

Objektorientierte Programmierung (OOP)

Am Anfang steht wohl die Frage was Objektorientierte Programmierung überhaupt ist. Dies ist nicht ganz einfach zu erklären und es gibt viele Definitionen die dies versuchen.

Grob gesagt ist **Objektorientierte Programmierung (OOP) ein Verfahren zur Strukturierung von Programmen, bei dem Programmlogik zusammen mit Zustandsinformationen (Datenstrukturen) in Einheiten, den Objekten, zusammengefasst werden.**

Es ist also möglich mehrere Objekte des gleichen Typs zu haben. Jedes dieser Objekte hat dann seinen individuellen Zustand. Dies bietet die Möglichkeit einer besseren Modularisierung der Programme, sowie einer höheren Wartbarkeit des Quellcodes.

Klassen

Die Klasse (class) ist die zentrale Datenstruktur in C + +. Sie kapselt zusammengehörige Daten und Funktionen vom Rest des Programmes ab. Sie ist das Herz der objektorientierten Programmierung (OOP).

Klassen sind ein erweitertes Konzept von Datenstrukturen denen es erlaubt ist, außer Daten, noch Methoden zu beinhalten. Ein Objekt ist eine Instanz einer Klasse, welches sich die Eigenschaften der Klasse zunutze machen kann. Es ist möglich mehrere Objekte einer Klasse zu instanziiieren, wobei jedes dieser Objekte zwar die selben Eigenschaften hat, intern aber einen ganz individuellen Zustand haben kann.

Beispiel: Zwei Instanzen der Klasse „Gegner“ werden erzeugt. Nachdem ein Gegner Schaden erlitten hat sind seine Lebenspunkte auf 50 gesunken. Die des anderen Gegners haben allerdings noch den Wert 100.

Klassen werden normalerweise mit dem Schlüsselwort **class** deklariert. Außerdem haben sie immer folgendes Format:

```
class Klassenname
{
  Zugriffsspezifizierung 1:
  Member 1;
  ...
  Zugriffsspezifizierung 2:
  Member 2;
  ...
  ...
};
```

Der Klassenname identifiziert die Klasse, der Objektname ist eine optionale Liste von Namen von Objekten dieser Klasse. Der Körper der Klasse wird durch geschweifte Klammern gekennzeichnet und kann verschiedene Member (Methoden, Membervariablen) beinhalten. Diesen können verschiedene Zugriffsspezifizierungen zugewiesen werden.

Zugriffsmodifikatoren

public: Auf Members kann von überall zugegriffen werden

private: Auf Member kann nur innerhalb der Instanz zugegriffen werden

protected: Auf Member kann nur innerhalb der Instanzen der selben Klassen oder vererbten (=abgeleiteten) Klassen zugegriffen werden

Attribute

Die Festlegung, welche Daten zu einem Typ gehören, erfolgt bei der Definition der Klasse. Die Objekte enthalten jeweils unabhängig voneinander einen Satz von Datenkomponenten (Attributen). Ihre Startwerte können durch Konstruktoren festgelegt werden. C++ erlaubt auch die Zuweisung von Anfangswerten bei der Definition:

```
class Enemy {  
    public:    //Zugriffsrecht public  
        void setHealth(int h);  
        int getHealth();  
        string name;  
            int id;  
    private: //Zugriffsrecht private  
        int health;  
};
```

Elementzugriff

Im Beispiel davor wird ein Objekt vom **Typ Enemy** angelegt. Das **Objekt** enthält die **3 Variablen health (integer), id (int) und name (String)**, wie es in der Klassendefinition zu sehen ist. Um auf eine öffentliche Elementvariable zugreifen zu können, wird an den Objektnamen ein **Punkt** und dann der in der Klassendefinition verwendete Elementname gehängt. Im Beispiel wird der Name auf Andreas gesetzt.

```
...  
//Zugriff auf öffentliche Variable name  
e1.name="Andreas";  
//Zugriff auf öffentliche Variable id  
e1.id=1;  
...  
//e1.health=100; ist nicht erlaubt, da health das Schlüsselwort private  
besitzt!!!
```

Klassenmethoden

Nun können Sie ein Objekt von der Klasse Enemy erzeugen. Aber was nützt Ihnen die schönste

