

Assoziationen

Einleitung:

Objekte leben nicht in Isolation, sondern in Beziehungen zu anderen Objekten.

- Beziehung über eine Referenz
- Beziehung über Vererbung

Über diese Beziehungen interagieren die Objekte miteinander.

Sie rufen über die Referenzen auf andere Objekte deren Methoden auf, nutzen der Daten, übergeben Daten und reagieren auf die Antworten (Return-Werte).

In diesem Dokument geht es um

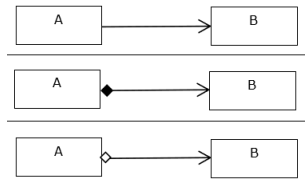
Assoziationen → Beziehung über eine Referenz

Ziel

- ⇒ Ich kann Zusammenhänge zwischen Klassen erkennen und als Assoziationen abbilden
- ⇒ Ich kann die unterschiedlichen Eigenschaften von Assoziationen erklären.
- ⇒ Ich kann die Kardinalitäten/Multiplizitäten einer Assoziation erklären, interpretieren und festlegen.
- ⇒ Ich kann Aggregation, Komposition, Verwendungsbeziehung erklären, interpretieren und festlegen.
- ⇒ Ich kann Assoziationen in einem Diagramm zeichnen und interpretieren.
- ⇒ Ich kann Assoziationen implementieren und anwenden.
(Name, Navigation, Aggregation, Komposition, Verwendungsbeziehung, Multiplizität, Kardinalität usw.)

Inhalte

Einleitung:	1
Ziel	1
Inhalte	1
Die Assoziation und ihre Eigenschaften	2
Die Abhängigkeitsbeziehung	4
Implementierung von Assoziation	5
Implementierung von Verwendungsbeziehung	9



Die Assoziation und ihre Eigenschaften

Quellen: 01.11.2022 https://openbook.rheinwerk-verlag.de/javainsel/07_001.html#u7

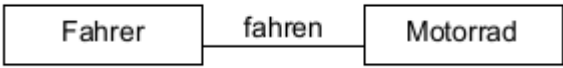
2.11.2022 https://openbook.rheinwerk-verlag.de/oop/oop_kapitel_04_003.htm

Eine wichtige Eigenschaft objektorientierter Systeme ist das Zusammenspiel von Objekten miteinander. Ein Objekt *kennt* andere Objekte und kann so über die Referenzen auf die Objekte Methoden anderer Objekte aufrufen, dabei Parameter übergeben und erhält via Return-Wert Antworten.

Es gibt Quellen, die sprechen vom "Senden von Nachrichten".

Dies muss im Sinne von Aufruf von Methoden verstanden werden.

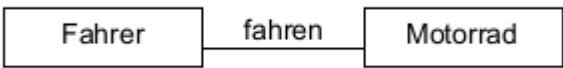
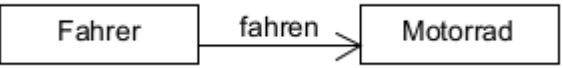
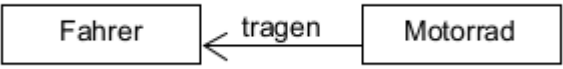
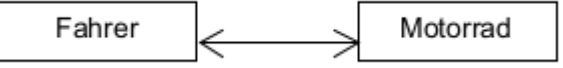
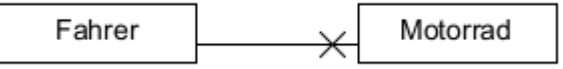
Assoziation:

	<p>Einfache Assoziation zwischen Fahrer und Motorrad.</p> <p>Hier gezeigt ist die Assoziation mit dem Namen <i>fahren</i>.</p>
---	---

Assoziation haben unterschiedliche Eigenschaften, die die Assoziation weiter definieren.

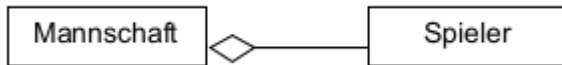
Navigation:

- Richtung der Interaktion.
- Z.B. vom Fahrer zum Motorrad oder vom Motorrad zum Fahrer.

