C# - Typkonzept (OO-1)

(Angepasstes Beispiel für eine Ausarbeitung zum Proseminar)

Anand Kapadia
mail@anand.de

ABSTRACT

Dieses Beispieldokument zeigt den Umgang mit LaTeX. Diese Ausarbeitung ist eine überarbeitete Fassung eines Beitrags aus den vergangenen Jahren. Die Ausarbeitung liegt als Quelltest vor, sodass sie sowohl als Anleitung für den Umgang mit LaTeX als auch als Anschauungsobjekt dienen kann. Insbesondere wird allen Teilnehmern des Seminars/Proseminars empfohlen, den LaTeX-Code intensiv zu studieren. Die meisten der verwendeten Konzepte und Pakete sollten selbsterklärend sein – im Zweifel empfiehlt es sich jedoch, ein wenig zu „spielen" und auszuprobieren.

1. EINLEITUNG

Im Jahre 2001 präsentierte Microsoft im Rahmen seiner .NET-Strategie eine neue objektorientierte Programmiersprache: C#. Besonderes im Fokus neben einer automatischen Speicherbereinigung (Garbage Collection) standen eine einfache Handhabung und starke Typisierung. Hierfür griff man auf wesentliche Konzepte der Programmiersprachen Java, C++ und Delphi zurück, passte sie an und änderte bzw. modernisierte sie [12].

Gegenstand dieser Ausarbeitung ist es einen allgemeinen Überblick über das Typkonzept in C# zu vermitteln, um ein Verständnis für die Programmiersprache C# aufzubauen.

2. TYPKLASSIFIZIERUNG IN .NET

Prinzipiell unterscheidet man im .NET Framework zwischen zwei Typen: **Referenz**typen und **Wert**typen. Ein genereller Unterschied zwischen diesen beiden Typen liegt in der *Allozierung des Speichers*. Referenztypen werden auf dem Heap gespeichert. Lediglich ein Verweis, mit der Speicheradresse im Heap, wird auf dem Stack hinterlegt. Werttypen hingegen werden ausschließlich auf dem Stack gespeichert. Das folgende Listing 1 macht noch auf einen weiteren Unterschied aufmerksam. Will man nämlich ein Objekte erzeugen, so muss man bei Referenztypen den `new` - Operator verwenden; bei Werttypen kann eine Instanziierung mittels des `new` - Operators erfolgen, muss jedoch nicht.

```
// new - Operator notwendig
MyClass mc = new MyClass();
// hier optional
int i = new int();
i = 1;
// alternativ
int i = 1;
```

Listing 1: new-Operator

Zu dem unterscheiden sich Referenz- und Werttypen hinsichtlich des *Inhalts* von Variablen. Im Gegensatz zu Variablen von Werttypen, die die eigentlichen Daten speichern, werden bei Variablen von Referenztypen Verweise gespeichert. Logischerweise werden bei Vergleichs- oder Kopieroperationen die Werte (Werttypen) bzw. die Zeiger (Referenztypen) verglichen oder kopiert.

Auch bezüglich der `null` - Referenz unterscheiden sich diese beiden Typen. Während Objekte von Referenztypen den Wert `null` annehmen können, ist dies bei Objekten von Werttypen eigentlich nicht möglich. Dies kann umgangen werden, indem ein sogenannter `null` - fähiger Werttyp erstellt wird. Dazu wird entweder der ? - Modifikators oder der generische Typ `Nullable<T>` benötigt (vgl. Listing 2: null-fähiger Werttyp). Genaueres zu diesem Thema lässt sich in [5] nachlesen.

```
// ? Modifikator
int? myDBInt = null;

//generischer Typ Nullable<T>
Nullable<int> myDBInt =
        new Nullable<int>();
```

Listing 2: null-fähiger Werttyp

Zu guter Letzt weisen Referenz- und Werttypen einen Unterschied bezüglich einer weiteren Ableitung (vgl. Abschnitt 3.3) auf. Anders als bei Referenztypen, können neue Typen nicht von einem Werttyp abgeleitet werden.

