



FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Teil 4 – Objektorientierung 2 | Objektbeziehungen

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de

Objektorientierung 2

Inhalt



FH MÜNSTER
University of Applied Sciences

- UML
 - Was ist das?
 - Wichtige Diagrammtypen.
 - Objekt- und Klassendiagramme.
- Beziehungen zwischen Objekten
- Beziehungen zwischen Klassen
 - Vererbung
 - Klassen Ableiten in C++



Objektorientierung 2

UML (Unified Modeling Language)



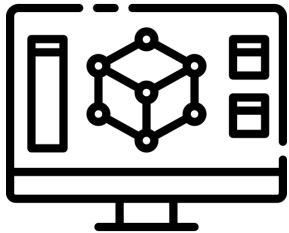
FH MÜNSTER
University of Applied Sciences



UML

Was ist ein Modell?

Ein Modell...



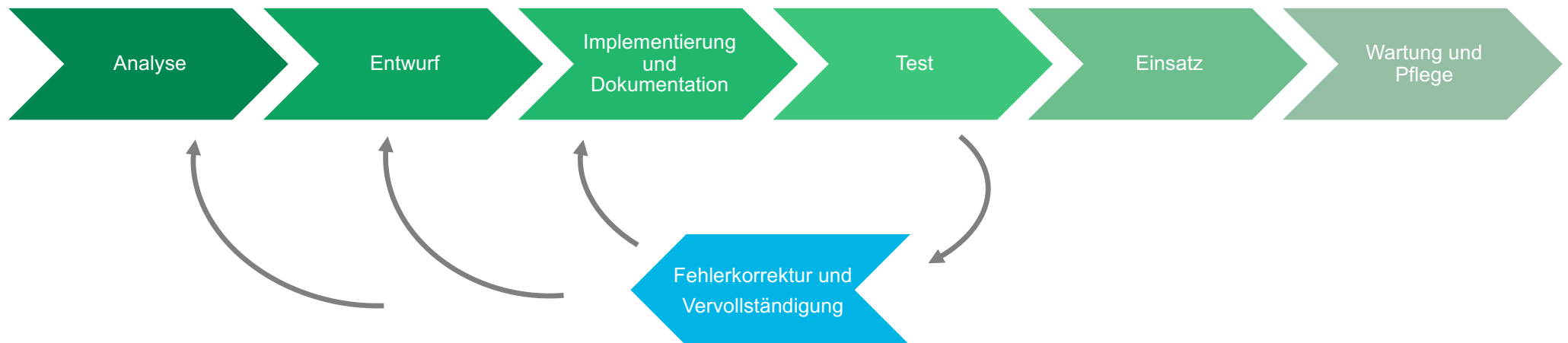
- abgeleitet aus dem lat. **modulus** = "Maß, Maßstab", bedeutet Muster, Vorbild, Entwurf
- kann ein ein **Abbild** oder ein **Vorbild** sein
- kann **konkret** sein oder **abstrakt**
- (absichtlich) **nicht originalgetreu**
- hebt manche **Eigenschaften** hervor und lässt andere weg
- der **Verwendungszweck** eines Modells bestimmt welche Eigenschaften modelliert werden

UML

Phasen der Softwareentwicklung



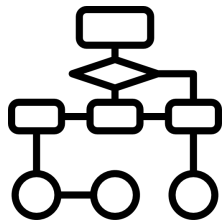
FH MÜNSTER
University of Applied Sciences



UML

Was ist UML?

UML...



- ist die Kurzform von **U**nified **M**odeling **L**anguage
- ist eine allgemein verwendbare **Modellierungssprache**
- kann **auch außerhalb der Softwareentwicklung** verwendet werden
- definiert **verschiedene Diagrammtypen**, mit denen statische und dynamische Aspekte beliebiger Anwendungsgebiete modelliert werden können

Diagrammtypen

Class Diagram Elements

Package::AbstractClass



Object : Class

<http://creativecommons.org/licenses/by-nc-sa/2.5/>

dependency



Page 10 of 10



0.



<http://creativecommons.org/licenses/by-nc-sa/2.5/>

UML

Diagrammtypen



Strukturdiagramme

- **Klassendiagramm**
- **Objektdiagramm**
- Verteilungsdiagramm
- Komponentendiagramm
- ...

statisch

Verhaltensdiagramme

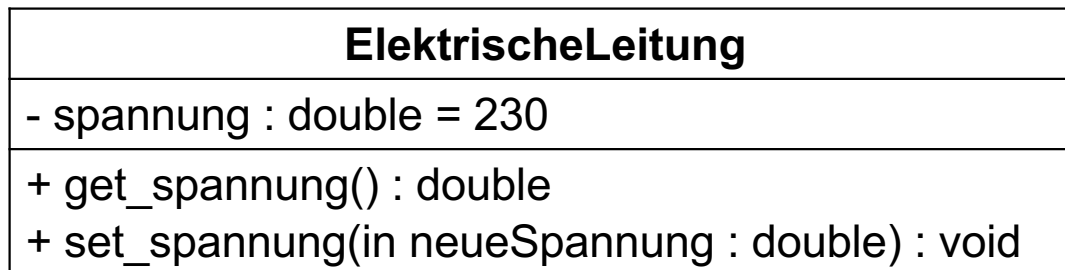
- Anwendungsfalldiagramm
- Aktivitätsdiagramm
- Zustandsdiagramm
- Interaktionsdiagramm
- Sequenzdiagramm
- ...

dynamisch

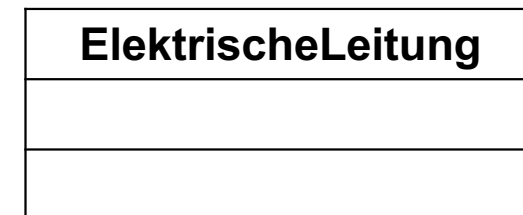
UML

Klassendiagramm

- Zentrales Konzept der UML.
- Modelliert die statischen Elemente von objektorientierter Software.
- Zeigt die Beziehungen der Elemente untereinander.
- Kann das System in verschiedenen Detailgraden darstellen.