	<p>Einfache Assoziation zwischen Fahrer und Motorrad.</p> <p>Keine Aussage über Navigation</p>
 	<p>Unidirektional gerichtete Assoziation Der Fahrer fährt mit dem Motorrad</p> <p>Das Motorrad trägt den Fahrer</p>
	<p>Bidirektionale gerichtete Assoziation. Der Fahrer interagiert mit dem Motorrad Das Motorrad interagiert mit dem Fahrer</p>
	<p>Navigation von Fahrer zu Motorrad ist nicht erlaubt.</p>

Aggregation:

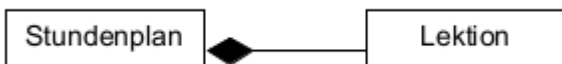
Beispiel für eine Aggregation:



- Eine Mannschaft besteht aus vielen Spielern. Wird die Mannschaft aufgelöst, so werden die Spieler nicht automatisch aufgelöst, sondern können z.B. in andere Mannschaften wechseln.
- Die Lebensdauer der Objekte der Spieler ist nicht von der Lebensdauer des Objektes der Mannschaft abhängig.
Das Objekt der Mannschaft kann gelöscht sein, die Objekte der Spieler können weiterhin existieren.
- Die Mannschaft *besitzt* die Objekte Spieler *nicht*, sondern **kennt** sie nur.

Komposition:

Beispiel für eine Komposition:



- Ein Stundenplan an der BBW umfasst vielen Lektionen.
Die Lektionen sind mit dem Stundenplan eng verbunden.
- Wird der Stundenplan gelöscht, so werden damit auch die Lektionen gelöscht.
- Die Lebensdauer der Objekte der Lektionen ist von der Lebensdauer des Objektes des Stundenplanes abhängig.
- Der Stundenplan **besitzt** die Objekte der Lektionen.

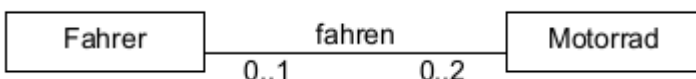
Multiplizität / Kardinalität:

In UML wird klar zwischen der *Multiplizität* und der *Kardinalität* einer Beziehung unterschieden:

- Die Multiplizität legt die möglichen Kardinalitäten einer Beziehung fest, meist indem die minimale und maximale Kardinalität angegeben wird.
- Die tatsächliche Kardinalität einer Beziehung kann nur für konkret bekannte Objekte angegeben werden. So gibt es in Deutschland zum Beispiel genau 2076 Städte. Die *Kardinalität* der Rolle Stadt in dieser Beziehung wäre also 2076. Die zugehörige *Multiplizität* der Rolle ist aber 1..*. Diese beschreibt einfach die möglichen Kardinalitäten, von denen die 2076 eben auch eine ist.

Die begriffliche Trennung zwischen Multiplizität und Kardinalität ist nicht immer so klar, wie sie innerhalb der UML definiert wird. Häufig wird der Begriff der Kardinalität analog zur Multiplizität in UML verwendet

Eine Multiplizität ist ein Intervall und hat eine obere und eine untere Grenze.



Ein Fahrer fährt 0 bis 2 Motorräder.

Ein Motorrad wird von einem oder keinem Fahrer gefahren.

Weitere Beispiele (nicht abschliessend)

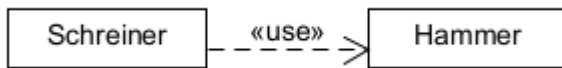
Min	Max	Darstellung
1	1	1
0	1	0..1
1	5	1..5
0	beliebig	0..*

Die Abhängigkeitsbeziehung

Die Abhängigkeitsbeziehung ist ein weiteres Modellierungselement in der UML.

Es gibt verschiedene Arten von Abhängigkeitsbeziehungen

Verwendungsbeziehung:



Ein Schreiner benützt einen Hammer.

Die Abhängigkeitsbeziehung bindet den Hammer und Schreiner schwächer aneinander als eine Assoziation.

Beispiele für weitere Abhängigkeitsbeziehungen sind:

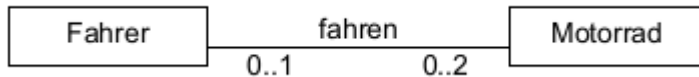
- Abstraktionsbeziehung <abstract>, <derive>, <trace>, <refine>
- Verteilungsbeziehung <deploy>

(Liste ist nicht vollständig)

Implementierung von Assoziation

Implementierung in Bezug auf die Navigation

Variante ohne Navigation

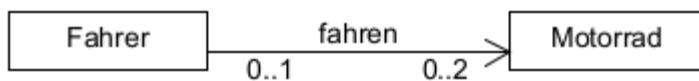


Ein Fahrer fährt 0 bis 2 Motorräder.