3. REFERENZTYP

Im folgenden Kapitel werden die Referenztypen in C# behandelt. Dabei kann man Referenztypen kategorisieren in:

- Klassen
- Schnittstellen
- Felder
- Delegate

3.1 Klassen

Klassen sind die wohl meist verwendeten Referenztypen in C#. Sie werden durch das Schlüsselwort (auch: reserviertes Wort) `class` gekennzeichnet. Klassen sind *benutzerdefinierte Typen*, d.h. ein Programmierer fasst Variablen anderer Typen, sowie Methoden und Ereignisse innerhalb einer Klasse zusammen. Die Bestandteile einer Klasse bezeichnet man als *Mitglieder* (engl.: *member*). Die Member einer Klasse sind:

- Methoden (engl.: *method*)
- Eigenschaften (engl.: *property*)

- Ereignisse (engl.: *event*)
- Konstruktoren (engl.: *constructor*) und
- Felder (engl.: *field*).

Listing 3 zeigt, wie eine Klasse `Person` exemplarisch in C# implementiert werden kann.

```
public class Person
{
    // Feld: In name wird der Name
    // der Person gespeichert.
    private string name;

    //Eigenschaften
    public string Name
    {
        set {name = value;}
        get {return name;}
    }

    // Konstruktor
    public Person() { }

    // Methode
    public void Eat() { }
}
```

Listing 3: Eine Klasse Person

Die Klasse `Person` besitzt ein Feld `name` vom Typ String, in dem der Name einer Person gespeichert wird. Zusätzlich besitzt sie eine Eigenschaft `Name`, bestehend aus einem set- und get-Accessor. Mit dem set-Accessor kann ein Wert zugewiesen werden, während der get-Accessor einzig den Wert von `name` zurückgibt. Das Schlüsselwort `value` ist implizit Parameter des set-Accessors und definiert den Wert von name. Mit dem Konstruktor, der genauso heißen muss wie die Klasse selbst - also in diesem Fall `Person` - werden in der `Main`-Methode neue Objekte vom Typ `Person` erzeugt. Schlussendlich besitzt die Beispielklasse `Person` noch eine Methode `Eat()` ohne Rückgabewert (gekennzeichnet durch das Schlüsselwort `void`).

3.2 Zugriffsmodfikatoren

Mit Hilfe von Zugriffsmodifikatoren kann man festlegen, inwieweit eine Klasse und ihre jeweiligen Member für andere Klassen verfügbar sind. Hierbei zu beachten ist, dass ähnlich wie bei Klassen, Zugriffsmodifikatoren reservierte Wörter sind. C# verfügt über fünf Zugriffsmodifikatoren:

- `public`: auf den Typ oder Typmember kann öffentlich zugegriffen werden, d.h. er obliegt *keinerlei Beschränkungen*
- `private`: Typmember sind *ausschließlich innerhalb der Klasse sichtbar*
- `protected`: ähnlich wie private, aber *auch abgeleitete Klassen* können auf Typ oder Typmember zugreifen
- `internal`: Typmember sind nur *innerhalb einer Assembly*, d.h. in einer exe.- oder dll.-Datei, nutzbar
- `protected internal`: Kombination der Zugriffsmodifikatoren `protected` und `internal`, d.h. *Klassen der selben Assembly, sowie abgeleitete Klassen in anderen Assemblies* haben Zugriff auf das jeweilige Mitglied.

Ein Zugriffsmodifikator, der festlegt, dass nur abgeleitete Klassen innerhalb einer Assembly Zugriff auf einen Typmember haben, existiert nicht in C#.

3.3 Vererbung

Vererbung ist ein grundlegendes Merkmal der Objektorientierung. Unter Vererbung versteht man die *Wiederverwendung bzw. Erweiterung einer Klasse*, d.h. aus einer bestehende Klasse, der sogenannten *Basisklasse*, wird eine neue Klasse (*abgeleitete Klasse* oder *Subklasse*) erzeugt. Die abgeleitete Klasse stellt somit eine *Sonderform der Basisklasse* dar, zu mal sie weitere Felder oder Methoden besitzt.

Genauso wie Java unterstützt C# die *einfache Vererbung*. Somit wird festgelegt, dass eine Klasse *maximal eine Basisklasse* besitzen darf. Jedoch kann sie *mehrere Schnittstellen* (engl.: *interfaces*, Abschnitt 3.3.4) implementieren.