Datenstruktur in Ihrem Programm, wenn sie nicht durch Funktionen zum Leben erweckt wird? Sie werden z.B. einen Namen und Lebenspunkte eingeben und ausgeben wollen. Vielleicht wollen Sie die Lebenspunkte verringern oder erhöhen. Kurz gesagt, ein Datenverbund ist nichts wert ohne Funktionen, die auf ihn wirken. Aber die Funktionen sind auch nur im Zusammenhang mit ihrem Datenverbund sinnvoll. Aus diesem Grund werden die Funktionen ebenso in die Klasse integriert wie die Datenelemente. Eine Funktion, die zu einer Klasse gehört, nennt man Elementfunktion oder auf englisch member function. In anderen objektorientierten Programmiersprachen spricht man auch von einer Methode oder Operation. Aufruf So, wie Sie auf Datenelemente nur über ein real existierendes Objekt, also eine Variable dieser Klasse zugreifen können, kann auch eine Funktion nur über ein Objekt aufgerufen werden. Objekt und Funktionsnamen werden dabei durch einen Punkt getrennt. Die Funktion arbeitet mit den Daten des Objekts, über das sie gerufen wurde. Aus prozeduraler Sicht könnte man es so sehen, dass eine Elementfunktion immer bereits einen Parameter mit sich trägt, nämlich das Objekt, über das sie aufgerufen wurde.

Beispiel:

```
#include <iostream>
#include <conio.h>
#include <string.h>

using namespace std;

//Klasse Enemy
class Enemy {
public:    //Zugriffsrecht public
    void setHealth(int h);
    int getHealth();
    string name;
           int id;
private: //Zugriffsrecht private
    int health;
};

//Methode setHealth zum Setzen der Lebenspunkte
void Enemy::setHealth(int h)
{
    health=h;
}

//Methode getHealth() zum Auslesen der Lebenspunkte
int Enemy::getHealth()
{
    return health;
}

//Hauptprogramm
int main()
{
    Enemy e1;    //Objekt e1 der Klasse Enemy wird erzeugt

    //Zugriff auf öffentliche Variable name
    e1.name="Andreas";
```

```

//Zugriff auf öffentliche Variable id
e1.id=1;

//Aufruf der Methode setHealth
e1.setHealth( 100 );
//Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";
//Aufruf der Methode setHealth
e1.setHealth( 50 );
//Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";

return 0;
}

```

Methodendefinition innerhalb einer Klasse

Alternativ können Sie die Elementfunktion auch direkt in der Klasse definieren. Das wird leicht unübersichtlich, darum sollten Sie das nur bei sehr kurzen Funktionen tun.

```

#include <iostream>
#include <conio.h>
#include <string.h>

using namespace std;

class Enemy {
public:      //Zugriffsrecht public
           /****** INLINE - METHODENDEFINITION *****/
           //Methode setHealth zum Setzen der Lebenspunkte
           void setHealth(int h)
           {
               health=h;
           }
           //Methode getHealth() zum Auslesen der Lebenspunkte
           int getHealth()
           {
               return health;
           }

           string name;
           int id;

private:   //Zugriffsrecht private
           int health;

```

```
};

//Hauptprogramm
int main()
{
    Enemy e1;    //Objekt e1 der Klasse Enemy wird erzeugt

    //Zugriff auf öffentliche Variable name
    e1.name="Andreas"

    //Aufruf der Methode setHealth
    e1.setHealth( 100 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";
    //Aufruf der Methode setHealth
    e1.setHealth( 50 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";

    return 0;
}
```

Aufruf

So, wie Sie auf Datenelemente nur über ein real existierendes Objekt, also eine Variable dieser Klasse zugreifen können, kann auch eine Funktion **nur über ein Objekt aufgerufen werden**.

