Einführung in die OOP mit C#

Teil 2: Beziehungen

BITLC | Tom Selig
19. Dezember 2025

this

- Mit dem Schlüsselwort `this` kann ein Objekt seine eigene Referenz (Speicheradresse) abfragen.
- Es kann eingesetzt werden, um Namenskonflikte aufzulösen.
- Es kann auch genutzt werden, wenn das Objekt "sich selbst" weitergeben soll.

```
class Person {
    private Person partner;
    public string Name { get; set; }

    public void SetPartner(Person partner) {
        if (this.partner == null) {
            this.partner = partner;
            this.partner.SetPartner(this);
        }
    }
    public Person GetPartner() {
        return partner;
    }
}
```

```
static void Main(string[] args) {
    Person adam = new Person("Adam");
    Person eva = new Person("Eva");
    eva.SetPartner(adam);
    Console.WriteLine(eva.GetPartner().Name);
    Console.WriteLine(adam.GetPartner().Name);
}
```

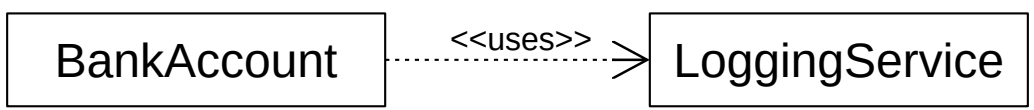
Beziehungen zwischen Klassen

- Klassen können zueinander in Verbindung stehen.
- Dabei unterscheidet man verschiedene Arten von Beziehungen:

Abhängigkeit:	(Referenz)	
Assoziation:	(hat ein)	
Aggregation:	(hat ein + Ganzes/Teil)	
Komposition:	(hat ein + Ganzes/Teil + Besitz)	
Vererbung:	(ist ein)	=> siehe OOP - Teil 3
Implementierung:	(Vertrag)	=> siehe OOP - Teil 4

Abhängigkeit

- Eine Abhängigkeit besagt, dass eine Klasse, der **Client**, von einer anderen Klasse, dem **Supplier**, abhängig ist.
- Der **Client** braucht den **Supplier** um zu funktionieren. Er nutzt Elemente (Attribute und/oder Methoden) des **Suppliers**.
- Der **Client** besitzt den **Supplier** aber nicht.
- Änderungen am **Supplier** werden in der Regel auch Änderungen am **Client** erfordern.



Abhängigkeit

```
internal class LoggingService {
    public void Log(int accountId, string message) {
        Console.WriteLine(accountId);
        Console.WriteLine("   " + message);
    }
}
```

```
static void Main(string[] args) {
    var acc = new BankAccount();

    var logger = new LoggingService();
    acc.Deposit(1000, logger);
}
```

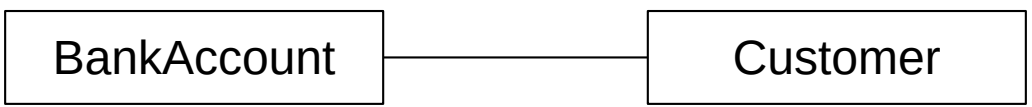
```
internal class BankAccount {
    private readonly int _accountId;
    private decimal _balance;

    // ...
    public void Deposit(decimal amount, LoggingService logger) {
        _balance += amount;
        logger.Log(_accountId, $"Einzahlung {amount}");
    }
}
```

- `BankAccount` hat keinen eigenen `LoggingService`. Der Methode `Deposit` wird beim Aufruf ein `LoggingService` übergeben.

Assoziation

- Eine Assoziation besagt, dass es eine feste Verbindung zwischen zwei Klassen gibt.
- In einer Klasse (oder in beiden) gibt es ein Attribut vom Typ der jeweils anderen Klasse („Hat-ein-Beziehung“).
- Die Objekte beider Klassen können auch alleine existieren.
- Die beiden Objekte begegnen sich „auf Augenhöhe“, es gibt keine Hierarchie zwischen ihnen.



