Inhaltsverzeichnis

1. Assoziationen

Klassen existieren fast nie unabhängig, denn Software besteht immer auch vielen Klassen und weiteren Datenstrukturen, die nur **gemeinsam** funktionieren und somit die ganze Software bilden.

1.1. Beziehungsarten

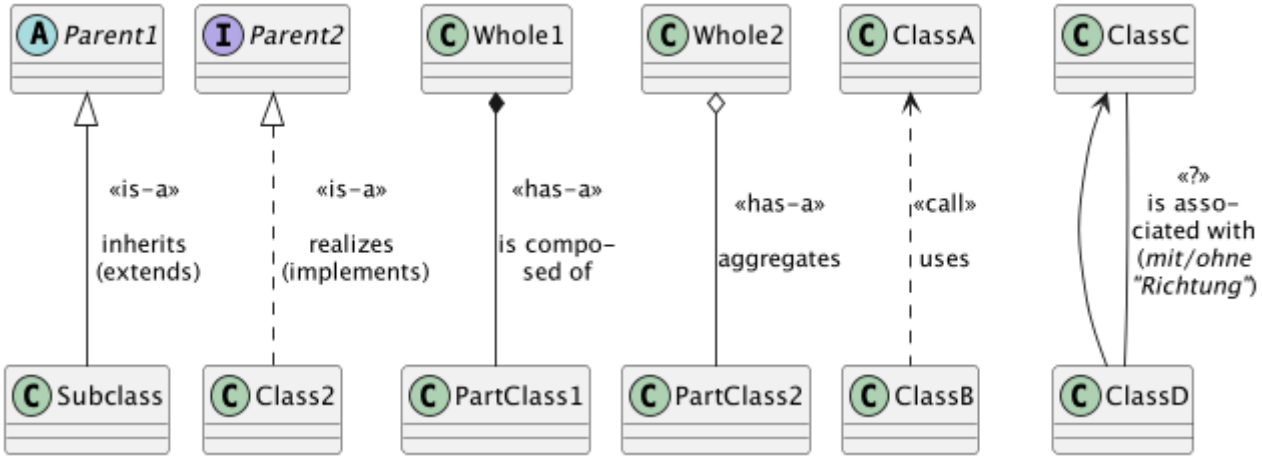
Klassen gehen also mit anderen Klassen viele sogenannte **"Beziehungen"** ein. Dabei gibt es verschiedene Arten von Beziehungen. Die wichtigsten sind:

- **Aufruf** (*dynamische Beziehung zur Laufzeit*), kann innerhalb einer Software sein oder zwischen ganzen Systemen!
- **Vererbung**, **Realisierung** (*statische Beziehung zur Compilezeit*)
- **Besitz** (*statische Beziehung zur Compilezeit*) → wird (neben weiteren Varianten) oft auch als **Assoziation** bezeichnet.

Vielfach wird bzgl. der Beziehungen zwischen Klassen zwischen folgenden zwei grundsätzlichen Varianten unterschieden:

1. `is-a` - Klasse A *ist eine* Klasse vom Typ B ⇒ "Vererbung"
2. `has-a` - Klasse A *hat eine* Klasse B ⇒ "Zusammensetzung"

Zusammengefasst als Übersicht:



1.1.1. Kopplung

Art & Grad der Abhängigkeit **zwischen** (einzelnen) Softwaremodulen, insb. Klassen.

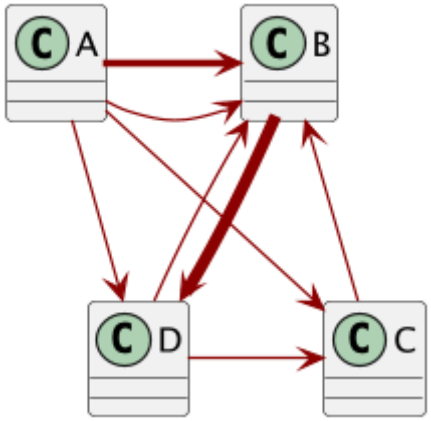


Figure 1. Kopplung

Die **Kopplung** im obigen Diagramm ist die (Art und) **Anzahl der Pfeile** zwischen den Klassen. Sie gibt an, wie stark die einzelnen Programmmodule (hier Klassen) von den anderen abhängig sind. Wechselwirkungen zwischen zwei Klassen/Objekten treten auf, weil es eine Kopplung gibt. Lose gekoppelte Programme sind flexibel und erweiterbar, je stärker die Kopplung, desto schwieriger wird die Änderbarkeit der Klassen.