Objekt und **Funktionsnamen** werden dabei durch einen **Punkt** getrennt. Die Funktion arbeitet mit den Daten des Objekts, über das sie gerufen wurde. Aus prozeduraler Sicht könnte man es so sehen, dass eine Elementfunktion immer bereits einen Parameter mit sich trägt, nämlich das Objekt, über das sie aufgerufen wurde.

```
....
Enemy e1;    //Objekt e1 der Klasse Enemy wird erzeugt

//Aufruf der Methode setHealth
e1.setHealth( 100 );
....
```

Konstruktor

Die Elementfunktion, die beim Erzeugen eines Objekts aufgerufen wird, nennt man Konstruktor. In dieser Funktion können Sie dafür sorgen, dass alle Elemente des Objekts korrekt initialisiert sind. Der Konstruktor trägt immer den Namen der Klasse selbst und hat keinen Rückgabetypp, auch nicht void. Der Standardkonstruktor hat keine Parameter.

```
class Enemy {
    public:          //Zugriffsrecht public

        /**** Konstruktor ***/
        Enemy()
        {
            health=0;
            name="";
            id=0;
        }

        string name;
        int id;

    private:        //Zugriffsrecht private
        int health;
};
```

Destruktor

Im Falle einer Datumsklasse wäre es sinnvoll, dass der Konstruktor alle Elemente auf 0 setzt. Daran kann jede Elementfunktion leicht erkennen, dass das Datum noch nicht festgelegt wurde. Sie könnten alternativ das aktuelle Datum ermitteln und eintragen. Im Beispiel ist auch ein Destruktor definiert worden, obwohl er im Falle eines Datums keine Aufgabe hat.

```
class Enemy {
    public:          //Zugriffsrecht public

        /**** Destruktor ***/
        ~Enemy()
        {
            cout << "Ressourcen von " << name << " wurden
freigegeben!";
        }

        char name[50];
        int id;

    private:        //Zugriffsrecht private
        int health;
};
```

Überladen von Konstruktor

Konstrukturen können genauso überladen werden wie normale Funktionen auch. Es kann neben dem Standardkonstruktor auch mehrere weitere Konstrukturen mit verschiedenen Parametern geben. Der

Compiler wird anhand der Aufrufparameter unterscheiden, welcher Konstruktor verwendet wird.

```
class Enemy {
    public:        //Zugriffsrecht public

        /**** Konstruktor ****/
        Enemy()
        {
            name="Max Mustermann";
        }
        Enemy(string n)
        {
            name=n;
        }
        Enemy(string n, int h)
        {
            name=n;
            health=h;
        }
        /**** Destruktor ****/
        ~Enemy()
        {
            cout << "Ressourcen von " << name << " wurden
freigegeben!";
        }

        char name[50];
        int id;

    private:      //Zugriffsrecht private
        int health;
};
```

Je nachdem wie viele Parameter beim Konstruktoraufbau übergeben werden, wird der passende Konstruktor ausgewählt. Existiert gar kein Konstruktor so wird vom Compiler ein leerer Konstruktor eingefügt.

```
int main (void)
{

    Enemy e;                //führt den Standardkonstruktor Enemy() aus
    -> name="Max Mustermann"
    Enemy e("Fritz Phantom"); //führt den 2. Konstruktor mit den
    passenden Argumenten aus -> name="Fritz Phantom"
    Enemy e("Hans Wurst", 100); //führt den 3. Konstruktor mit den
    passenden Argumenten aus -> name="Hans Wurst", health=100

    return 0;
}
```



Vererbung (=Klassenbeziehungen)

Die Informatik steckt voller schöner Analogien. So hat die objektorientierte Programmierung den Begriff der »Vererbung« eingeführt, wenn eine Klasse von einer anderen Klasse abgeleitet wird und deren Eigenschaften übernimmt.

Basisklasse

Eine Klasse kann als Basis zur Entwicklung einer neuen Klasse dienen, ohne dass ihr Code geändert werden muss. Dazu wird die neue Klasse definiert und dabei angegeben, dass sie eine abgeleitete Klasse der Basisklasse ist. Daraufhin gehören alle öffentlichen Elemente der Basisklasse auch zur neuen Klasse, ohne dass sie erneut deklariert werden müssen. Man sagt, die neue Klasse erbt die Eigenschaften der Basisklasse.

Spezialisierung

Durch die Elemente, die in der abgeleiteten Klasse definiert werden, wird die abgeleitete Klasse zu einem besonderen Fall der Basisklasse. Sie besitzt alle Eigenschaften der Basisklasse. Sie können neue Elemente hinzufügen. Am folgenden Beispiel wird deutlich, warum das Hinzufügen von Eigenschaften eine Spezialisierung ist.