Im Gegensatz zu Java wird in C# die Vererbung mittels des „ `:` “ - Operators dargestellt. Dieser hat direkt nach dem Klassennamen zu stehen. Danach folgt an erster Stelle die Basisklasse, von der die abgeleitete Klasse erbt, gefolgt von sämtlichen Interfaces, die zusätzlich noch implementiert werden. Wird für eine Klasse nicht explizit eine Basisklasse angegeben, so erbt diese implizit von der Klasse `object`. Folglich erben sämtliche Klasse direkt oder indirekt von der Klasse `object` [3].

Im folgenden Beispiel ist die Klasse `X` Basisklasse der Klasse `Y` und stellt dieser sämtliche sichtbaren Mitglieder zur Verfügung. Außerdem implementiert `Y` die Schnittstellen `IA` und `IB`.

```
class X
{
  ...
}

interface IA
{
  void someMethod(){ ... }
}

interface IB
{
  void someOtherMethod(){ ... }
}

class Y : X,IA,IB
{
  ...
  void someMethod(){ ... }
  void someOtherMethod(){ ... }
}
```

Listing 4: Vererbung

3.3.1 Modfizierer: virtual und override

C# stellt Programmieren die Möglichkeit zur Verfügung, Methoden in abgeleiteten Klassen zu überschreiben. Hierfür ist ein Konstrukt der beiden Modifizierer `virtual` und `override` zwingend notwendig. `virtual` kann auf Methoden, Eigenschaften, Ereignisdeklarationen und Indexer angewandt werden. Dies hat zur Folge, dass sie in einer abgeleiteten Klasse neu definiert werden können.

Das Überschreiben einer virtuellen Methode erfolgt dann mit dem `override`-Modifizierer. Im Gegensatz zu Java, wo lediglich ein @override Tag reicht, muss in C# explizit mit dem `override`-Modifizierer angegeben, dass eine Methode überschrieben wird. Dieser kann auch nur auf als virtuelle (bzw. abstrakte) Methoden verwendet werden.

Im folgenden Beispiel werden die beiden Methoden `F()` und `F2()` der Klasse `X` zunächst als `virtual` deklariert und

anschließend in der abgeleiteten Klasse `Y` überschrieben.

```
class X
{
  // virtuelle Methode F()
  protected virtual void F()
  {
    Console.WriteLine("X.F");
  }
}

class Y : X
{
  // Überschreiben der Methode F()
  protected override void F()
  {
    Console.WriteLine("Y.F");
  }
}
```

Listing 5: virtual und override (Beispiel nach [10])

3.3.2 Modifzierer: sealed

Ein Gegensatz zu dem Modifzierer `virtual` stellt der Modifzierer `sealed` dar. `sealed` ist das Äquivalent zu Javas `final`, d.h. mit dem Modifzierer `sealed` kann man Klassen, sowie Methoden oder Eigenschaften versiegeln. Konkret bedeutet das: wird `sealed` auf eine Klasse angewandt, so steht diese Klasse anderen Klassen nicht mehr als Basisklasse zur Verfügung. Somit können andere Klassen nicht mehr von dieser Klasse erben.

Wird der `sealed` - Modifizierer auf Methoden oder Eigenschaften verwendet, kann die Klasse zwar immer noch als Basisklasse dienen, jedoch können ihre versiegelte Methoden oder Eigenschaften in abgeleitete Klassen nicht mehr überschrieben werden (s. Listing 6).

```
class X
{
  protected virtual void F() {
    Console.WriteLine("X.F");
  }
  protected virtual void F2() {
        Console.WriteLine("X.F2");
  }
}

class Y : X
{
  // F() wird nun versiegelt und
  // darf nicht mehr überschrieben werden
  sealed protected override void F() {
        Console.WriteLine("Y.F");
  }
  protected override void F2() {
        Console.WriteLine("Y.F2");
  }
}

class Z : Y
{
  // Methode F kann nicht überschrieben
  // werden, sonst compiler error
  // F2 darf Überschrieben werden
  protected override void F2() {
        Console.WriteLine("Z.F2");
  }
}
```

Listing 6: sealed (Beispiel nach [10])

3.3.3 Abstrakte Klassen

Abstrakte Klassen stellen Sonderformen von Klassen dar. Sie werden durch das Schlüsselwort `abstract`, welches vor `class` zu stehen hat, deklariert. Im Gegensatz zu einer konkreten („normalen“) Klasse, sind abstrakte Klassen nicht *direkt* instanziierbar. Sie stellen somit lediglich eine Definition einer Basisklasse bereit. Da es sich bei abstrakten Klassen ebenfalls um Klassen handelt, gilt auch hier: eine abgeleitete Klasse kann maximal von einer abstrakten Klasse erben.

