Vorlesung Softwareentwicklung 2021

https://github.com/SebastianZug/CsharpCourse

André Dietrich Christoph Pooch Fabian Bär Fritz Apelt Galina Rudolf JohannaKlinke Jonas Treumer KoKoKotlin Lesestein LinaTeumer MMachel Sebastian Zug Snikker123 Yannik Höll Florian2501 DEVensiv fb89zila

Abstrakte Klassen und Interfaces

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2021`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Konzepte Abstrakter Klassen und Interfaces`
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/10_AbstrakteKlassenUndInterfaces.md
Autoren	@author

Auf Nachfrage ...

Wozu brauche ich das? Bisher bin ich auch gut ohne OOP ausgekommen ...

Nutzung von objektorientierten Konzepten im Python Projekt github2pandas.

Abstrakte Klassen / Abstrakte Methoden

Mit `virtual` werden einzelne Methoden spezifiziert, die durch die abgeleiteten Klassen implmentiert werden. Die Basisklasse hält aber eine "default" Implementierung bereit. Letztendich kann man diesen Gedanken konsequent weiter treiben und die Methoden der Basisklasse auf ein reines Muster reduzieren, dass keine eigenen Implementierungen umfasst.

Diese Aufgabe übernehmen abstrakte Klassen und abstrakte Methoden. Eine abstrakte Klasse:

- kann nicht instantiiert werden
- kann abstrakte Methoden umfassen
- ist oft als Startpunkt(e) einer Vererbungshierarchie gedacht sind.

Innerhalb der Klasse können abstrakte Methoden integriert werden, die

- implizit als virtuelle Methode implementiert angelegt werden
- entsprechend keinen Methodenkörper umfassen

Eine nicht abstrakte Klasse, die von einer abstrakten Klasse abgeleitet wurde, muss Implementierungen aller geerbten abstrakten Methoden und Accessoren enthalten.

```
using System;

public abstract class Animal
{
  public string Name;
```

```
  public Animal(string name){
    Name = name;
  }
  public abstract void makeSound();
}

public class Corcodile : Animal{
  public Corcodile(string name) : base(name){
    Name = name;
  }
  public override void makeSound(){
    Console.WriteLine("I'm a Crocodile");
  }
}

public class Program
{
  public static void Main(string[] args){
    Corcodile A = new Corcodile("Tuffy");
    A.makeSound();
  }
}
```

Abstrakte Klassen dienen somit als Template für nachgeordnete Unterklassen. Neben Methoden können auch Properties und Indexer als abstrakt deklariert werden.

Warum macht es keinen Sinn eine abstrakte Klasse als `sealed` zu deklarieren?

Interfaces

Interfaces setzen die Idee der abstrakten Klassen konsequent fort und umfassen nur abstrakte Member. Sie bilden die Signatur einer Klasse, in der Methoden, Properties, Indexer und Events erfasst werden.

> Merke: Interfaces umfassen keine Felder!

Charakteristik von Interfaces:

- alle Bestandteile aus einem Interface müssen implementiert werden
- Klassen „implementieren“ Interfaces und „erben“ von Basisklassen
- Interfaces haben das Schlüsselwort `interface` und fangen im allgemeinen mit dem Buchstaben I an
- alle Elemente sind implizit `abstract` und `public`

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

interface IShape
{
  double Area();
  double Scope();
}

class Rectangular : IShape   // Rectangular implementiert das Interface IShape
{
  double area;
  double scope;
  public double Area() => area;
  public double Scope() => scope;
  public Rectangular(double sideA, double sideB)
  {
    area = sideA * sideB;
    scope = 2 * sideA + 2 * sideB;
  }
}
```

```
public class Program
{
  public static void Main(string[] args)
  {
    Rectangular rect = new Rectangular(2, 3);
    Console.WriteLine("Area: {0}, " + "Scope: {1}", rect.Area(), rect.Scope());
  }
}
```

Eine Klasse kann nur von einer anderen Klasse erben, aber beliebig viele Interfaces implementieren.

Schnittstellen werden verwendet:

- um eine lose Kopplung zu erreichen.
- um eine vollständige Abstraktion zu erreichen.
- um komponentenbasierte Programmierung zu erreichen
- um Mehrfachvererbung und Abstraktion zu erreichen.

Vererbung

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

interface IBaseInterface { void M(); }
interface IDerivedInterface : IBaseInterface { void N(); }

class A : IBaseInterface
{
  public void M()
  {
    Console.WriteLine("Methode M in {0}", this.GetType().Name);
  }
}

class B : IDerivedInterface
{
  public void M()
  {
    Console.WriteLine("Methode M in {0}", this.GetType().Name);
  }
  public void N()
  {
    Console.WriteLine("Methode N in {0}", this.GetType().Name);
  }
}

public class Program
{
  public static void Main(string[] args)
  {
    IBaseInterface t1 = new A();    // Statischer Typ IBaseInterface, dynamischer class A
    IBaseInterface t2 = new B();    // Statischer Typ IBaseInterface, dynamischer class B
    t1.M();
    t2.M();
    Console.WriteLine(t2 is IDerivedInterface);
    (t2 as IDerivedInterface).N();
    (t2 as B).N();
  }
}
```

Es besteht keine Vererbungshierarchie zwischen den beiden Klassen `A` und `B`! Vielmehr ergibt sich ein neuer Zusammenhang, die gemeinsame Implementierung eines Musters an Membern.