Dies ist eine detaillierte Beschreibung der Klasse ElektrischeLeitung



Dies ist die einfachste Art die Klasse in UML darzustellen

UML

Objektdiagramm

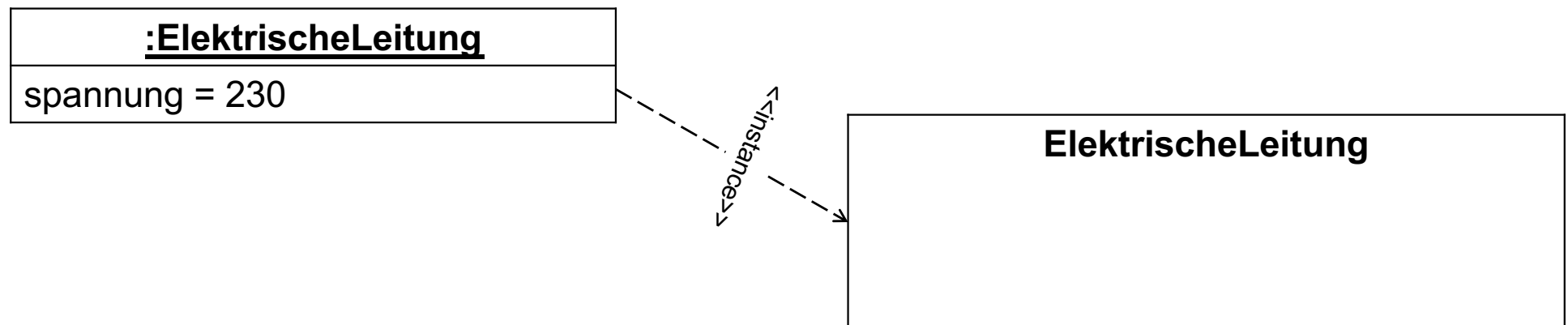
- Zeigt den aktuellen Zustand eines objektorientierten Systems.
- Verdeutlicht Beziehungen zwischen Objekten.



UML

Objektdiagramm

- Zeigt den aktuellen Zustand eines objektorientierten Systems.
- Verdeutlicht Beziehungen zwischen Objekten.
- Wird häufig parallel oder ergänzend zu einem Klassendiagramm verwendet.
- Kann auch Klassenbeschreibungen aus Klassendiagrammen enthalten.



Objektorientierung 2

Beziehungen zwischen Objekten

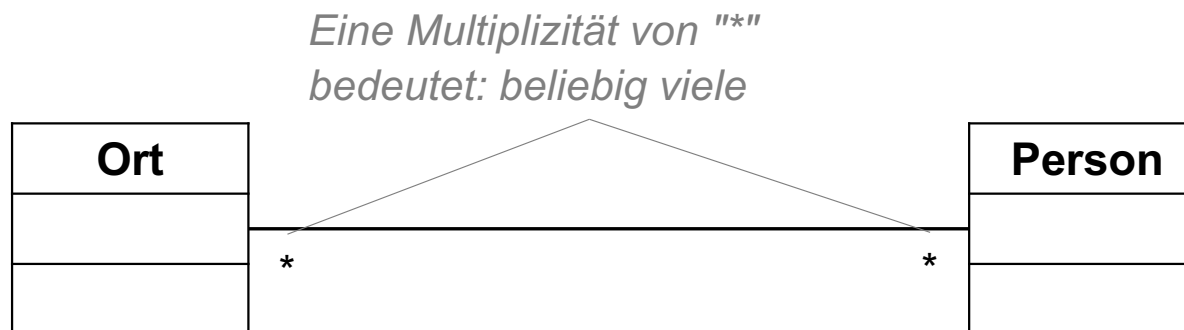


FH MÜNSTER
University of Applied Sciences

Beziehungen zwischen Objekten

Assoziation

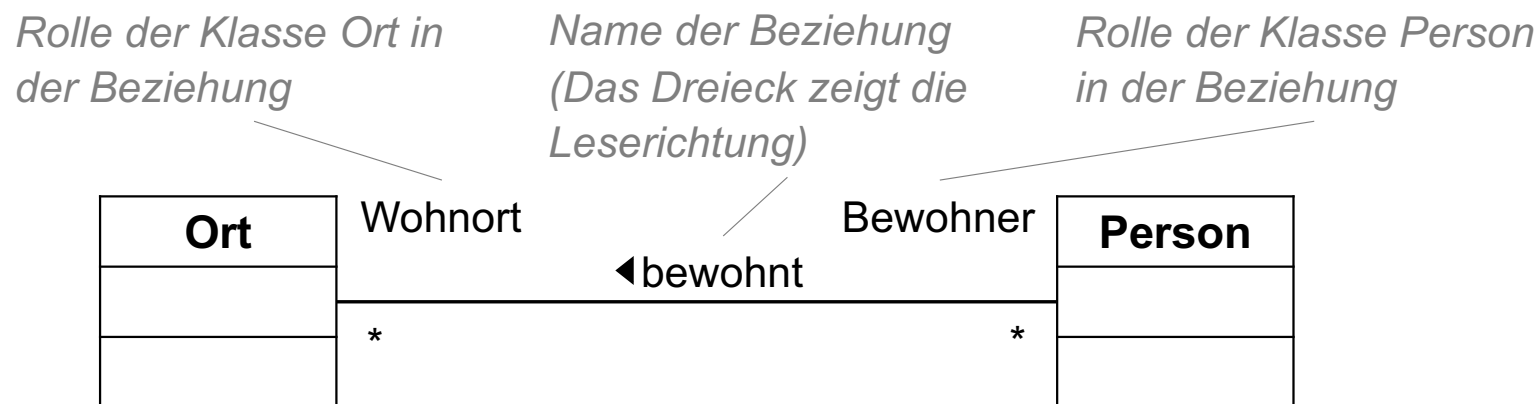
- Eine Assoziation ...**
- beschreibt ganz allgemein eine **Beziehung zwischen Objekten**.
 - also eine Beziehung, hat eine **Multiplizität**:
 - legt die Anzahl der an der Beziehung beteiligten Exemplare fest
 - kann auch als Intervall "**min .. max**" angegeben werden