Auch Methoden können als `abstract` deklariert werden. In diesem Fall wird das Schlüsselwort `abstract` vor dem Rückgabetyp angegeben. Hierbei zu beachten ist einerseits, dass *abstrakte Methoden nur in abstrakten Klassen* vorkommen und andererseits, dass sie einzig eine Methodensignatur beinhalten und *keine* Implementierung (siehe Listig 7). Folglich können abstrakte Methoden in unterschiedlichen Klassen, die von einer abstrakten Klasse erben, unterschiedlich realisiert werden. Außerdem muss eine abgeleitete Klasse sämtliche abstrakte Methoden einer abstrakten Klasse implementieren. Das Überschreiben erfolgt hierbei mit dem Modifizierer `override`.

3.3.4 Schnittstellen

Ein Referenztyp, der vor allem im Bezug mit Vererbung interessant ist, ist die Schnittstelle (engl.: *interface*). Schnittstellen werden durch das reservierte Wort `interface` deklariert und können, ähnlich wie abstrakte Klassen, nicht direkt instanziiert werden. Zu dem enthalten sie nur Signaturen von Methoden, Eigenschaften oder Ereignissen. Der Unterschied zwischen Schnittstellen und abstrakten Klassen ist jedoch, während eine abstrakte Klasse auch eine implementierte Methode besitzen kann, enthält eine Schnittstelle *ausschließlich* abstrakte Methoden. Des Weiteren können neben Klassen auch andere Schnittstellen, sowie Strukturen (vgl. Abschnitt 4.2) mehr als eine Schnittstelle implementieren. Per Konvention ist festgesetzt, dass Namen von Schnittstellen immer mit „I“ beginnen.

```
interface IControl {
  void Paint(); }
interface ISurface {
  void Paint(); }

public class SampleClass : IControl,
    ISurface {
  // Paint() aus Schnittstelle IControl
  void IControl.Paint() {
    System.Console.WriteLine("IControl.
        Paint");
  }
  // Paint() aus Schnittstelle ISurface
  void ISurface.Paint() {
    System.Console.WriteLine("ISurface.
        Paint");
  }
}

class TestClass {
  static void Main(string[] args){
    SampleClass sc = new SampleClass();
    // die Methode Paint() von der
    // Schnittstelle IControl wird
    // aufgerufen - Cast notwendig
    ((IControl)sc).Paint();
  }
}
```

Listing 8: Schnittstellen und ihre Implementierung

```
abstract class ShapesClass
{
    public abstract int Area();

    public void SuperClassMethod(){
        System.Console.WriteLine("My␣super␣class␣is␣shapes␣class.");
    }
}

class Square : ShapesClass
{
    int side = 0;

    public Square(int n)
    {
        side = n;
    }
    // Methode Area muss implementiert werden, sonst compile-time
    // error
    public override int Area()
    {
        return side * side;
    }
    // Methode SuperClassMethod muss ebenfalls implementiert werden
    public void SuperClassMethod(){
        System.Console.WriteLine("My␣super␣class␣is␣shapes␣class.");
    }
}
```

Listing 7: Abstrakte Klasse

Zusätzlich wird in C# die Möglichkeit angeboten, Schnittstellen explizit zu implementieren. Dies wird am folgenden Beispiel deutlich. Sowohl das `interface IControl` als auch `ISurface` bieten die Methode `Paint()` an. Die Klasse `Sample Class` implementiert nun beide Schnittstellen. Eine Implementierung der Methode `Paint()` hätte zur Folge, dass sowohl die Schnittstelle `IControl` als auch `ISurface` als ihrige benutzen. Soll aber `Paint()` unterschiedliche Funktionen übernehmen, handelt es sich in diesem Fall um eine falsche Implementierung. Stattdessen kann man explizit angeben, aus welcher Schnittstelle die Methode `Paint()` implementiert wird (durch explizite Angabe der Schnittstelle, z.B. `ISurface.Paint()` oder `IControl.Paint()`) [8].