Assoziation

```
internal class BankAccount {
    // Felder
    // ...
    private decimal _balance;
    private Customer _owner;

    // Konstruktor
    public BankAccount(Customer owner,
            decimal balance) {
        _owner = owner;
        _balance = balance;
    }

    // Methoden
    public Customer GetOwner() {
        return _owner;
    }
    public void SetOwner(Customer owner) {
        _owner = owner;
    }
    // ...
}
```

```
internal class Customer {
    public string FirstName { get; init; }
    public string LastName { get; init; }
    public Customer(string first, string last) {
        FirstName = first;
        LastName = last;
    }
}
```

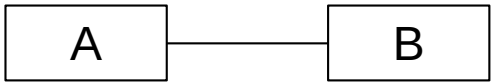
```
static void Main(string[] args)
{
    var customer = new Customer("John", "Doe");
    var account = new BankAccount(customer, 10);
    string name = account.GetOwner().LastName;
}
```

- Das `Customer`-Objekt wird im `BankAccount` gespeichert, existiert aber auch außerhalb.

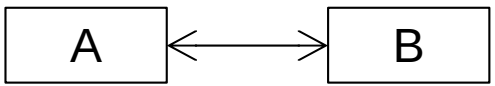
Navigationsrichtung

- Bei einer Assoziation kann durch einen Pfeil bzw. ein Kreuz die Navigationsrichtung angegeben werden.
- Dadurch wird gesagt, in welche Richtung die Navigation von einem Objekt zu seinem Partnerobjekt möglich sein soll.
- Ein Pfeil zeigt eine mögliche Navigation an, ein Kreuz dagegen eine nicht mögliche Navigation.
- Wird weder Pfeil noch Kreuz notiert, ist die Navigation unbestimmt.
- Somit wird vorgegeben, in welcher Klasse ein Attribut der Partnerklasse angelegt werden muss (oder kann).

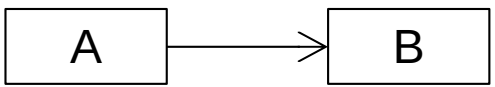
Navigationsrichtung



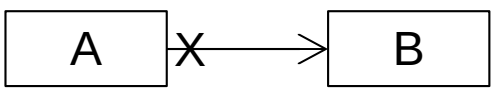
Ungerichtete Kanten besagen in der UML, dass die Navigationsrichtung nicht definiert ist. In einer der beiden Klassen muss aber ein Attribut der anderen Klasse vorgesehen werden.



Bei einer bidirektionalen Assoziation muss die Navigation in beide Richtungen möglich sein. In beiden Klassen gibt es daher ein Attribut vom Typ der jeweils anderen Klasse.



Bei dieser Variante ist die Navigation von A nach B möglich, von B nach A aber undefiniert. In Klasse A muss ein Attribut vom Typ B vorhanden sein, in B kann eines vom Typ A vorhanden sein.



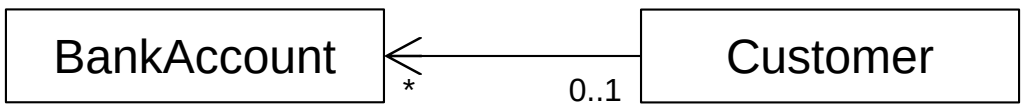
Bei dieser Variante ist die Navigation von A nach B möglich, aber von B nach A nicht erlaubt. Hier darf nur ein Attribut vom Typ B in A sein, nicht aber umgekehrt.

Multiplizitäten

- Multiplizitäten geben an, wie viele Verbindungen es zwischen den Objekten der Klassen gibt.
- Um sie zu ermitteln, müssen immer zwei Sätze gebildet werden, die mit „***Ein*** *Objekt der Klasse …*“ beginnen:

  Ein Objekt der Klasse A steht mit ***n*** Objekten der Klasse B in Beziehung.