Beziehungen zwischen Objekten

Assoziation

- Eine Assoziation ...**
- beschreibt ganz allgemein eine **Beziehung zwischen Objekten**.
 - also eine Beziehung, hat eine **Multiplizität**:
 - legt die Anzahl der an der Beziehung beteiligten Exemplare fest
 - kann auch als Intervall "**min .. max**" angegeben werden

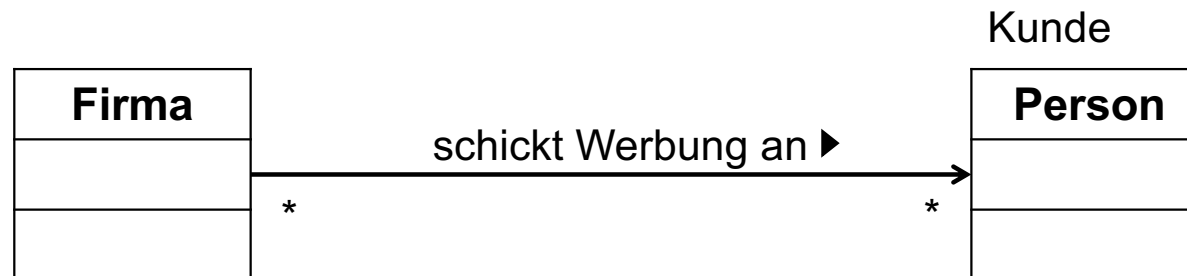


Beziehungen zwischen Objekten

Assoziation: gerichtet

Eine gerichtete Assoziation ...

- ist eine Beziehung zwischen zwei Objekten, die nur in eine Richtung geht.
- nennt man auch **navigierbar**.
- wird mit einem Pfeil \longrightarrow angegeben.



Beziehungen zwischen Objekten

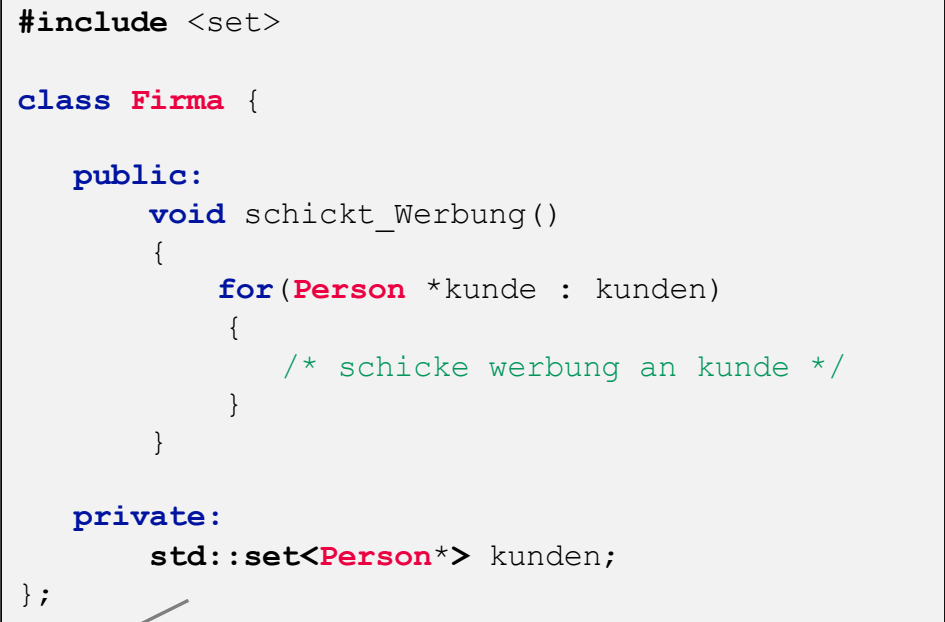
Beispiel: Eine gerichtete Assoziation in C++

```
#include <set>

class Firma {

    public:
        void schickt_Werbung()
        {
            for(Person *kunde : kunden)
            {
                /* schicke werbung an kunde */
            }
        }


    private:
        std::set<Person*> kunden;
};
```