Ein Motorrad wird von einem oder keinem Fahrer gefahren.

In der Assoziation ist keine Navigation enthalten. Damit ist einer der unten gezeigten Möglichkeiten zu Implementierung möglich.

Variante Navigation von Fahrer zu Motorrad:

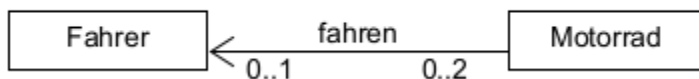


```

/* Fahrer hat zwei Motorräder
 * Die Referenzen können auch null sein
 */
public class Fahrer {
    private Motorrad bikeA;
    private Motorrad bikeB;
}
  
```

//In der Klasse Motorrad hat es kein Attribut Fahrer!

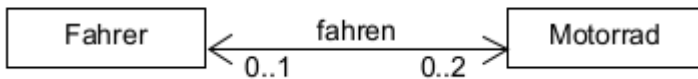
Variante Navigation von Motorrad zu Fahrer:



```

/* Motorrad hat einen Fahrer
 * Die Referenz kann auch null sein
 */
public class Motorrad {
    private Fahrer driver;
}
  
```

//In der Klasse Fahrer hat es keine Attribut Motorrad!

Variante bidirektionale Navigation:

```

/* Fahrer hat zwei Motorräder
 * Die Referenzen können auch null sein
 */
public class Fahrer {
    private Motorrad bikeA;
    private Motorrad bikeB;
}

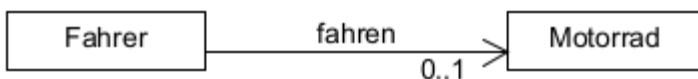
//und gleichzeitig

/* Motorrad hat einen Fahrer
 * Die Referenz kann auch null sein
 */
public class Motorrad {
    private Fahrer driver;
}
  
```

ACHTUNG:

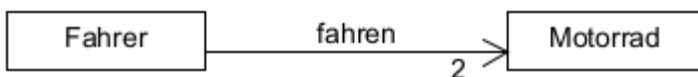
Die bidirektionale Implementierung ist nicht empfohlen, da es sehr schwierig wird die Konsistenz der Beziehung aufrecht zu erhalten.

Die bidirektionale Implementierung wird oft auch als schwerwiegender Designfehler angesehen.

Implementierung in Bezug auf Multiplizität

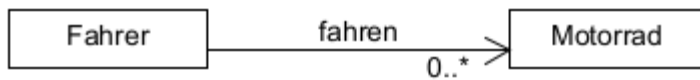
```

/* Fahrer hat ein oder kein Motorrad
 * Die Referenz kann auch null sein
 */
public class Fahrer {
    private Motorrad bike;
}
  
```



```

/* Fahrer hat genau zwei Motorräder
 * Die Referenzen dürfen hier NICHT null sein.
 * was eventuell nicht so einfach zu gewährleisten ist.
 */
public class Fahrer {
    private Motorrad bikeA;
    private Motorrad bikeB;
}
  
```



```

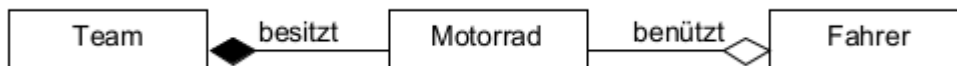
/* Fahrer hat beliebig viele Motorräder.
 * Die Liste kann auch leer sein
 */
public class Fahrer {
    private List<Motorrad> bikes;
}
  
```

Implementierung in Bezug auf Aggregation und Komposition

Die Unterscheidung zwischen Aggregation und Komposition kann in Java nicht allein durch die Sprache Java gemacht werden.

Denn die Lebensdauer eines Objektes ist bei Java nicht an die Lebensdauer eines anderen Objektes gebunden, sondern daran, ob noch jemand eine Referenz auf das Objekt hält.

Beispiel:



Im Motorradrennsport besitzen die Teams Motorräder, die von den Fahrern des Teams benützt (gefahren) werden. Der Fahrer besitzt das Motorrad nicht, er benützt es nur.

