Informatik im Bau- und Umweltwesen 1

TECHNISCHE UNIVERSITÄT DARMSTADT

WiSe 2018/19

Skript - Kap. 4

Einführung in die Sprache C#
(Vorbereitung 1. Hörsaalübung)

Prof. Dr.-Ing. Uwe Rüppel
Meiling Shi, M.Sc.

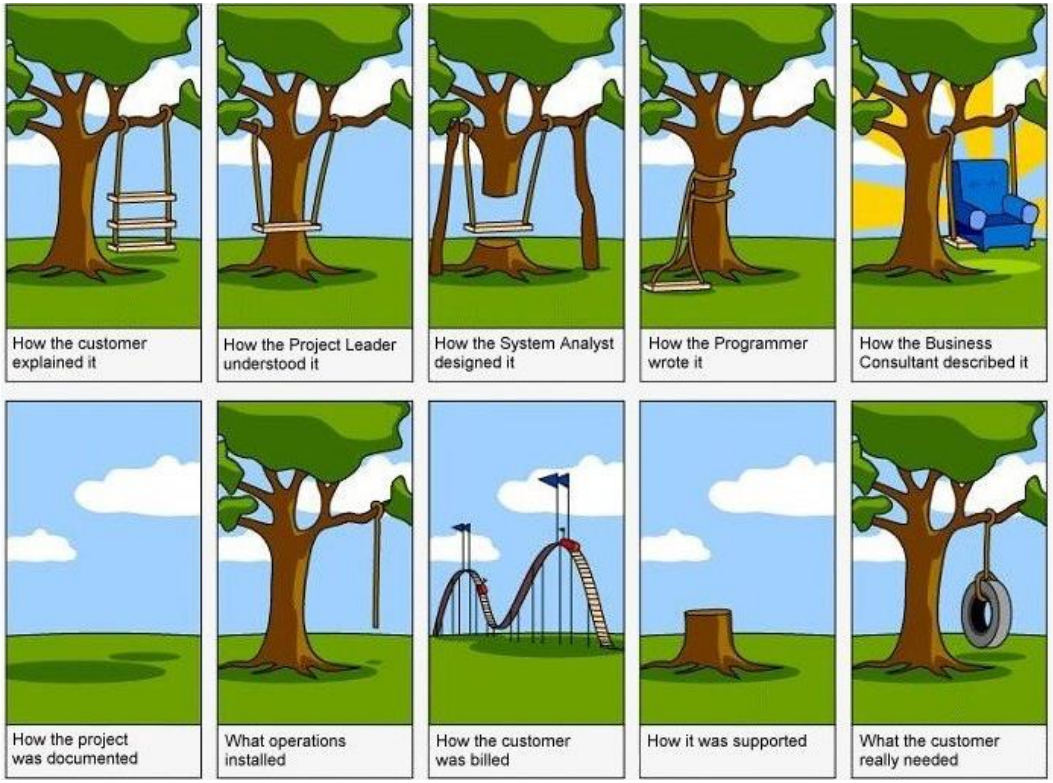
Agenda

TECHNISCHE UNIVERSITÄT DARMSTADT

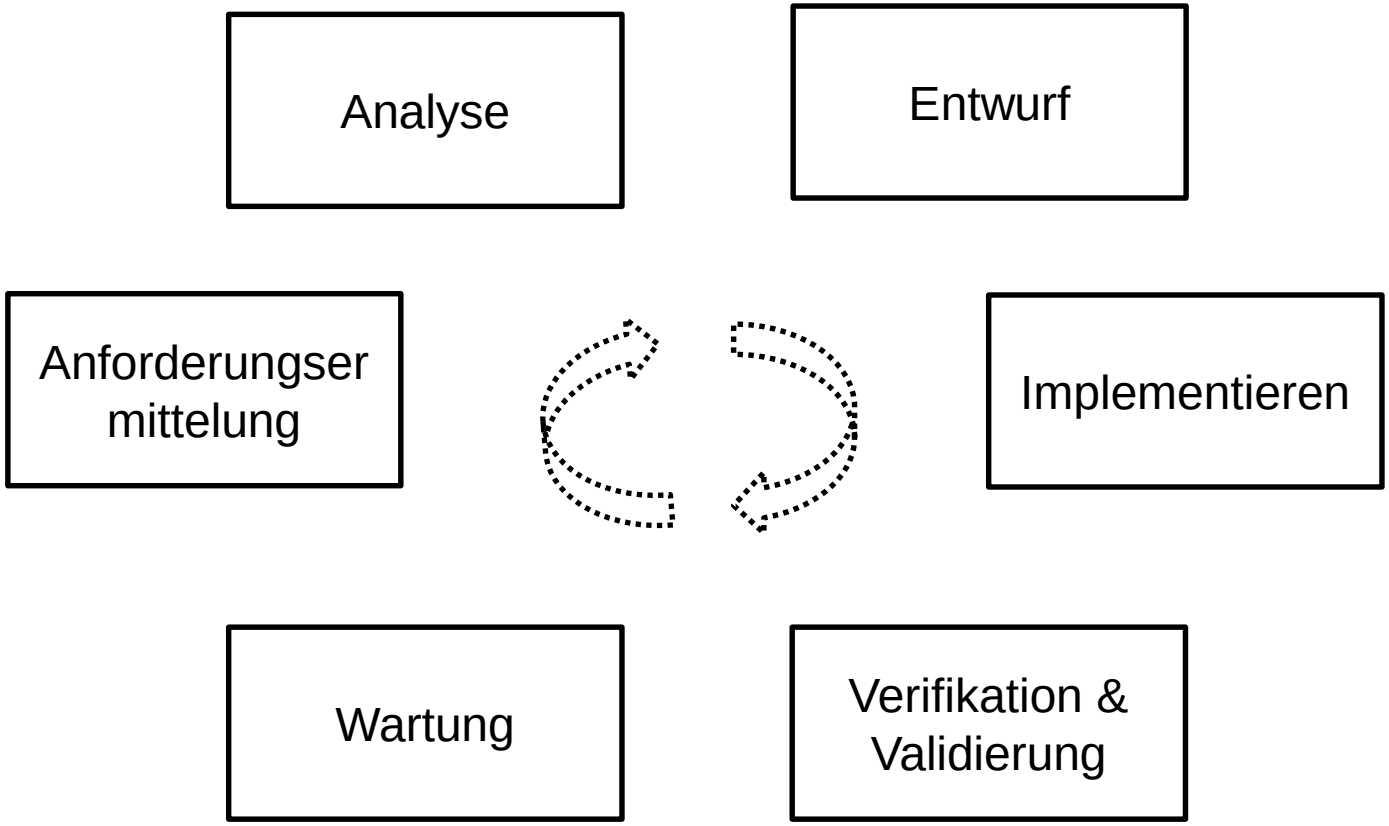
1. Requirements Engineering
2. UML
3. Recap: Objektorientiertes Programmierparadigma
4. Einführung in Visual Studio und C#
   - Klasse
   - Methode
   - Andere Befehle
5. Ausgabe Hausübung

1. Requirements Engineering

TECHNISCHE UNIVERSITÄT DARMSTADT



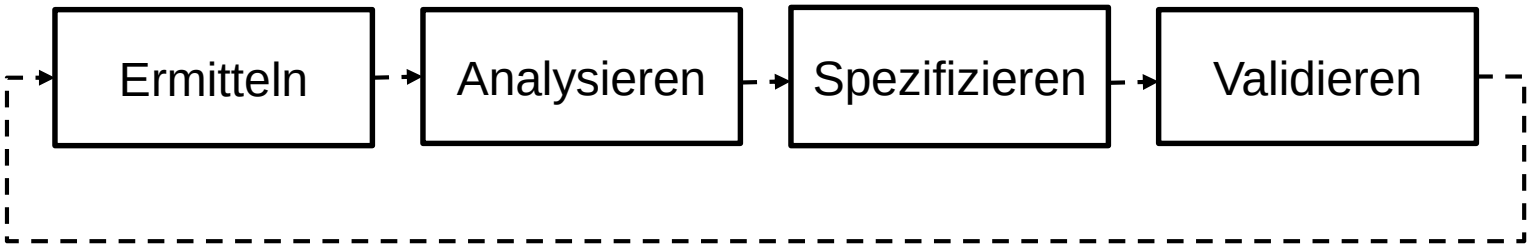
Recap: Aktivitäten zur Software Entwicklung



Definition

Requirements Engineering ist die Domäne, die alle Aktivitäten umfasst, Anforderungen zu entwickeln und im Folgendem die Anforderungen durch den Entwicklungsprozess bzw. während der Lebenszeit des Systems zu managen

Quelle: van Lamsweerde, A. (2009) Requirements Engineering S.3, Chichester, John Wiley and Sons



Stufe der Anforderungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Geschäftsanforderung: Beschreiben den High-level Zweck und die Bedürfnisse, die das Produkt erfüllen soll um z.B. Umsatz zu steigern, Betriebsprozess zu optimieren, Kundenzufriedenheit zu steigern usw.
- Benutzersanforderung: Definition der Produkt Requirements aus der Sicht der Anwender.
  - Verfasst in natürliche Sprache mit verständlicher Diagrammen
- Systemanforderungen: Anforderungen, die nötig sind um User Requirements und Business Requirements zu erfüllen
  - Verfasst in natürliche Sprache mit standardisierte Diagrammen/Notation
  - UML

Anforderungstypen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionale Anforderungen: gewünschte Funktionalitäten (was soll das System tun/können) eines Systems bzw. Produkts, dessen Daten oder Verhalten.
  - Hierfür müssen meistens Code geschrieben werden
- Nicht-funktionale Anforderung: Anforderungen an die "Qualität„, in welcher die geforderte Funktionalität zu erbringen ist.
  - Betrifft das ganze System

(Nicht-functionale Anforderungen sind manchmal kritischer als einzelne functionale Anforderungen)

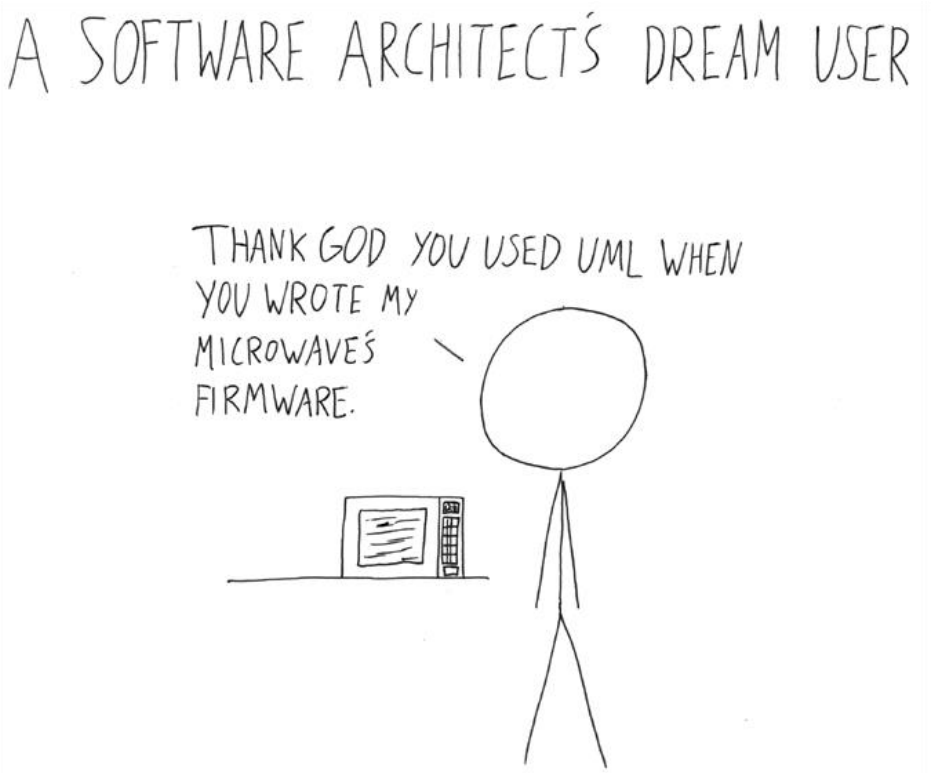
Best Practice Wording

TECHNISCHE UNIVERSITÄT DARMSTADT

- Definiere active Anforderungen
  - Soll
- Vermeiden negative und passive Anforderungen
  - Soll nicht
- Vermeinden abstrakte Adverbien
  - manchmal, immer

2. Unified Modeling Language

TECHNISCHE UNIVERSITÄT DARMSTADT



UML – Unified Modeling Language

TECHNISCHE UNIVERSITÄT DARMSTADT

- graphische Modellierungssprache zur Spezifikation, Konstruktion und Dokumentation von Teilen von Software und anderen Systemen. (ISO-Standard)

Verhaltensdiagramme der UML

- Aktivitätsdiagramm
- Anwendungsfalldiagramm
- Interaktionsübersichtsdiagramm
- Kommunikationsdiagramm
- Sequenzdiagramm
- Zeitverlaufsdiagramm
- Zustandsdiagramm

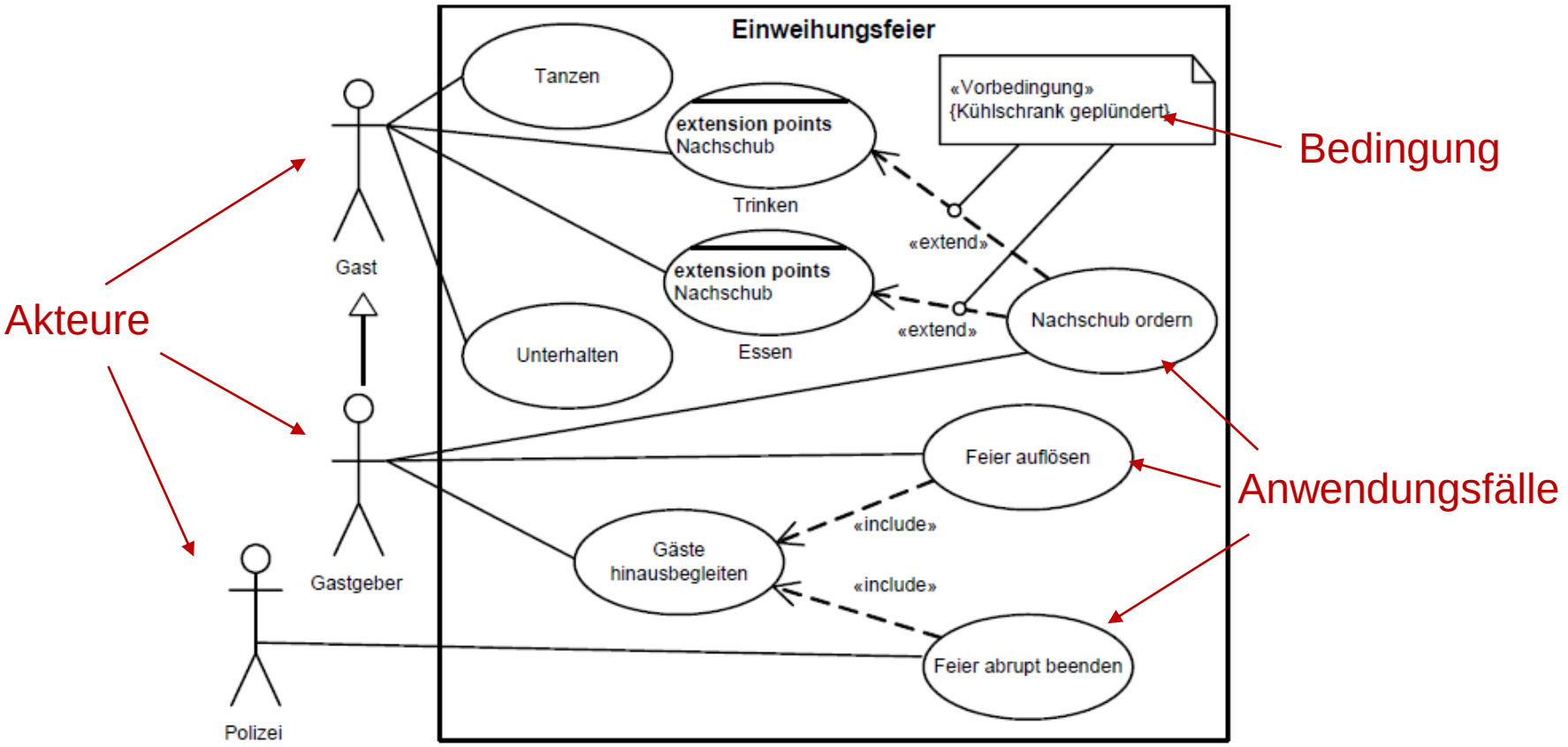
Strukturdiagramme der UML

- Klassendiagramm
- Komponentendiagramm
- Kompositionsstrukturdiagramm
- Objektdiagramm
- Paketdiagramm
- Profildiagramm
- Verteilungsdiagramm

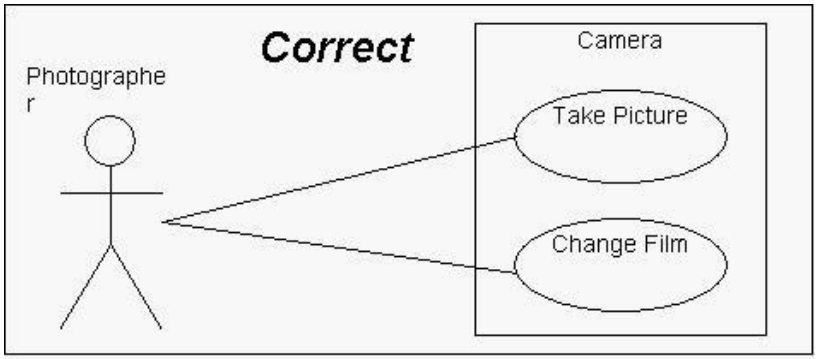
Übersicht unter → http://www.oose.de/wp-content/uploads/2012/05/UML-Notations%C3%BCbersicht-2.5.pdf

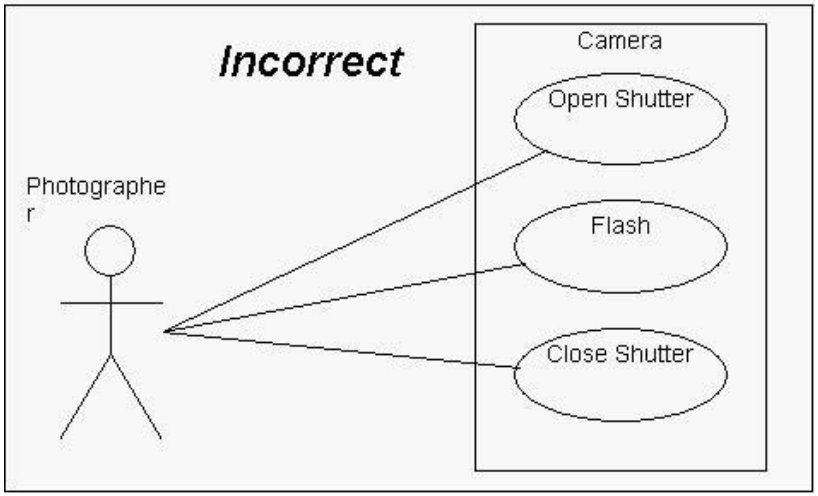
2.1 Anwendungsfall-Diagramm (Use-Case)

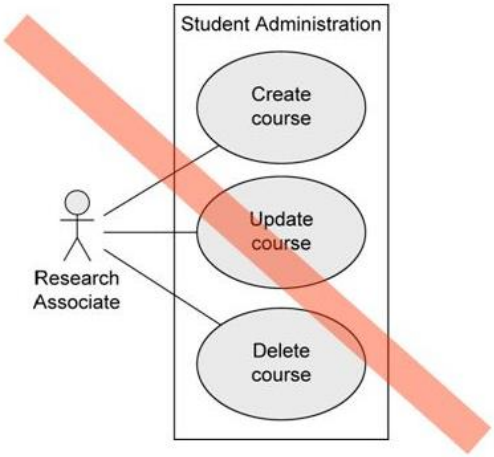
- Zu Beginn der Softwareentwicklung: Was muss das System leisten?