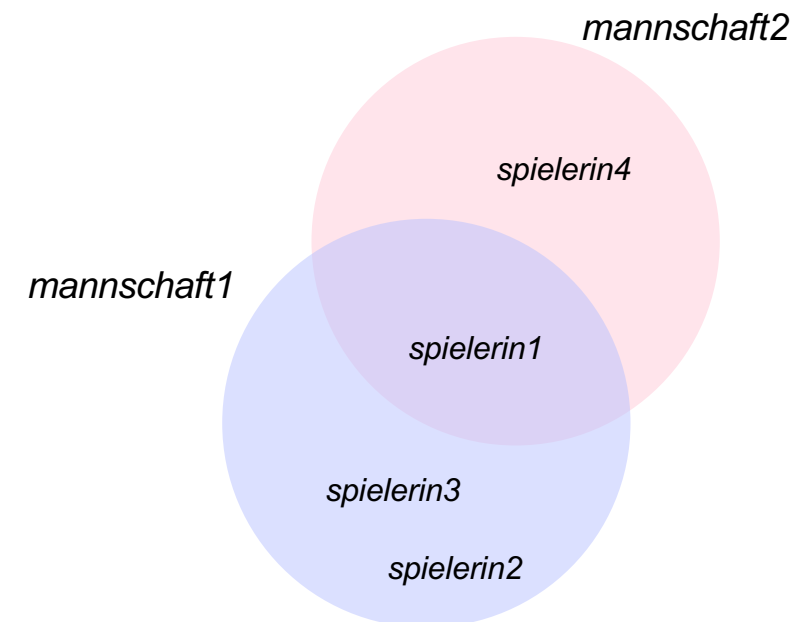
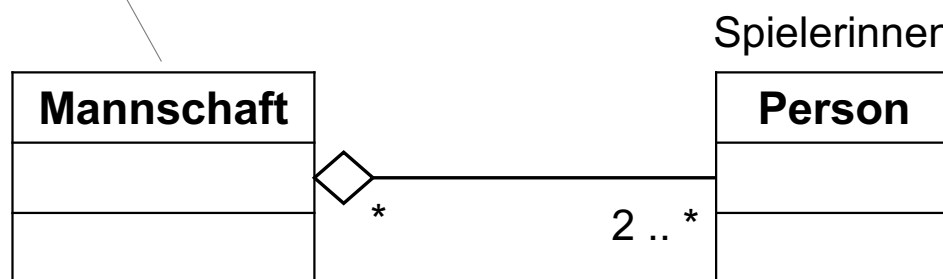
`std::set<Typ>` ist eine Klasse aus der C++ Standardbibliothek und erzeugt einen Datenbehälter (Container) mit Objekten vom Typ **Typ** (Hier: **Person***). Jedes Objekt ist einzigartig, wird also genau einmal in `set` gespeichert. `std::set` wird durch die Headerdatei `<set>` im Programmcode bekannt gemacht.

Beziehungen zwischen Objekten

Assoziation: Aggregation


- Eine Aggregation ...**
- ist eine spezielle Form der Assoziation.
 - liegt vor, wenn ein Objekt Teil von **mehreren zusammengesetzten** Objekten (Aggregate) sein kann.
 - wird mit  gekennzeichnet.

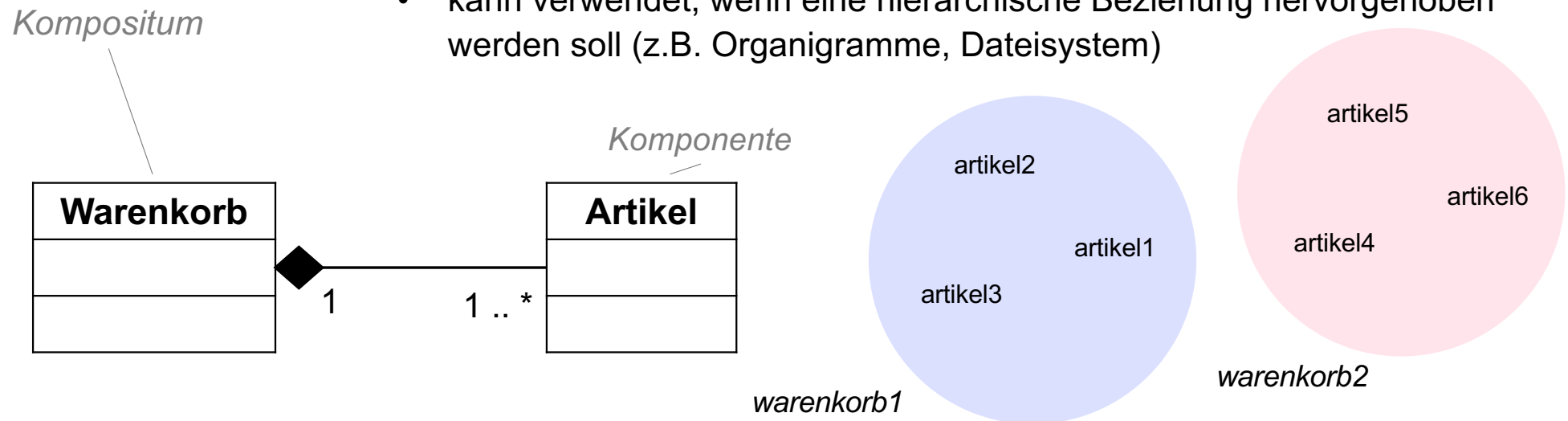
Aggregat



Beziehungen zwischen Objekten

Assoziation: Komposition

- Eine Komposition ...**
- ist eine spezielle Form der Aggregation.
 - liegt vor, wenn ein Objekt Teil (Komponente) von **genau einem zusammengesetzten** Objekt (Kompositum) sein kann.
 - wird mit  gekennzeichnet.
 - kann verwendet, wenn eine hierarchische Beziehung hervorgehoben werden soll (z.B. Organigramme, Dateisystem)



Beziehungen zwischen Objekten

Beispiel: Eine Komposition in C++

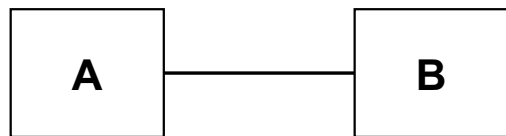
```
#include <vector>

class Warenkorb {

    public:
        void fuege_artikel_hinzu(int artikelNummer)
        {
            Artikel artikel {artikelNummer};
            artikelListe.push_back(artikel)
        }
    private:
        std::vector<Artikel> artikelListe;
};
```

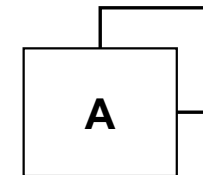
Beziehung zwischen Objekten

Assoziation: Übersicht



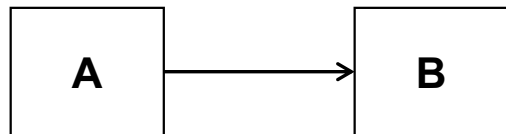
Assoziation

Navigierbarkeit in beide Richtungen
(auch durch Doppelpfeil möglich)



