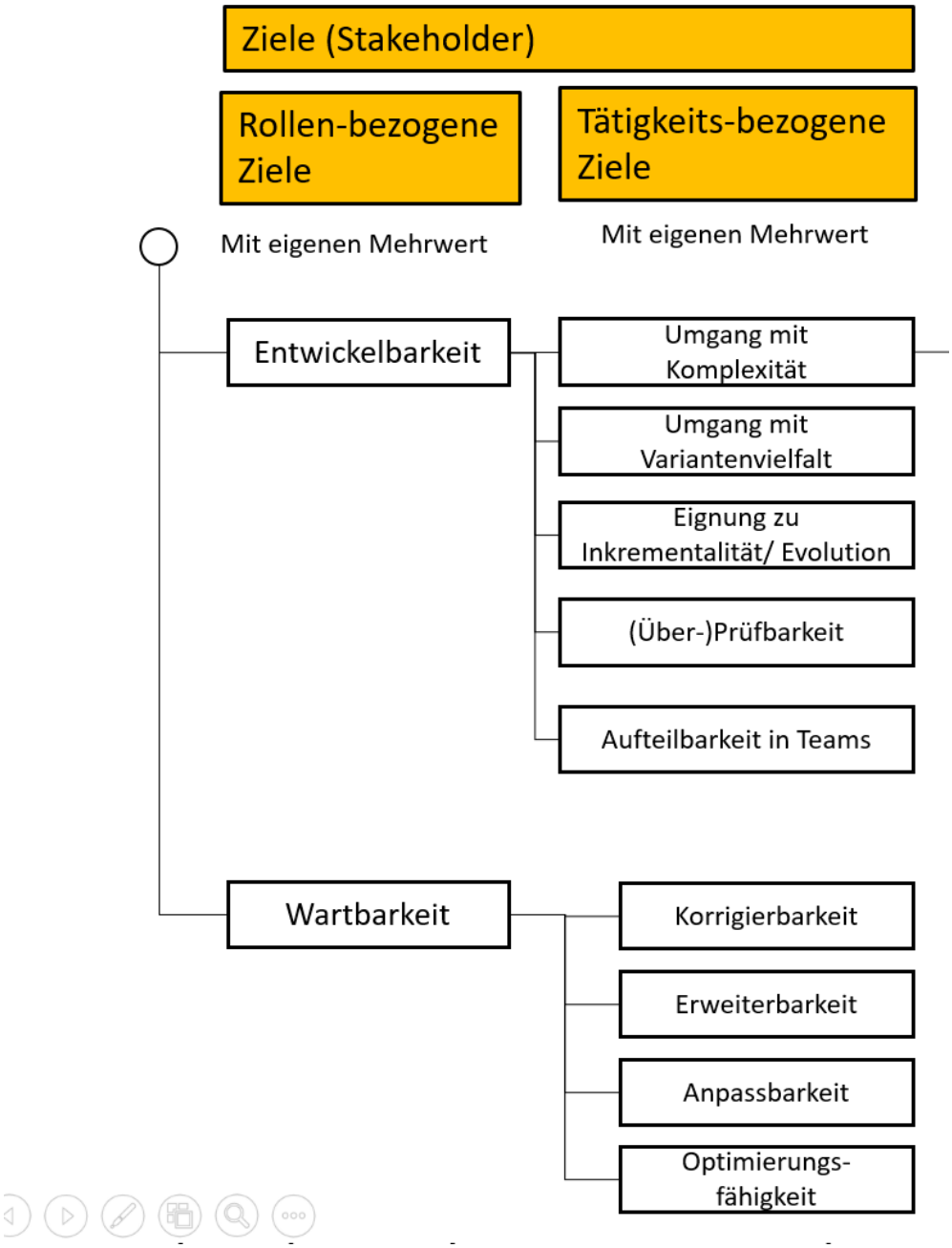
## Beobachter-Muster

| Prinzip: | Begründung: |
|---|---|
| **Separation of concern** | Die Oberserver haben die Aufgabe ein Objekt zu überwachen. Die Objekte, welche überwacht werden (observable), haben die Aufgabe, ihre Informationen an den Observer weiterzuleiten. |
| **Prinzip der Strukturierung** | |
| **Prinzip der Abstraktion** | |
| **Information Hiding** | Dem Observable-Objekt bleibt verborgen, was innerhalb der update() Methode eines Observers passiert. |
| **Trennung von Interface und Implementierung** | Die abstrakte Klasse „Subject“ dient nur als Interface zur Überwachung der Daten. |
| **Rigor and Formality** | Es liegt eine festgesetzte UML-Struktur für die Darstellung des Observers und Observable vor (Observer hat immer eine Beziehung zum observable). |
| **Prinzip der Allgemeinheit** | Man kann beliebig viele Observable-Objekte einfügen, welche jedoch alle eine Beziehung zu einem Observer haben müssen. |
| **Prinzip der Evolution/ Inkrementalität** | |

## Schablonenmethoden-Muster

| Prinzip: | Begründung: |
|---|---|
| **Separation of concern** | Jede Klasse hat eine eigene Verantwortung. |
| **Prinzip der Strukturierung** | |
| **Prinzip der Abstraktion** | Die Unterklassen lassen sich beliebig durch weitere Klassen erweitern. |
| **Information Hiding** | Die abstrakten Methoden sind in der nicht abstrakten Methode vorhanden, jedoch nach außen nicht sichtbar.<br><br>Beispiel am Schablonenmuster-Beispiel aus Übung 1: zeigen() und löschen() sind innerhalb der verschieben() Methode, jedoch ist nicht direkt erkennbar, welche Methoden innerhalb der verschieben() Methode ausgeführt werden. |
| **Trennung von Interface und Implementierung** | Die Abstrakte Klasse liefert nur die Schnittstelle für die Methoden, die in den konkreten Klassen zu implementieren sind. |
| **Rigor and Formality** | Das Muster wird durch eine festgelegte UML-Struktur dargestellt. |
| **Prinzip der Allgemeinheit** | |
| **Prinzip der Evolution/ Inkrementalität** | |

Welche Ziele habe ich bei der Software-Architektur?



Die nachstehenden Prinzipien wurden in der Vorlesung vorgestellt.

| | Beispiele, die den Prinzipien gehorchen |
|---|---|
| Prinzip der Strukturierung | • Funktionale Zerlegung (**Wann** wird **was** gemacht vs. **wie** wird es gemacht |
| Separation of Concern | • Objektorientierung<br>• Layer-Architekturen<br>• Treiber-Architekturen<br>• Entwurfsmuster |
| Prinzip der Abstraktion | • Virtuelle Maschine<br>• Interpreter |
| Information Hiding | • Sichtbarkeit (private)<br>• Fassade |
| Trennung von Interface und Implementierung | • Schnittstellen in Java<br>• Komponenten-Schnittstellen<br>• Collection-Interfaces |
| Rigor and Formality | • Programmierrichtlinien<br>• Dokumentationsstandards<br>• Schablonen<br>• UML |
| Prinzip der Allgemeinheit | • Auch über Schnittstellen, z.B. Beispiel Druckerschnittstellen |
| Prinzip der Evolution/ Inkrementalität | • Generische Schnittstellen |
| | |
| | |

# Warum begünstigen die Prinzipien die Zielerfüllung?

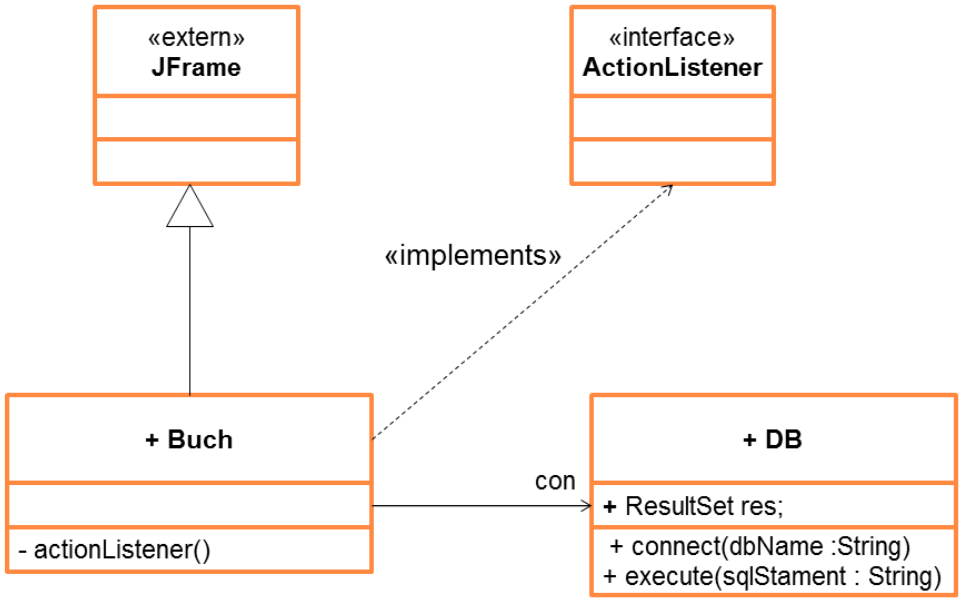
Überlegungen am Beispiel 'Interface und Implementierung'