3.3.5 Abstrakte Klasse vs. Schnittstelle

Sowohl abstrakte Klassen als auch Schnittstellen trennen Vertrag (bezeichnet die Abmachung, dass eine erbende Klasse sämtliche Member einer abstrakten Klasse/Schnittstelle implementiert) und Implementierung. Jedoch sind abstrakte Klassen flexibler als Schnittstellen, da neue Funktionalität einfacher hinzugefügt werden können. Änderungen oder Erweiterungen bei Schnittstellen sind hingegen mit wesentlich mehr Arbeitsaufwand verbunden, da auch abgeleitete Klassen dementsprechend angepasst werden müssen.

Da abstrakte Klassen und Schnittstellen sich im Bezug auf Vererbung ähneln, wirft sich nun die Frage auf, wann man das eine und wann man das andere verwenden soll. Grundsätzlich gilt:

- bei „Ist-Ein(e)“ – Beziehungen (z.B. abstrakte Klassen: Objekte von Square sind Shapes) bietet sich eine abstrakte Klasse an
- bei „Objekt-Kann“ – Beziehungen sollte man auf Schnittstellen zurückgreifen [2]
- ist ein Vertrag voraussichtlich über längeren Zeitraum stabil, d.h. die Basisklasse einer Klasse ändert sich nicht, kann man Schnittstellen verwenden; ist dies nicht der Fall sollte eine abstrakte Klasse in Betracht gezogen werden
- soll ein Typ mehrere Verträge implementieren, d.h. man will eine Mehrfachvererbung simulieren, kann dies mit einer Schnittstelle erfolgen

3.4 Felder

Abgesehen von Klassen und Schnittstellen gibt es einen weiteren Referenztypen: Felder (engl.: *array*). Arrays sind Objekte bzw. Datenstrukturen, die verwendet werden, ummehrere typgleiche Variablen oder Objekte zu speichern und zu verwalten. Ein Array kann dabei eindimensional, mehrdimensional oder verzweigt (vgl. Listings 9 bzw. 10) sein.

```
// ein eindimensionales Array
int[] arrayA;

// ein mehrdimensionales Array
int[,] mulDimArrayA;
```

Listing 9: Ein- und Mehrdimensionale Arrays

Arrays müssen mit dem `new`-Operator instanziiert werden:

```
int[] arrayB = new int[3];
// Zwei dimensionales Array wird
// deklariert mit Länge 3 bzw. 4
int[,] mulDimArrayA = new int[3, 4];
```

Dabei wird nochmals der Typ angegeben und zusätzlich noch die Elementzahl. Des Weiteren gilt für ein Array der Länge n: die Indizierung eines Arrays beginnt bei 0 und endet bei $n - 1$. Natürlich besteht aber auch die Möglichkeit ein Feld direkt mit einer Wertliste zu initialisieren. Hierbei

zu beachten ist, dass die Werte nicht in eckigen, sondern in geschweiften Klammern zu stehen haben. Auf die einzelnen Arrayelemente kann man mit Hilfe des Indexes zugreifen.

```
// Beispiel für ein verzweigtes Array
int[][] jaggedArray = new int[4][];

jaggedArray[0] = new int[]{1,2,3};
jaggedArray[1] = new int[]{4,5,6};
jaggedArray[2] = new int[]{7,8,9,10};
jaggedArray[3] = new int[]{11,12,13,14};

// Array wird deklariert mit Werten 1,2,3
int[] arrayC = new int[] {1, 2, 3};

// Alternative Schreibweise
int[] arrayD = {1, 2, 3};

// der Wert an der ersten Stelle im Array
// arrayD wird nun auf zwei gesetzt
arrayD[0] = 2; // arrayD = {2,2,3}

// Zwei-Dim. Array mit Werten 1,2,3
// bzw. 4,5,6,7 wird deklariert
int[,] mulDimArrayB = {{1,2,3},{4,5,6,7}};

// string-Array mit den Wochentagen
string[] daysOfWeek = {"Mon","Tue",
        "Wed","Thu","Fri","Sat","Sun"};
```