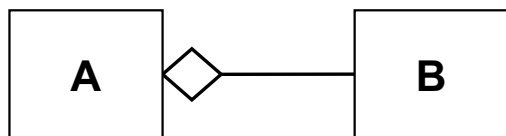
Spezialfall:

reflexive Assoziation



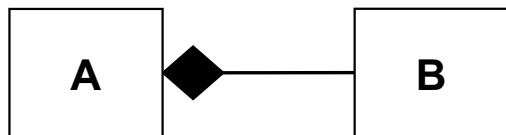
gerichtet Assoziation

Erlaubt Navigieren von A nach B, d.h. A kennt B, nicht umgekehrt



Aggregation

B ist Teil von A



Komposition

B ist teil von genau einer Instanz von A

Objektorientierung 2

Beziehungen zwischen Klassen

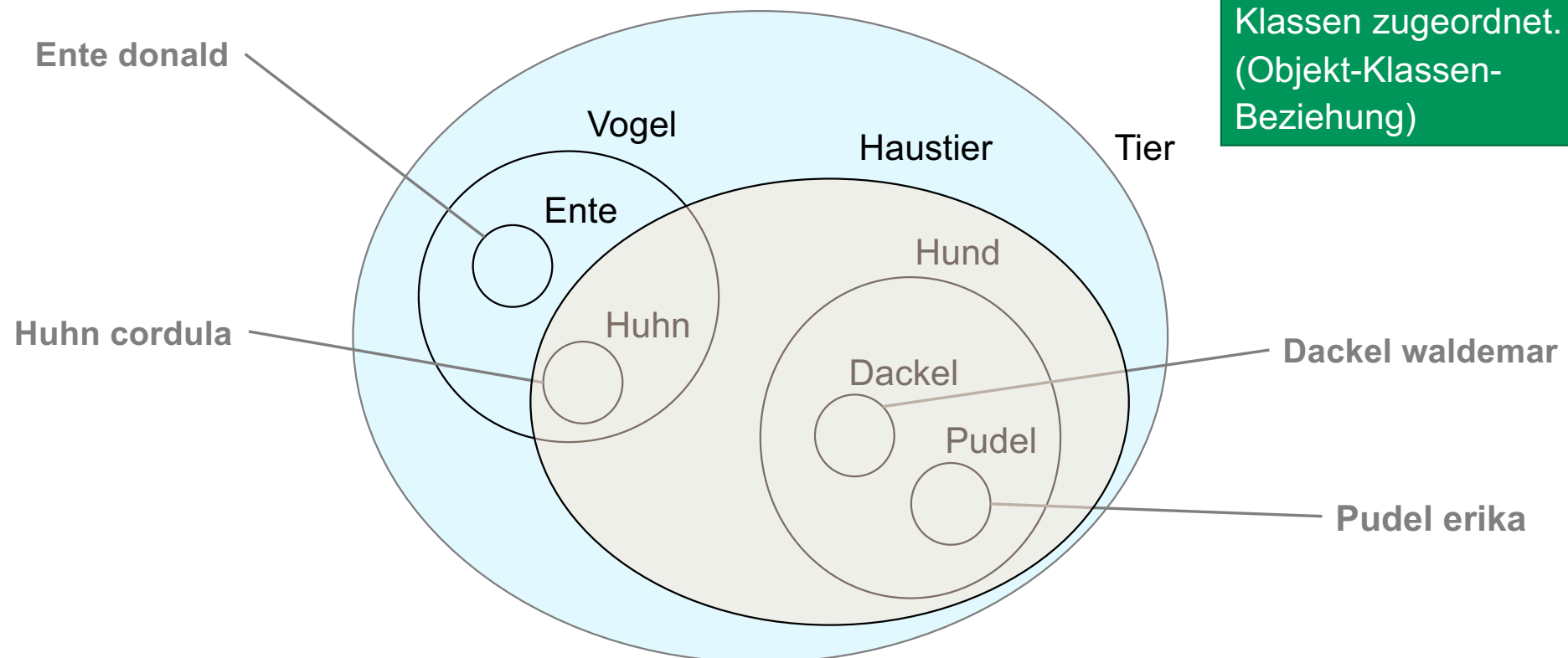


FH MÜNSTER
University of Applied Sciences

Beziehungen zwischen Klassen

Klassifizierung von Objekten

Merke: Objekte haben eine Identität. Objekte werden Klassen zugeordnet. (Objekt-Klassen-Beziehung)

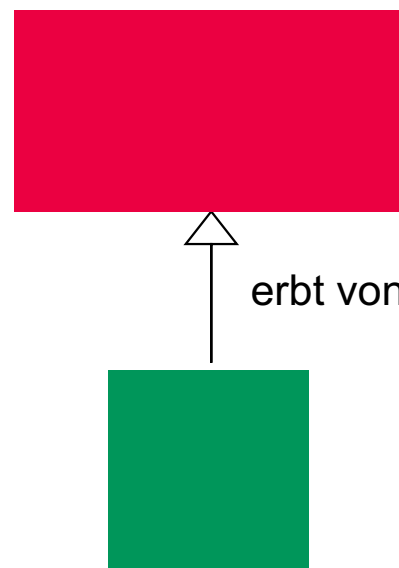


Beziehungen zwischen Klassen

Vererbung

- **Vererbung in der OOP** (Objekt Orientierten Programmierung) nicht im Sinne einer Erbfolge.
- Vielmehr ist eine Vererbung eine **Kopie** einer Klasse und eine daraus folgende **Spezialisierung**.
- Man spricht bei Vererbung auch von **Ableitung von einer Klasse**.

Beispiel:



Rechteck mit der Eigenschaft: 4 Kanten und 4 Ecken, alle Kanten stehen an den Eckpunkten im rechten Winkel zueinander. Kann beschrieben werden durch Breite und Höhe.