Team besitzt Motorrad → Komposition

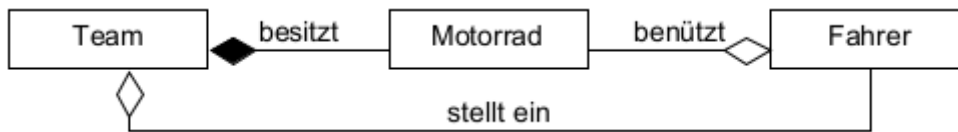
Fahrer benützt Motorrad → Aggregation

Wird nun das Team gelöscht, dann müssten auch die Motorräder des Teams gelöscht werden. Doch es kann sein, dass ein Fahrer noch eine Referenz auf ein Motorrad hält, und damit würde Java dieses Motorrad nicht löschen.

Java kann dieses Problem nicht für uns lösen. Wir als Programmieren können aber dazu Sorge tragen, dass Aggregation und Komposition eingehalten werden.

Beispiel nächste Seite.

Dazu erweitern wir das Beispiel, so dass das Team auch den Fahrer kennt.



```

public class Team {
    private Motorrad bike;
    private Fahrer driver;

    public Team() {
        bike = new Motorrad();
    }

    public setFahrer(Fahrer driver) {
        this.driver = driver;
        driver.setMotorrad(bike);
    }

    public reset() {
        driver.resetMotorrad();
    }
}

```

Das Team übernimmt die Verantwortung für das Motorrad.

Wird dem Team ein Fahrer übergeben, so "leiht" das Team dem Fahrer das Motorrad.

Wird nun das Team gelöscht, so muss man zuvor die Methode reset() aufrufen und kann dort dem Fahrer das Motorrad wieder wegnehmen.

Wenn Sie wissen, wann der Zeitpunkt gekommen ist, die Motorräder den Fahrer wegzunehmen, so können Sie das selbst implementieren.

Bei Spring können Sie über eine Annotierung das Framework anweisen, eine Methode unmittelbar vor dem Löschen des Objektes aufzurufen.

```

@Component
public class Team {
    private Motorrad bike;
    private Fahrer driver;

    public Team() {
        bike = new Motorrad();
    }

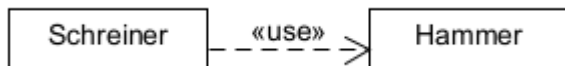
    public setFahrer(Fahrer driver) {
        this.driver = driver;
        driver.setMotorrad(bike);
    }

    @PreDestroy
    public reset() {
        driver.resetMotorrad();
    }
}

```

Was wann wie angewandt wird, hängt von der Idee der Lösung ab.

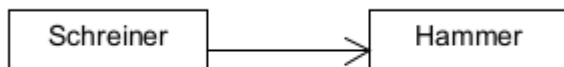
Implementierung von Verwendungsbeziehung



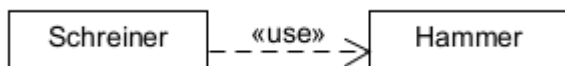
Die Verwendungsbeziehung bindet den Hammer und Schreiner schwächer aneinander als eine Assoziation.

Man kann die Beziehung allenfalls auch zu einer Aggregation wandeln.
 Und mit einer der zuvor gezeigten Lösungen implementieren.

Das ist insbesondere der Fall, wenn der Schreiner durch den Aufruf einer Methode des Objektes Hammers, dieses Objekt hammer in seinem Zustand verändert.
 Denn dann muss der Schreiner das Objekt hammer mindestens kennen.



Und wenn es eine <<use>> Beziehung bleiben soll?



Dann wäre eine Möglichkeit eine **statische Methode** zu verwenden.

Dann wird beim Aufruf der Methode kein Objekt verwendet, sondern **eine Klassen-Methode**.

ACHTUNG: damit entfällt die Möglichkeit den Zustand eines Objekts hammer zu verändern, es gibt beim Aufruf kein Objekt hammer mit einem Zustand.

Verwenden Sie Klassen-Methoden vorsichtig und *sparsam*.

```

public class Schreiner {

    public void verbindeBretter(){
        //Aufruf der Klassenmethode über den Klassennamen
        Hammer.nageln();
    }

}

public class Hammer {

    //static macht die Methode zur Klassenmethode
    public static void nageln(){
        //to be defined;
    }

}
  
```