| | | |
|---|---|---|
| Entwickelbarkeit | Umgang mit Komplexität | Trennung zwischen Gebrauch und Implementierung |
| | Umgang mit Variantenvielfalt | Alternative Implementierungen |
| | Eignung zur Inkrementalität/Evolution | Schnittstellen vorsehen, später sukzessive Implementierungen liefern |
| | Überprüfbarkeit | Testtreiber für Schnittstellen |
| | Aufteilung in Teams | Trennung über Schnittstelle |
| Wartbarkeit | Korrigierbarkeit | Testtreiber für Schnittstellen |
| | Erweiterbarkeit | ... zusätzliche Implementierungen |
| | Anpassbarkeit | ... generische Schnittstellen |
| | Optimierungsfähigkeit | ... Austausch von Implementierungen |

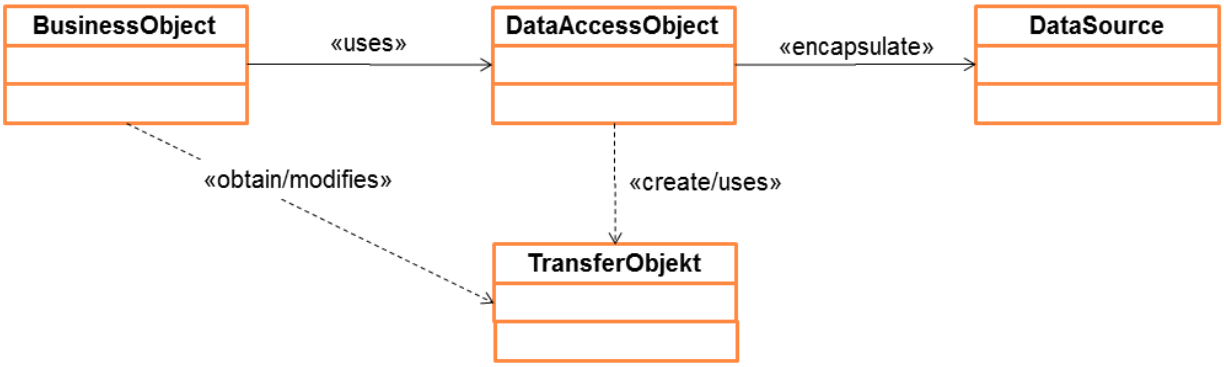
# Aufgabe 1

Gegeben ist eine kleine Anwendung zur Buchverwaltung.

```
public void actionPerformed(ActionEvent event)
{

        if(event.getActionCommand().equals("Next"))
        {
            System.out.println("Next");
            einBuch = dieBuchVerwalt.getNext();
            updateView(einBuch);
            repaint();
        }
        else if (event.getActionCommand().equals("Rewind"))
        {
            System.out.println("Rewind");
            einBuch = dieBuchVerwalt.getFirst();
            updateView(einBuch);
            repaint();
        }
        else if (event.getActionCommand().equals("Aendern"))
        {
            System.out.println("Aendern");
            Buch einBuch = SaveView();
            dieBuchVerwalt.aendern(einBuch);
            repaint();
       }

}
```



Setzen Sie diese Anwendung um in eine 3-Schichten-Architektur mit dem DAO/DTO-Architekturmuster.

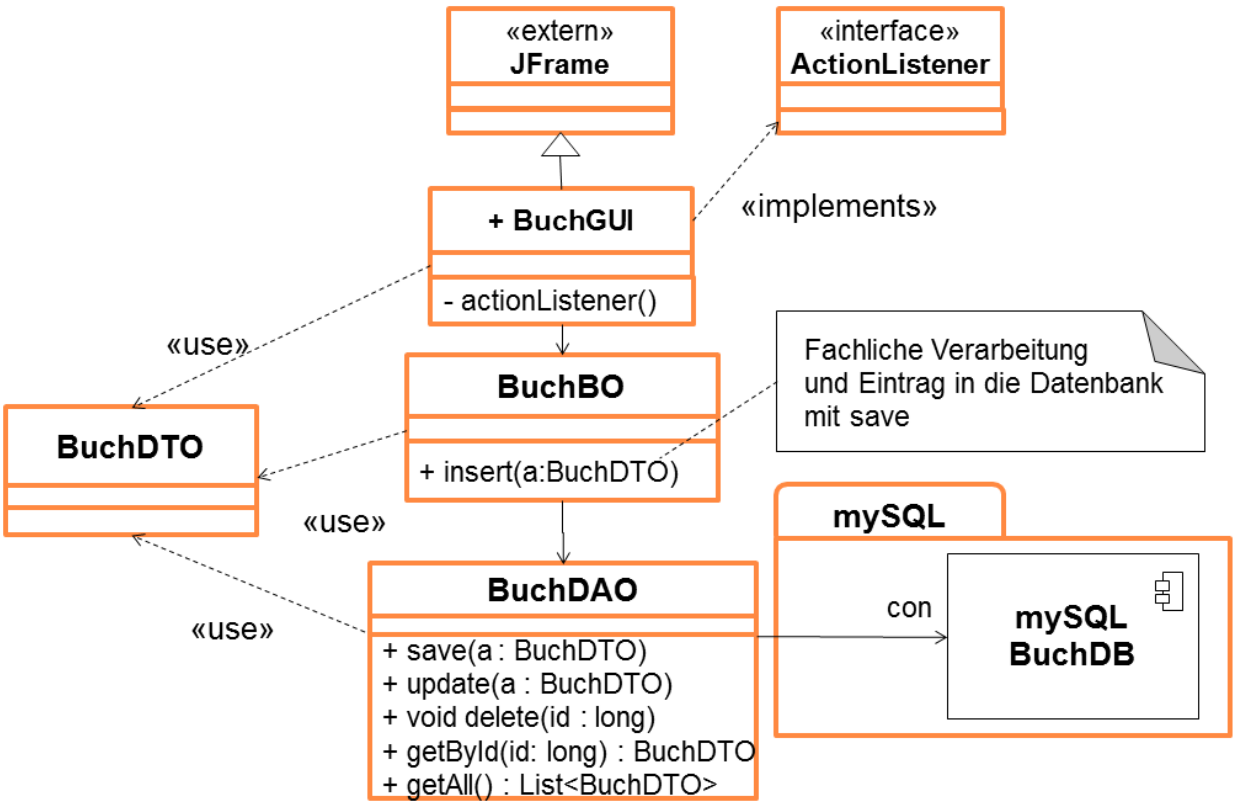



## Aufgabe 2

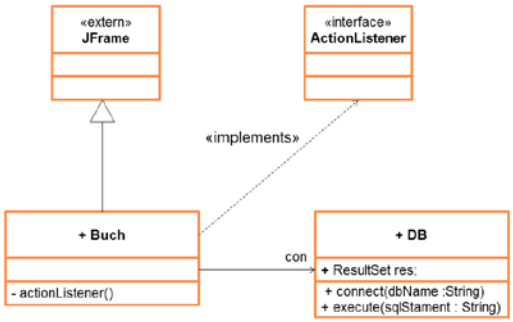
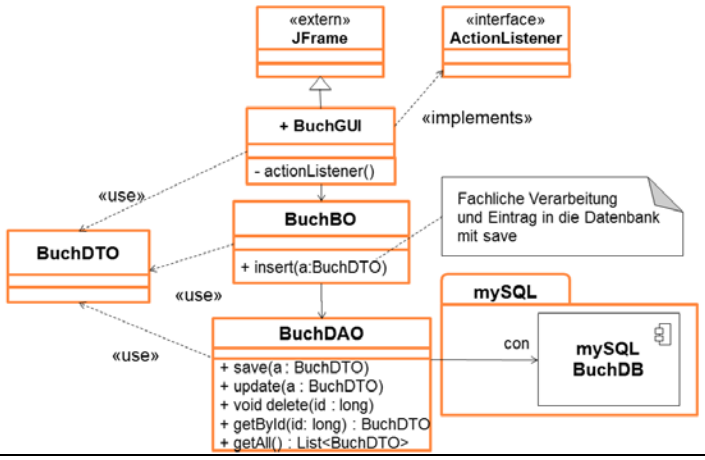
Welche Vorteile ergeben sich aus der Verwendung des DAO/DTO-Entwurfsmuster?