Quadrat ist eine Spezialisierung eines Rechtecks bei dem alle 4 Kanten gleich lang sind.

Vererbung

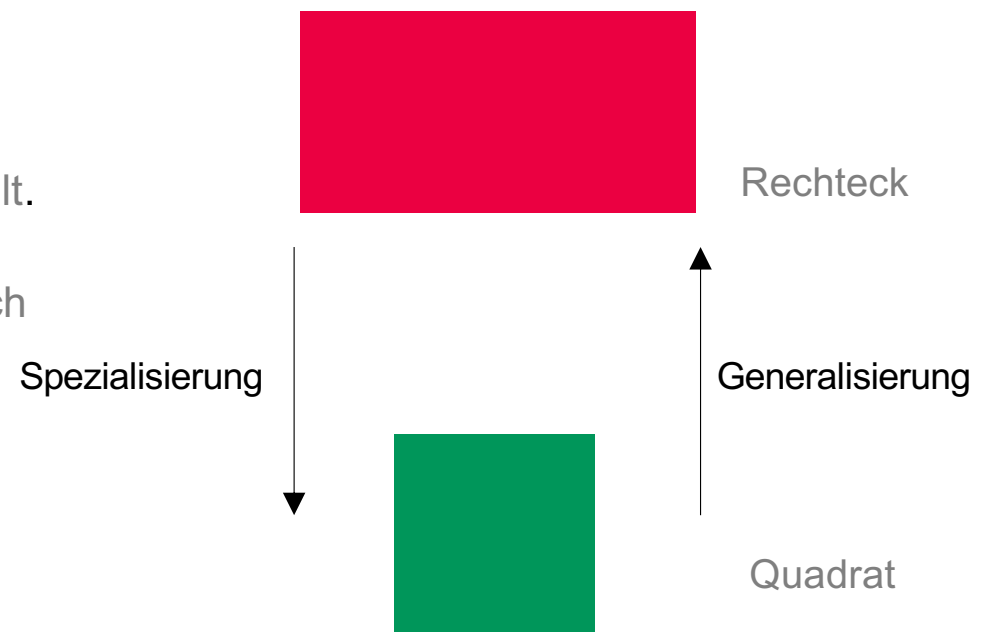
Konzepte

1. Vererbung der Spezifikation
 - Vererbt wird der grundsätzliche Aufbau einer Klasse.
2. Vererbung der Implementierung
 - Vererbt wird neben dem Aufbau auch die Implementierung.
 - Dient vor allem der Vermeidung von Codedoppelung.
3. Mehrfachvererbung

Vererbung

Unterklassen und Oberklassen

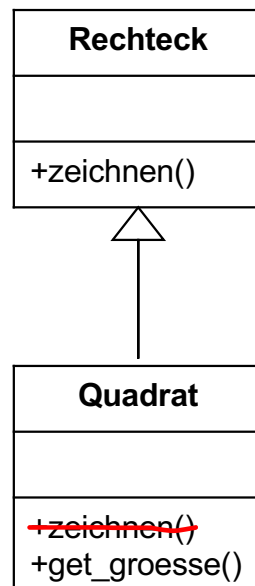
- Eine Klasse ist dann eine **Unterklasse** einer allgemeinen **Oberklasse**, wenn sie die Spezifikation der Oberklasse erfüllt, z.B. ein Quadrat ist auch ein Rechteck, da es die Definition eines Rechtecks erfüllt.
- Eine **Oberklasse** erfüllt nicht die Spezifikation der Unterklasse, z.B. ein Rechteck ist kein Quadrat (auch wenn manche Rechtecke auch Quadrate sein können).
- Ein Objekt einer **Unterklasse** kann anstelle eines Objektes der **Oberklasse** genutzt werden, da es die Spezifikation der Oberklasse erfüllt (Prinzip der Ersetzbarkeit).



Vererbung

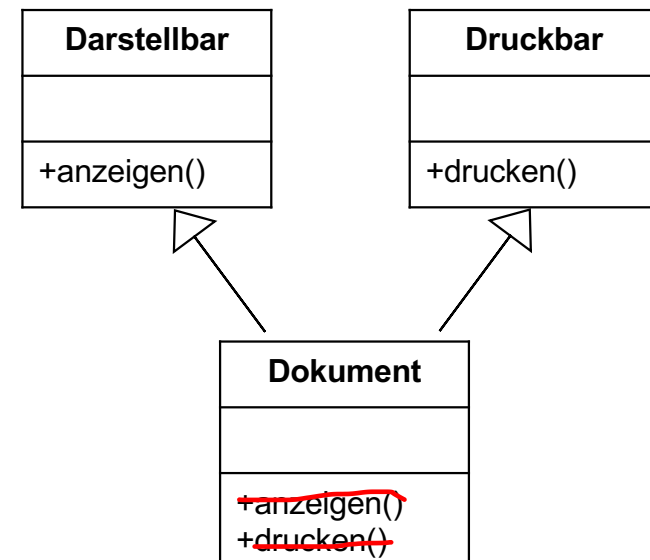
Hierarchien von Unter- und Oberklassen

Beispiel 1:



Einfachvererbung

Beispiel 2:



Mehrfachvererbung

Vererbung in C++

Schlüsselwörter in diesem Kapitel

alignas	const	dynamic_cast	long	short	typedef
alignof	constexpr	else	mutable	signed	typeid
asm	constexpr	enum	namespace	sizeof	typename
auto	constinit	explicit	new	static	union
bool	const_cast	export	noexcept	static_assert	unsigned
break	continue	extern	nullptr	static_cast	using
case	co_await	false	operator	struct	virtual
catch	co_return	float	private	switch	void
char	co_yield	for	protected	template	volatile
char8_t	decltype	friend	public	this	wchar_t
char16_t	default	goto	register	thread_local	while
char32_t	delete	if	reinterpret_cast	throw	
class	do	inline	requires	true	
concept	double	int	return	try	

Klassen ableiten in C++

- Syntax für die Ableitung einer Klasse (**Unterklasse**) von seiner Elternklasse (**Oberklasse**):

```
class Oberklasse {  
    ...  
};
```

```
class Klasse : public Oberklasse {  
    ...  
};
```

- Es ist nur ein Zugriff auf den öffentlichen Teil der Oberklasse erlaubt.
- Auf private Variablen und Methoden der Oberklasse kann aus der abgeleiteten Klasse heraus nicht zugegriffen werden (Datenkapselung).