1.1.2. Kohäsion

Grad des Zusammenhangs (z.B.) von Klassen **innerhalb** eines Moduls/Pakets.

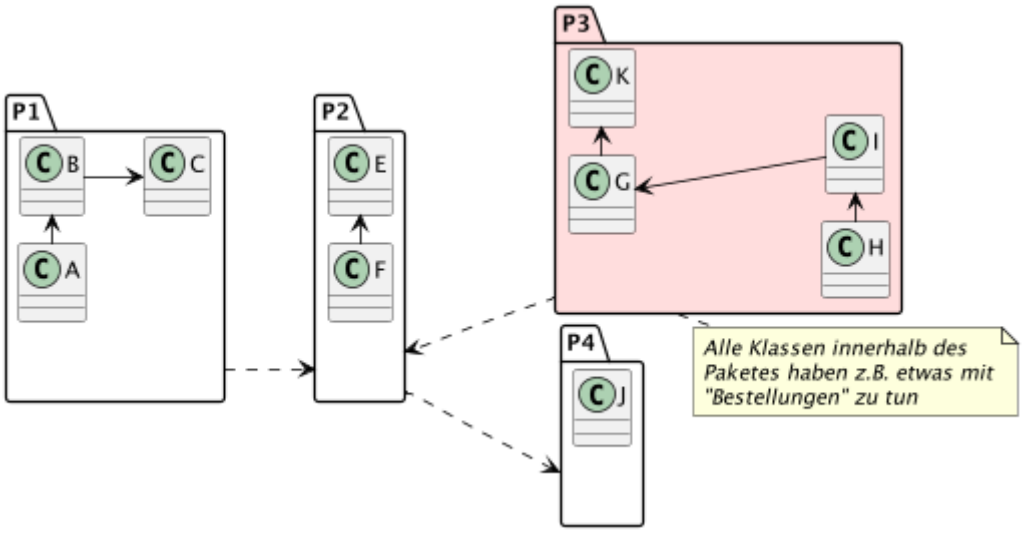


Figure 2. Kohäsion (hier z.B. im rot hinterlegten Paket)

Kohäsion misst, wie stark die einzelnen Klassen innerhalb eines Programmmoduls - z.B. eines Paketes - zusammenhängen, vor allem wie stark sie zu *einem/wenigen* Aspekt oder auch Zusammenhang gehören. Gut **strukturierte** Software führen oft zu stark zusammenhängenden Anwendungen.

*Der Grad - also die Anzahl und Qualität - der Summe aller dieser Beziehungen zwischen den Klassen einer Software wird unter dem Begriffspaar **Kopplung & Kohäsion** zusammengefasst (**"loosely coupled along with high cohesion"**). Ein allgemeines Qualitätsziel von Software ist bspw., einen niedrigen Grad von Kopplung der Komponenten zu erreichen. Je geringer die Kopplung ist, desto besser ist die Software änderbar. Gleichzeitig sollen die Klassen innerhalb eines Paketes/Moduls zur gleichen Fachlichkeit gehören!*

Beispiel:

Die **Organisation dieser LV** ist so aufgebaut. Es gibt viele einzelne Module, in denen Klassen enthalten sind, die eben zu diesem Modul bzw. zu dem jeweiligen Thema gehören (*"hoch kohäsiv"*).