Ein Beispiel

Aus Sicht eines Computerprogramms haben alle **Personen** (=Basisklasse)

- Namen
- Adressen und
- Telefonnummern.

Geschäftspartner haben darüber hinaus eine

- Bankverbindung.

Da die Geschäftspartner auch Personen sind, haben sie neben ihrer Bankverbindung auch Namen, Adressen und Telefonnummern (von der Basisklasse).

Einige Geschäftspartner können auch **Kunden** sein. **Kunden** haben zusätzlich zu den Eigenschaften eines Geschäftspartners noch eine

- Lieferanschrift.

Lieferanten sind keine Kunden, aber auch Geschäftspartner. Sie haben noch

- eine Anzahl an offenen Rechnungen.

Selbst **Mitarbeiter** sind eigentlich Geschäftspartner, denn sie haben eine Bankverbindung. Darüber

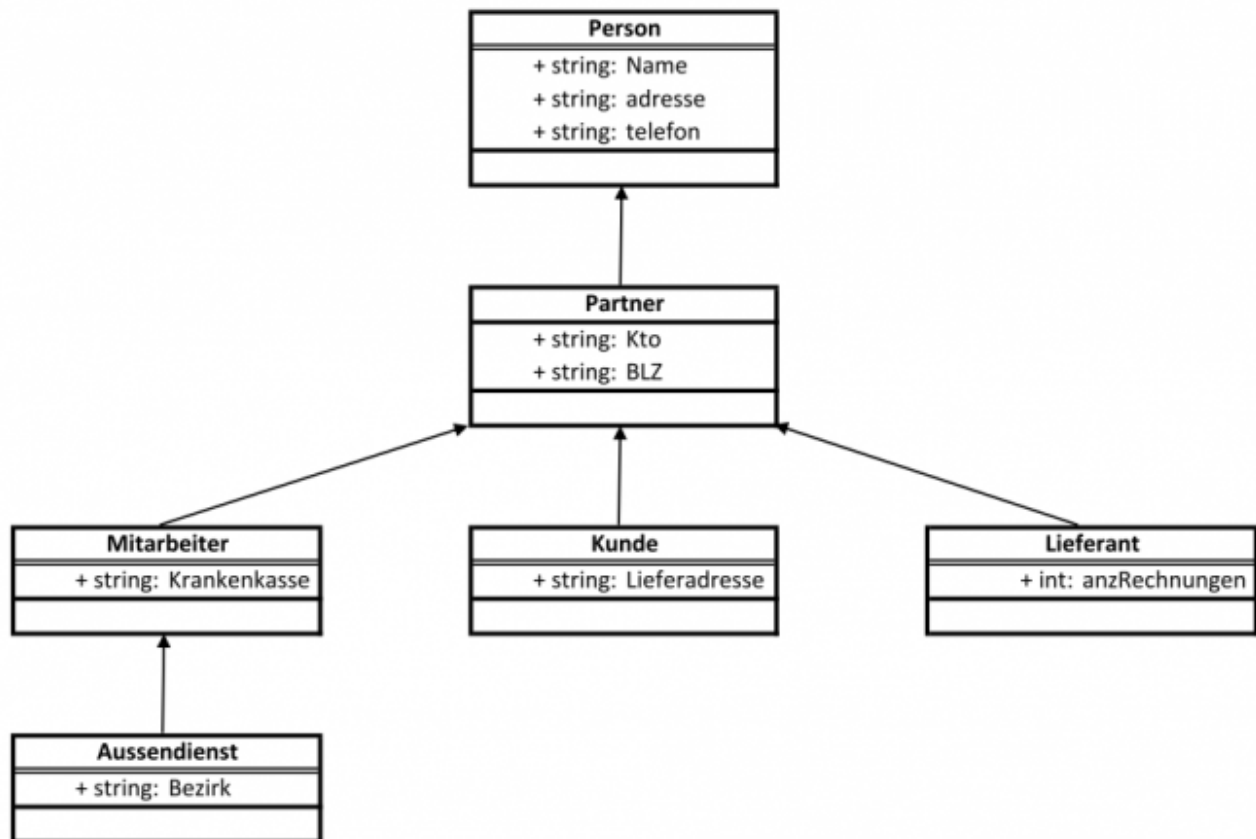
hinaus haben sie eine

- Krankenkasse.