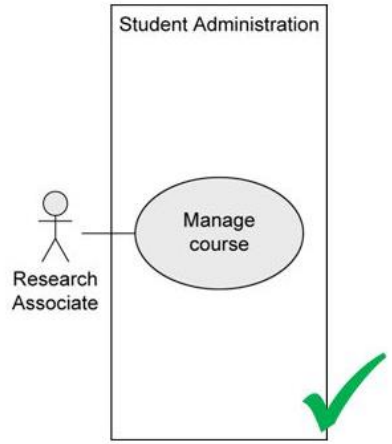


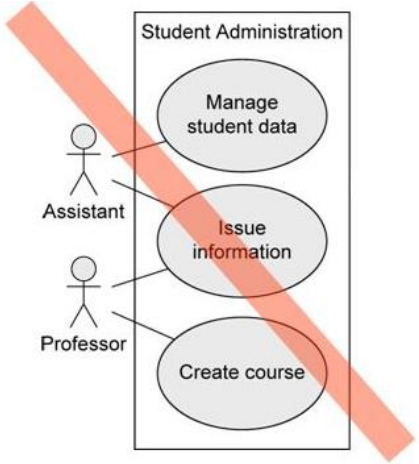
Anwendungsfall-Diagramm (Use-Case) – DON‘Ts

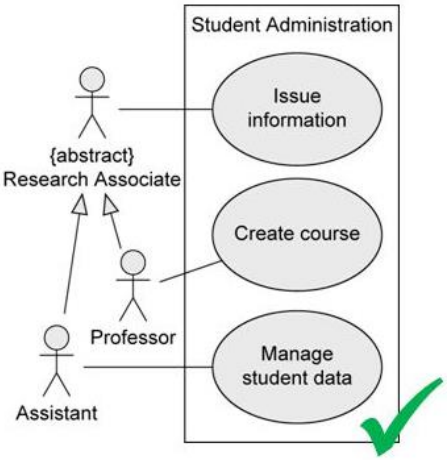






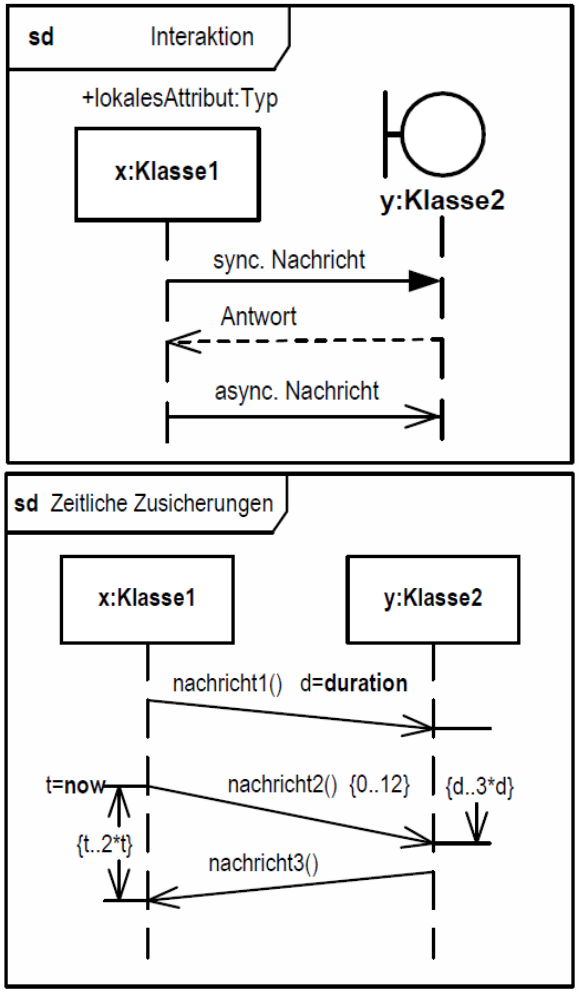




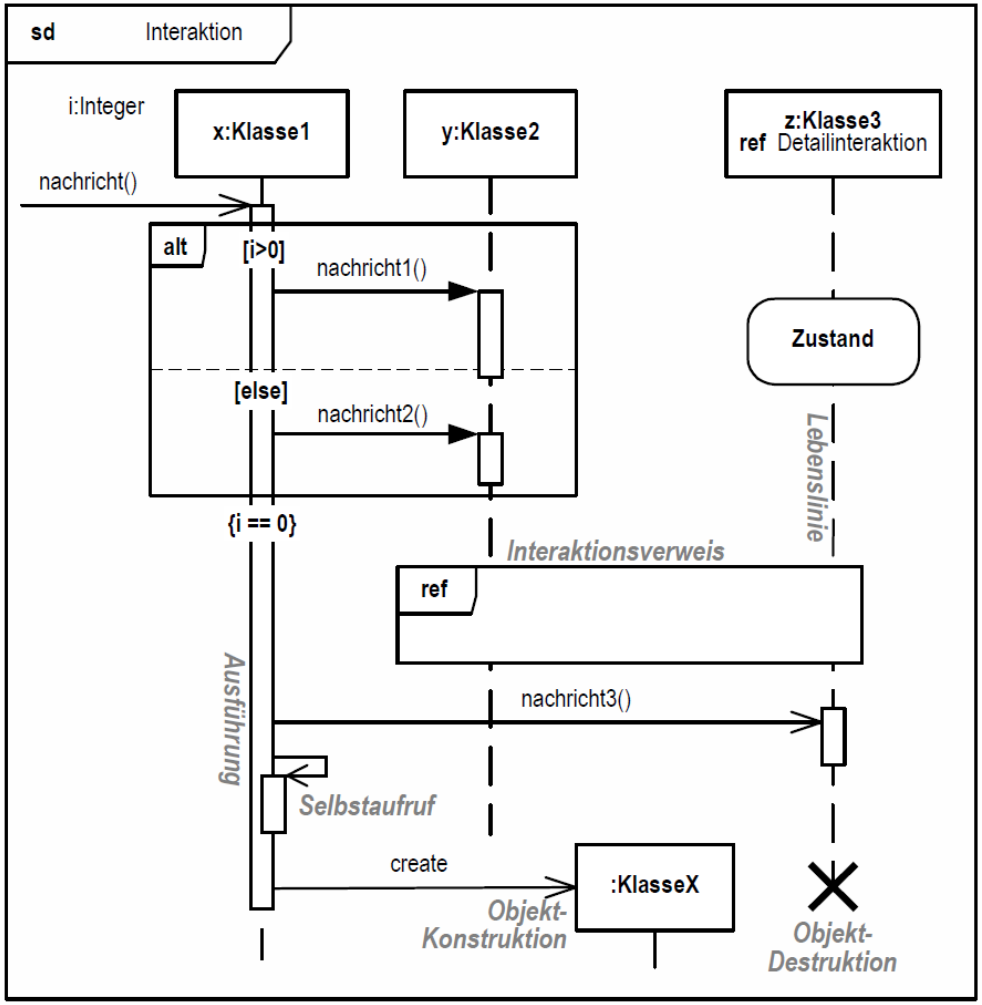


2.2 Sequenzdiagramm

- Gehört zur Gruppe der Verhaltensdiagramme, Untergruppe Interaktionsdiagramme
- Exemplarische Darstellung von Interaktionen/Nachrichtenaustausch zwischen Systemkomponenten (z.B. Klassen/Objekten)
  → Darstellung von Abläufen / Szenarien
- Es werden nicht alle möglichen Abläufe dargestellt, hierfür wären Aktivitäts-/Zustandsdiagramme besser geeignet
- In der Regel vertikale Darstellung entlang von Lebenslinien (chronologisch von oben nach unten)



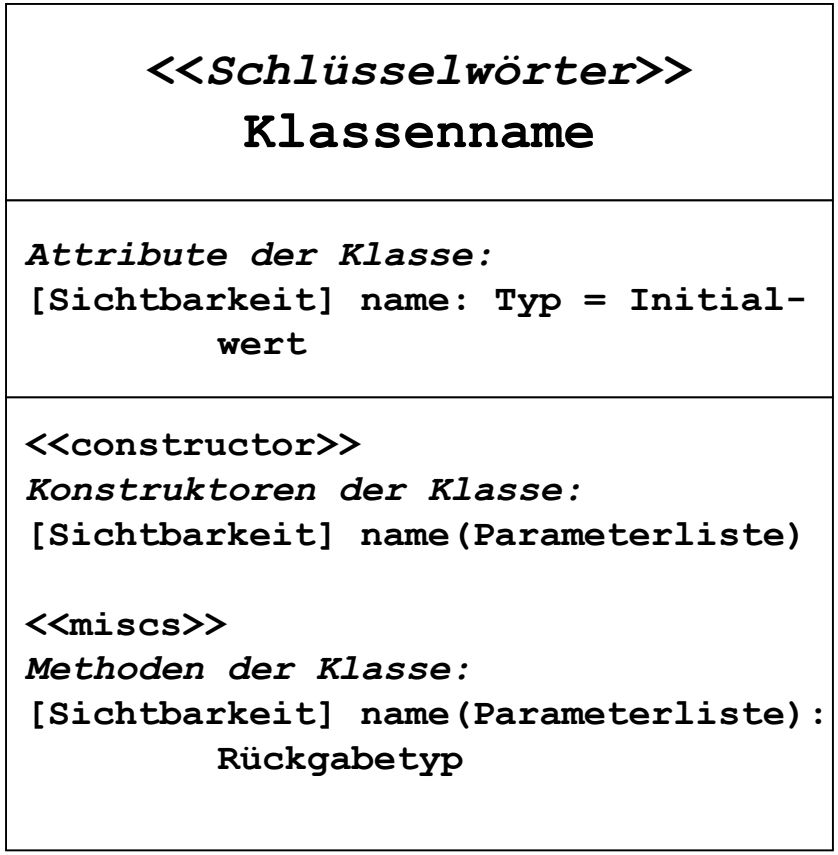
Sequenzdiagramm



2.3 UML - Klassendiagramme

TECHNISCHE UNIVERSITÄT DARMSTADT

Allgemeine Beschreibung einer Klasse:



Zeichen:

+	public
#	protected
-	private
~	internal

Kommentare:

«constructor»	Konstruktoren
«miscs»	allg. Methoden

Parameterliste:

```
name: Typ [=defaultwert]
```

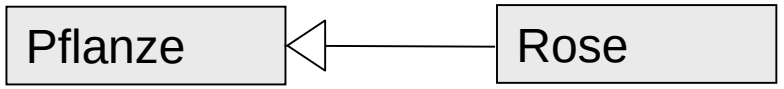
Beispiel – Klasse Kuh

TECHNISCHE UNIVERSITÄT DARMSTADT

Beziehungen zwischen Klassen

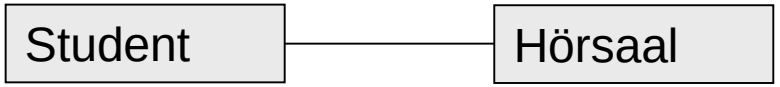
TECHNISCHE UNIVERSITÄT DARMSTADT

- **Vererbung**



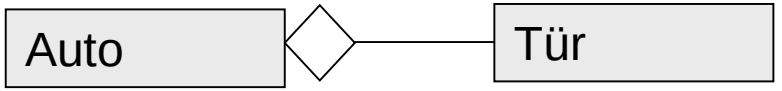
(Weitergabe von Eigenschaften und Methoden)

- **Assoziation**



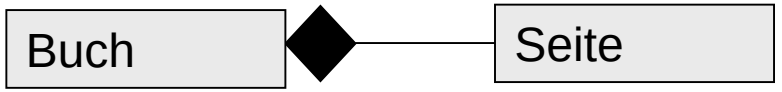
(Verbindung zwischen zwei gleichwertigen Klassen)

- **Aggregation**



(Assoziation mit Teil-Ganzes-Beziehung)

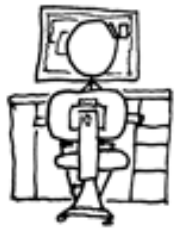
- **Komposition**



(Aggregation mit existentieller Abhängigkeit)

3. Recap: Objektorientiertes Programmierparadigma

TECHNISCHE UNIVERSITÄT DARMSTADT



Grundkonzepte der OOP (aus GDI)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bildung von **Objekten**
- **Abstraktion** der Objekte in **Klassen**

Außerdem:

- **Datenkapselung**
- **Vererbung**
- **Polymorphie**

Objekt

Schaukelstuhl
-anzahlBeine:int -sitzflaeche:double -name:String -armlehne:Lehne
+schaukeln() +istGanz():bool

Klasse

3.1 Klassen und Objekte

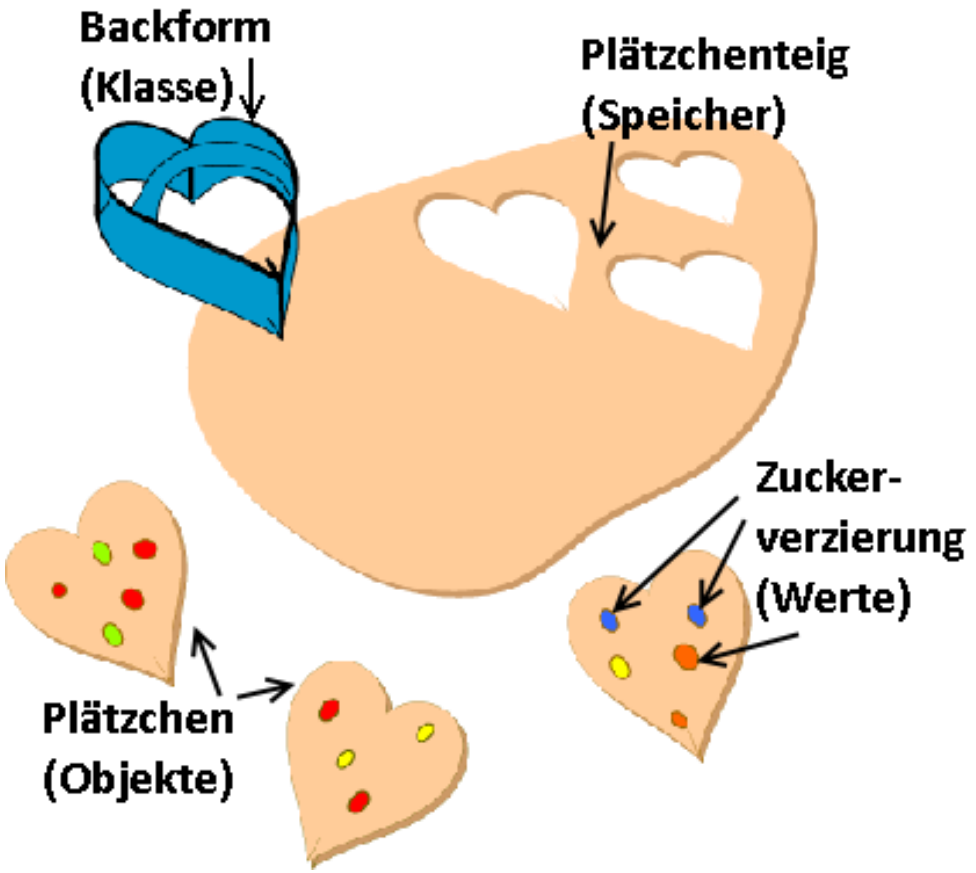
TECHNISCHE UNIVERSITÄT DARMSTADT

- **Klasse:**

  Quellcode zur Beschreibung/Definition einer Datenstruktur.

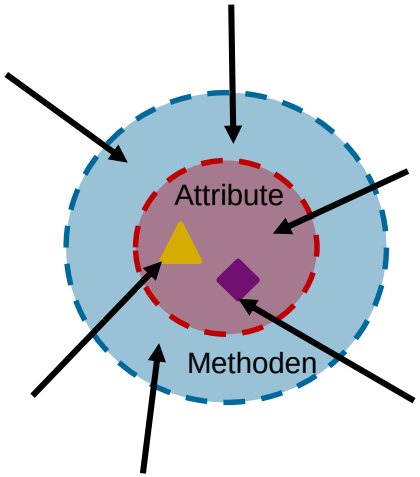
- **Objekt = Instanz einer Klasse:**

  Ein zur Laufzeit existierender Bereich des Speichers, auf den die Definition der Klasse angewendet wird.

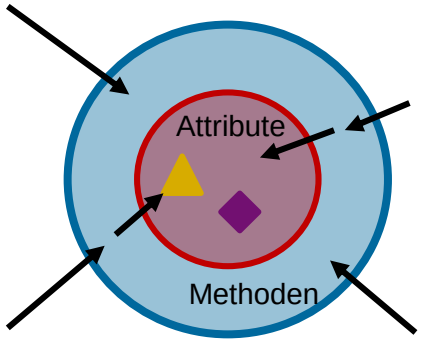


3.2 Datenkapselung

- Verwendung von Zugriffs-Modifikatoren:
  - `public`: öffentlicher Zugriff
  - `private`: Zugriff nur innerhalb der Klasse
- Zugriffs-Modifikatoren können für Attribute und Methoden verwendet werden



Öffentliche Attribute: Direkter Zugriff auf Attribute der Instanz einer Klasse möglich.