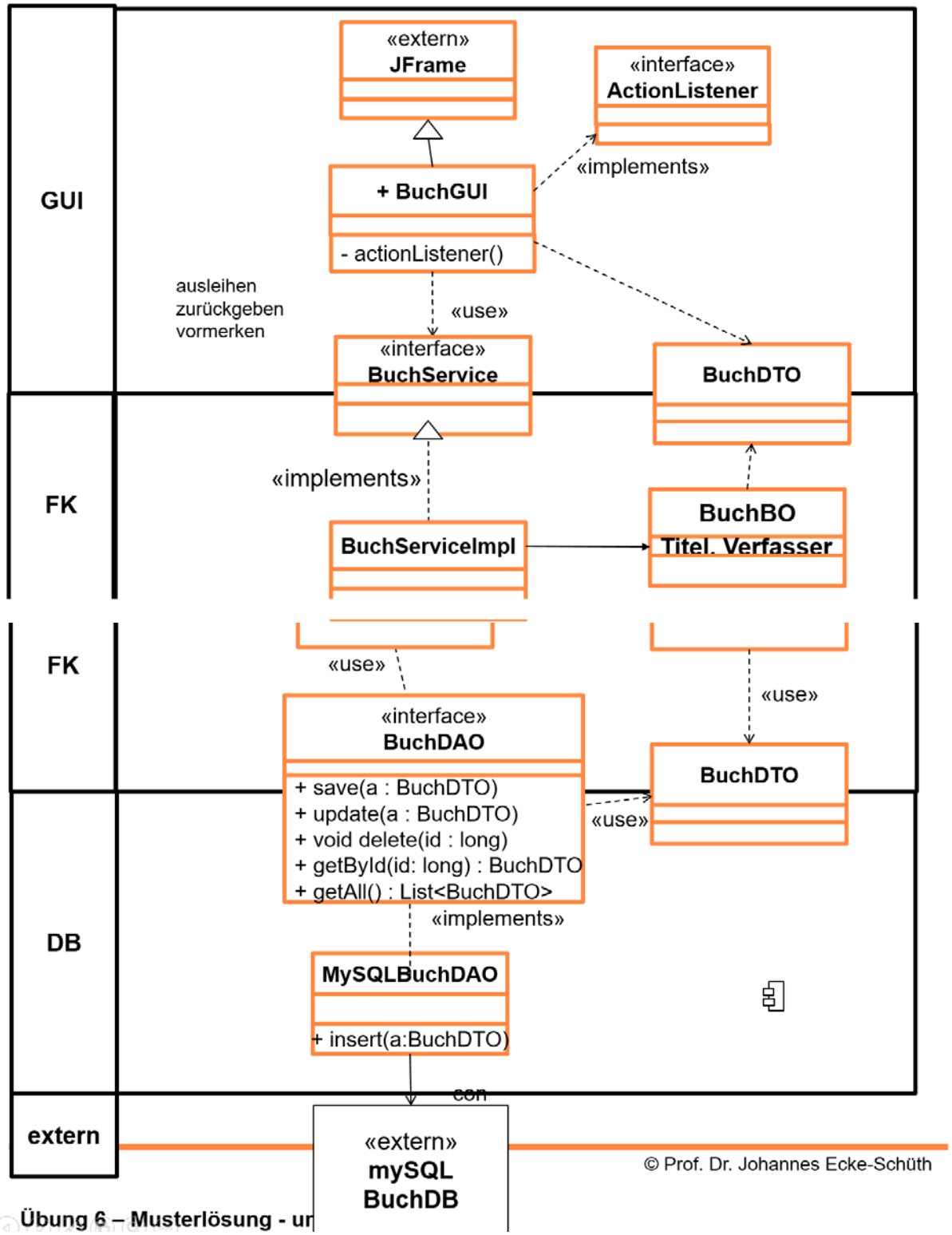
# Aufgabe 1

Die 3-Schichten-Architektur mit dem DAO/DTO-Architekturmuster für die Anwendung 'Buch' sieht wie folgt aus:



# Aufgabe 2

| Ursprüngliche Architektur | Neue Architektur |
|---|---|
|  |  |
| **Separation of Concern** | |
| Es gibt keine Trennung der Verantwortlichkeiten von GUI-, Fachkonzept- und Datenbankaspekten | Es gibt eine sauber Trennung der Verantwortlichkeiten der einzelnen Klassen:<br>• Die Klasse `BuchGUI` ist für die Präsentation verantwortlich.<br>• Die Klasse BuchBO ist für die Fachlogik verantwortlich.<br>• Die Klasse BuchDAO ist für die Realisierung der Anbindung an die Datenhaltung verantwortlich. |
| Die Klasse `Buch` enthält nicht nur die reine Fachlogik, sondern auch Datenbank-spezifische Aufrufe und GUI-spezifische Aufrufe. | Die Klasse `BuchBO` enthält keine GUI- bzw. Datenbank-spezifischen Anweisungen. |
| **Abhängigkeiten** | |
| Die Klasse `Buch` ist abhängig von der speziellen Datenbank-Implementierung und von der speziellen GUI-Implementierung. | Die Fachkonzeptklasse `Buch` ist unabhängig von der speziellen GUI-Implementierung.<br><br>Die Fachkonzeptklasse `BuchBO` ist nicht mehr direkt von der speziellen Datenbank-Implementierung abhängig, sondern nur über die 'Fassade' BuchDAO |
| **Stabilität** | |
| | Die Schnittstellen werden im Wesentlichen von den Klassen `BuchDTO` und `BuchDAO` bestimmt, die früh bestimmt und dann 'stabil' (unverändert) gehalten werden können. |




Übung 6 – Musterlösung - ur

# Aufgabe 1

Für eine studentische Hilfskraft werden folgende Daten benötigt:

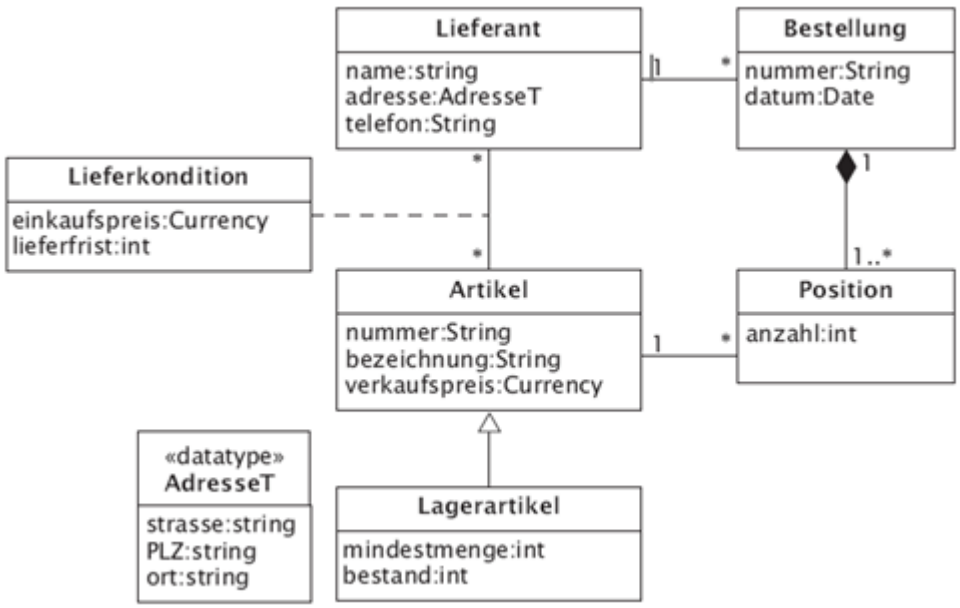
- eine 7-stellige Matrikelnummer
- der Vorname und der Nachname
- die Adresse, bestehend aus Straße, PLZ und Ort,
- der aktueller Stundenlohn, der für alle studentischen Hilfskräfte gleich ist (Klassenattribut)
- ein oder mehrere Arbeitsverträge, wobei für jeden Arbeitsvertrag Beginn, Ende und die vereinbarte Stundenzahl gespeichert wird. Die Menge der Arbeitsverträge darf nicht begrenzt werden.

Diese Informationen sollen zukünftig in Tabellen einer relationalen Datenbank abgespeichert werden.

Stellen sie dazu das physische Datenmodell in UML-Notation dar.
Verwenden Sie die SQL-Datentypen.