Außendienstler haben alle Eigenschaften eines Mitarbeiters und zusätzlich ihren

- Bezirk.

Beispiel als UML-Diagramm:



Beispiel in C++

```

class Person
{
    public:
        string Name, Adresse, Telefon;
};

class Partner : public Person
{
    public:
        string Kto, BLZ;
};

class Mitarbeiter : public Partner
{
    public:
        string Krankenkasse;
};
  
```

```
};

class Kunde : public Partner
{
    public:
        string Lieferadresse;
};

class Lieferant : public Partner
{
    public:
        int anzRechnungen;
};

class Aussendienst : public Mitarbeiter
{
    public:
        string Bezirk;
};

int main (void) {

    Aussendienst a1;

    a1.Bezirk="Amstetten";
    a1.Name="Hans Wurst";
    a1.Krankenkasse="Gebietskrankenkasse";
    a1.BLZ="14200";

    return 0;
}
```

Zusammenfassung

Die nachfolgende Tabelle zeigt nochmals in einer Übersicht, wie sich das Zugriffsrecht beim Ableiten auf die Basisklassen-Member auswirkt:

ACCESS	Member Superklasse	Wird zu ... der Subklasse
private	private	kein Zugriff
	protected	private-Member
	public	private-Member
protected	private	kein Zugriff
	protected	protected-Member
	public	protected-Member
public	private	kein Zugriff
	protected	protected-Member
	public	public-Member

Vorteile von Vererbungen

Damit stellt die Ausgangsklasse Person die Verallgemeinerung dar und jede abgeleitete Klasse eine

Spezialisierung. Der Vorteil dieser Technik ist, dass der bestehende Code der Basisklasse nicht noch einmal für die neu geschaffene Klasse geschrieben werden muss. Beispielsweise würde eine Prüffunktion der Adresse, die der Klasse Person hinzugefügt wird, automatisch auch allen anderen Klassen, die direkt oder indirekt von Person abgeleitet wurden, hinzugefügt, ohne dass eine Zeile Code mehr geschrieben werden müsste. Eine Änderung in der Klasse Mitarbeiter würde immer auch auf die Außendienstler durchschlagen. Es gibt aber keine Rückwirkung auf die Geschäftspartner.