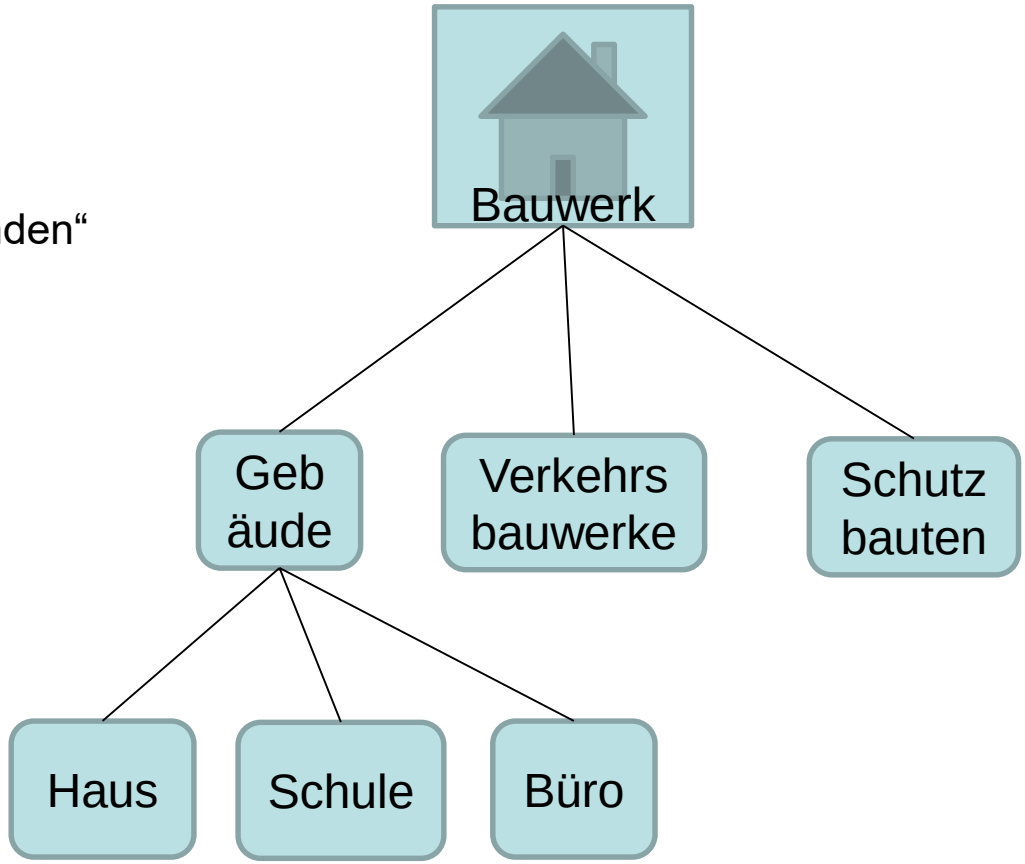


Private Attribute: Zugriff auf Attribute einer Instanz ist nur über Methoden der Klasse möglich.

3.3 Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Hierarchischer Austausch von Klasseneigenschaften
  - Attribute und Methoden der Superklasse sind auch in „erbenden“ Klassen vorhanden und können aufgerufen werden



Abstrakte Klassen und Methoden (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Von abstrakten Klassen können keine Instanzen gebildet werden.
- Besitzt eine Klasse eine abstrakte Methode ist die Klasse abstrakt.
- Erbt eine Klasse eine abstrakte Methode, muss sie diese implementieren, andernfalls ist die Klasse selbst abstrakt.
- Abstrakte Methoden haben keinen Inhalt (Methodenkörper). Auch keine geschweiften Klammern.

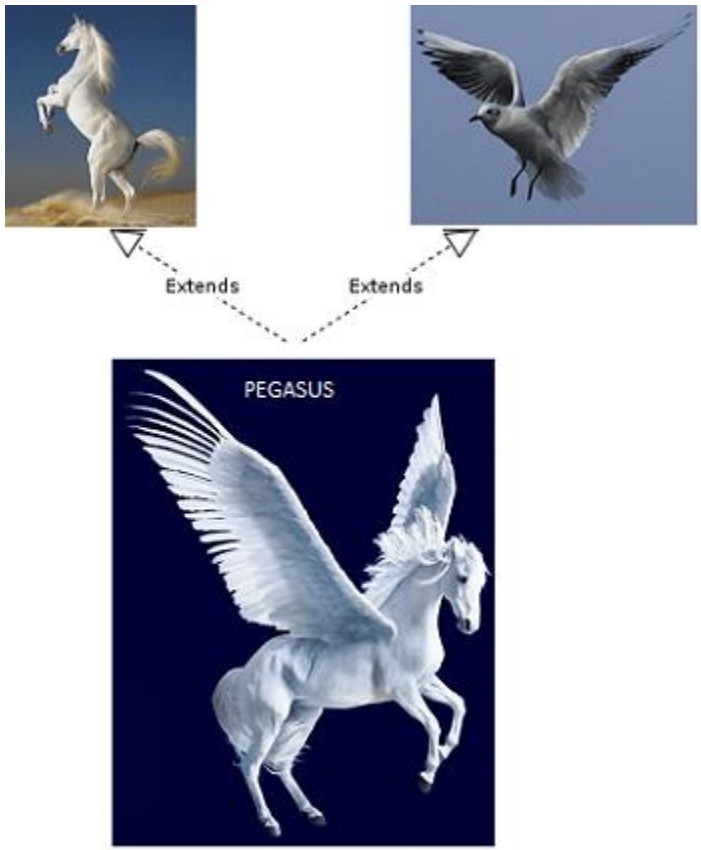
Abstrakte Klassen und Methoden (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Anwendungsfälle für abstrakte Klassen:
  - Basisklasse ist so allgemein, dass es unsinnig ist eine Instanz davon zu bilden.
  - Basisklasse ist so allgemein, dass es unsinnig ist eine ausführbare Methode zu implementieren. Aber: Man möchte für mögliche Subklassen bestimmte (zwingende) Vorgaben machen (Methoden vorgeben, die implementiert werden **müssen**).
  - Beispiel:
    Basisklasse `Saeugetier`
    → Was soll das für ein Tier sein? Hat es jemals Sinn davon eine Instanz zu bilden?
    → Ist es sinnvoll für diese Klasse eine Methode `fressen` zu implementieren?

Interfaces

- Fortführung der Idee der abstrakten Klassen
- Ein Interface enthält nur Signaturen von Methoden, Eigenschaften, Ereignissen oder Indexen.
- Eine Klasse, die ein Interface implementiert, muss alle Member des Interface implementieren.
- Eine Klasse kann mehrere Interfaces implementieren. („Mehrfachvererbung“)



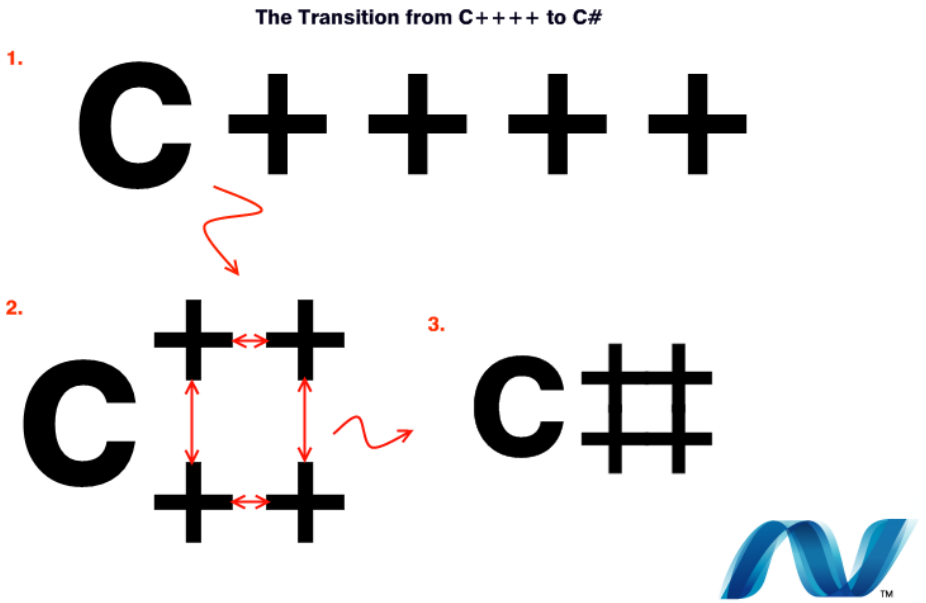
Polymorphismus und Late Binding

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Polymorphismus, [griech.] „Vielgestaltigkeit“:**
  - Der deklarierte Typ eines Objekts kann von seinem Laufzeittyp abweichen.
    (z.B. Instanz einer abgeleiteten Klasse wird wie Instanz der Basisklasse behandelt)
  - Verschiedene Objekte reagieren auf gleiche Informationen verschieden.
    (z.B. überdecken von Methoden)
- **Late Binding, „Späte Bindung“:**
  - Was ist **binding**?: Die Assoziierung von Syntax-Elementen mit logischen Programm-Elementen.
  - Was macht es **late**?: Das **binding** erfolgt (zum Teil) erst zur Laufzeit.
  - **Late Binding**: Zuordnung von Daten und Methoden zum spätmöglichsten Zeitpunkt.
    (z.B. Vererbungsstruktur → erst zur Laufzeit wird entschieden ob eine Methode aus der Basis- oder Subklasse aufgerufen wird)

4. Einführung in Visual Studio und C#

TECHNISCHE UNIVERSITÄT DARMSTADT

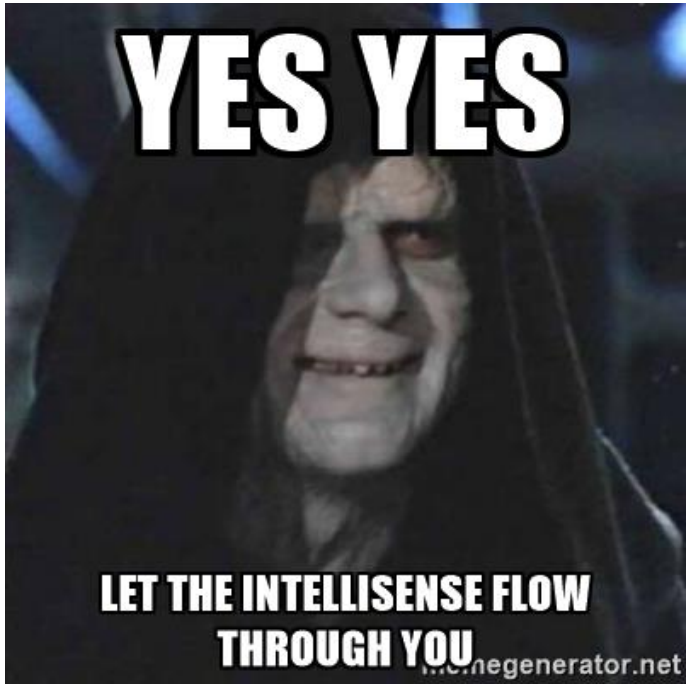


Quelle: http://blog.donnfelker.com/2008/10/28/how-did-c-get-its-name/

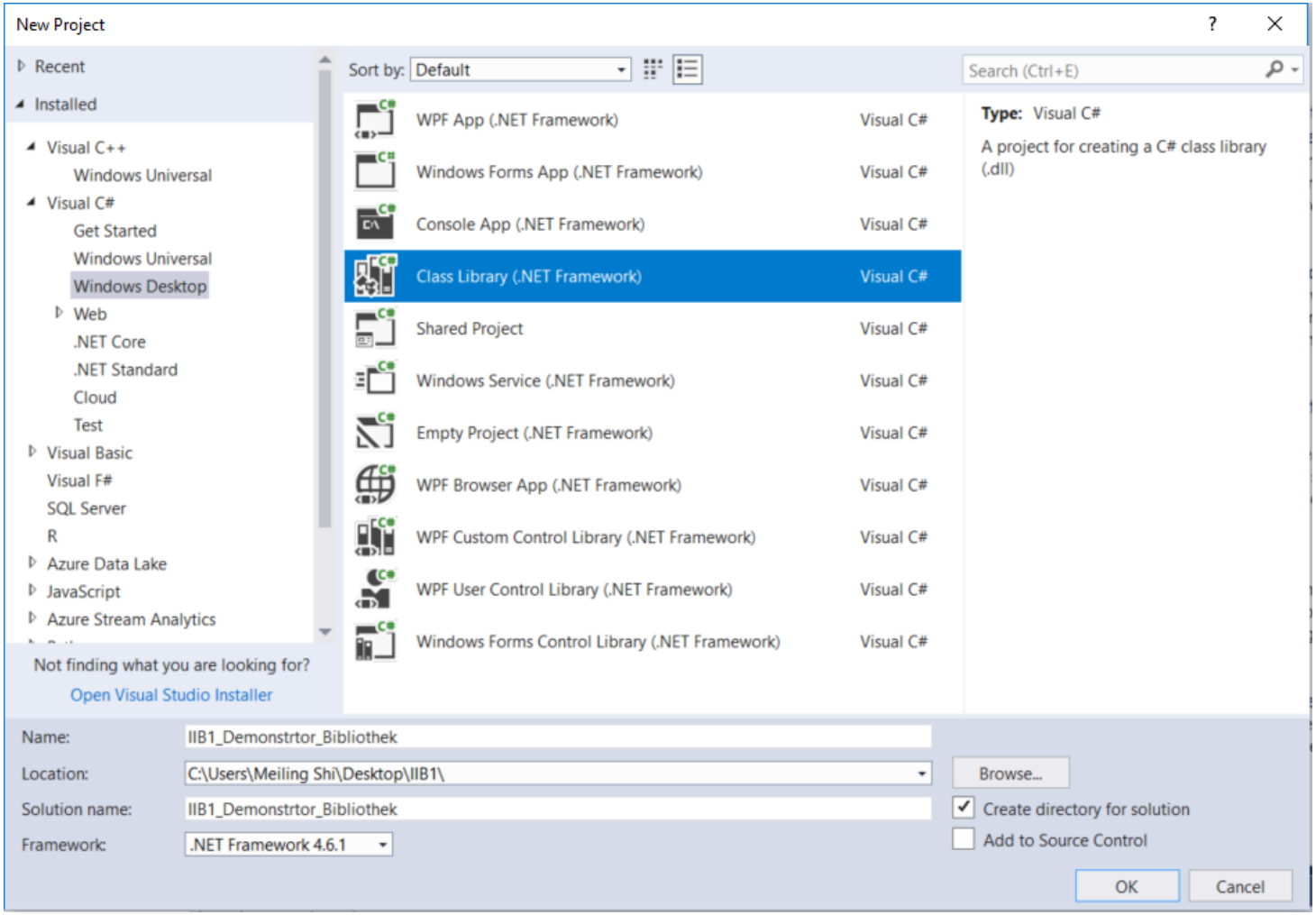
4.1 Visual Studio

TECHNISCHE UNIVERSITÄT DARMSTADT

- Integrierte Entwicklungsumgebung (**IDE**) von Microsoft
- Unterstützt VB, .NET, C, C++, C#, Python, HTML, JS, CSS…
- Bietet viele Funktionen
  - automatische Methoden- und Funktionsergänzung
  - Graphische Schnittstellen …
- .NET Framework
  - Klassenbibliothek
  - Common Language Runtime(CLR)



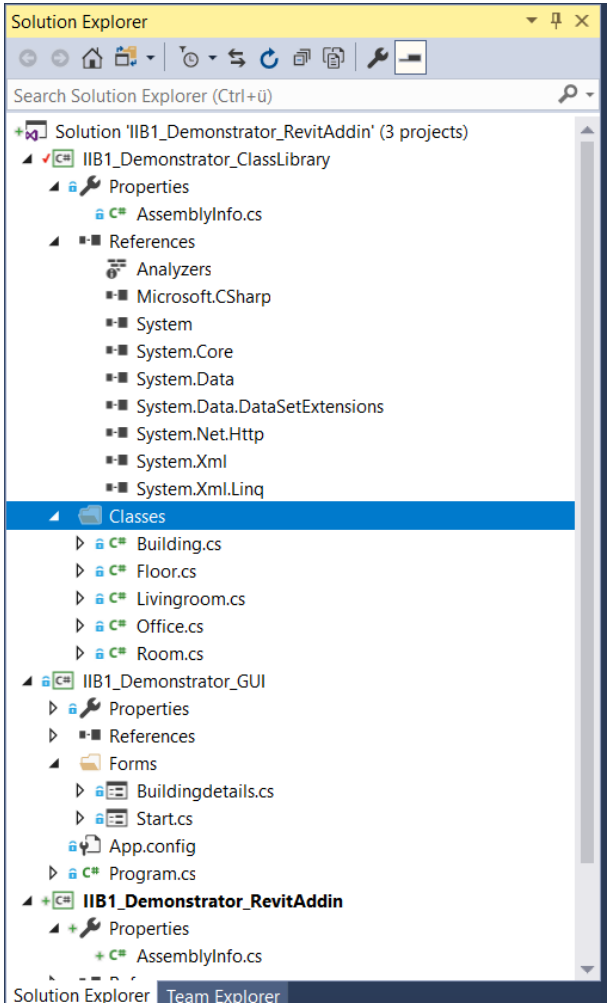
Erstellen eines neuen Projektes



Projektmappen-Explorer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Enthält Projektdaten
  - Konfigurationsdateien
  - Referenz
  - Quellcode
  - UML-(Klassen)Diagramme
  - Daten
  - Bilder…



4.2 Allgemeines zu C#

TECHNISCHE UNIVERSITÄT DARMSTADT

- Objektorientierte Programmiersprache von Microsoft
- Vorbilder: C++, Delphi, Java (Starke Ähnlichkeit zu Java!)
- Einfache Syntax
- Zeigerarithmetik nicht standardmäßig unterstützt, aber via `unsafe` Keyword möglich
- Integriert in .NET-Framework von Microsoft
- IDE: Visual Studio

Gruppen-Aufgabe

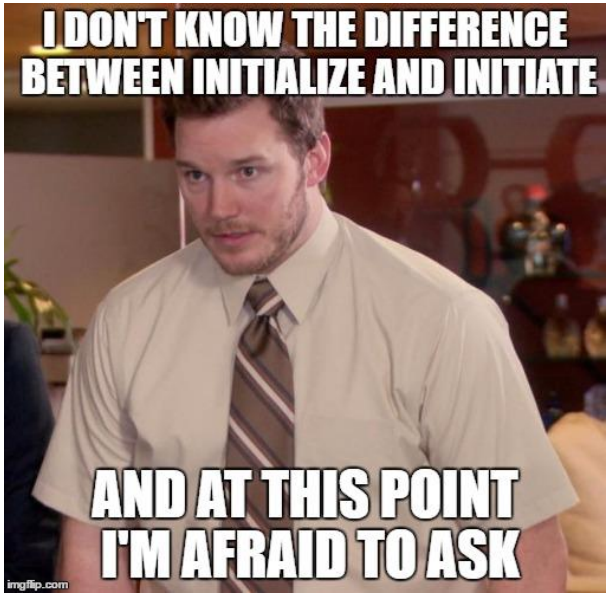
TECHNISCHE UNIVERSITÄT DARMSTADT

Was war das nochmal?