Listing 10: Verzweigtes Array und Wertzuweisung

Da Felder Referenztypen sind, gilt auch hier: bei der Zuweisung zweier Arrayvariablen wird nur ein Verweis auf die Arrayvariable hinterlegt und nicht der Wert kopiert.

```
int[] arrayE = {1, 2, 3, 4, 5};

// weitere Referenz auf das Feld arrayE
int[] arrayF = arrayE;
```

Listing 11: Referenzzuweisung bei Arrays

Um Arrayelemente zu kopieren muss man die Methode `Clone()`, die jedes Feld besitzt, da die Schnittstelle `ICloneable` implementiert wird, verwenden. Alternativ kann auch die Methode `Copy` verwendet werden. Beim folgenden Listing sollte auf die unterschiedliche Realisierung der `Copy`-Methode beachtet werden.

```
// mit Clone()
int[] source = {1, 2, 3, 4, 5};

// target = {1,2,3,4,5}
int[] target = (int[])source.Clone();

// mit Copy()
int[] target2 = new int[5];
Array.Copy(source, target2, 5);

foreach (int value in target2)
{
        Console.WriteLine(value);
}

// Ausgabe:
// 1
// 2
// 3
// 4
// 5

// man kann auch bestimmte
// Arraysequenzen mit Copy() kopieren
int[] target3 = new int[3];
Array.Copy(source, 0, target3, 0, 3);

foreach (int value2 in target3)
        {
            Console.WriteLine(value2);
        }

// Ausgabe:
// 1
// 2
// 3
```

Listing 12: Wertkopie bei Arrays

Im Zusammenhang mit Arrays muss man zusätzlich noch berücksichtigen, dass C# keine dynamischen Arrays unterstützt, d.h. die Länge eines Arrays kann nach der Instanzierung nicht mehr verändert werden [6].

3.5 Delegate

Delegate sind Referentypen, die eine Methodensignatur beschreiben. Sie können für Referenzen auf Methoden verwendet werden und bilden die Grundlage für Ereignisse. Da C# von C++ abgeleitet wurde, weisen Delegates in C# Ähnlichkeiten mit den Funktionszeigern in C++ auf, sind jedoch objektorientierter bzw. „typsicherer" [7]. Listing 13 verdeutlicht das Prinzip von Delegates. Im Listing 13 wird zunächst ein Delegate `Del` erzeugt. Dieser kapselt ausschließlich Methoden, die zwei Strings als Parameter entgegennehmen und dabei als Rückgabetyp `void` besitzen. Anschließend werden zwei Methoden implementiert, die zwei Strings entgegennehmen und diese ausgeben. Hierbei zu beachten ist, dass die Methode `methodA` zunächst das erste Argument ausgibt und dann das zweite, während die Methode `methodB` zuerst das zweite Argument ausgibt und danach das erste.

In der `Main`-Methode der Klasse `MainClass` werden zwei Objekte `myDel1` und `myDel2` vom Typ `Del` angelegt, die die Methode `methodA` bzw. `methodB` als Parameter übergeben bekommen. Somit wird festgelegt, auf welche Methoden die Delegates `myDel1` bzw. `myDel2` verweisen. Nun erhalten die beiden Delegate-Objekte zwei identische Strings („Hello" und „World"). Da `myDel1` auf die Methode `methodA` verweist wird logischerweise „Hello World" ausgegeben. `myDel2` hingegen verweist auf `methodB`, weshalb „World Hello" auf der Konsole erscheint.

4. WERTTYPEN

Der folgende Abschnitt befasst sich mit Werttypen in C#. Werttypen lassen sich kategorisieren in:

- Basistypen
- Strukturen
- Enumerationen

4.1 Basistypen

C# bietet alle Datentypen an, die auch in Java angeboten werden (`char`, `byte`, `short`, `int`, `double`, `float`, `long`). Eine Ausnahme bildet dabei Javas `boolean`, welches in C# `bool` heißt. Zusätzlich gibt es noch `sbyte`, `uint`, `ushort`, `ulong` und `decimal`. Das „u" bzw. „s" steht dabei für „unsigned" bzw. „signed". Bei den Gleitkommatypen (`double`, `float`,

```
// Delegate Del kapselt eine Methode
public delegate void Del(string s1, string s2);

class TestClass{
  //methodA gibt erst s1 dann s2 aus, wichtig: muss gleiche Parameter haben wie Del
  private void methodA(string s1, string s2) {
     System.Console.WriteLine("␣"+ s1 + "␣" + s2);
  }
  //methodB gibt erst s2 dann s1 aus, wichtig: muss gleiche Parameter haben wie Del
  private void methodB(string s1, string s2) {
     System.Console.WriteLine("␣" + s2 + "␣" + s1);
  }

}

class MainClass{
  static void Main(string[] args) {
    // Objekt myDel1 des Delegates Del wird erzeugt mit Parameter methodA
    Del myDel1 = new Del(methodA);
    myDel1("Hello", "World");  // Ausgabe: Hello World
    // Objekt myDel2 des Delegates Del wird erzeugt mit Parameter methodB
    Del myDel2 = new Del(methodB);
    myDel2("Hello", "World");  // Ausgabe: World Hello
  }
}
```

Listing 13: Delegate (Beispiel nach [4])

`decimal`) muss der entsprechende Suffix (d für `double`, f für `float` und m für `decimal`) angegeben werden, da sonst von einem `double` ausgegangen wird. Eine Übersicht der einzelnen Werftenbereiche gibt Tabelle 1.

4.2 Strukturen

Neben Klassen sind Strukturen ein wichtiger Bestandteil des .NET Frameworks. Strukturen werden mit dem Schlüsselwort `struct` deklariert und können Konstruktoren, Konstanten, Felder, Methoden, Eigenschaften, Indexer, Operatoren, Ereignisse und geschachtelte Typen beinhalten. Im Gegensatz zu Klassen können Strukturen *nur Schnittstellen implementieren* und *nicht von anderen Strukturen bzw. Klassen erben*. Deshalb dürfen Strukturmember nicht als `protected` definiert werden. Ein weiterer Unterschied zu Klassen ist, dass Strukturen keinen parameterlosen Konstruktor besitzen dürfen. Dies würde zu einem Fehler führen. Auch besitzen Strukturen nur statische oder konstante Felder.

```
public struct Point {
 public double x;
 public double y;

 // Konstruktor
 public Point(double x, double y) {
  this.x = x;
  this.y = y;
 }
 // Methode
 public override string ToString() {
  return(string.Format(
        "x:␣{0},␣y:␣{1}", x, y));
 }
}
```

Listing 14: Struktur Point

Ein Beispiel, wie die Struktur `Point` implementiert werden kann, zeigt Listing 14. Dabei speichert die Variable x die x-Koordinate bzw. die Variable y die y-Koordinate des Punktes. Mit dem Konstruktor kann in der `Main`-Methode ein Objekt vom Typ `Point` erzeugt. Zusätzlich besitzen alle Objekte von `Point` eine `ToString`-Methode, die die Werte von x und y wiedergibt.

Man bemerkt, dass der Entwurf einer Klasse und einer Struktur nahezu identisch ist. Da Werttypen ressourcenschonender sind als Referenztypen - denn sie werden nur auf dem Stack gespeichert - mag sich nun der Gedanke verfestigen, dass es sinnvoller sei eine Struktur statt einer Klasse zu verwenden. Im Allgemeinen werden Strukturen verwendet um kleine, als Faustregel gilt: ein Objekt ist kleiner als 16 Bytes, und kurzlebige Typen darzustellen, d.h. kompakte Objekte und einfache Datenstrukturen, wie z.B. Points [9].

Will man komplexere Datenstrukturen abbilden oder soll das Objekt ein komplexes Verhalten vorweisen oder ist vorgesehen, dass die Datenstruktur vererben soll, so empfiehlt sich eine Klasse.

