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

Delegaten

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2021`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Deleganten - Konzepte und  Anwendung`
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/21_Delegaten.md
Autoren	@author

Motivation und Konzept der Delegaten

Ihre Aufgabe besteht darin folgendes Code-Fragment so umzuarbeiten, so dass unterschiedliche Formen der Nutzer-Notifikation (neben Konsolenausgaben auch Emails, Instant-Messager Nachrichten, Tonsignale) möglich sind. Welche Ideen haben Sie dazu?

```
using System;
using System.Reflection;
using System.Collections.Generic;

public class VideoEncodingService{

  private string userId;
  private string filename;

  public VideoEncodingService(string filename, string userId){
     this.userId = userId;
     this.filename = filename;
  }

  public void StartVideoEncoding(){
     Console.WriteLine("The encoding job takes a while!");
     NotifyUser();
  }

  public void NotifyUser(){
      Console.WriteLine("Dear user {0}, your encoding job {1} was finished",
                        userId, filename);
  }
```

```
}

public class Program{
  public static void Main(string[] args){
     VideoEncodingService myMovie = new VideoEncodingService("007.mpeg", "12321");
     myMovie.StartVideoEncoding();
  }
}
```

Gegen das Hinzufügen weiterer Ausgabemethoden in die Klasse `VideoEncodingService` spricht die Tatsache, dass dies nicht deren zentrale Aufgabe ist. Eigentlich sollte sich die Klasse gar nicht darum kümmern müssen, welche Art der Notifikation genutzt werden soll, dies sollte dem Nutzer überlassen bleiben.

Folglich wäre es sinnvoll, wenn wir `StartVideoEncoding` eine Funktion als Parameter übergeben könnten, die wir unabhängig von der eigentlichen Klasse definiert haben.

```
  public void TriggerMe(){
    // TODO
  }

  VideoEncodingService myMovie = new VideoEncodingService("007.mpeg",
                                                           "12321",
                                                           triggerMe);
```

In C würden wir an dieser Stelle von einem Funktionspointer sprechen.

```
// https://www.geeksforgeeks.org/function-pointer-in-c/

#include <stdio.h>
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}

void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}

void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
           "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

Grundidee

> Merke: Ein Delegat ist eine Methodentyp und dient zur Deklaration von Variablen, die auf eine Methode verweisen.

Für die Anwendung sind drei Vorgänge nötig:

1. Anlegen des Delegaten (Spezifikation einer Signatur)
2. Instantiierung (Zuweisung einer signaturkorrekten Methode)
3. Aufruf der Instanz

```
// Schritt 1
//[Zugriffsattribut] delegate Rückgabewert DelegatenName(Parameterliste);
public delegate int Rechenoperation(int x, int y);

static int Addition(int x, int y){
  return x + y;
}

static int Modulo(int dividend, int divisor){
  return divident % divisor;
}

// Schritt 2 - Instanzieren
Rechenoperation myCalc = new Rechenoperation(Addition);
// oder
Rechenoperation myCalc = Addition;

// Schritt 3 - Ausführen
myCalc(7, 9);
```

Lassen Sie uns dieses Konzept auf unsere `VideoEncodingService`-Klasse anwenden.

```
using System;
using System.Reflection;
using System.Collections.Generic;


// Schritt 1
public delegate void NotifyUser(string userId, string filename);

public class VideoEncodingService
{
  private string userId;
  private string filename;

  public VideoEncodingService(string filename, string userId){
     this.userId = userId;
     this.filename = filename;
  }

  public void StartVideoEncoding(NotifyUser notifier){
     Console.WriteLine("The encoding job takes a while!");
      // Schritt 3
      notifier(userId, filename);
  }
}

public class Program
{
  // Die Notifikationsmethode ist nun Bestandteil der "Nutzerklasse"
  public static void NotifyUserByText(string userId, string filename){
    Console.WriteLine("Dear user {0}, your encoding job {1} was finished",
                      userId, filename);
```