Versucht gemeinsam die Definitionen folgender Begriffe zu formulieren:

- Deklarieren
- Initialisieren
- Instanziieren
- Implementieren

Dauer: 5 Minuten



Lösung:

TECHNISCHE UNIVERSITÄT DARMSTADT

Deklarieren: Definition des Datentypes und Namens der Variable

Platz im Speicher reservieren

Bsp.:
```
int meinInteger;
Kuh meineKuh;
```

Initialisieren: Belegung der Variable mit einem Wert

Speicher mit Wert belegen

Bsp.:
```
meinInteger = 7;
```

Instanziieren: Erstellen eines neuen Objekts (Klasseninstanz)

Neues Objekt erzeugen

Bsp.: meineKuh = new Kuh(“Elsa”)

Implementieren: Umsetzen von Logik in Quellcode

Quellcode erzeugen

Einfache Datentypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterschiede zu Java!

C# Alias	.NET Klasse	Größe (Bit)	Beschreibung
bool	Boolean	8	Boolescher Wert
byte	Byte	8	Vorzeichenlos, numerisch, ganzzahlig
sbyte	Sbyte	8	Mit Vorzeichen, numerisch, ganzzahlig
char	Char	16	Unicode-Zeichen
decimal	Decimal	128	Numerisch, dezimal
double	Double	64	Numerisch, Gleitkomma
float	Single	32	Numerisch, Gleitkomma
int	Int32	32	Mit Vorzeichen, numerisch, ganzzahlig
uint	UInt32	32	Vorzeichenlos, numerisch, ganzzahlig
long	Int64	64	Mit Vorzeichen, numerisch, ganzzahlig
ulong	UInt64	64	Vorzeichenlos, numerisch, ganzzahlig
short	Int16	16	Mit Vorzeichen, numerisch, ganzzahlig
ushort	UInt16	16	Vorzeichenlos, numerisch, ganzzahlig

Komplexere Datentypen

TECHNISCHE UNIVERSITÄT DARMSTADT

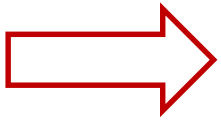
C# Alias	.NET Klasse	Beschreibung	Beispiel
enum		Gruppe benannter Konstanten (Enumeration)	`enum Pets { Dog, Cat, Zebra};` `enum Pets : byte { Dog=1, Cat, Zebra};`

object	Object	„Mutter aller Datentypen“	`object a = "the answer is";`
			`object b = 42;`
			`object c = true;`

string	String	Sequenz aus Unicode-Zeichen	`string a = "hello world";`

struct		Struktur mehrerer Variablen (auch Konstruktoren, Methoden, etc.)	`public struct Book`
			`{`
			`  public decimal price;`
			`  public string title;`
			`  public string author;`
			`}`

Die folgenden Deklarationen von a sind absolut gleichwertig:

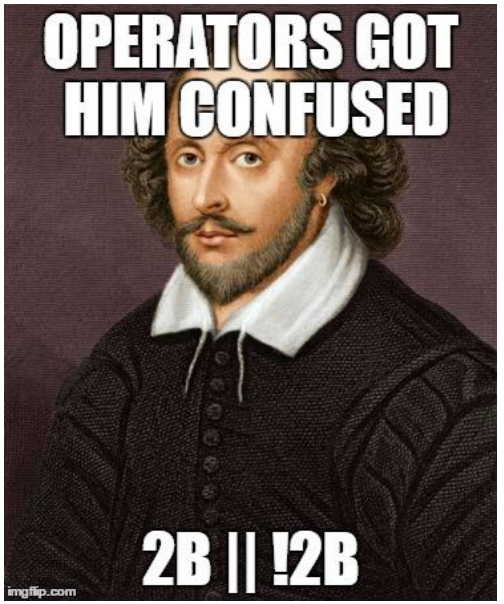


```
Int32 a;
int a;
```

Grundlegende Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Art	Beschreibung
Logische Operatoren	&& (und) || (oder) ! (nicht)
Relationale Operatoren	== (gleich) != (ungleich) <= (kleiner gleich) >= (größer gleich) > bzw. < (größer bzw. kleiner)
Arithmetische Operatoren	+ - * / %
Sonstige	++ (Inkrementierung) -- (Dekrementierung)
Shortcuts	a += b (kurz für a = a + b)



Klassen (Ordner Architektur)

TECHNISCHE UNIVERSITÄT DARMSTADT

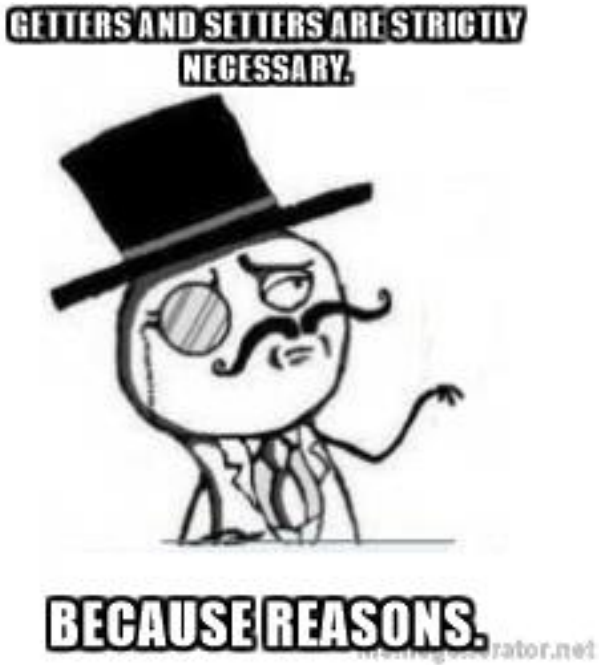
```
class Tagebuch
    {
        // Attribute (fields)
        private string dasKannJederWissen; // privat
        private string dunkleGeheimnisse; // privat
        private bool ichBinEs; // privat
        private const string passwort = "Top5ecret!?"; // private Konstante

        // Property
        public string DasKannJederWissen { get => dasKannJederWissen;
                                           set => dasKannJederWissen = value;}
        public string DunkleGeheimnisse { get => dunkleGeheimnisse;
                                          set => dunkleGeheimnisse = value; }

        // Default Konstruktor
        public Tagebuch() { }

        // Methode
        public void oeffnen(string pw)
        {
            if (pw == passwort) ichBinEs = true;
            else ichBinEs = false;
        }
    }
```

Properties (Getter und Setter)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Tagebuch
{
    private string dasKannJederWissen; // öffentlich
    private string dunkleGeheimnisse; // privat
    private bool ichBinEs; // privat
    private const string passwort = "Top5ecret!?"; // private Konstante
    // beschränkter Lese- und Schreib-Zugriff
    public string DunkleGeheimnisse {
        get { // Lesen
            if (ichBinEs) return dunkleGeheimnisse;
            else return "Elvis lebt!"
        }
        set { // Schreiben
            if (ichBinEs) dunkleGeheimnisse += value;
        }
    }
            ...
     public void oeffnen(string pw) {
        if (pw == passwort) ichBinEs = true;
        else ichBinEs = false;
    }
}
```

Unterschiede zu Java!



FYI:
Verschlüsselungen in .NET
System.Security.Cryptography.MD5
→ Klasse zur Generierung von 128-bit Hashcodes

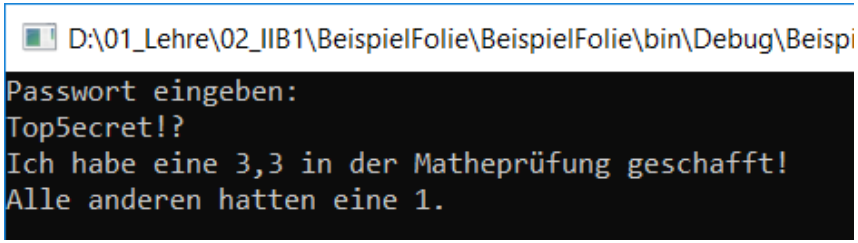
Nutzung von Properties (Getter und Setter)

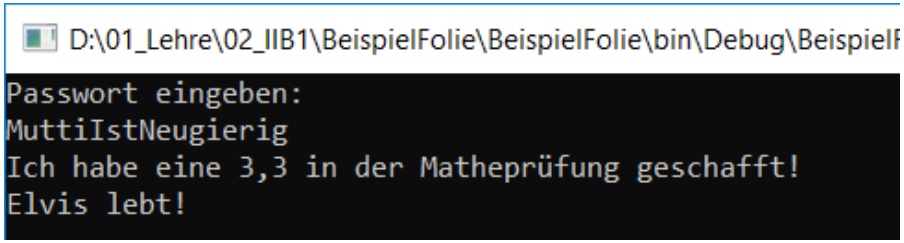
TECHNISCHE UNIVERSITÄT DARMSTADT

```
static void Main(string[] args)
{
    Tagebuch tagebuch = new Tagebuch();
    Console.WriteLine("Passwort eingeben:");
    tagebuch.oeffnen(Console.ReadLine());

    tagebuch.DasKannJederWissen = "Ich habe eine 3,3 in der Matheprüfung geschafft!";
    Console.WriteLine(tagebuch.DasKannJederWissen);

    tagebuch.DunkleGeheimnisse = "Alle anderen hatten eine 1.";
    Console.WriteLine(tagebuch.DunkleGeheimnisse);