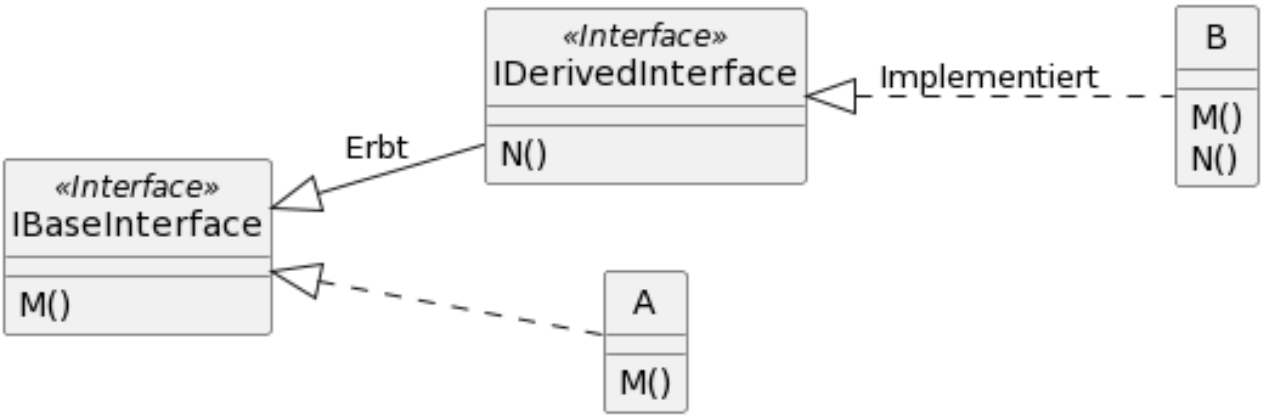

Figure 1: ClassStructure

Die Visualisierung von Klassen und deren Abhängigkeiten mit plantUML ist eine Möglichkeit einen raschen Überblick über bestimmte Zusammenhänge zu gewinnen. In den folgenden Materialien wird dies intensiv genutzt.

Interfaces vs. Abstrakte Klassen

interface	abstract class
viele Interfaces möglich	immer nur eine Basisklasse
speichert keine Daten	kann Felder umfassen
keine Konstruktorensignaturen	kann Konstruktoren umfassen
beinhaltet nur Methodensignaturen	kann Signaturen und Implementierungen integrieren
keine Zugriffsmodifizierer	beliebige Zugriffsmodifizierer
keine statischen Member	statische Member möglich

Merke: Interfaces geben keine Struktur vor, sondern nur ein Verhalten!

Bedeutung von Interfaces

Die C# Bibliothek implementiert eine Vielzahl von Interfaces, die insbesondere für die Handhabung von Datenstrukturen in jedem Fall genutzt werden sollten.

Informieren Sie sich unter Link über die wichtigsten davon wie:

- IEnumerable, IEnuerator
- IList
- IComparable
- ICollection
- …

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Cat: IComparable
{
    public string Name {get; set;}
    public int CompareTo(object obj)
    {
        if (!(obj is Cat))
        {
            throw new ArgumentException("Compared Object is not of Cat");
        }
        Cat cat = obj as Cat;
        return Name.CompareTo(cat.Name);
    }
}
```



```
public class Program
{
```

Auflösung von Namenskonflikten

```
using System;

interface IInterfaceA{
  void M();
}

interface IInterfaceB{
  void M();
}

public class SampleClass : IInterfaceA, IInterfaceB
{
    // Hier ist die zuordnung nicht eindeutig
    public void M()
    {
        Console.WriteLine("Gib irgendwas aus!");
    }
}

public class Program
{
  public static void Main(string[] args)
  {
    SampleClass sample = new SampleClass();
    sample.M();
    IInterfaceA A = sample;
    IInterfaceB B = sample;
    A.M();
    B.M();
  }
}
```

Wenn zwei Schnittstellenmember nicht dieselbe Funktion durchführen sollen muss diese separat implementiert werden. Hierzu wird ein Klassenmember erstellt, der sich explizit auf das Interface bezieht und den Namen der Schnittstelle benennt.

```
public class SampleClass : IInterfaceA, IInterfaceB
{
    // Hier ist die zuordnung nicht eindeutig
    void IInterfaceA.M()
    {
        Console.WriteLine("IInterfaceA - Gib irgendwas aus!");
    }

    void IInterfaceB.M()
    {
        Console.WriteLine("IInterfaceB - Gib irgendwas aus!");
    }
}
```

Allerdings kann diese Funktion dann nur über die Schnittstelle und nicht über die Klasse aufgerufen werden.

Aufgaben

- [ ] Setzen Sie sich mit den Konzepten von Interfaces auseinander!

!?Interfaces

!?Interfaces