Klassen ableiten

Beispiel: Deklaration der Ableitung

```
class Rechteck {  
    // öffentlicher Teil  
    public:  
        // Konstruktor initialisiert breite und hoehe  
        // b=Breite, h=Höhe  
        Rechteck(unsigned int b, unsigned int h);  
        // Methode zum Zeichnen des Quaders  
        void zeichnen();  
    // privater Teil  
    private:  
        unsigned int breite;  
        unsigned int hoehe;  
};
```

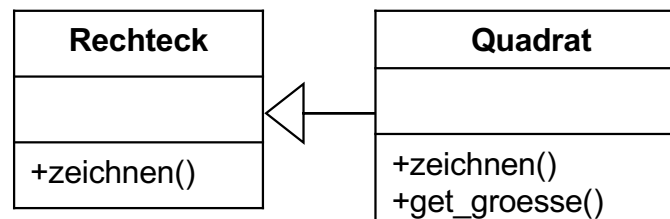
Rechteck
+zeichnen()

Klassen ableiten

Beispiel: Deklaration der Ableitung

```
class Rechteck {  
    // öffentlicher Teil  
    public:  
        // Konstruktor initialisiert breite und hoehe  
        // b=Breite, h=Höhe  
        Rechteck(unsigned int b, unsigned int h);  
        // Methode zum Zeichnen des Quaders  
        void zeichnen();  
    // privater Teil  
    private:  
        unsigned int breite;  
        unsigned int hoehe;  
};
```

```
class Quadrat : public Rechteck {  
    // öffentlicher Teil  
    public:  
        // Konstruktor initialisiert groesse  
        // g=Größe  
        Quadrat(unsigned int g);  
        // "Getter-Methode" gibt Größe des Quadrats  
        unsigned int get_groesse();  
    // privater Teil  
    private:  
        unsigned int groesse;  
};
```



Klassen ableiten

Beispiel: Implementierung der abgeleiteten Klasse

```
// quadrat.hpp
#ifndef _QUADRAT_HPP_
#define _QUADRAT_HPP_

#include "rechteck.hpp"

class Quadrat : public Rechteck {
    // öffentlicher Teil
public:
    // Konstruktor initialisiert Eckpunkte
    // g=Größe
    Quadrat(unsigned int g);
    // "Getter-Methode" gibt Größe des Quadrats
    unsigned int get_groesse();
    // privater Teil
private:
    unsigned int groesse;
};

#endif // _QUADRAT_HPP_
```

```
// quadrat.cpp
#include "quadrat.hpp"

// Implementierung des Konstruktors
Quadrat::Quadrat(unsigned int g) : Rechteck{g, g},
groesse{g} {}

// Implementierung der Methode get_groesse()
unsigned int Quadrat::get_groesse() {
    return this->groesse;
}
```

Auch hier geschweifte
Klammern { } nutzen
für die Typsicherheit!
(ab C++11)

Klassen ableiten

Konstruktor und Destruktor

- Der **Konstruktor** der **Oberklasse** wird automatisch aufgerufen, wenn ein Objekt der **abgeleiteten Klasse (Unterklasse)** erzeugt wird.
 - Bei Objekt Erzeugung erfolgt:
 1. Aufruf Konstruktor **Oberklasse**
 2. Aufruf Konstruktor **abgeleitete Klasse (Unterklasse)**
- Der **Destruktor** der **Oberklasse** wird automatisch aufgerufen, wenn ein Objekt der **abgeleiteten Klasse (Unterklasse)** im Speicher freigegeben wird.
 - Bei Objekt Freigabe erfolgt:
 1. Aufruf Destruktor **abgeleitete Klasse (Unterklasse)**
 2. Aufruf Destruktor **Oberklasse**

Klassen ableiten

Konstruktor und Destruktor

- Hat die Oberklasse einen **Konstruktor mit Parameterliste** muss dieser in der Initialisierungsliste des Konstruktors der abgeleiteten Klasse aufgerufen werden.
- Hat die Oberklasse **mehrere Konstruktoren**, muss ein geeigneter Konstruktor bei der Initialisierung gewählt werden.
- Wird **kein Konstruktor der Oberklasse** in der Initialisierungsliste aufgerufen, wird der Standardkonstruktor der **Oberklasse** aufgerufen, wenn er vorhanden ist.
- Konstruktoren können auch Default-Werte für die Parameterliste bereitstellen:

Beispiel:

```
Rechteck(unsigned int b=1, unsigned int h=1);
```

- In diesem Fall lässt sich der Konstruktor ohne Parameterliste und somit wie ein Standardkonstruktor verwenden.