    Console.Read();
}
```

direkter Zugriff, da öffentliches Attribut

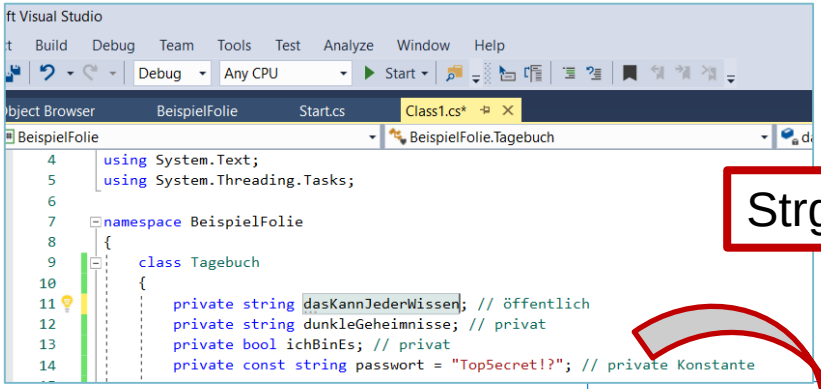
Zugriff nur über Getter/Setter, da privates Attribut



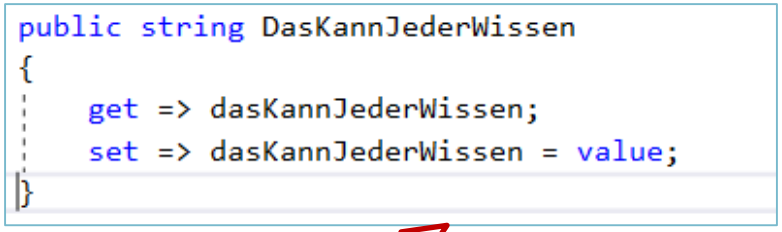


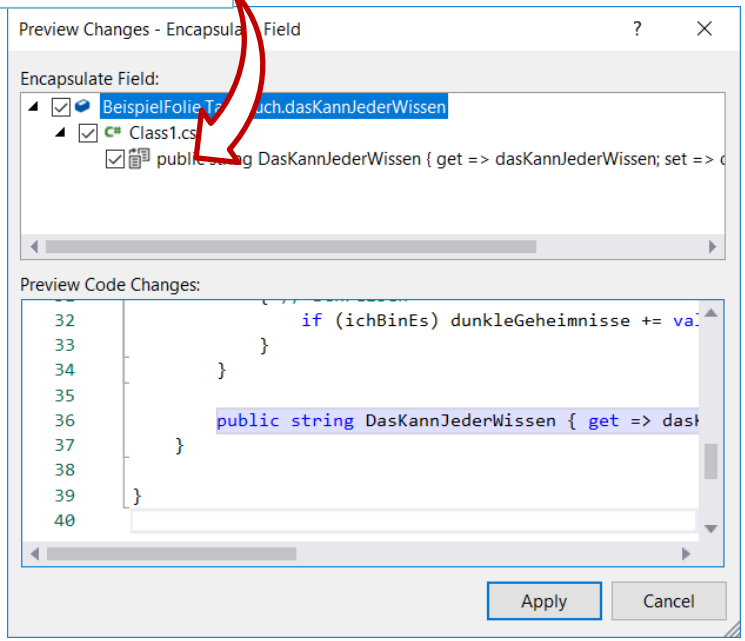
„Feld kapseln“

TECHNISCHE UNIVERSITÄT DARMSTADT



Strg +R+ E

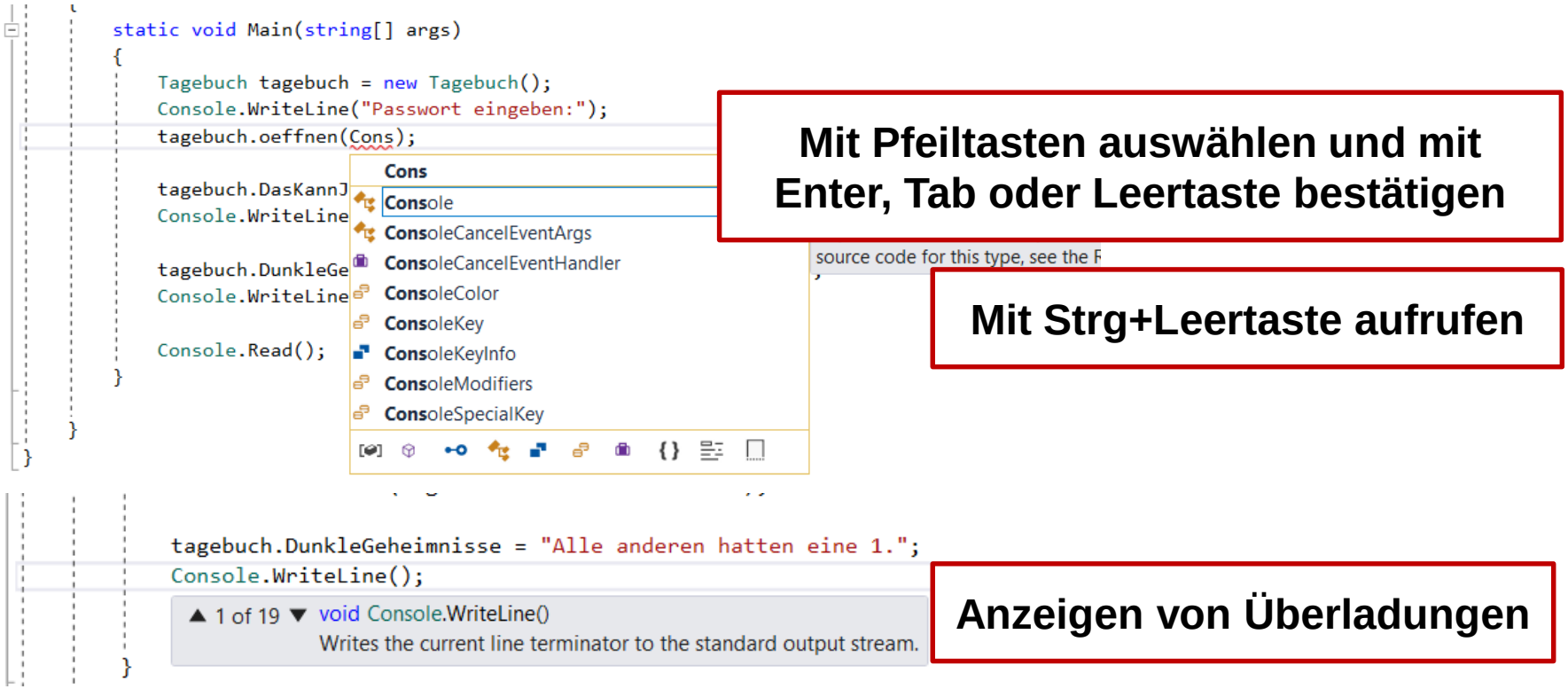




Codevervollständigung

TECHNISCHE UNIVERSITÄT DARMSTADT

- „Intellisense“ (automatische Codevervollständigung)
- Für fast alles verfügbar (auch bei Config Files)

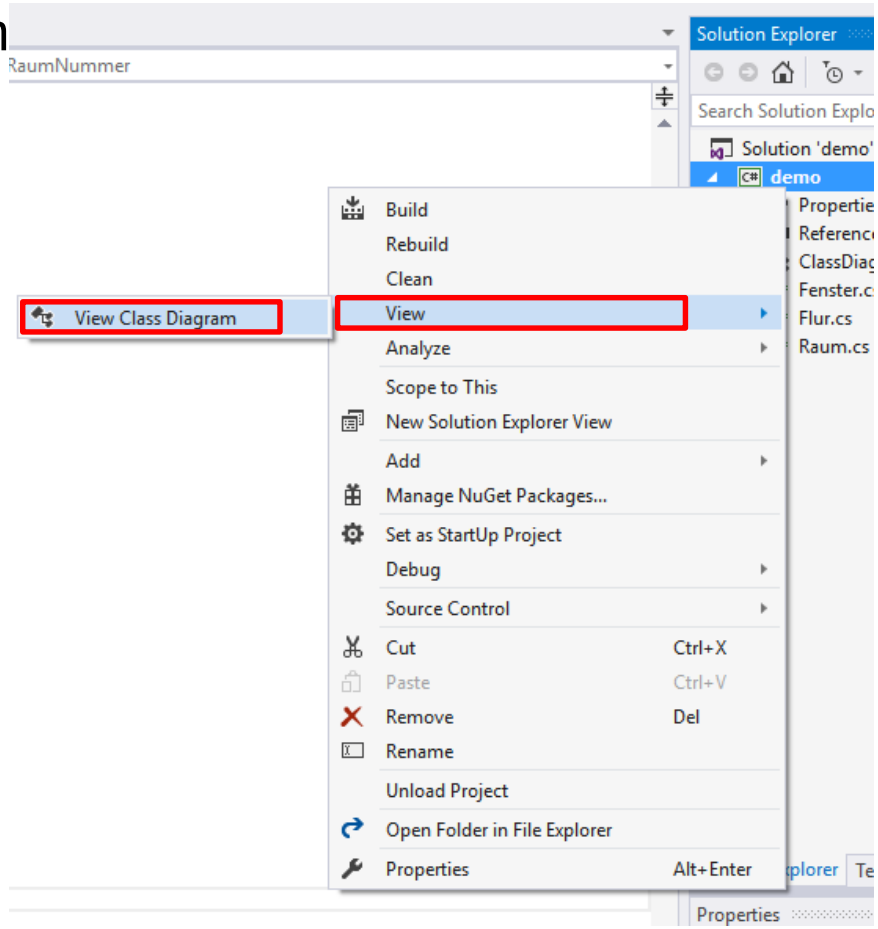


4.3 Programmierung mit Klassendiagramm

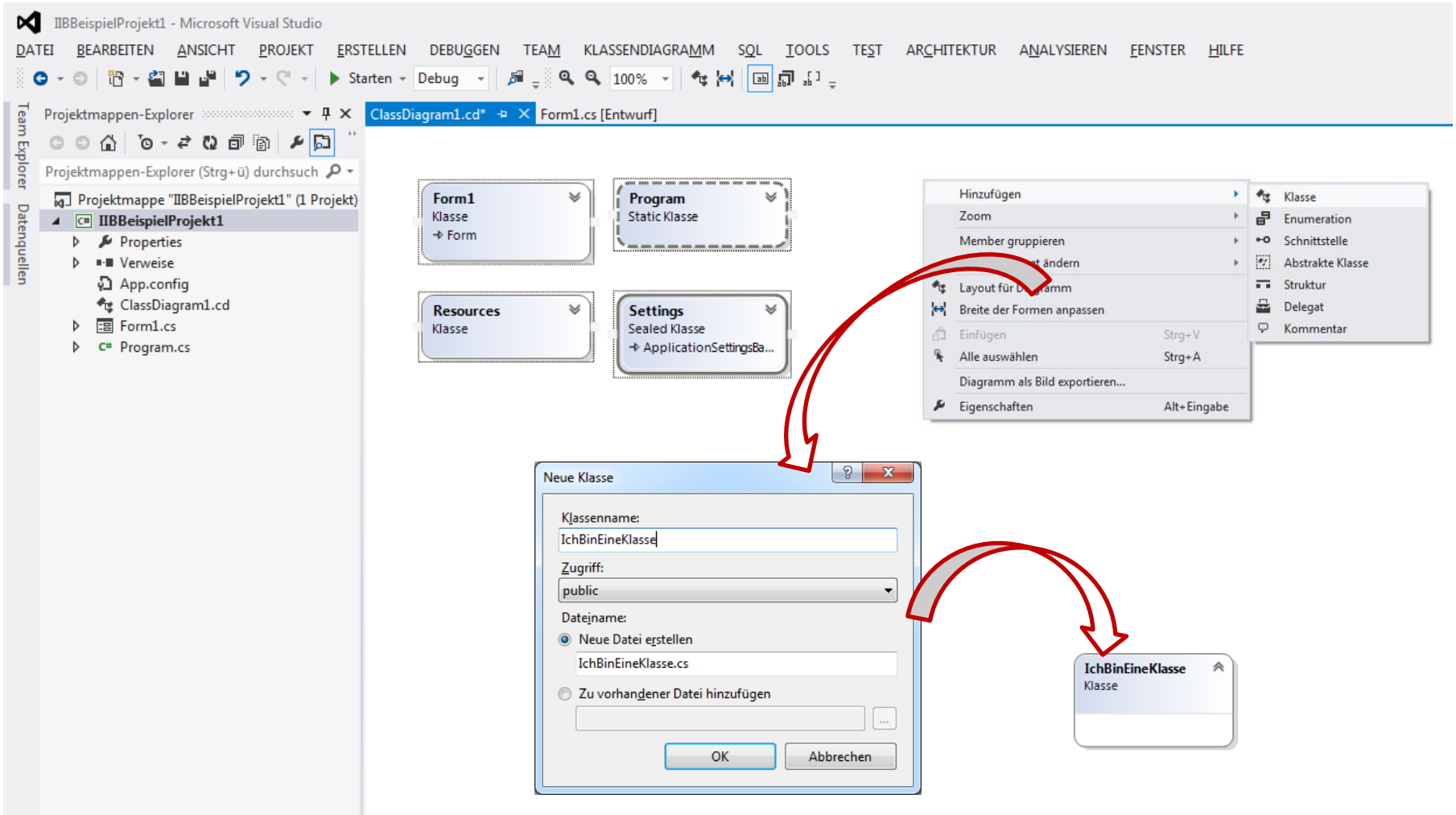
TECHNISCHE UNIVERSITÄT DARMSTADT

- Modellierung mit Klassendiagramm
  - Klassendiagramm anlegen/ in Klassenansicht wechseln
- Erstellen und Modifizieren von
  - Klassen
  - Methoden
  - Attributen
  - Vererbungsbeziehungen

➢ Generierung von Quellcode

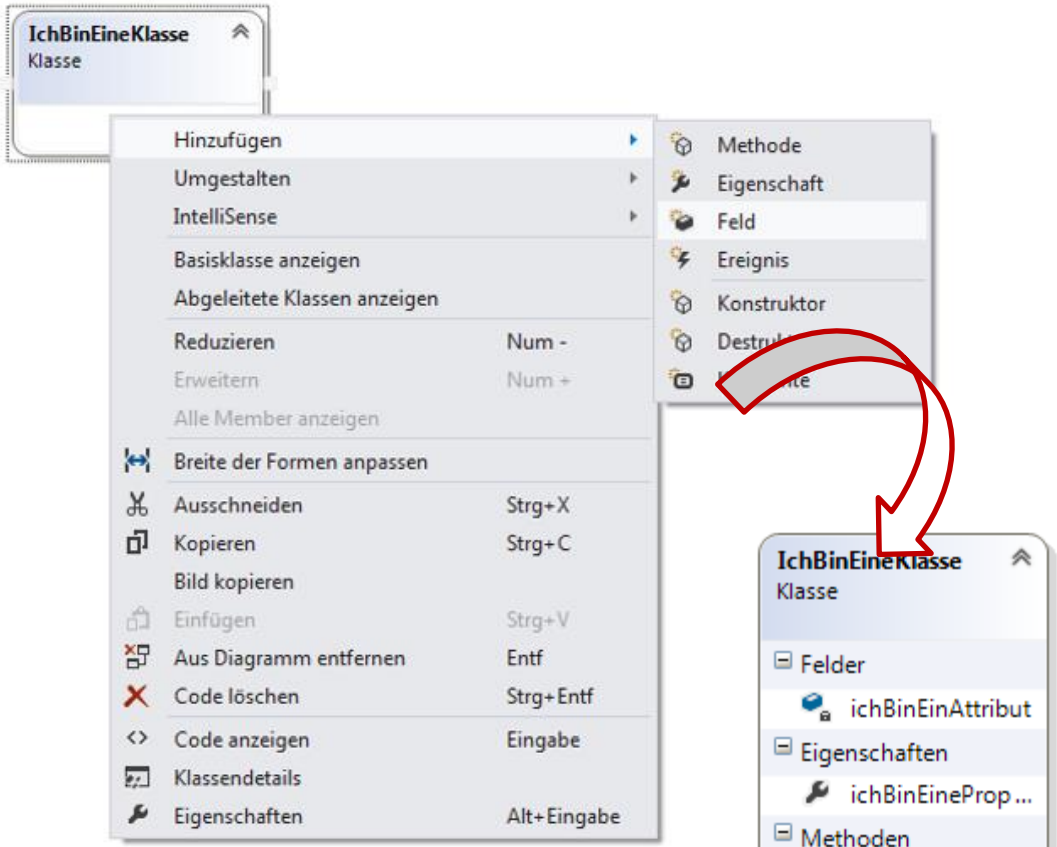


Beispiel: Klasse erstellen



Beispiel: Attribute, Properties und Methoden hinzufügen

TECHNISCHE UNIVERSITÄT DARMSTADT

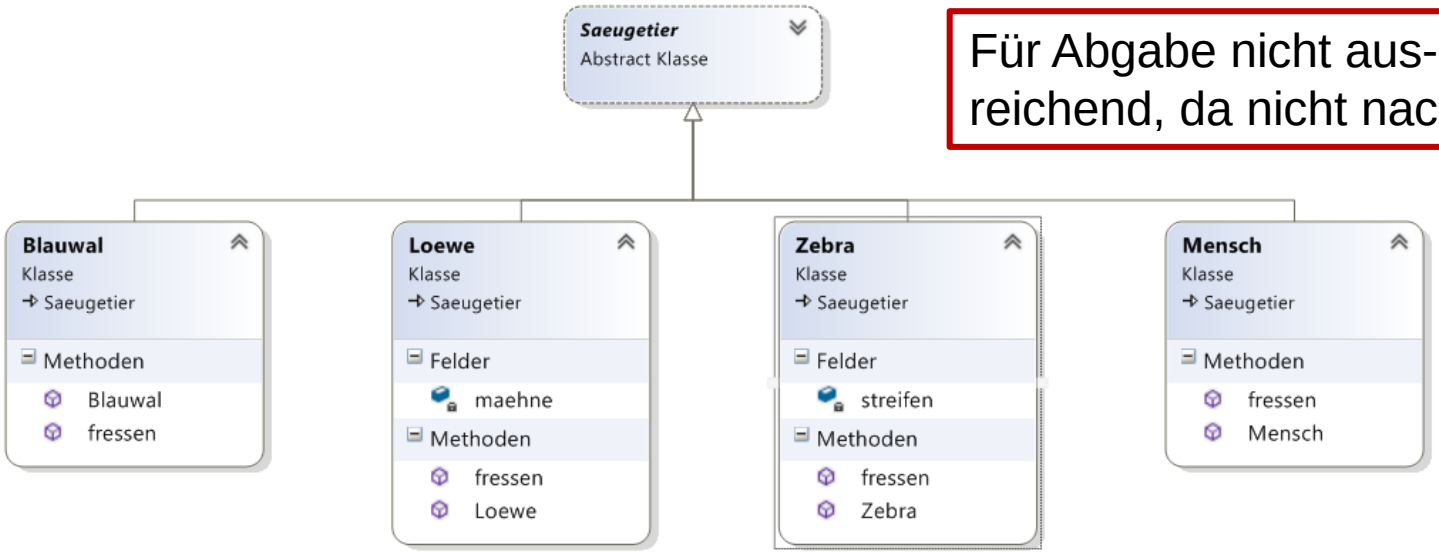



```
public class IchBinEineKlasse
{
    private double ichBinEinAttribut;

    public int ichBinEineProperty
    {
        get
        {
            throw new System.NotImplementedException();
        }
        set
        {
        }
    }

    public void myMethod()
    {
        throw new System.NotImplementedException();
    }
}
```

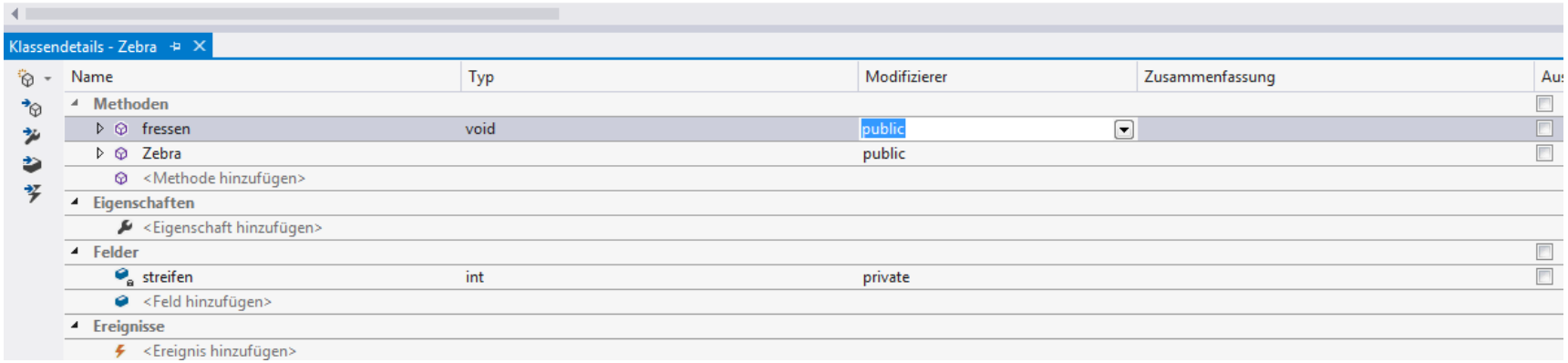
Generierter Quellcode

Beispiel: Modellierte Klassen und Vererbungsbeziehungen

TECHNISCHE UNIVERSITÄT DARMSTADT



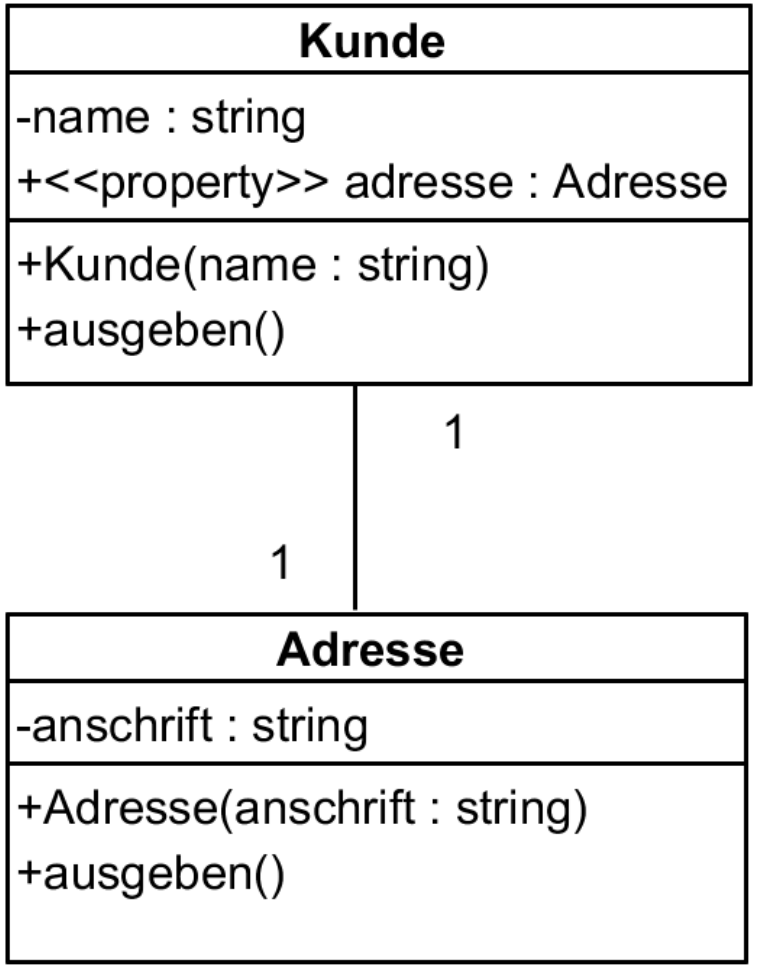
Für Abgabe nicht aus-reichend, da nicht nach UML!



4.3.1 Aggregation (1:1) – Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

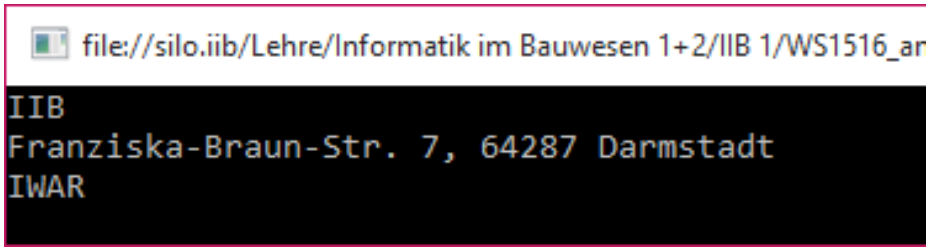
```
class Adresse {
    private string anschrift;
    public Adresse(string anschrift) {
        this.anschrift = anschrift;
    }

    public void ausgeben() {
        Console.WriteLine(anschrift);
    }
}

class Kunde {
    private string name;
    public Adresse adresse { get; set; }
    public Kunde(string name) {
        this.name = name;
    }
    public void ausgeben() {
        Console.WriteLine(name);
        if(adresse != null) adresse.ausgeben();
    }
}
```

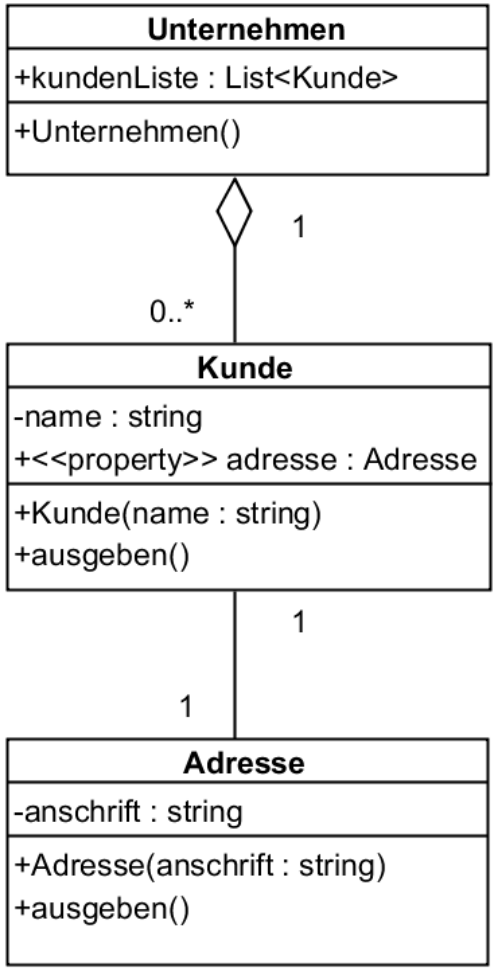


Aggregation (1:n) – Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Unternehmen {
    public List<Kunde> kundenListe;

    public Unternehmen() {
        kundenListe = new List<Kunde>();
    }
}

static void Main(string[] args) {
    Unternehmen u = new Unternehmen();
    Kunde k1 = new Kunde("IIB");
    k1.adresse = new Adresse("Franziska-Braun-Str. 7, 64287 Darmstadt");
    u.kundenListe.Add(k1);
    u.kundenListe.Add(new Kunde("IWAR"));
    foreach (Kunde k in u.kundenListe) {
        k.ausgeben();
    }
    Console.Read();
}
```