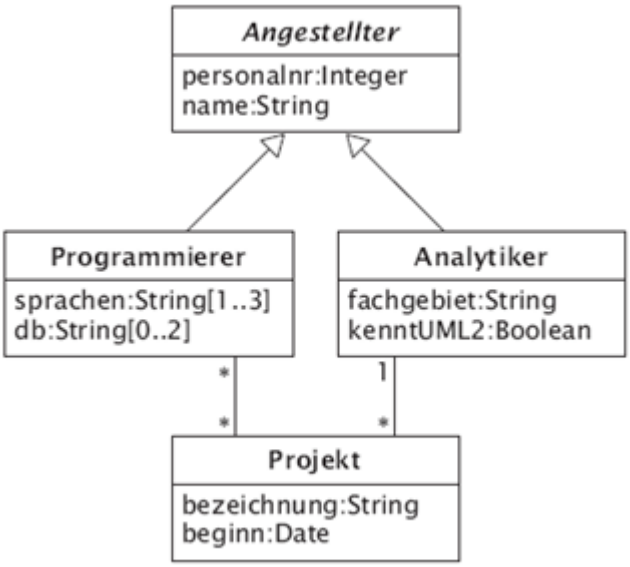
# Aufgabe 2

Bilden Sie folgendes Klassendiagramm auf Tabellen einer relationalen Datenbank ab. Stellen Sie das physische Datenmodell einschließlich der Datentypen in UML-Notation dar.



# Aufgabe 3

Bilden Sie folgendes Klassendiagramm auf Tabellen einer relationalen Datenbank ab.
Stellen Sie das physische Datenmodell einschließlich der Datentypen in UML-Notation dar. Gehen Sie davon aus, dass die verwendete Datenbank nur einfache Datentypen unterstützt.

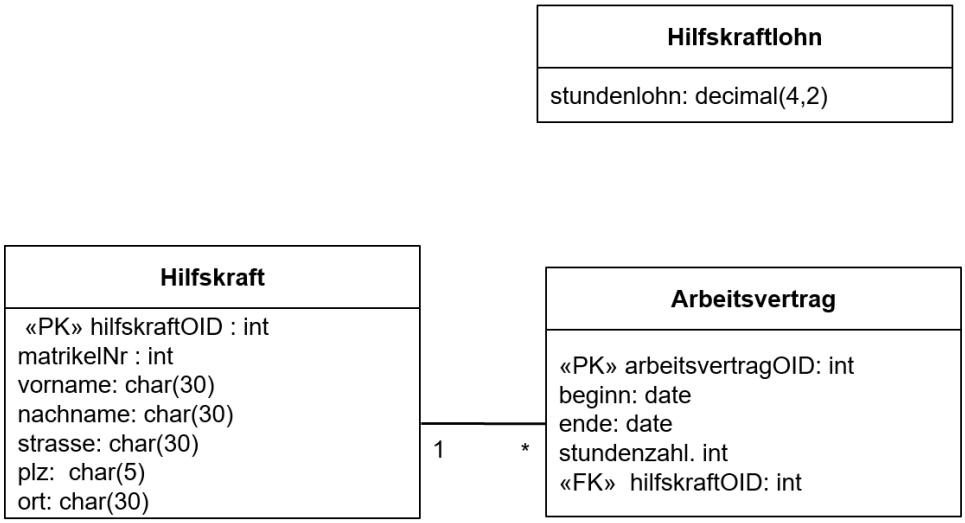



# Aufgabe 4

(a) Erläutern Sie den Unterschied zwischen dem Schlüsselattribut einer relationalen Datenbank und der Objektidentität der objektorientierten Modellierung.
(b) Erläutern Sie die Unterschiede zwischen einer Klasse und einer Tabelle.
(c) Erläutern Sie die Unterschiede zwischen der Assoziation zwischen Klassen und der Schlüssel-Fremdschlüssel-Beziehung zwischen Tabellen.
(d) Warum muss für eine Klasse Kunde, die bereits ein eindeutiges Attribut Kundennummer besitzt, bei der Abbildung auf eine Tabelle ein zusätzliches OID-Attribut eingefügt werden?

# Aufgabe 1

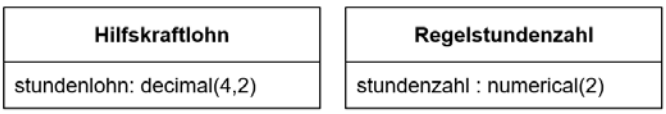
**cd** Physische Datenbank in UML-Notation



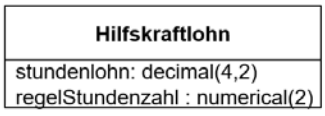

Anmerkungen

- Abbildung der Klassen 'Hilfskraft' und ‚Arbeitsvertrag‘ auf getrennte Tabellen
- Es werden die Datenbank-Datentypen verwendet.
- Der Hilfskraftlohn wird in einer eigenen Tabelle abgelegt (`static`-Attribut).
- In diesem Beispiel gab es nur ein statisches Attribut. Bei zwei und mehr Attributen hat man unterschiedliche Umsetzungsmöglichkeiten:
  Variante 1: getrennte Tabellen
  Variante 2: je Attribut eine Spalte
  Variante 3: eine Spalte für die Attribut-Bezeichnung, eine

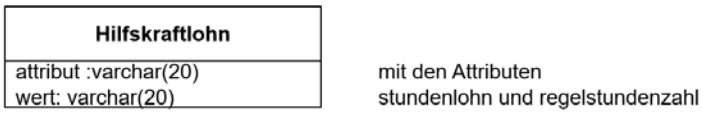
**Variante 1**




**Variante 2**



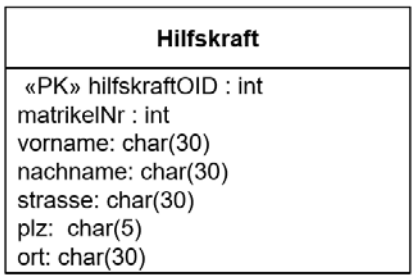

**Variante 3**



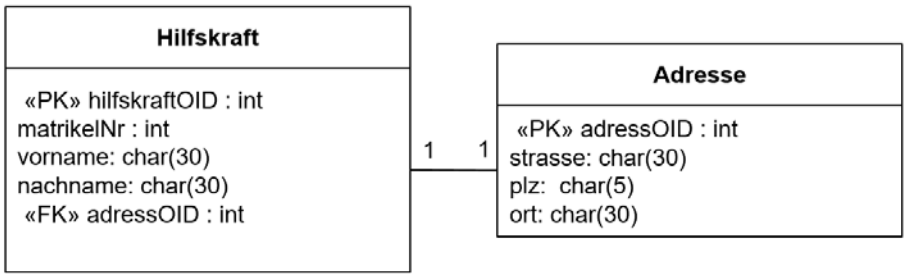

Es ist hier nicht richtig, eine „Vererbung einzubauen der Art:
Person → Student → Hilfskraft.

Bei 1:1-Beziehungen hat man die grundsätzliche Wahlfreiheit, sich für eine Tabelle oder für zwei Tabellen zu entscheiden.