Objektbeziehungen

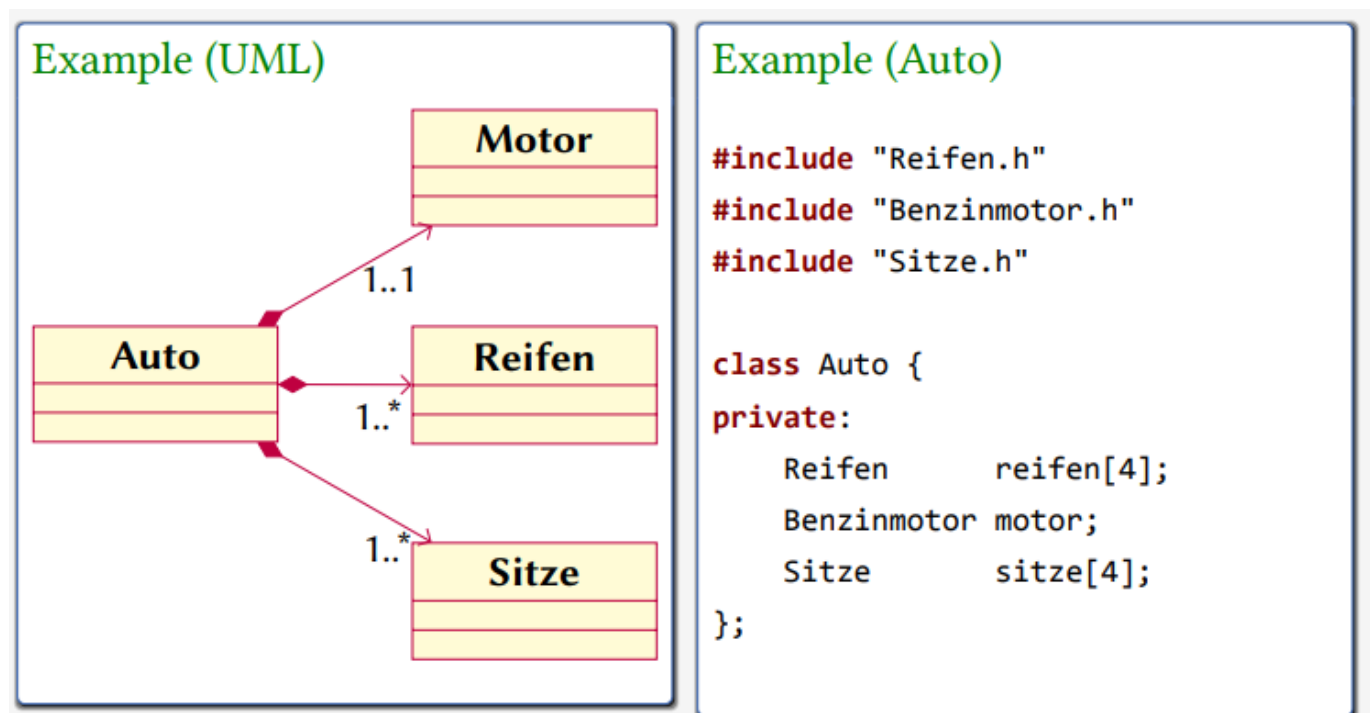
Komposition

Reale und komplexe Objekte bestehen meist aus kleinen und einfachen Objekte

- Auto besteht aus Reifen, Motor, Sitzen, . . .
- PC besteht aus CPU, Motherboard, RAM, . . .

Im objektorientierten Paradigma nennt man diese Beziehung **Komposition**. Diese bildet eine „besteht aus“ oder „hat ein“ - Beziehung ab.

- Komposition wird in C++ mit Attributen umgesetzt



Besonderheiten

- Oberklasse besitzt die Subklassen