Klassen ableiten

Das **protected**-Privileg

- Manchmal ist es notwendig, dass eine **Klasse auf geschützte Daten oder Methoden seiner Oberklasse** zugreifen kann.
- Diese Bereiche einer Klasse werden mit dem Schlüsselwort **protected** gekennzeichnet.

```
class Oberklasse {  
    ...  
  
    // geschuetzter Teil  
    protected:  
        double geschuetzte_variable;  
        void geschuetzte_funktion();  
  
    ...  
};
```

```
class Klasse : public Oberklasse {  
    ...  
    void foo();  
    ...  
};
```

Aufruf von geschützten Daten und Methoden der Oberklasse aus Unterklasse heraus:

```
Klasse::foo() {  
    this->geschuetzte_variable = 1.3;  
    this->geschuetzte_funktion();  
}
```

Klassen ableiten

Zugriffsprivilegien bei Ableitungen

```
class Klasse : public Oberklasse {  
    ...  
};
```

```
class Klasse : protected Oberklasse {  
    ...  
};
```

```
class Klasse : private Oberklasse {  
    ...  
};
```

Klassen ableiten

Zugriffsprivilegien

```
class Klasse : public Oberklasse {...};
```

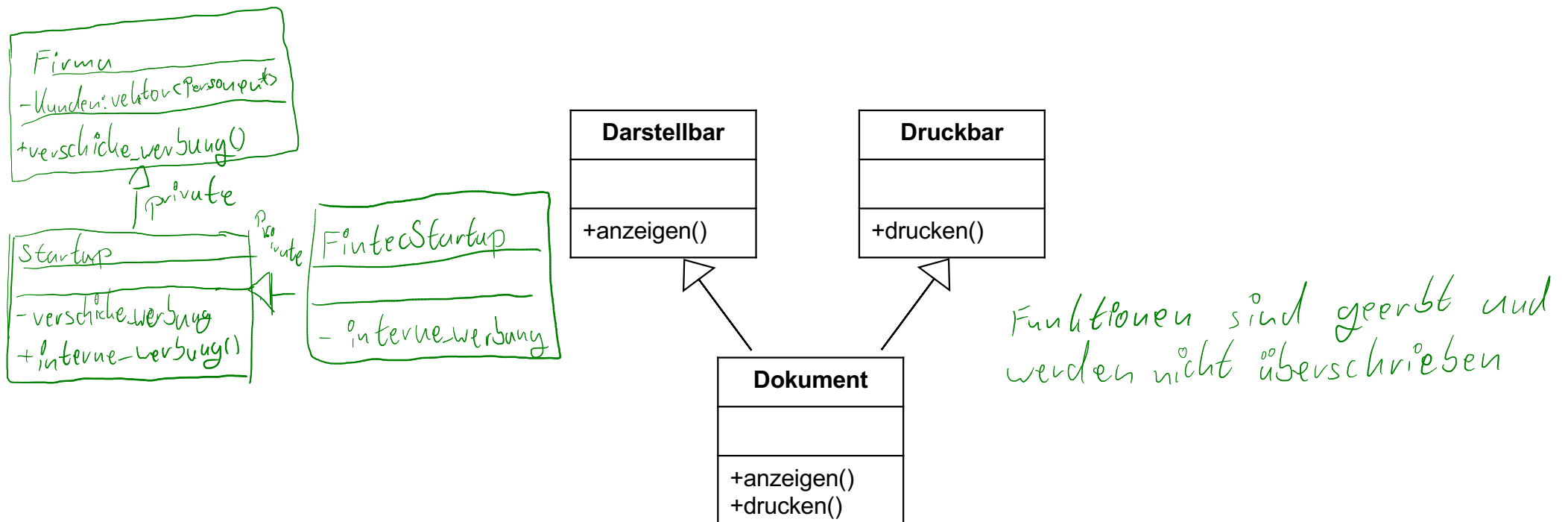
```
class Klasse : protected Oberklasse {...};
```

```
class Klasse : private Oberklasse {...};
```

Ableitung	Privileg der Basisklasse	Privileg der abgeleiteten Klasse
public +	public protected private	public protected <i>kein Zugriff</i>
protected #	public protected private	protected protected <i>kein Zugriff</i>
private -	public protected private	private private <i>kein Zugriff</i>

Mehrfachvererbung

Beispiel



Mehrfachvererbung

Syntax

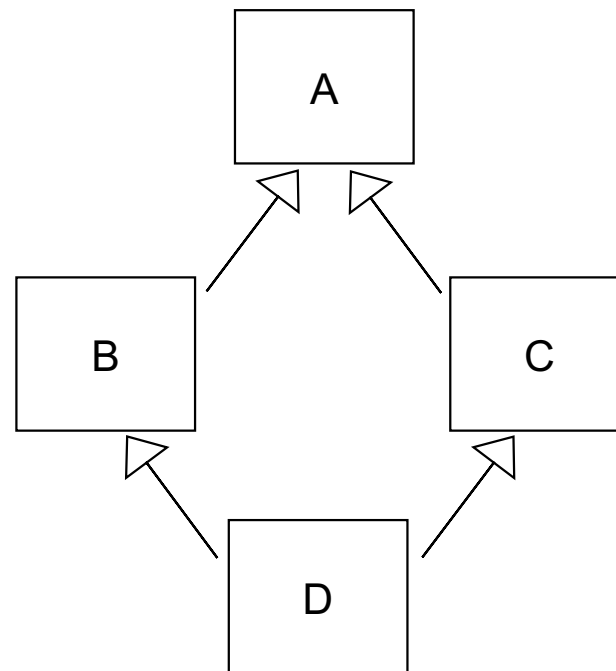
- Das Privileg **public**, lässt sich jeweils gegen die Privilegien **private** oder **protected** austauschen.

```
class Klasse : public Oberklasse1, public Oberklasse2, ...  
{  
    ...  
};
```

- Für Konstruktoren und Destruktoren gelten die gleichen Regeln, wie bei der Einfachvererbung:
 - alle Konstruktoren der Oberklassen werden in der Reihenfolge der Deklaration der Ableitung durchlaufen
 - für Destruktoren gilt die umgekehrte Reihenfolge
 - alle Konstruktoren der Oberklasse müssen im Konstruktor der Ableitung initialisiert werden, wenn es keinen jeweiligen Standardkonstruktor gibt

Mehrfachvererbung

Diamond-Problem





FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Prof. Dr. Kathrin Ungru

Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de