  Ein Objekt der Klasse B steht mit ***m*** Objekten der Klasse A in Beziehung.
- Das Ergebnis (n oder m) wird dann am jeweils gegenüber-liegenden Ende der Assoziation notiert.



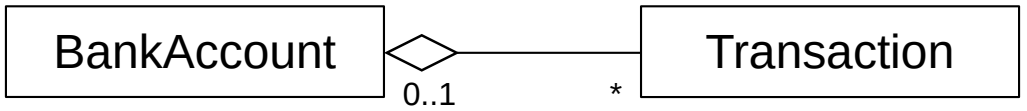
Multiplizitäten

- Multiplizitäten werden in einer der folgenden Formen notiert:

n	genau n
*	kein, ein oder viele (optional)
1..*	ein oder viele
0..1	kein oder ein (optional)
m..n	m bis n
n..*	mindestens n
m,n,p	m oder n oder p
m,n..p,q	m oder n bis p oder q

Aggregation

- Ein Aggregation ist eine Sonderform der Assoziation und beschreibt neben der „Hat-ein-“ eine „Ganzes-Teil-Beziehung“.
- Die Teile bilden dabei ein größeres Ganzes, es besteht eine Art von Rangordnung oder Hierarchie zwischen den Objekten.
- Die Lebensdauer der Teile ist von der des Ganzen unabhängig, d.h. die Teile können auch ohne das Ganze existieren.
- Ein Teil kann ggf. auch zu mehreren Ganzen gehören.



Aggregation

```
internal class Transaction {
  public DateTime Timestamp { get; init; }
  public decimal Amount { get; init; }
  public string Description { get; init; }
}
```

```
static void Main(string[] args)
{
  var customer = new Customer("J.","Doe");
  var account = new BankAccount(
    customer, 1000);

  var trans1 = new Transaction() {
    Timestamp = DateTime.Now,
    Amount = 1_000,
    Description = "Intitial Deposit"
  };

  account.AddTransaction(trans1);
}
```

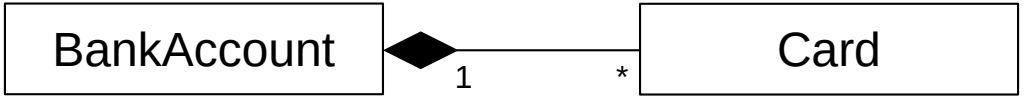
```
internal class BankAccount {
  // Felder
  // ...
  private List<Transaction> _transactions = [];

  // Konstruktoren
  // ...

  // Methoden
  // ...
  public List<Transaction> GetTransactions() {
      return _transactions;
  }
  public void AddTransaction(Transaction t) {
      _transactions.Add(t);
  }
}
```

Komposition

- Eine Komposition ist eine starke Form der Aggregation und beschreibt ebenfalls eine Ganzes-Teil-Beziehung.
- Dabei wird aber unterstellt, dass das Ganze und seine Teile fest miteinander verbunden sind.
- Wird das Ganze zerstört, existieren auch die Teile nicht mehr.
- In der Regel werden die Teile im Inneren des Ganzen erzeugt.
- Ein Teil kann daher auch nur zu genau einem Ganzen gehören.



Komposition

```
internal class Card {
  public string CardNumber { get; }
  public DateTime ExpiryDate { get; }

  public Card(string number) {
    CardNumber = number;
    ExpiryDate = DateTime.Now.AddYears(3);
  }

  // ...
}
```

```
internal class BankAccount {
  // Felder
  // ...
  public Card AccountCard { get; }

  // Konstruktor
  public BankAccount(Customer o, decimal b) {
    _owner = o;
    _balance = b;
    _accountId = _nextId++;

    AccountCard = new Card("CARD-" + _accountId);
  }

  // ...
}
```

- Das `Card`-Objekt wird im `BankAccount` erzeugt, es existiert nur dort. Wird der das `BankAccount`-Objekt gelöscht, wird auch das `Card`-Objekt gelöscht.

Übersicht Beziehungen