 - entscheidet über deren Leben und Tod
- Subklassen werden erstellt wenn die Oberklasse erstellt wird
- Subklassen werden gelöscht wenn die Oberklasse gelöscht wird

Vor- und Nachteile der Komposition

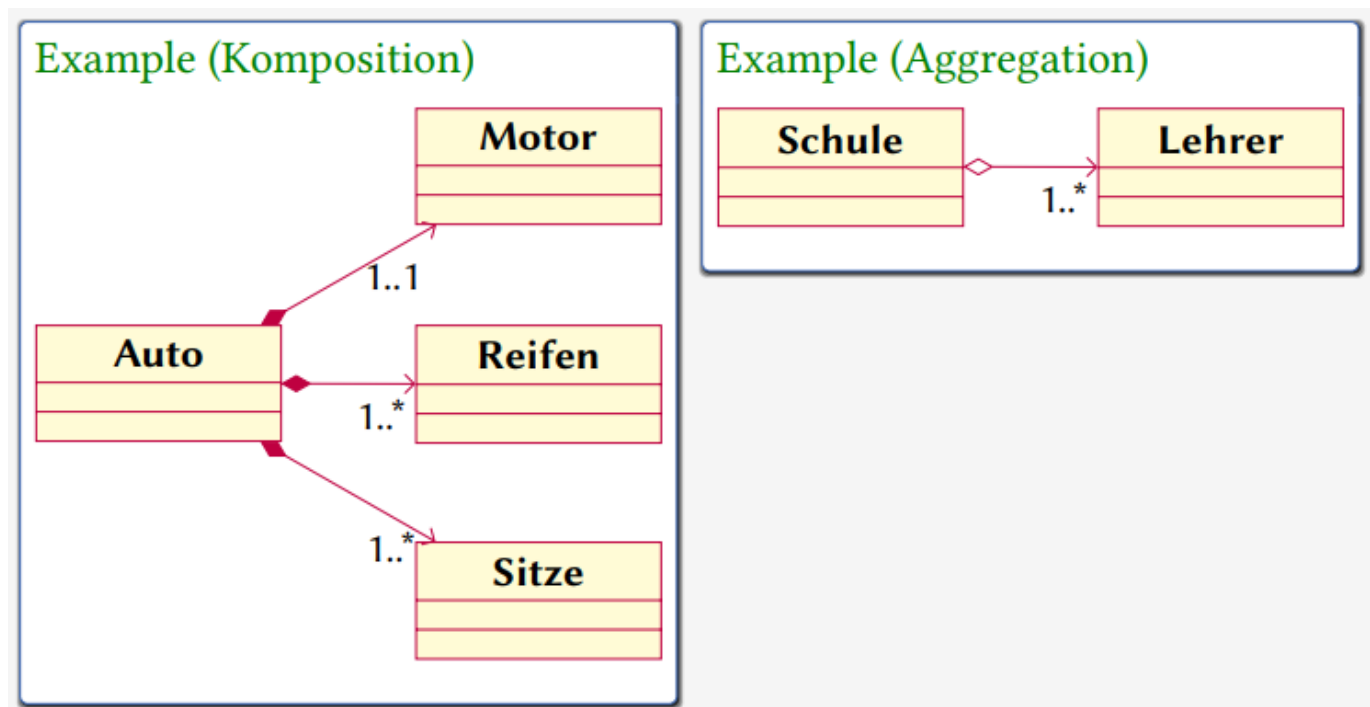
- Vorteile
 - jede Klasse kann klein gehalten werden
 - Subklassen können wieder verwendet werden
 - z.B. Fahrrad braucht Reifen
 - Komplexität verteilt sich dadurch auf verschiedene Klassen, statt auf einer einzigen
- Nachteile
 - Oberklassen sind schlechter wiederverwendbar