Gleichzeitig haben die Module untereinander keine bzw. sehr wenige Abhängigkeiten (*"loose gekoppelt"*)

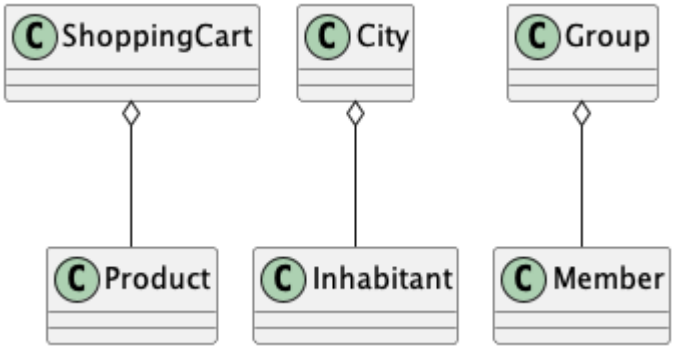
1.2. Aggregation & Komposition

Assoziationen können auch **qualitativ und semantisch** recht unterschiedlich sein. In diesem Zusammenhang sind **Aggregation** und **Komposition** von Bedeutung.

1.2.1. Aggregation

Die Aggregation, sowie auch die Komposition, ist eine Assoziation von **Teilen zu einem Ganzen**. Jedes Teil ist zu dem Ganzen mit einer Assoziation "ist-teil-von" (part-of) verbunden.

Ein paar Beispiele dazu:



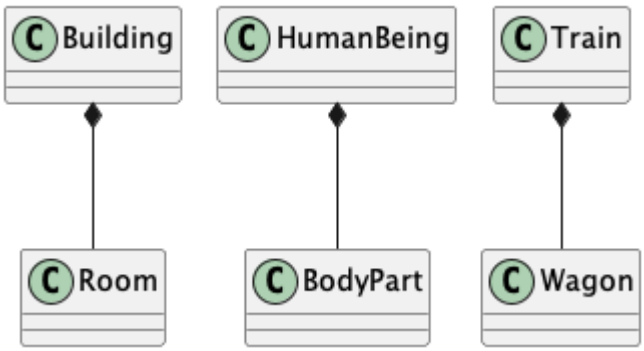
Demo:

```
src/test/java/de/dhbw/demo/AssociationsDemoTest.demo1()
```

1.2.2. Komposition

Der Unterschied einer Komposition zur Aggregation ist, dass die Teile, die ein Objekt enthält, von dem Ganzen **existentiell abhängig** sind.

Ein paar Beispiele dazu:



Die Unterschiede sind also fein, aber wichtig.

> *Es stellt sich außerdem die Frage, wie man so eine **"existentielle Abhängigkeit"** im Code ausdrücken könnte?*

Demo:

```
src/test/java/de/dhbw/demo/AssociationsDemoTest.demo2()
```

1.3. Navigierbarkeit

Sobald Klassen durch Assoziationen verbunden werden, kommt der Aspekt der **Navigierbarkeit** (von Instanz zu Instanz) hinzu, d.h. man überlegt, welche Klasse welche als Attribut enthält und ob auch die assoziierte Klasse etwas über diese Beziehung weiß.

Letztlich drückt sich die Navigierbarkeit darin aus, ob assoziierte Klassen

- als Attribut mit `Typ` → (2) oder
- nur als `Id` in Form eines primitiven Datentyps → (3)

vorhanden sind. Auch die "besitzende" Klasse hat in aller Regel selbst eine ID → (1).

Am Beispiel:

```
public class Train {
    public long id; ①
    public Locomotive locomotive; ②
    public long locomotiveId; ③
}                           |
                            |
public class Locomotive {   |
    public long id; <-------+
    public Train train; // ist das hier sinnvoll?
}
```

1.4. One-to-One-Assoziation

Siehe Unit-Test:

```
src/test/java/de/dhbw/demo/AssociationsDemoTest.demo3()
```

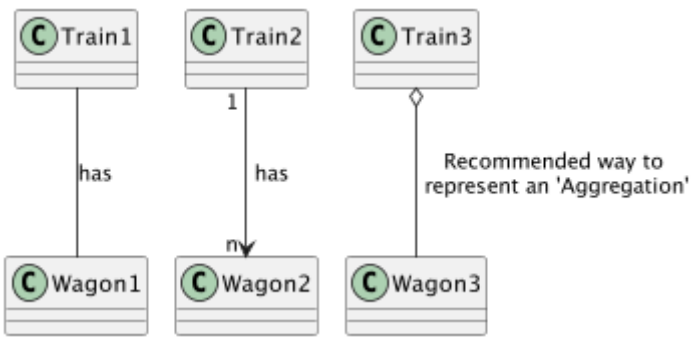
Ein Beispiel

> *Jede natürliche Person besitzt **genau** einen Personalausweis, der Ausweis kann verloren gehen, die Person kann sterben, dann muss aber auch der Personalausweis vernichtet werden, das wurde oben bereits demonstriert.*

Es geht hier also oft um die Überlegung zur **Semantik bzw. Qualität** einer Beziehung zwischen zwei abhängigen Objekten!