**Variante 1**

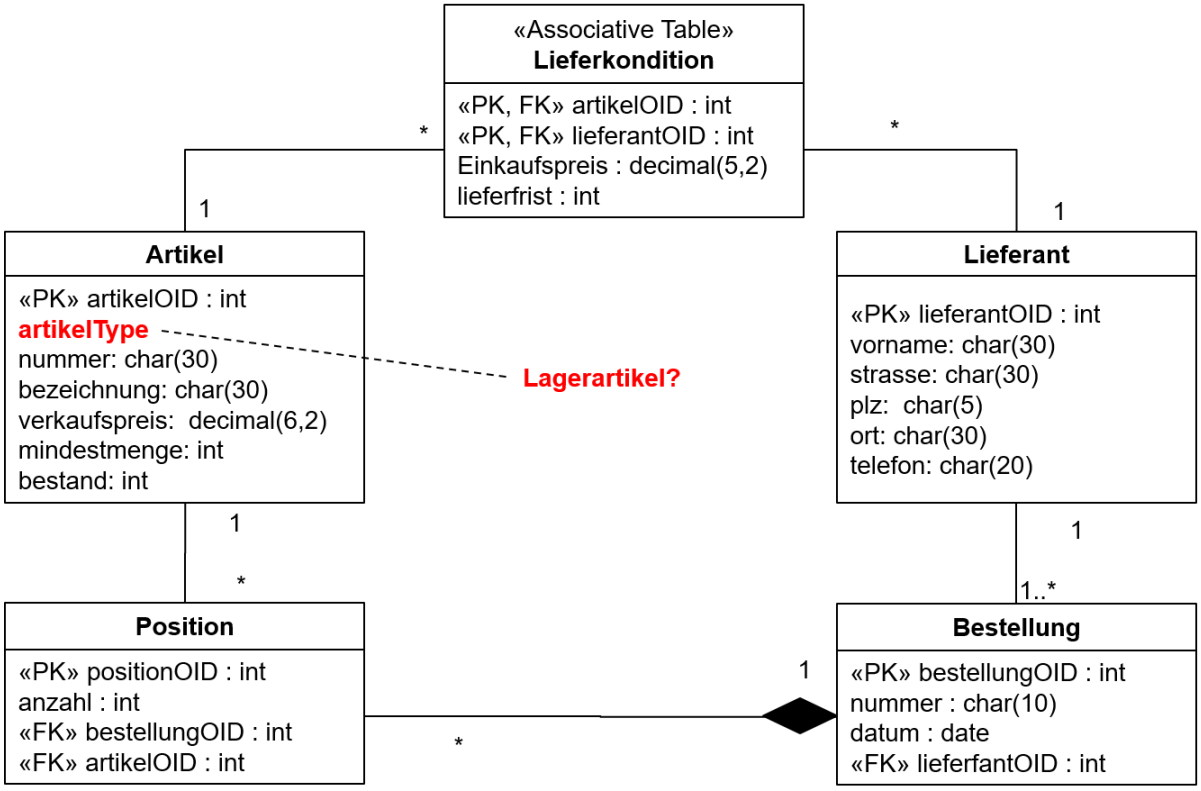



**Variante 2**



# Aufgabe 2

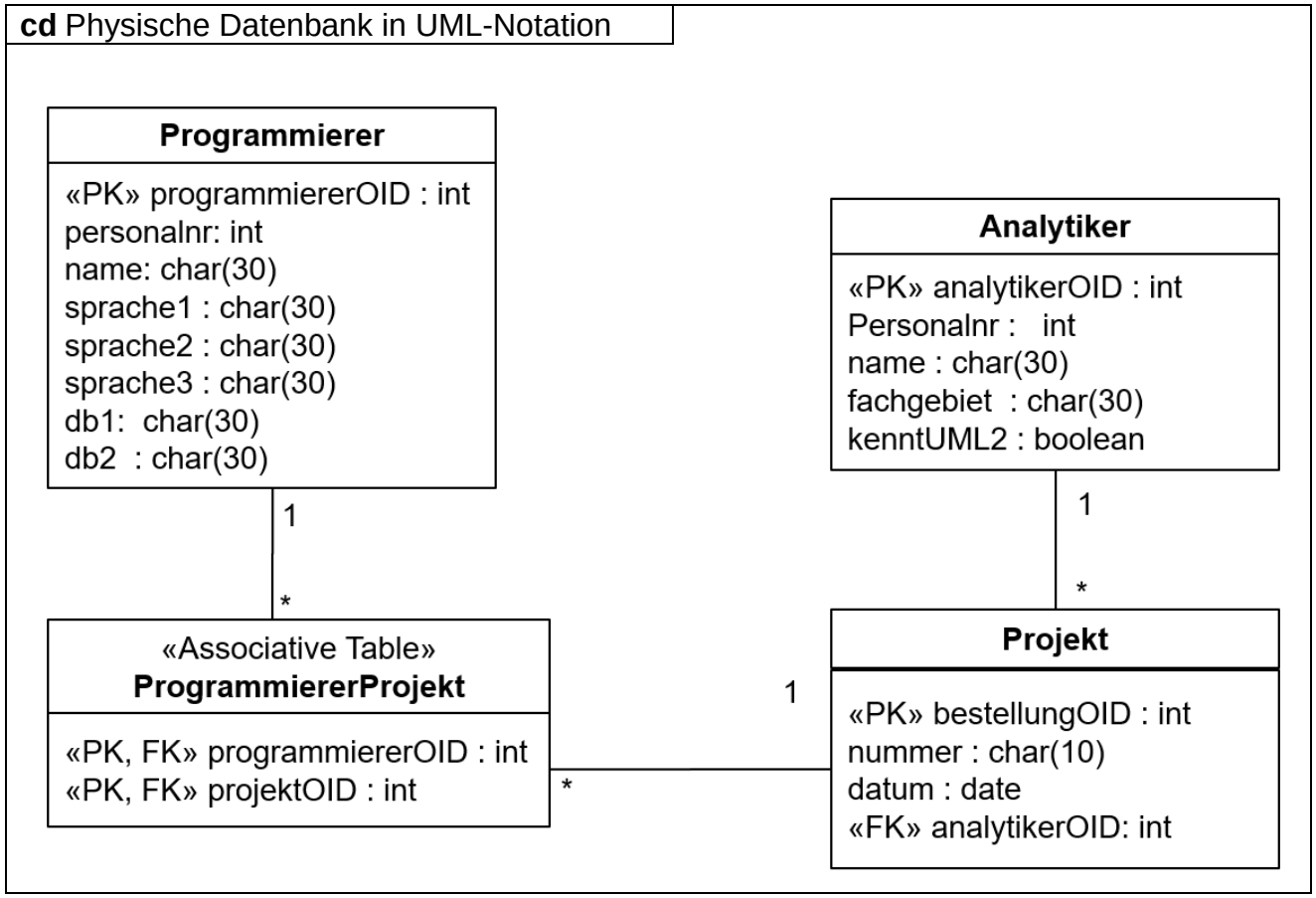
**cd** Physische Datenbank in UML-Notation




Anmerkungen:

- Die Generalisierungsstruktur wird auf eine einzige Tabelle abgebildet.
- Das Attribut artikelType gibt an, ob es sich um einen allg. Artikel, oder aber um einen Lagerartikel handelt.
- Die Adressenstruktur wird in die Tabelle `Lieferant` integriert.

# Aufgabe 3




Anmerkungen:

- Die Generalisierungsstruktur wurde hier für jede konkrete Klasse auf eine Tabelle abgebildet, weil die Klassen Programmierer und Analytiker unterschiedliche Assoziateten zum Projekt haben.
- Die Attribute `sprache` und `db` werden aufgelöst (bei max. 3 Elementen)
- Die 1:m-Assoziation zwischen `Analytiker` und `Projekt` wird durch einen Fremdschlüssel in der Tabelle `Projekt` realisiert.
- Für die n:m-Assoziation ist die Assoziationstabelle ProgrammiererProjekt notwendig.

# Aufgabe 4

(a)

**Schlüsselattribut in einer relationalen Datenbank**

Jedes Tupel in einer Tabelle muss einen expliziten Schlüssel besitzen.

- Der Schlüssel besteht aus einem oder mehreren Attributen
- Die Schlüsselattribute sind äußerlich nicht von den anderen Attributen zu unterscheiden.
- Die Schlüsselattribute identifizieren jedes Tupel einer Tabelle eindeutig.
- Es können fachliche, wie auch künstliche Schlüssel verwendet werden; fachliche Schlüssel sollten aber in der Praxis vermieden werden.
  Gründe hierfür:
  - Künstliche Schlüssel können so Domäne-unabhängig gleich behandelt werden
  - Es gibt weniger Probleme, wenn Veränderungen an den fachlichen Schlüsseln vorgenommen werden.

**Objektidentität in der objektorientierten Modellierung**

- Jedes Objekt besitzt eine implizite Objektidentität.
- Sie ist nicht nur in der Klasse, sondern systemweit eindeutig.
- Die Objektidentität besitzt keine semantische Bedeutung.

(b)

| **Klasse** | **Tabelle** |
|---|---|
| • Definition einer Kollektion von Objekten:<br>* Strukturen (Attribute)<br>* Verhalten (Methoden)<br>* Beziehungen<br>• Mechanismus zur Konstruktion von Objekten (Konstruktor)<br>• Objektidentität | • Menge von Tupeln<br>• Die Identifikation erfolgt durch einen eindeutigen Primäschlüssel<br>• Der Schlüssel kann auch aus mehreren Attributen bestehen. |

(c)

| **Assoziation zwischen Klassen** | **Schlüssel-Fremdschlüssel-Beziehung zwischen Tabellen** |
|---|---|
| • Kann in der Analyse unspezifiziert sein, im Entwurf unidirektional oder bidirektional<br>• Wird nicht durch beteiligte Attribute ausgedrückt (durch Rollen)<br>• Wer welche Attribute kennt, wird durch Assoziationen ausgedrückt. | • Ein Fremdschlüssel ist ein Referenzattribut, das dem Primärschlüssel eines Tupels einer anderen Tabelle entspricht.<br>• Die Schlüssel-Fremdschlüssel-Beziehung ist bidirektional (SQL). |

(d)

- Das OID-Attribut ist ein künstliches Attribut, um das jede Tabelle erweitert wird.
- Es erfüllt in der Datenbank alle Eigenschaften der Objektidentität.
- Das OID-Attribut ist in der Datenbank nicht von den anderen Attributen zu unterscheiden.
- Häufig werden Zahlen erzeugt – ohne semantische Bedeutung.