Aggregation

Aggregation ist eine spezielle Form der Komposition bildet auch eine besteht aus oder hat ein Beziehung ab. Die Oberklasse hat jedoch keine Besitzansprüche

- Subklassen leben weiter und werden **nicht** zerstört wenn die Oberklasse zerstört wird
- Subklassen werden auch **nicht** automatisch erstellt wenn die Oberklasse erstellt wird

Unterschied Aggregation/Komposition



Example (Klasse)

```
class Lehrer {
private:
    string name;
public:
    Lehrer(string name) : name(name){}
};

class Schule {
private:
    Lehrer* lehrer;
public:
    Schule(Lehrer* m) : lehrer(m){}
};
```

Example (Main)

```
int main(void) {
    Lehrer* bob = new Lehrer("Bob");

    {
        Schule schule(bob);
    }

    // Lehrer existiert immer noch obwohl
    // die Schule schon zerstört ist
    delete bob;
}
```

Assoziation

Eine Assoziation in C++ ist eine lose Verbindung zwischen Objekten. Die Objekte kennen sich und die Umsetzung erfolgt über Argumenten in Methoden.

Example (UML)



Example (Kreis)

```
class Kreis {
public:
    setzeMittelpunkt(Punkt p);
};
```

```
class Bank {
public:
    void transferMoney(Account* fromAccount, Account* toAccount, double
amount) {
        // Code to transfer money from one account to another
    }
};

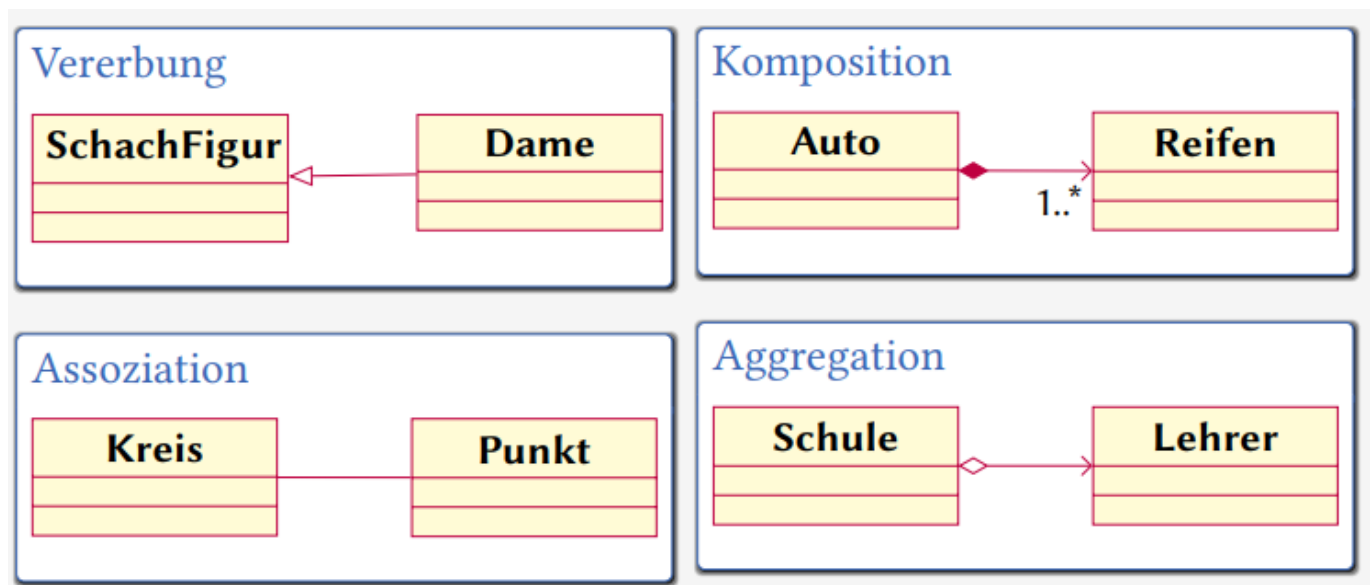
class Account {
public:
    Account(int id, double balance) : id(id), balance(balance) {}
    int getId() { return id; }
    double getBalance() { return balance; }
private:
```

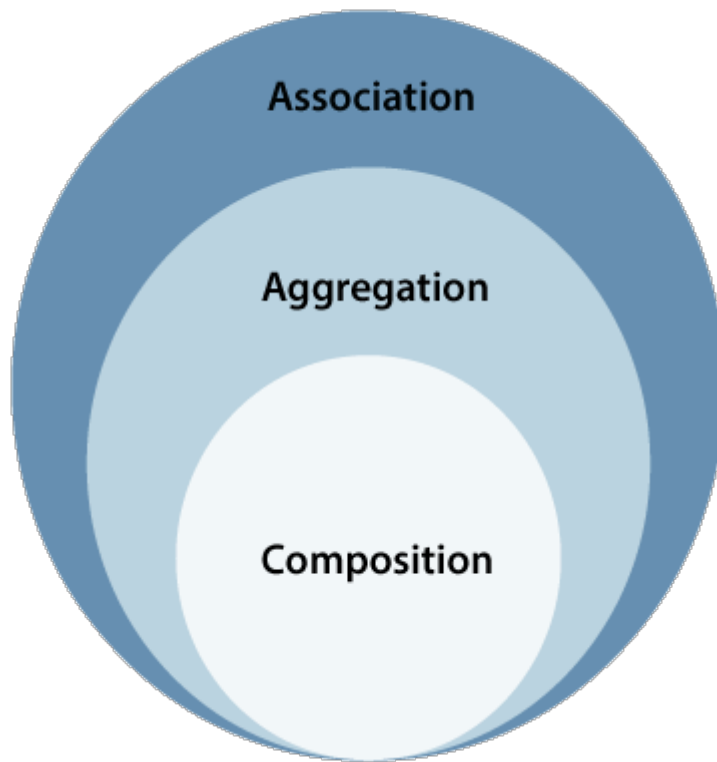
```
int id;
double balance;
};

int main() {
    Account* account1 = new Account(123, 1000.00)
    Account* account2 = new Account(456, 500.00);
    Bank bank;
    bank.transferMoney(account1, account2, 250.00);
    return 0;
}
```

Zusammenfassung

- Assoziation beschreibt Beziehungen zwischen Klassen.
- Aggregation repräsentiert eine „hat eine“ Beziehung, bei der Objekte unabhängig existieren können.
- Komposition identifiziert eine „Teil von“ Beziehung, bei der Objekte in anderen Objekten integriert sind.





From:

<http://elearn.bgamstetten.ac.at/wiki/> - Wiki

Permanent link:

http://elearn.bgamstetten.ac.at/wiki/doku.php?id=inf:inf7ai_202526:10:10_06

Last update: **2025/11/18 15:12**