1.5. One-to-Many-Assoziation

Ein Beispiel für eine Eins-zu-Viele Beziehung:



*Man beachte immer auch die Richtung der Pfeile, die in aller Regel etwas über die **Lesart** und **Navigierbarkeit** aussagt!*

Zugehörige Unit-Tests zur **Demonstration**:

```
src/test/java/de/dhbw/demo/AssociationsDemoTest.demo4a()
src/test/java/de/dhbw/demo/AssociationsDemoTest.demo4b()
```

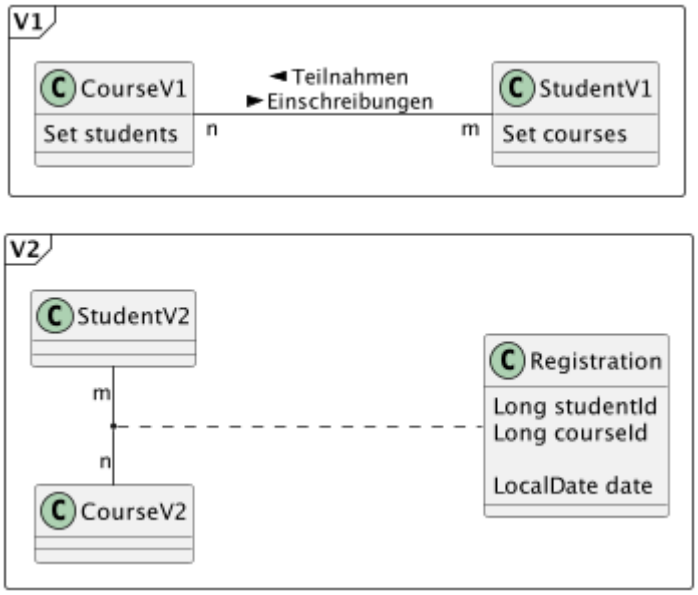
sowie Implementierungen für V1 und V2:

```
src/main/java/de/dhbw/demo/associations.n.*.java
```

→ Übung 1

1.6. Many-to-Many-Assoziation

Die folgenden zwei Varianten kommen am häufigsten vor:



Assoziationen zwischen Java Klassen werden manchmal auch als **Relationen** bezeichnet. Dieser Begriff stammt aus dem Umfeld der **Datenbanken**, meint aber das Gleiche. Dies ist ein optionales Thema und kann bei Gelegenheit eingeschoben werden.

Übungen:

Die **Übungen** sollen in Form von **Unit-Tests** in folgendem *Package* implementiert werden:

```
src/test/java/de/dhbw/exercises/AssociationsExerciseTest.java
```

Die **Testobjekte**, also die *echten* Klassen, Interfaces oder anderer Sourcecode wie zuvor auch in:

```
src/main/java/de/dhbw/exercises
```

Übung 1

Erstelle zwei Klassen

- `Course` und
- `Student`

Der Kurs kann von **mehreren** Student:innen besucht werden. Setze diese Beziehung zwischen den beiden Klassen in der Klasse `Course` um.

Überlege hier auch, welcher **Listentyp** sich dafür am besten eignet.

Übung 2 (optional)

Umsetzung des Prinzips **Information Hiding**

In dem folgenden Paket finden sich die Klassen aus der Demonstration:

```
src/test/java/de/dhbw/demo/associations.nm.v2
```

Diese Klassen enthalten aber noch keine Methoden!

Füge die zu den Attributen gehörenden `getter` und `setter` Methoden hinzu und passe auch den zugehörigen Unit-Test entsprechend an. Der findet sich hier:

```
src/test/java/de/dhbw/demo/AssociationsExerciseTest.exercise2()
```