# **Aufgabe 1** (Implementierungssicht)

| | |
|---|---|
| Im Rahmen einer Softwareentwicklung für das Beschwerdesystem NÖRGL sind folgende Ergebnisse entstanden. | |
| (1) | Im Rahmen der Anforderungsanalyse wurden bereits zwei Anforderungsdokumente erstellt:<br>"NÖRGL - Anforderungen" und<br>"NÖRGL – Anforderungen 2.0"<br><br>Das zweite Anforderungsdokument wurde erforderlich, da in der ersten Version die Auswertemöglichkeiten nicht berücksichtigt wurden. Zudem fehlten die Anforderungen hinsichtlich einer SMS-Unterstützung und eine Unterscheidung zwischen der "Basis-Version" und der "Champion-Version", wobei in diesen Varianten zwischen Basis- Auswertemöglichkeiten und fortgeschrittenen Auswertemöglichkeiten unterschieden wird |
| (2) | Aufbauend auf den Vorgaben aus „NÖRGL- Anforderungen“ wurden die Java-Klassen NörglGUI.java (mit der main-Methode) und NörglListe.java erstellt. |
| (3) | NörglGUI.class und NörglList.class wurden<br>zusammen mit der Datei README (Version 1.0) in Nörgl1.0.jar zusammengeführt.<br><br>Diese Version ist in dieser Form an verschiedenen Kunden ausgeliefert worden. |
| (4) | Die aktuellen Arbeiten basieren auf den Anforderungen aus dem Dokument „NÖRGL – Anforderungen 2.0“.<br><br>Zu den bisherigen Quelltexten wurden weitere Quelltexte erforderlich:<br>• NörglAuswertGemeinsam.java<br>• NörglAuswertBasis.java<br>• NörglAuswertChampion.java<br><br>Zu jeder Quelltextdatei wird eine entsprechende Historien-Datei angelegt, in der vermerkt wird, welche Änderungen – bezogen auf die Vorgängerversionen eingebracht wurden. |
| (5) | Nunmehr gibt es zwei zusätzliche Versionen des Programmes in Form von Nörgl-Basis-2.0.jar und Nörgl-Champion-2.0. jar.<br><br>In beiden Archiven sind die aktualisierten Versionen der Basisklassen NörglGUI.class und NörglList.class, sowie NörglAuswertGemeinsam.class zu finden.<br><br>Zudem findet sich dort – je nach Produktvariante - NörglAuswertBasis.class |

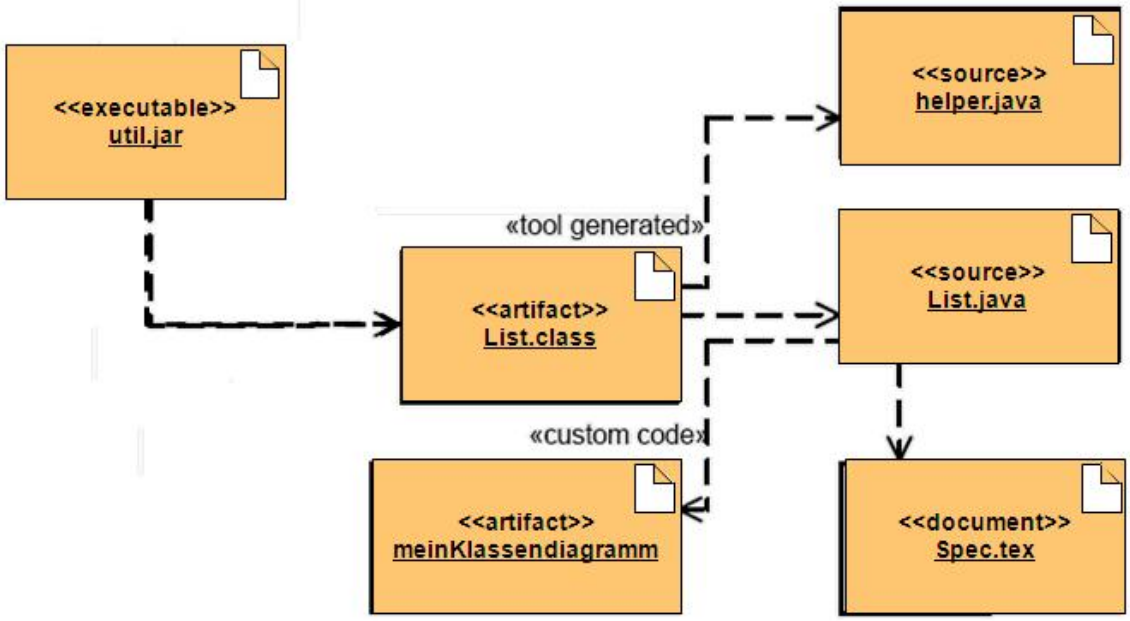
| | |
|---|---|
| | oder NörglAuswertChampion.class<br><br>Ergänzt wird das Archiv durch die Datei README.<br>In dieser Datei werden – als Release-Note - die Historie der betroffenen Dateien zusammengeführt. |

Stellen Sie zu den vorstehenden Zusammenhängen die Implementierungssicht da.

Wählen sie für die beiden Programmversionen zwei getrennte Diagramme!