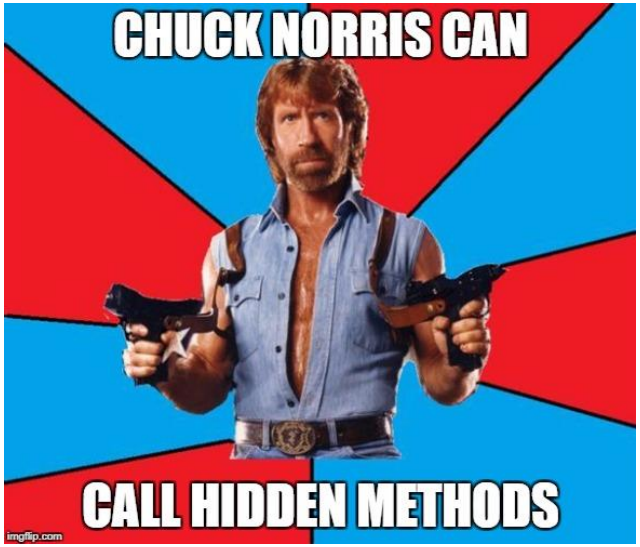




4.3.2 Vererbung – Hiding (Methoden verdecken)

TECHNISCHE UNIVERSITÄT DARMSTADT

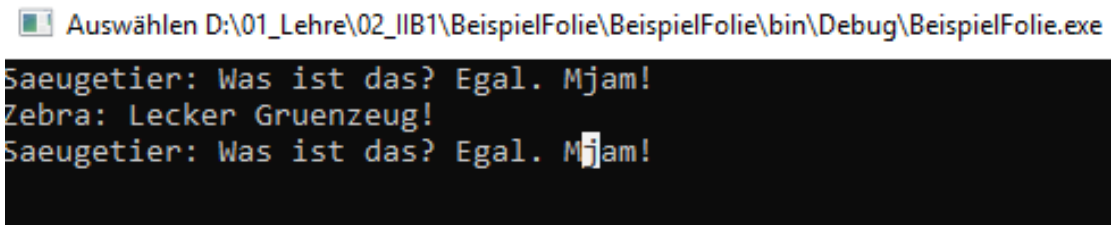
```
public class Pflanze{ /* ... */ }

public class Saeugetier { // Basisklasse
    // ...
    public void fressen(object futter) {
        Console.WriteLine("Was ist das? Egal. Mjam!");
    }
}

public class Zebra : Saeugetier { // Abgeleitete Klasse
    // ...
    // durch den "new"-Modifizierer wird die Methode der Basisklasse explizit überdeckt
    public new void fressen(object futter) {
        if (futter is Pflanze) Console.WriteLine("Lecker Gruenzeug!");
    }
}
```



Vererbung – Hiding (Methoden verdecken)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
static void Main(string[] args)
        {
            Pflanze pf = new Pflanze();
            Saeugetier st = new Saeugetier();
            Zebra zb = new Zebra();

            st.fressen(pf);
            zb.fressen(pf);

            Saeugetier szb = new Zebra();
            szb.fressen(pf);

            Console.ReadLine();
        }
```



Vererbung – Overriding (Methoden überschreiben)

TECHNISCHE UNIVERSITÄT DARMSTADT

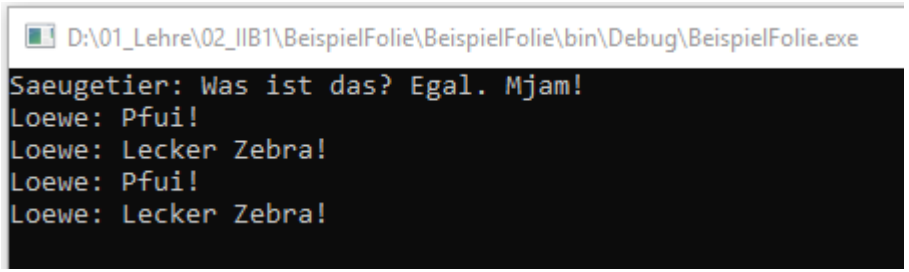
```
public class Saeugetier { // Basisklasse
    // ...
    public virtual void fressen(object futter) {
        Console.WriteLine("Was ist das? Egal. Mjam!");
    }
}


 public class Loewe: Saeugetier {// Abgeleitete Klasse
        // ...

       // durch den „override" Modifizierer wird die Methode der
       // Basisklasse überschrieben
        public override void fressen(object futter)
        {
            if (futter is Saeugetier)
                Console.WriteLine("Loewe: Lecker " + futter.GetType().Name +
"!");

            else Console.WriteLine("Loewe: Pfui!");
        }
}
```

Um in abgeleiteten Klassen "Override" Keyword zu nutzen muss die Methoden in Basisklassen mit "*virtual*" modifiziert werden

Vererbung – Overriding (Methoden überschreiben)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
static void Main(string[] args)
        {
            Pflanze pf = new Pflanze();
            Saeugetier st = new Saeugetier();
            Zebra zb = new Zebra();

            st.fressen(pf);

            Loewe lw = new Loewe();
            lw.fressen(pf);
            lw.fressen(zb);

            Saeugetier slw = new Loewe();
            slw.fressen(pf);
            slw.fressen(zb);

            Console.ReadLine();
        }
```



Vererbung – Overloading

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Saeugetier { // Basisklasse
    // ...
    public virtual void fressen(object futter) {
        Console.WriteLine("Was ist das? Egal. Mjam!");
    }
}

public class Loewe: Saeugetier { // Abgeleitete Klasse
    // ...
    public override void fressen(object futter){
         if (futter is Saeugetier)
            Console.WriteLine("Loewe: Lecker " + futter.GetType().Name + "!");
         else Console.WriteLine("Loewe: Pfui!");
        }
    // Methode der Basisklasse wird überladen
    public void fressen(Saeugetier futter) {
        if (!(futter is Loewe)) Console.WriteLine("Lecker " + futter.GetType().Name + "!");
    }
}
```

Overladen: Gleiche Name, unterschiedliche Signatur

4.4 Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Methodenkopf

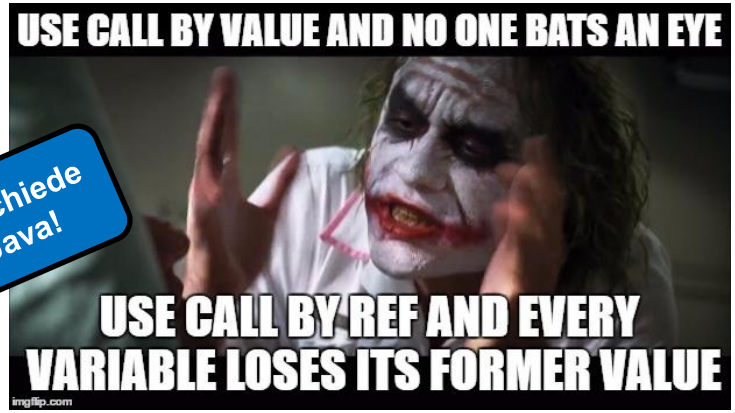
```
[Sichtbarkeit] [Rückgabetyp/void] [Methodenname]([Übergabeparameter: <Typ> <Name>])
{
   // Methodeninhalt
   return  [Rückgabewert]; //außer bei void
}
```

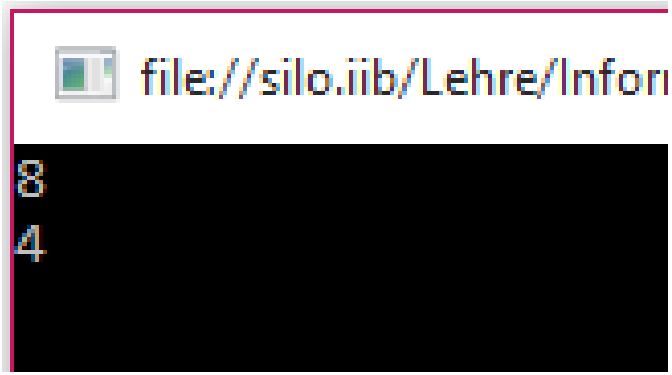
Methodenkörper

Call by Value / Call by Reference

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class GutUndBoese
{
    public void liebeMethode(int ueberschreib_mich_nicht) {
  // Übergabe per Wert (by Value)
        ueberschreibe_mich_nicht = 4;
    }

    public void boeseMethode(ref int ueberschreib_mich) {
  // Übergabe per Referenz (by Reference)
        ueberschreibe_mich = 4;
    }

    static void Main(string[] args) {
        int hilflos = 8;
        GutUndBoese gub = new GutUndBoese();
        gub.liebeMethode(hilflos);
        Console.WriteLine(hilflos);
        gub.boeseMethode(ref hilflos);
        Console.WriteLine(hilflos);
        Console.Read();
    }
}
```





Call by Reference
Unterschied zwischen out und ref (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
static void skalarProduktByRef(double[] v1, double[] v2, ref double skalarProd)
{
    for (int i = 0; i < v1.Length; i++)
    {
        skalarProd += v1[i] * v2[i];
    }
}

static void skalarProduktOut(double[] v1, double[] v2, out double skalarProd)
{
    skalarProd = 0; // Die Variable muss innerhalb der Methode initialisiert werden

    for (int i = 0; i < v1.Length; i++)
    {
        skalarProd += v1[i] * v2[i];
    }
}
```

Call by Reference
Unterschied zwischen out und ref (2)

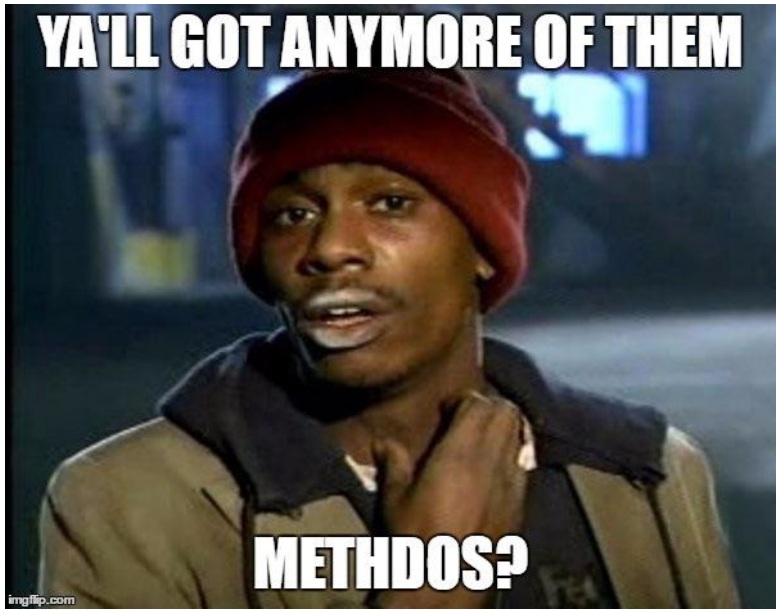
TECHNISCHE UNIVERSITÄT DARMSTADT

```
static void Main(string[] args)
{
    double[] v1 = { 1, 1, 2 };
    double[] v2 = { 3, 1, 4 };

    // call by reference

    double skalarProdByRef = 0; // Die Variable muss vor dem Methodenaufruf initialisiert werden

    skalarProduktByRef(v1, v2, ref skalarProdByRef);
    Console.WriteLine(skalarProdByRef);

    // out
    double skalarProdOut;
    skalarProduktOut(v1, v2, out skalarProdOut);
    Console.WriteLine(skalarProdOut);

    Console.Read();
}
```

Overloading (Methoden überladen)

TECHNISCHE UNIVERSITÄT DARMSTADT

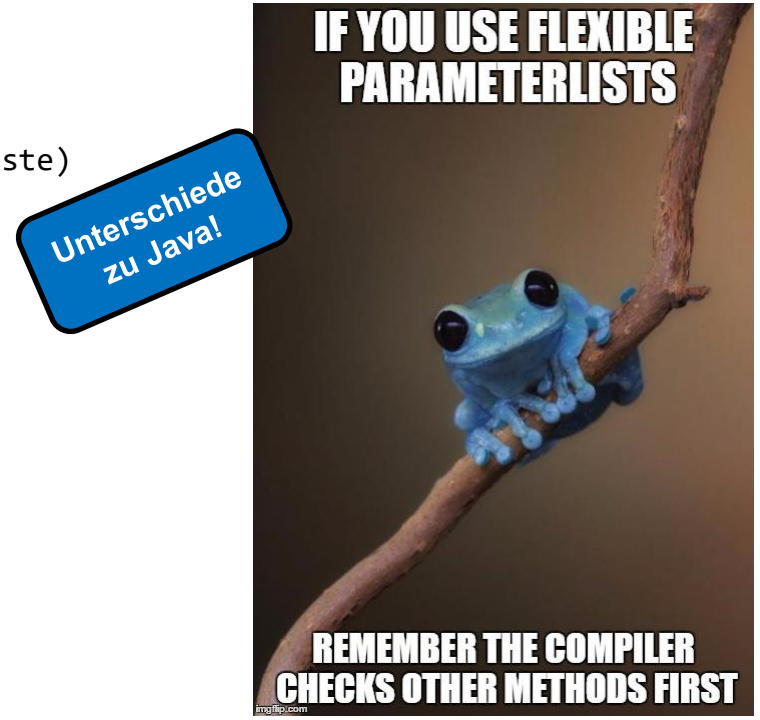
```
class Taschenrechner {
    public int mal(int x, int y) {
        return x * y;
    }
    public double mal(double x, double y) {
        return x * y;
    }
    public float mal(float x, float y, float z) {
        return x * y * z;
    }
    static void Main(string[] args) {
        Taschenrechner t = new Taschenrechner();
        Console.WriteLine(t.mal(3, 2));
        Console.WriteLine(t.mal(3.0, 1.5));
        Console.WriteLine(t.mal(3.0f, 1.5f, 2.25f));
        Console.Read();
    }
}
```

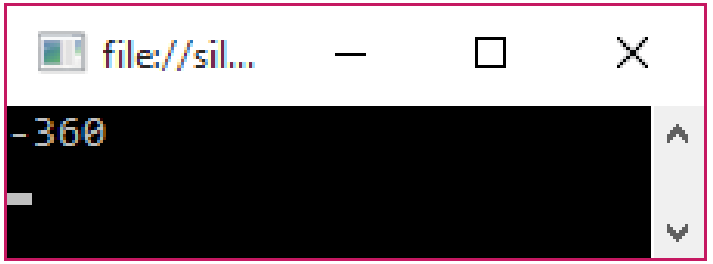


Flexible Parameterlisten

TECHNISCHE UNIVERSITÄT DARMSTADT

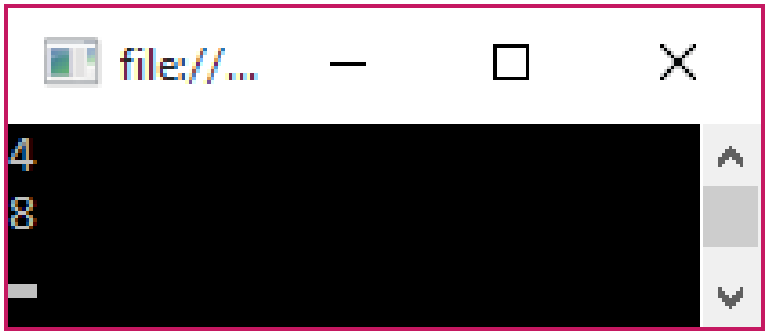
```
class Taschenrechner {

    public int mal_ganzviele(int x, int y, params int[] liste)
    {
        int ergebnis = x * y;
        for (int i = 0; i < liste.Length; i++)
        {
            ergebnis *= liste[i];
        }
        return ergebnis;
    }

    static void Main(string[] args) {
        Taschenrechner t = new Taschenrechner();
        Console.WriteLine(t.mal_ganzviele(5, 6, 2, 2, -3));
        Console.Read();
    }
}
```





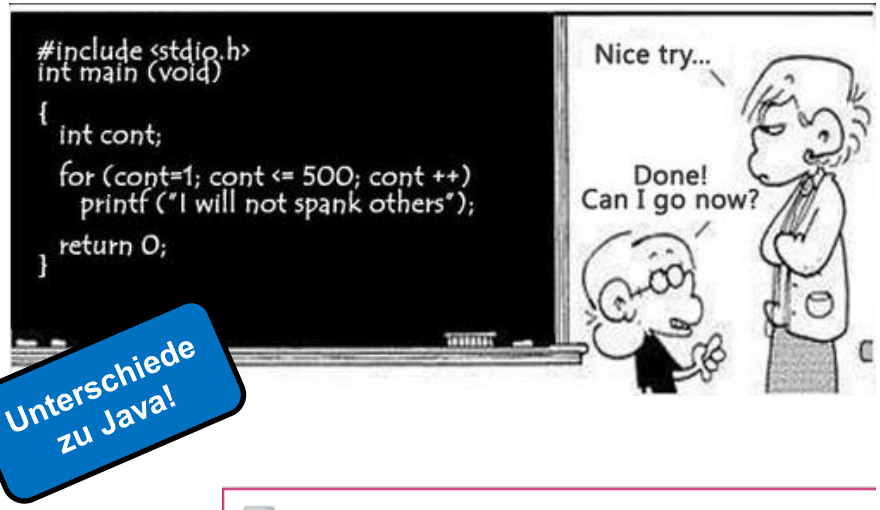
Optionale Parameter

```
class Taschenrechner {

    public int xHochY(int x, int y = 2)
    {
        int ergebnis = 1;
        for (int i = 0; i < y; i++)
        {
           ergebnis *= x;
        }
        return ergebnis;
    }

    static void Main(string[] args) {
        Taschenrechner t = new Taschenrechner();
        Console.WriteLine(t.xHochY(2));
        Console.WriteLine(t.xHochY(2, y: 3));
        Console.Read();
    }
}
```



4.5 Schleifen

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine("Opa, was machst Du?");
}

string[] feld = { "Malen", "Rechnen", "Aufpassen" };
foreach (string feldElement in feld)
{
    Console.WriteLine(feldElement);
}

int j = 0;
do
{
    Console.WriteLine("Opa, was machst Du?");
    j++;
} while (j < 5);
```