4.3 Enumerationen

Enumerationen (oder kurz: Enum) sind Werttypen, welche durch das reservierte Wort `enum` eingeleitet werden. Enums bezeichnen die Aufzählung einer Gruppe benannter Konstanten, die sogenannte Enumeratorliste. Am besten werden sie innerhalb eines Namespaces oder ggf. auch in Klassen oder Strukturen geschachtelt. Eine Aufzählung beginnt standardmäßig mit dem Wert 0 und wird pro Konstante jeweils um 1 erhöht. Außerdem ist der standardmäßige Typ bei Enums `int`:

```
{
 A ,   // 0
 B ,   // 1
 C ,   // 2
 D     // 3
}
```

Alternativ können auch `byte`, `sbyte`, `short`, `ushort`, `uint`, `long` oder `ulong` benutzt werden. C# unterstützt eine Mani-

Datentyp	.NET Laufzeittyp	Wertebereich
bool	System.Boolean	true oder false
byte	System.Byte	0 bis 255
sbyte	System.SByte	-128 bis 127
char	System.Char	U+0000 bis U+ffff
short	System.Int16	-32.768 bis 32.767
ushort	System.UInt16	0 bis 65.535
int	System.Int32	-2.147.483.648 bis 2.147.483.647
uint	System.UInt32	0 bis 4.294.967.295
float	System.Single	-3.402823e38 bis 3.402823e38
long	System.Int64	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
ulong	System.UInt64	0 bis 18.446.744.073.709.551.615
double	System.Double	-1.79769313486232e308 bis 1.79769313486232e308
decimal	System.Decimal	(-7.9 x 1028 bis 7.9 x 10^{28}) / ($10^{0\ bis\ 28}$)

Table 1: Basistypen in C# und ihre Wertebereiche

pulation der Enumeratorliste, d.h. einzelne Konstanten können mit einem bestimmten Wert belegt werden. Das folgende Glied der Enumeratorliste besitzt, falls nicht manipuliert, einen um eins erhöhten Wert:

```
{
  Mon   =  1 ,
  Tue ,     // 2
  Wed ,     // 3
  Thu ,     // 4
  Fri ,     // 5
  Sat ,     // 6
  Sun       // 7
}
```

5. ZUSAMMENFASSUNG

Obwohl C# eine relativ neue Programmiersprache ist, stellt man alles in allem fest, dass C# wesentliche Konzepte aus den bisherigen Programmiersprachen, wie z.B. die Objektorientierung, eine einfache Vererbung wie in Java, etc. beibehalten hat.

Durch das Einführen neuer Schlüsselwörter (`virtual` und `override`) oder das Speichern von Objekten am Stack durch Strukturen wird zu dem Code „lesbarer“ und Programme ressourcensparender. Schließlich wurden Schwachstellen, beispielsweise die Funktionszeiger in C++, in ähnlicher Form umgesetzt und typensicher gemacht.

6. REFERENCES

[1] Dieter Bremes Arne Schäpers, Rudolf Huttary. *C# : Windows- und Web-Programmierung mit Visual Studio .NET*. Markt+Technik Verlag, 2002.

[2] Gregor Raýman Bernhard Lahres. Praxisbuch objektorientierung. http://openbook.galileocomputing.de/oo/, 17.11.2011.

[3] Eric Gunnerson. C# - galileo openbook. http://openbook.galileocomputing.de/csharp/kap01.htm#t22 , 20.11.2011.

[4] hinzberg.net. Delegates. http://www.hinzberg.net/csharp/csharp/csharp/delegates.html, 01.12.2011.

[5] Stephen Teilhet Jay Hilyard. *C# Kochbuch*. O'Reilly, 2006.

[6] MSDN Library. Arrays (c#-programmierhandbuch). http://msdn.microsoft.com/de-de/library/9b9dty7d(v=VS.100).aspx, 16.11.2011.

[7] MSDN Library. delegate (c#-referenz). http://msdn.microsoft.com/de-de/library/900fyy8e(v=VS.100).aspx, 16.11.2011.

[8] MSDN Library. Explizite schnittstellenimplementierung (c#-programmierhandbuch). http://msdn.microsoft.com/de-de/library/ms173157(v=VS.100).aspx, 22.11.2011.

[9] MSDN Library. Klassen und strukturen (.net framework 4). http://msdn.microsoft.com/de-de/library/ms229017.aspx, 01.12.2011.

[10] MSDN Library. override (c#-referenz). http://msdn.microsoft.com/de-de/library/ebca9ah3(v=VS.100).aspx, 16.11.2011.

[11] MSDN Library. Typen (c#-referenz). http://msdn.microsoft.com/de-de/library/3ewxz6et.aspx, 16.11.2011.

[12] Wikipedia. C-sharp. http://de.wikipedia.org/wiki/C-Sharp , 20.11.2011.