Hinweise aus der Vorlesung

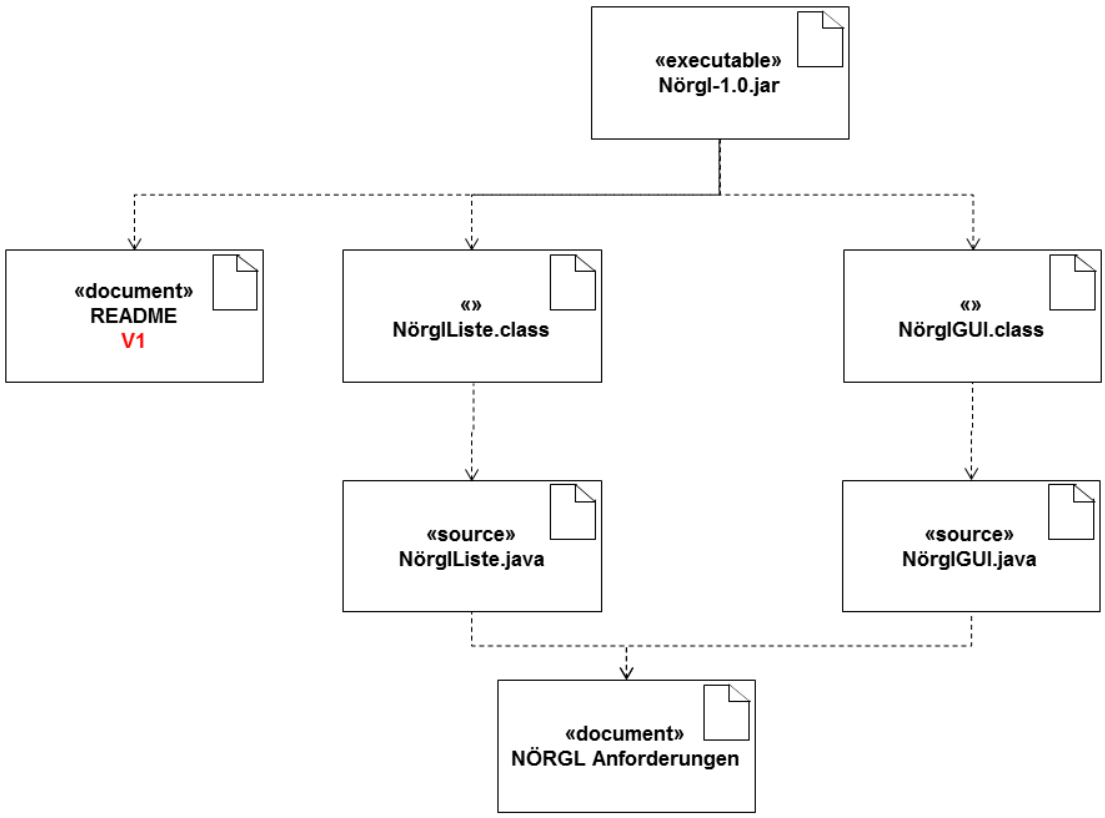

Quelle: UML-Spezifikation

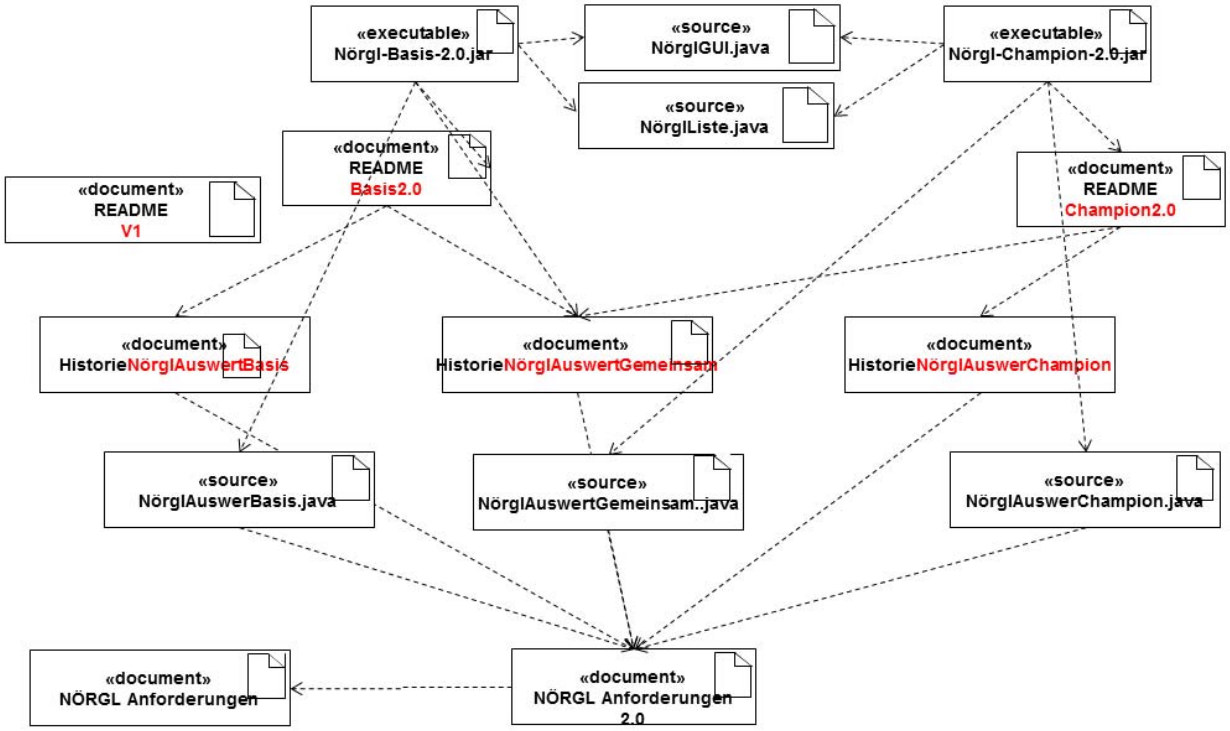



Idee: http://www.jeckle.de/vorlesung/uml/kap2-5--2-6.pdf

# **Aufgabe 1** (Implementierungssicht)

Die Fragestellung wurde hier in zwei Zeichnungen aufgeteilt, für die Version 1 und für die Version 2.






Eine Umsetzung aus der textuellen Beschreibung in ein UML-Diagramm lässt oftmals fehlende Eindeutigkeit und Inkonsistenzen erkennen.

- Baut das Anforderungsdokument "NÖRGL – Anforderungen 2.0" auf "NÖRGL - Anforderungen" auf – oder ersetzt es dieses Dokument?
- Wie sind die Historie-Dateien bezeichnet?
- Auf der Darstellung der class-Dateien wurde - aus Gründen der Übersichtlichkeit – in der zweiten Zeichnung verzichtet.
- Inwieweit fließen die Informationen aus "README V1" in "README Basis 2.0" und "README Campion 2.0"

## Zwecke von Dokumentation

- ➔ Für sich (... zu Erinnerungszwecken - nach langer Zeit)
- ➔ Für Andere
    - o andere Teammitglieder
    - o Betreiber sind im Allg. nicht die Entwickler
    - o Nachfolger in der Entwicklung
    - o Wartungsteammitglieder

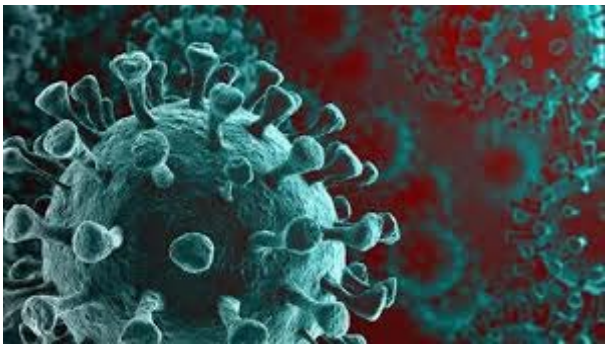
## Informationsgehalt der Implementierungssicht

Was erkennt man in der Implementierungssicht

- ➔ Strukturierung in Projekte (Prozesse wie z.B. Client, Server, Service) und Pakete
- ➔ Auslieferungseinheiten zur Softwareübergabe (z.B. Executables, Libraries und Konfigurationsdateien)
- ➔ Technologische Abhängigkeiten
- ➔ Einsatz von Fremdprodukten (Technologie-Stack)
    - o Bibliotheken und/oder Frameworks
    - o bestenfalls Dokumentation des verwendeten Subsets!
- ➔ Reichweite von Fremdpaketen
- ➔ GUI (z.B. Java-FX-Abhängigkeiten )
- ➔ Abhängigkeiten zu Plattform, Netzwerk und Peripherie
- ➔ DBMS (insbesondere die Fehlermeldungen)
  Technische Exception sind zu loggen und auf eigene Exceptions umzubiegen
- ➔ Versionen, Verträglichkeit bzw. Unverträglichkeiten von Versionen
- ➔ Zwang, Updates zu machen vs. Zwang Updates nicht machen zu dürfen (Constraints)

## Framework-Abhängigkeiten am Beispiel von Spring/Spring Boot

Frameworks bieten unzählige Schnittstellen an (*provided interface*)



Allerdings werden in einer Anwendung oftmals nur wenige Schnittstellen genutzt (*used interface*)

## Differnzierung zwischen Artifakten

- ➔ Direkt Editierbares (.java, .sh, .xml, .sq, build.xml)
  ... zur Kunden-spezifischen Konfiguration
  ... oder zur Programmierung (Anpassung)
- ➔ Indirekte Manipulation ( Wizard)
- ➔ Generierte Zwischenergebnisse
- ➔ Auslieferungseinheiten
  - Executables, Konfigurationsdateien, Readme-Dateien

## Rollen-/ Tätigkeitsbezogene Betrachtung

**Wofür benötigt man die Implementierungssicht?**

How-To (Wie werden die Aufgaben erledigt, wie kann man unterstützen?)

- ➔ Build (woraus, in welcher Reihenfolge)
- ➔ Installation, Deployment
- ➔ Traceability
  Change-Request → Architekturdokumente →Quellcode → Auslieferungseinheit → Release-Note
- ➔ Betrieb:
  - Verteilung
  - Gestaltung Projekt-übergreifender Systemlandschaften
  - Unterschiedliche Verantwortlichkeiten
  - Sicherstellung der Koexistenz
  - Monitoring
  - Laufzeitsysteme (Koexistenz)
  - Datensicherung, Backup- und Recovery
- ➔ How-To's für die „Was-wäre-wenn-Szenarien" in derWartung
  - wie wird eine andere Datenbank angebunden?
  - Wie werden neue Peripheriegeräte eingebunden?
  - PlugIn-Fähigkeiten

## Dokumentationsmittel—

**UML-Diagramm mit Artefakten und deren Abhängigkeiten**

Zusätzlich, alternativ oder ergänzend:

- Matrixdarstellung
  Artefakte (Spalten) x Artefakte (Zeilen)
  x für Abhängigkeiten
- Repository (z.B. in eclipse)
- Fließtext (als Ergänzung)
- Build-Datei (nicht unkommentiert)

**Zeichnet man Alles in einem Diagramm?**

- Nein, das wäre zu unübersichtlich.
- **Z**eichnungen, die das Prinzip erkennen lassen ergänzt
  z.B. .java → .class. → .jar als Zeichnung

# Aufgabe 1

| (a) | Stellen Sie das nachstehene Szenario in einem Verteilungsdiagramm da. |
|---|---|
| | Hinweis: In einem Verteilungsdiagramm werden sämtliche Elemente der Systemsicht beschreiben.<br>Zusätzlich werden die Artifakte ergänzt, die auf den Knoten abzulegen sind (auszuliefern, zu installieren). |