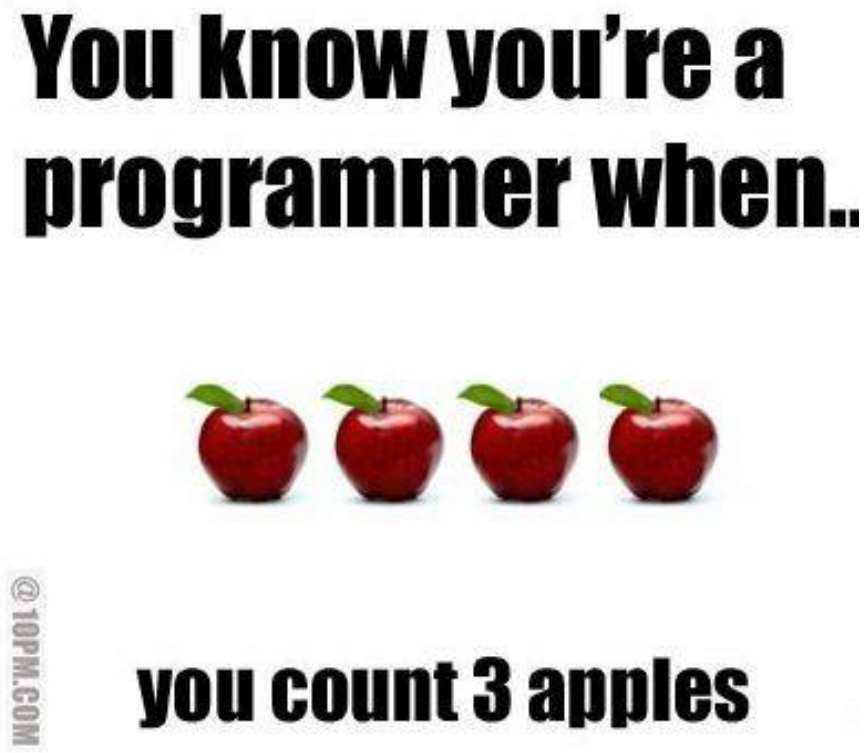


Unterschiede zu Java!

file://silo.iib/Lehre/Informatik im Bauwesen 1+2/IIB 1/W

```
Opa, was machst Du?
Opa, was machst Du?
Opa, was machst Du?
Opa, was machst Du?
Opa, was machst Du?
Malen
Rechnen
Aufpassen
Opa, was machst Du?
Opa, was machst Du?
Opa, was machst Du?
Opa, was machst Du?
Opa, was machst Du?
```

4.6 Listen und Arrays

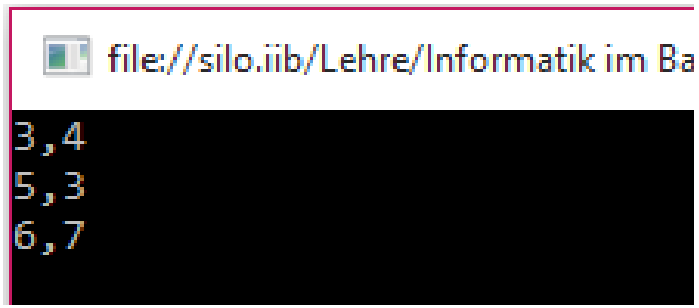


Eindimensionale Arrays (Felder)

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterschiede zu Java!

```
// Deklaration/Initialisierung eines Integer-Arrays mit
// 1000 Elementen
int[] ein_feld = new int[1000];
// Belegung des 1. Elements mit dem Wert 60
ein_feld[0] = 60;

double[] zweites_feld = new double[3] { 3.4, 5.3, 6.7 };
// oder
double[] noch_ein_feld = new double[] { 3.4, 5.3, 6.7 };
// oder
double[] schon_klar = { 3.4, 5.3, 6.7 };

// Zugriff auf Elemente des Arrays
for (int i = 0; i < zweites_feld.Length; i++)
{
    Console.WriteLine(zweites_feld[i]);
}
```

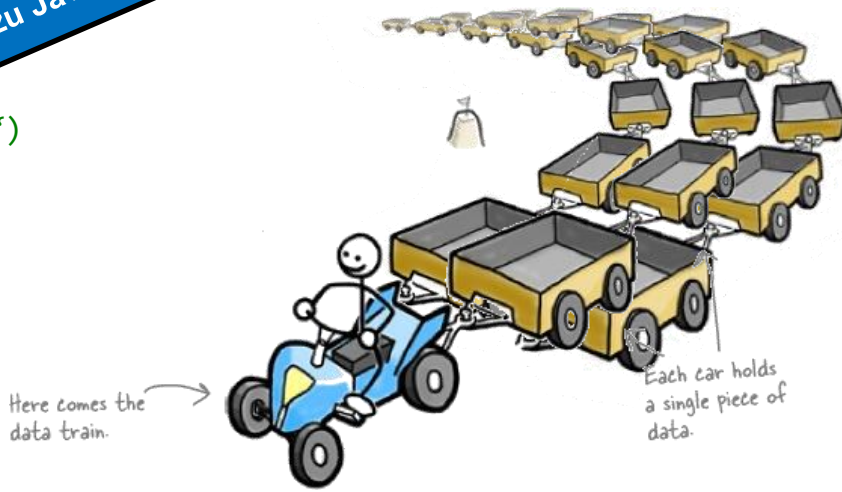


Mehrdimensionale Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Deklaration eines 4-dimensionalen Arrays
double[, , ,] raumzeit;

// Initialisierung eines 2-dimensionalen Arrays
int[,] feld_2D = new int[2, 3]; // („2 Zeilen, 3 Spalten“)
// oder
feld_2D = new int[,] { { 1, 2, 3 }, { 4, 5, 7 } };

// Wertzuweisung einzelnes Element
feld_2D[1, 2] = 6; // („2. Zeile, 3. Spalte“)

// GetLength(d) --> Array-Länge in der Dimension d
for (int x = 0; x < feld_2D.GetLength(0); x++)
{
    for (int y = 0; y < feld_2D.GetLength(1); y++)
    {
        Console.Write(feld_2D[x, y] + " ");
    }
    Console.Write("\n");
}
```

Unterschiede zu Java!




```
file://silo.iib/Lehre/Informatik im B
1 2 3
4 5 6
```

Generic Collections

TECHNISCHE UNIVERSITÄT DARMSTADT

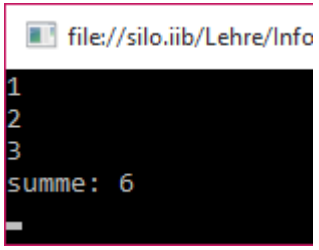
- Klassen zum Verwalten von Daten
- Unterstützung für stacks, queues, lists und hash-tables
- Seit .NET 2.0 integriert, sollten die nicht-generischen Pendants ablösen (ArrayList, ListDictionary, HashTable, SortedList)
- Vorteile gegenüber nicht-generischen Collections:
  - Typsicher
  - Performanter
  - Mehr Funktionen
- Beispiele: `List<T>, SortedList<K,V>, Dictionary<K,V>, Queue<T>, Stack<T>`
- Einbinden mit: `using System.Collections.Generic;`

List<T> (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Geordnete Sammlung von Objekten eines Typs T
- Dynamisch
- Wichtige Funktionen:
  - Add(T item)
  - Clear()
  - Contains(T item)
  - FindIndex(Predicate<int> match)
  - FindAll(Predicate<int> match)
  - Insert(int index, T item)
  - Remove(T item)
  - ToArray()
  - Sort(Comparison<T>)
  - ...

```
List<int> myInts = new List<int>();
myInts.Add(1);
myInts.Add(2);
myInts.Add(3);

for (int i = 0; i < myInts.Count; i++)
{
    Console.WriteLine("" + myInts[i]);
}
int summe = 0;
foreach (int x in myInts)
{
    summe += x;
}
Console.WriteLine("summe: " + summe);
```

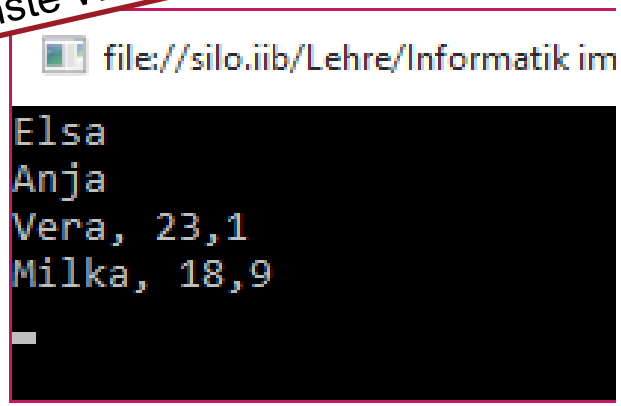


List<T> (2) - Durchsuchen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Kuh> kuehe = new List<Kuh>();
kuehe.Add(new Kuh("Elsa", 26.7, Color.Black));
kuehe.Add(new Kuh("Vera", 23.1, Color.Brown));
kuehe.Add(new Kuh("Anja", 35.5, Color.White));
kuehe.Add(new Kuh("Milka", 18.9, Color.Violet));
// Liste durchsuchen mit Prädikaten
List<Kuh> hoheMilchleistung = kuehe.FindAll(kuh => kuh.MilchLeistung > 25.0);
foreach (Kuh k in hoheMilchleistung)
{
    Console.WriteLine(k.name);
}
// Liste durchsuchen mit LINQ (Language-Integrated Query)
var geringeLeistung = from k in kuehe
                      where k.MilchLeistung < 25
                      select new { k.name, k.MilchLeistung };
foreach (var g in geringeLeistung)
{
    Console.WriteLine(g.name + ", " + g.MilchLeistung);
}
```

Mehr dazu nächste Woche !



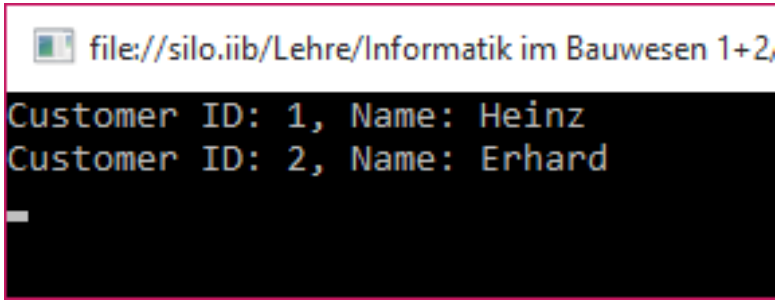
Dictionary<K,V>

TECHNISCHE UNIVERSITÄT DARMSTADT

- Assoziative, ungeordnete Collection von Schlüssel-Wert-Paaren (`KeyValuePair`)
- Gut geeignet für Datensätze mit unique identifiers

```
class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }

    public Customer(int id, string n)
    {
                ID = id;
                Name = n;
    }
}
```

```
Dictionary<int, Customer> customerDic;
customerDic = new Dictionary<int, Customer>();

Customer c1 = new Customer(1, "Heinz");
Customer c2 = new Customer(2, "Erhard");
customerDic.Add(c1.ID, c1);
customerDic[c2.ID] = c2;

foreach (KeyValuePair<int, Customer> kv in customerDic)
{
    Console.WriteLine(
    "Customer ID: {0}, Name: {1}",
    kv.Key, kv.Value.Name);
}
```

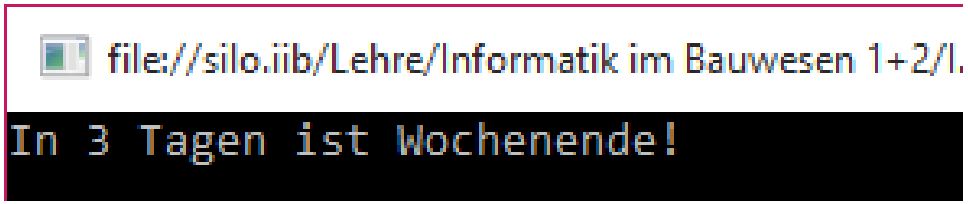


Enum (Enumerationen)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Benannte Konstanten, die (meistens) Integer-Werte repräsentieren
- Können deklariert und verwendet werden wie normale Variablen
- Können mit einer bitweisen OR-Operation kombiniert werden

```
enum Tag {
    Montag = 1, // Nummerierung soll bei 1 beginnen
    Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag,
    Sonnabend = Samstag // Samstag wird auch als Sonnabend bezeichnet
}

Tag Wochenende = Tag.Samstag | Tag.Sonntag;
Tag t = Tag.Dienstag;
if ((Wochenende) != t)
{
    int x = (int)t;
    int y = (int)Tag.Samstag;
    Console.WriteLine("In " + (y - x) + " Tagen ist Wochenende!");
}
else Console.WriteLine("Jipiee!");
```



4.7 Kontrollstrukturen

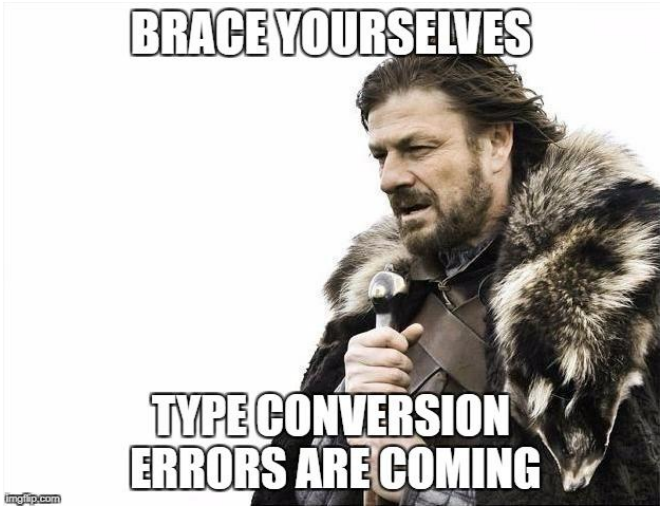
```
int a = (int)(new Random().NextDouble()*10);
if (a == 5)
{
   Console.WriteLine("a ist gleich 5");
}
else
{
   Console.WriteLine("a = "+ a +" ist ungleich 5");
}
int glueckszahl = 3;
switch (glueckszahl)
{
  case 1:
      Console.WriteLine("So ein Pech.");
      break;
  case 2:
      Console.WriteLine("Auch hier nix gewonnen.");
      break;
  case 3:
      Console.WriteLine("Die Waschmaschine ist Ihnen!");
      break;
}
```

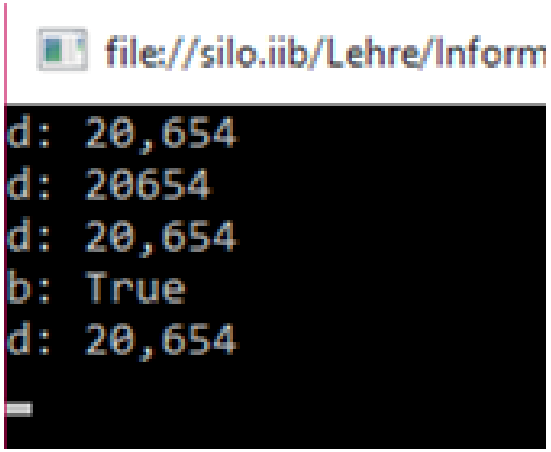

```
file://silo.iib/Lehre/Informatik im Bauwesen 1+2
a = 0 ist ungleich 5
Die Waschmaschine ist Ihnen!
```

4.8 Typkonvertierung - Parsing

TECHNISCHE UNIVERSITÄT DARMSTADT

```
string s = "20,654";
double d;
bool b;

// Ziel: string --> double
// Parse()
d = double.Parse(s);
Console.WriteLine("d: " + d);
d = double.Parse(s, System.Globalization.CultureInfo
          .GetCultureInfo("en-US").NumberFormat);
Console.WriteLine("d: " + d);

// oder Convert()
d = Convert.ToDouble(s);
Console.WriteLine("d: " + d);

// oder TryParse()
bool b = double.TryParse(s, out d);
Console.WriteLine("b: " + b);
Console.WriteLine("d: " + d);
```





Typkonvertierung - Casting

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int i = 5; double d = 5.6;

/* implizites Casting
Nur möglich von Typen geringerer Genauigkeit zu
Typen höherer Genauigkeit */
double x = i; // funktioniert
Console.WriteLine("x: " + x);
int y = d; // FEHLERMLEDUNG!

/* explizites Casting
(ggf. mit Informationsverlust) */
int y = (int)d;
Console.WriteLine("y: " + y);
```

⇨ abgeschnitten

```
y = Convert.ToInt32(d);
Console.WriteLine("y: " + y);
```

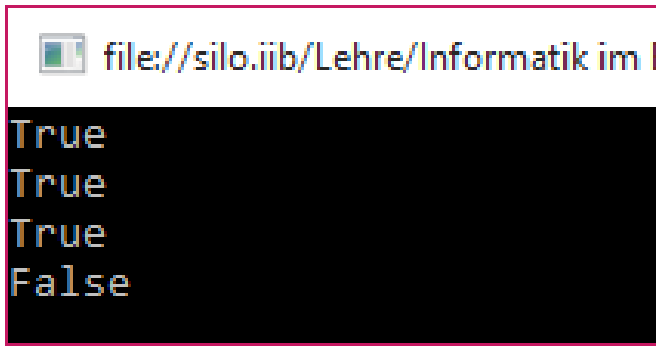
⇨ gerundet

4.9 is und as Operatoren (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Person { }
class Angestellter : Person { }
class Kunde : Person { }
class Baum { }

static bool test(Object o) {
    return (o is Person);
}

static void Main(string[] args) {
    object p = new Person();
    object a = new Angestellter();
    object k = new Kunde();
    object b = new Baum();
    Console.WriteLine(test(p));
    Console.WriteLine(test(a));
    Console.WriteLine(test(k));
    Console.WriteLine(test(b));
}
```

- Mit **`is`** lässt sich überprüfen ob ein Objekt die Instanz eines bestimmten Typs ist

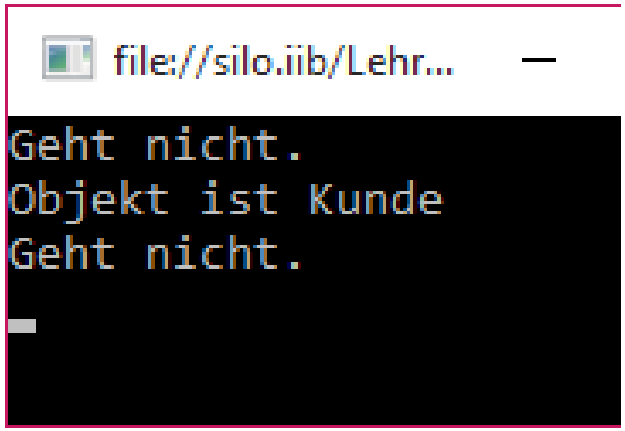


is und as Operatoren (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
static void castToKunde(object obj) {
    Kunde k = obj as Kunde;
    if (k == null) Console.WriteLine("Geht nicht.");
    else Console.WriteLine("Objekt ist {0}",
                                k.GetType().Name);
}

static void isOperator() {
    object p = new Person();
    object k = new Kunde();
    object b = new Baum();
    castToKunde(p);
    castToKunde(k);
    castToKunde(b);
}
```

- **as** führt eine Typüberprüfung durch und castet, wenn das Objekt vom gefragten Typ ist, wenn nicht wird das Objekt mit null belegt



4.10 Exception Handling

Exception-Handling (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

Warum?

- Während des Programmablaufs kann es zu unvorhergesehenen Problemen kommen, die einen plötzlichen Programmabsturz zur Folge haben. (z.B. durch fehlerhafte Benutzereingaben)

Abhilfe

- Schlecht: Erkennen von Fehlern z.B. anhand von Rückgabewerten von Methoden. → Keine Trennung von Datenfluss und Fehlerbehandlung.
- Besser: „**Werfen**“ von speziellen Fehlerobjekten (**Exceptions**) um in eine vom normalen Programmablauf getrennte Fehlerbehandlung zu wechseln.

Aufbau einer Fehlerbehandlung:

```
try
{
// tue etwas das schief gehen kann
}
catch (/* Exception-Typ A*/)
{
// Fehlerbehandlung A
}
catch (/* Exception-Typ B*/)
{
// Fehlerbehandlung B
}
finally // optional
{
// so oder so ausführen
}
```

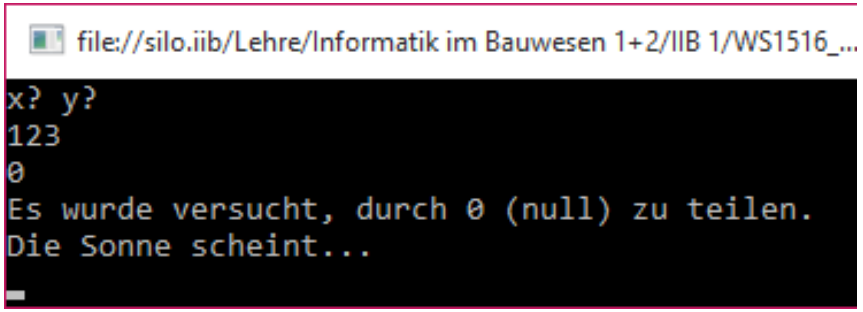
Exception-Handling (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Taschenrechner {
    public double teile(double x, double y) {
        if(y==0) throw new DivideByZeroException(); // Exception "auswerfen"
        else return x/y;
    }
}
static void Main(string[] args) {
    Taschenrechner t = new Taschenrechner();
    try {
        Console.WriteLine("x? y?");
        double x = Convert.ToDouble(Console.ReadLine());
        double y = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("x/y = " + t.teile(x, y));
    } catch (Exception ex) {
        Console.WriteLine(ex.Message);
    } finally {
        Console.WriteLine("Die Sonne scheint...");
    }
    Console.Read();
}
```

FormatException, OverflowException

DivideByZeroException

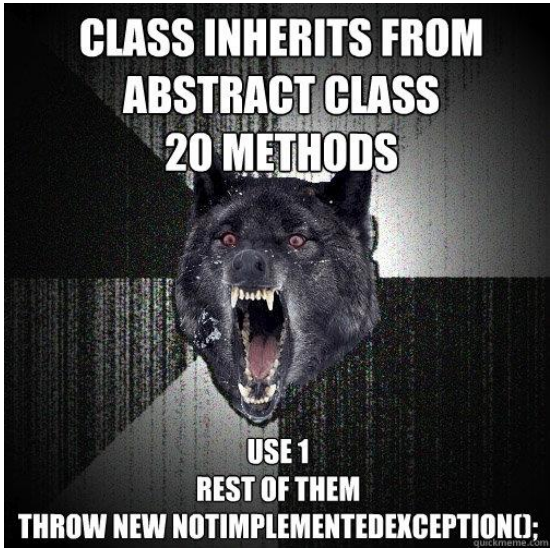


4.11 Abstrakte & Virtuelle Klassen/Methoden, Interfaces

Abstrakte Klassen und Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public abstract class Saeugetier { // abstrakte Basisklasse
    // ...
    public abstract void fressen(object futter); // abstrakte Methode
}
public class Loewe: Saeugetier { // Abgeleitete Klasse
    // ...
    public override void fressen(object futter) { // geerbte
  abstrakte Methode wird implementiert
        if (futter is Saeugetier && !(futter is Loewe))
            Console.WriteLine("Lecker " + futter.GetType().Name +
  "!");
        else Console.WriteLine("Pfui!");
    }
}
class Nahrungskette {
    static void Main(string[] args) {
        Saeugetier irgendwas = new Saeugetier(10000); // FEHLER!
        Saeugetier loewe = new Loewe(12, true);
        loewe.fressen(new Blauwal(90));
    }
}
```



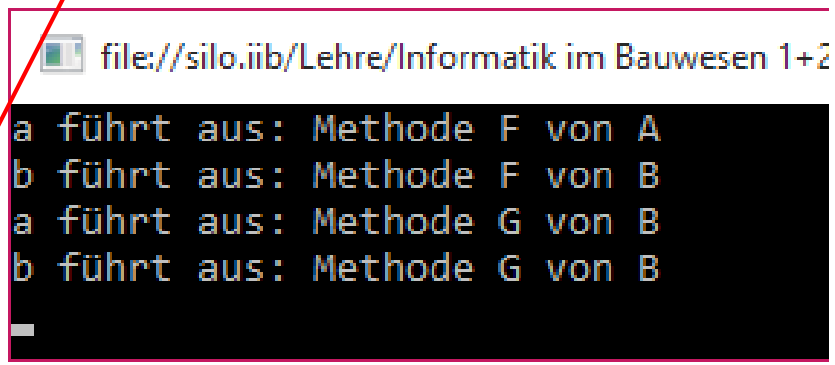

```
file://silo.iib/Lehre...
Lecker Blauwal!
```

Virtuelle Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class A {
    public string F() { return "Methode F von A"; } // nicht-virtuelle Methode
    public virtual string G() { return "Methode G von A"; } // virtuelle Methode
}
class B : A {
    new public string F() { return "Methode F von B"; } // Methode A.F() wird überdeckt
    public override string G() { return "Methode G von B"; } // Methode A.G() wird überschrieben
}

class VirtualTest {
    static void Main() {
        B b = new B();
        A a = b;
        Console.WriteLine("a führt aus: " + a.F());
        Console.WriteLine("b führt aus: " + b.F());
        Console.WriteLine("a führt aus: " + a.G());
        Console.WriteLine("b führt aus: " + b.G());
        Console.Read();
    }
}
```

Laufzeittyp != Kompilierungstyp



Wann new, wann override? http://msdn.microsoft.com/de-de/library/vstudio/ms173153.aspx

Interface (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Fortführung der Idee der abstrakten Klassen
- Ein Interface enthält nur Signaturen von Methoden, Eigenschaften, Ereignissen oder Indexen.
- Eine Klasse, die ein Interface implementiert, muss alle Member des Interface implementieren.
- Eine Klasse kann mehrere Interfaces implementieren.

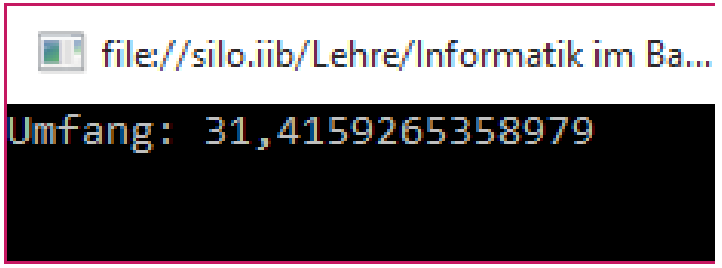
```
interface IGeometrie
{
    double umfang { get; }

    void berechneUmfang();
}
```

Interface (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Kreis : IGeometrie {
    private double radius;
    private double _umfang;

    public Kreis(double r) {
        radius = r;
    }

    public double umfang {
        get { return _umfang; }
    }

    public void berechneUmfang() {
        _umfang = Math.PI * 2 * radius;
    }
}
```

```
interface IGeometrie {
    double umfang { get; }

    void berechneUmfang();
}

static void Main(string[] args)
{
    Kreis k = new Kreis(5);
    k.berechneUmfang();
    Console.WriteLine("Umfang: " + k.umfang);
    Console.Read();
}
```



5. Ausgabe Übung 1

Die Abgabe hat bis zum 09.12.2018, 23:55 Uhr zu erfolgen!

TECHNISCHE UNIVERSITÄT DARMSTADT

Aufgabenstellung(en)
→ siehe Moodle

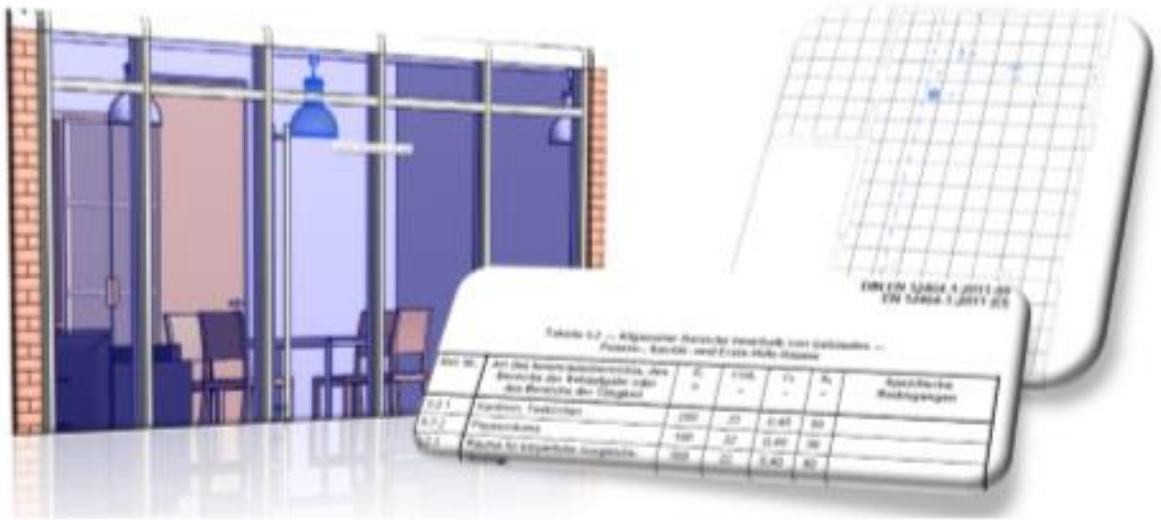
Mitteilung der gewählten Aufgabe bis zum 21.11.2018 per Moodle-Abstimmung - Bitte nur ein Student pro Gruppe abstimmen!!

Abgabe
Moodle

Übung 1 – Variante 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Erweiterung einer Modellierungs-Software um ein Beleuchtungsplanung-Tool
Teil 1: Windows Forms-Applikation



Übung 1 – Variante 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Erweiterung einer Modellierungs-Software um ein Flächenmanagement-Tool
Teil 1: Windows Forms-Applikation

Übung 1 – Variante 3

TECHNISCHE UNIVERSITÄT DARMSTADT

Erweiterung einer Modellierungs-Software um ein Planung-Tool für Photovoltaikanlagen
Teil 1: Windows Forms-Applikation

Kochrezept für eine Applikation mit GUI

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Anforderungsanalyse und Use-Case**
- **Klassendiagramm**
- **Berücksichtigung Aggregation/Vererbung**
- Hilfsklassen z.B. für immer wieder auftretende Berechnungsfunktionen
- Erzeugung geeigneter Forms (Vorstellung nächste Woche)
- **Funktionscode von Darstellungscode trennen!**
- Sauberen Code verwenden!

Hilfe

TECHNISCHE UNIVERSITÄT DARMSTADT

Visual Studio Hilfe

TECHNISCHE UNIVERSITÄT DARMSTADT

- Erreichbar über
  - Startseite von Visual Studio
  - Menü → Hilfe
    - Einzelne Inhalte über „Hilfeinhalte hinzufügen oder entfernen“ auch offline verfügbar machen
- Beinhaltet Informationen aus
  - MSDN (Microsoft Developer Network)
  - Community Sites (z.B. Codezone)
- Ergebnisse filterbar nach
  - Produkt
  - Programmiersprache
- Rename Project kann sehr aufwendig sein! https://www.youtube.com/watch?v=ZImkQ548vdg

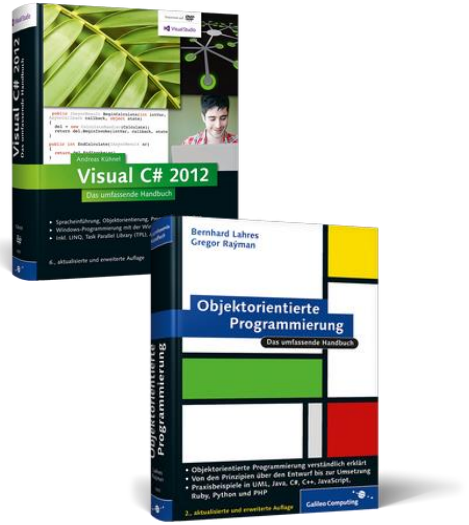
Literatur

- Visual C# 2012 – Das umfassende Handbuch (von Andreas Kühne) [openbook = kostenfrei]

  http://openbook.galileocomputing.de/visual_csharp_2012/

- Objektorientierte Programmierung (von B. Lahres, G. Rayman) [openbook = kostenfrei]

  http://www.boles.de/hamster/band2.html



- Objektorientierte Programmierung spielend gelernt mit dem Java-Hamster-Modell (von D. Boles, C. Boles)

  http://www.boles.de/hamster/band2.html

- HRZ – Handbücher: C# - Einführung

  https://www.hrz.tu-darmstadt.de/weiterbildung/handbuecher/index.de.jsp

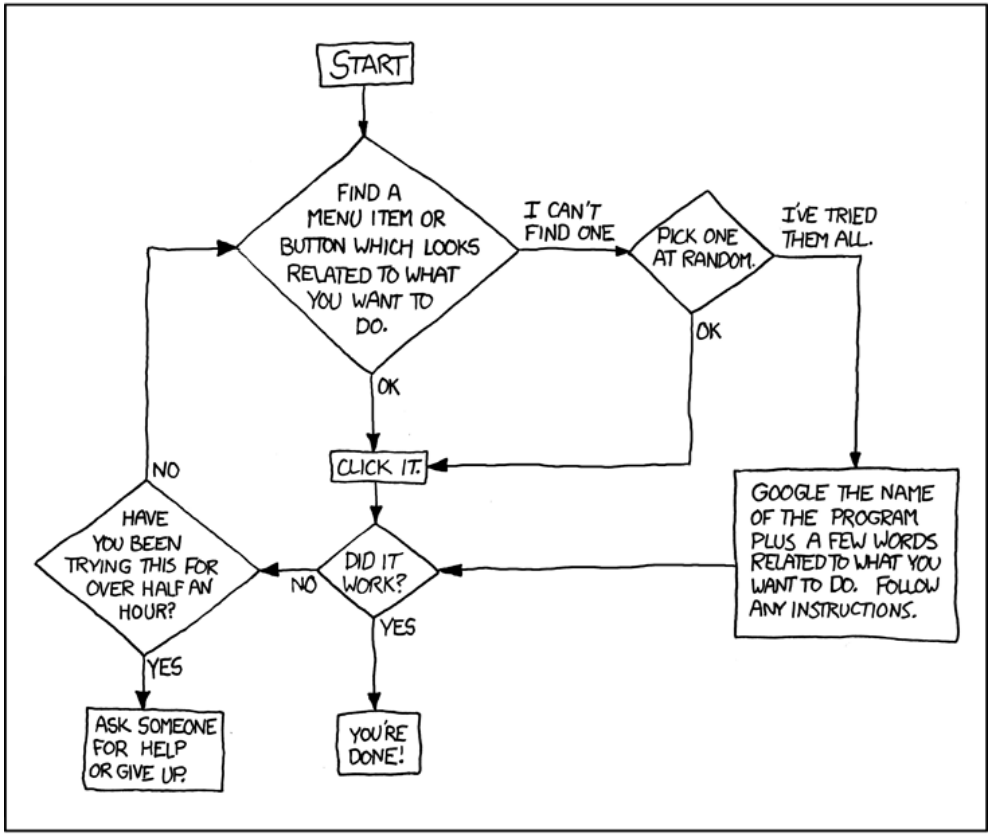
Hilfreiche Foren / Internetadressen

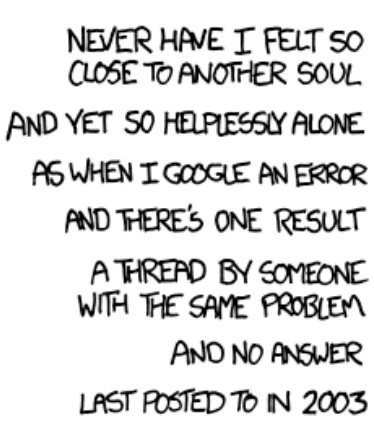
- MSDN Library (https://msdn.microsoft.com/de-de/library/w0x726c2(v=vs.110).aspx)

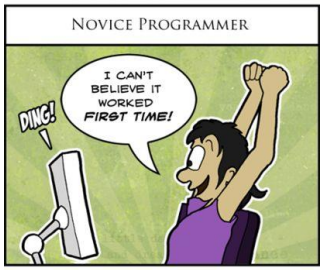
- Stackoverflow (http://stackoverflow.com/)
  - Beachtet den grünen Haken!

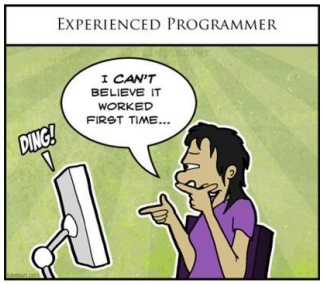
- Google.

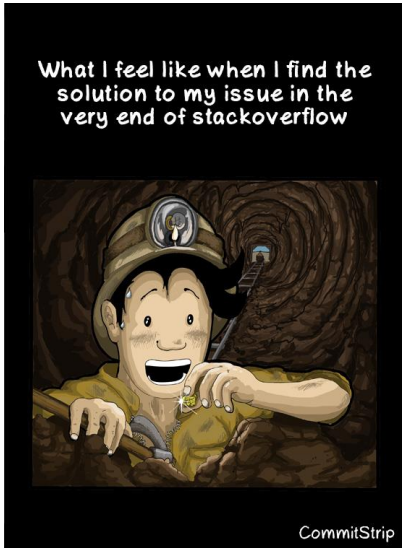
Bei Problemen…











Hinweise für‘s nächste Mal

TECHNISCHE UNIVERSITÄT DARMSTADT

- PCs im PC-Pool nutzbar

- Benutzen von eigenem Laptop/PC empfohlen
  - Benötigte Software:
    - Visual Studio 2017 Community Edition
      https://www.visualstudio.com/de/downloads/?rr=https%3A%2F%2Fwww.google.de%2F