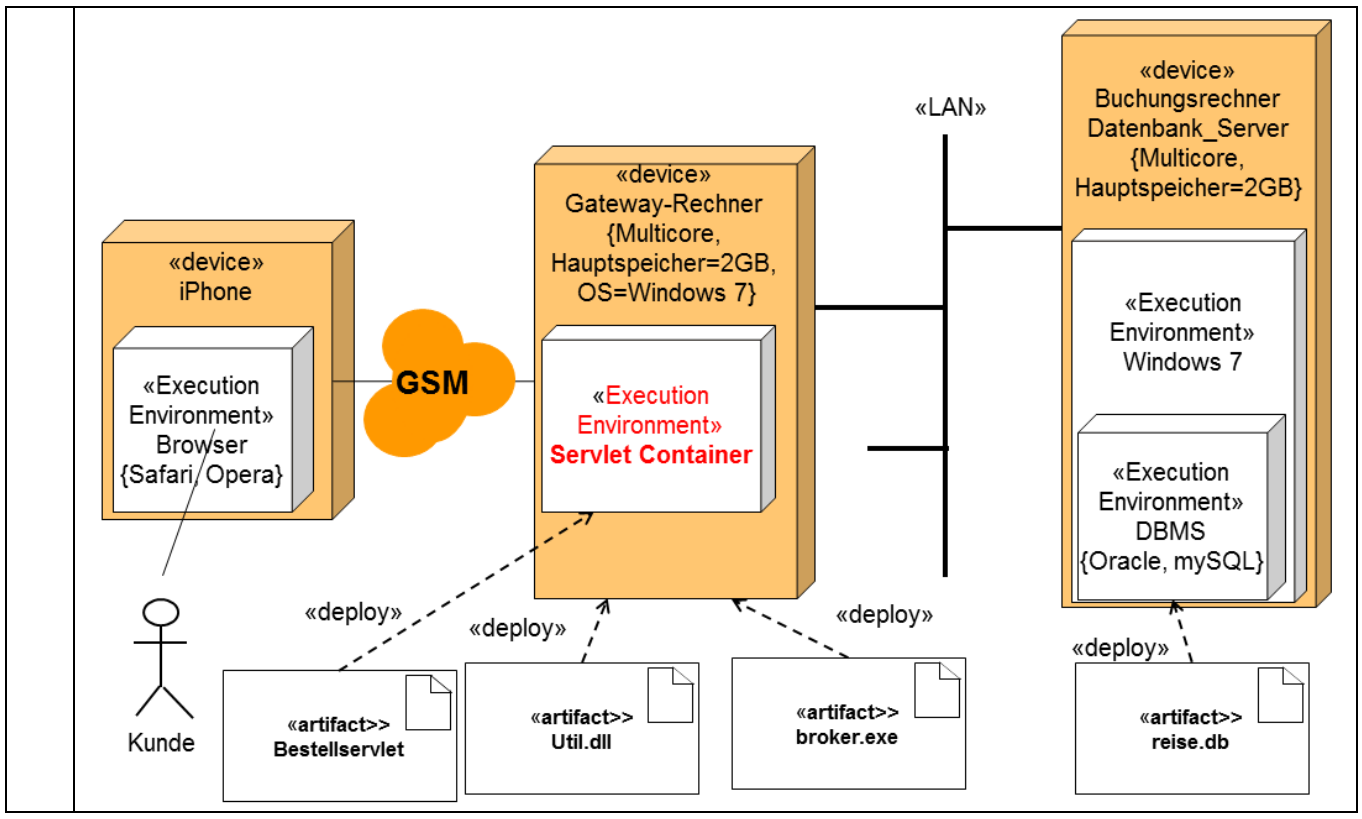
- Ein Kunde kann mit einem Mobiltelefon verschiedenartige Reisen reservieren.
- Der Kunde benötigt ein iPhone, mit dem er über das GSM-Netz auf entfernte Rechner zugreifen kann.
- Als Browser werden 'Opera Mini' und 'Safari' unterstützt.
- Auf dem iPhone werden im Browser die Reservierungswünsche erfasst.
- Mit einem Touch auf den 'Senden-Button' wird die Bestellung ausgelöst: Es wird – über das GSM-Netz - ein Bestell-Servlet auf einem Gateway-Rechner der Firma ausgeführt.
- Bei dem Gateway-Rechner handelt es sich um einen Broker, der je nach Reservierungsanfrage (Kreuzfahrt, Kulturreise oder Cluburlaub) die Anfrage an den jeweiligen Datenbank-Server weiterleitet.
- Hierzu sind auf dem Gateway-Rechner die Programme broker.exe und util.dll zu installieren.
- Die Buchungsrechner sind am Standort Buxtehude über ein LAN sowohl untereinander, als auch mit dem Gateway-Rechner verbunden.
- Sowohl die drei Buchungsrechner, als und der Gateway-Rechner sind baugleich. Sie haben einen Multi-Core-Prozessor und 2GB Hauptspeicher.
- Auf sämtlichen Rechnern ist zudem das Betriebssystem Windows 7 installiert.
- Auf den Buchungsservern ist zusätzlich ein 'Datenbank-Managements-System‘ wahlweise ORACLE oder mySQL installiert.
- In diesem DBMS ist jeweils die Datenbank reise.db aufgespielt.

| b) | Nicht alle Aspekte aus der vorliegenden Beschreibung lassen sich im UML-Verteilungsdiagramm darstellen.<br>Welche der nicht dargestellten Begriffe gehören zu welcher Architektur-Sicht? |
|---|---|
| | |

# Aufgabe 1

(a) Stellen Sie das nachstehende Szenario in einem Verteilungsdiagramm da.
Hinweis: In einem Verteilungsdiagramm werden sämtliche Elemente der Systemsicht beschreiben.
Zusätzlich werden die Artifakte ergänzt, die auf den Knoten abzulegen sind (auszuliefern, zu installieren).

| | |
|---|---|
| | • Ein Kunde kann mit einem Mobiltelefon verschiedenartige Reisen reservieren.<br>• Der Kunde benötigt ein iPhone, mit dem er über das GSM-Netz auf entfernte Rechner zugreifen kann.<br>• Als Browser werden 'Opera Mini' und 'Safari' unterstützt.<br>• Auf dem iPhone werden im Browser die Reservierungswünsche erfasst.<br>• Mit einem Touch auf den 'Senden-Button' wird die Bestellung ausgelöst: Es wird – über das GSM-Netz - ein Bestell-Servlet auf einem Gateway-Rechner der Firma ausgeführt.<br>• Bei dem Gateway-Rechner handelt es sich um einen Broker, der je nach Reservierungsanfrage (Kreuzfahrt, Kulturreise oder Cluburlaub) die Anfrage an den jeweiligen Datenbank-Server weiterleitet.<br>• Hierzu sind auf dem Gateway-Rechner die Programme broker.exe und util.dll zu installieren.<br>• Die Buchungsrechner sind am Standort Buxtehude über ein LAN sowohl untereinander, als auch mit dem Gateway-Rechner verbunden.<br>• Sowohl die drei Buchungsrechner, als und der Gateway-Rechner sind baugleich. Sie haben einen Multi-Core-Prozessor und 2GB Hauptspeicher.<br>• Auf sämtlichen Rechnern ist zudem das Betriebssystem Windows 7 installiert.<br>• Auf den Buchungsservern ist zusätzlich ein 'Datenbank-Managements-System' wahlweise ORACLE oder mySQL installiert.<br>In diesem DBMS ist jeweils die Datenbank reise.db aufgespielt. |
| | |




Anmerkung: Die Angabe eines Servlet-Containers fehlte in der Aufgabe!

b) Nicht alle Aspekte aus der vorliegenden Beschreibung lassen sich im UML-Verteilungsdiagramm darstellen.
Welche der nicht dargestellten Begriffe gehören zu welcher Architektur-Sicht?

| **Begriff** | **Architektur-Sicht** |
|---|---|
| Reisen reservieren | Use-Case / logische Sicht |
| Reservierungswünsche erfasst | |
| Mit einem Touch auf den 'Senden-Button' wird die Bestellung ausgelöst | Prozess-Sicht |
| Bestell-Dienst | Prozess-Sicht / Komponente |
| Broker | Prozess-Sicht |
| Reservierungsanfrage (Kreuzfahrt, Kulturreise oder Cluburlaub) | Logische Sicht |
| Standort | |
| | |
| iPhone, Windows 7, mySQL, ORACLE,… | auch Produktbezeichnunen |
| | |
| Util.dll, reisedb | auch Implementierungssicht (Artefakte) |
| Weiterleiten der Nachricht | |
| Servlet | Implementierungsicht, Prozessicht |