```
  }

  public static void Main(string[] args){
     VideoEncodingService myMovie = new VideoEncodingService("007.mpeg", "12321");
     // Schritt 2
     NotifyUser notifyMe = new NotifyUser(NotifyUserByText);
     myMovie.StartVideoEncoding(notifyMe);
  }
}
```

Was passiert hinter den Kulissen?

Was wird anhand des Aufrufes

`NotifyUser notifyMe = new NotifyUser(NotifyUserByText);`

deutlich? Handelt es sich bei `notifyMe` wirklich nur um eine Methode?

Delegattypen werden von der `Delegate`-Klasse im .NET Framework abgeleitet.

https://docs.microsoft.com/de-de/dotnet/api/system.delegate?view=netframework-4.8

Wenn der C#-Compiler Delegiertentypen verarbeitet, erzeugt er automatisch eine versiegelte Klassenableitung aus `System.MulticastDelegate`. Diese Klasse (in Verbindung mit ihrer Basisklasse, `System.Delegate`) stellt die notwendige Infrastruktur zur Verfügung, damit der Delegierte eine Liste von Methoden vorhalten kann. Der Compiler erzeugt insbesondere drei Methoden, um diese aufzurufen:

- die synchrone Invoke()-Methode, die aber nicht explizit von Ihrem C#-Code aufgerufen wird
- die asynchrone `BeginInvoke()` und
- `EndInvoke()` als Methoden, die die Möglichkeit bieten, die die eigentliche Methode in einem separaten Ausführungsthread zu handhaben.

Entsprechend der Codezeile `delegate int Transformer(int x);` generiert der Compiler eine spezielle `sealed class Transformers`

```
sealed class Transformer : System.MulticastDelegate
{
  ...
  public int Invoke(int x);
  public IAsyncResult BeginInvoke(int x, AsyncCallback cb, object state);
  public int EndInvoke(IAsyncResult resut);
  ...
}
```

Seit C# 2.0 ist die Syntax für die Zuweisung einer Methode an eine Delegate-Variable vereinfacht. Statt

`NotifyUser notifyMe = new NotifyUser(this.NotifyUserByText);`

kann nunmehr auch

```
NotifyUser notifyMe = this.NotifyUserByText;
NotifyUser notifyMe = NotifyUserByText;
```

verwendet werden.

Multicast Delegaten

Sollten wir uns mit dem Aufruf einer Methode zufrieden geben?

```
using System;
using System.Reflection;
using System.Collections.Generic;


public class Program
{
  delegate int Calc(int x, int y);
```

```
  static int Add(int x, int y){
      Console.WriteLine("x + y");
      return x + y;
  }

  static int Multiply(int x, int y){
      Console.WriteLine("x * y");
      return x * y;
  }

  static int Divide(int x, int y){
      Console.WriteLine("x / y");
      return x / y;
  }

  public static void Main(string[] args){
    // alte Variante
    // Calc computer1 = new Calc(Divide);
    // neue Variante:
    // Calc computer2 = Divide;
    Calc computer3 = Add;
    computer3 += Multiply;
    computer3 += Multiply;
    computer3 += Divide;
    computer3 -= Add;
    Console.WriteLine("Zahl von eingebundenen Delegates {0}",
                      computer3.GetInvocationList().GetLength(0));
    Console.WriteLine("Ergebnis des letzten Methodenaufrufes {0}",
                                                        computer3(15, 5));
  }
}
```

Merke: Der Rückgabewert des Aufrufes entspricht dem der letzten Methode.

Schnittstellen vs. Delegaten

```
void delegate XYZ(int p);

interface IXyz {
    void doit(int p);
}

class One {
    // All four methods below can be used to implement the XYZ delegate
    void XYZ1(int p) {...}
    void XYZ2(int p) {...}
    void XYZ3(int p) {...}
    void XYZ4(int p) {...}
}

class Two : IXyz {
    public void doit(int p) {
        // Only this method could be used to call an implementation through an interface
    }
}
```

Sowohl Delegaten als auch Schnittstellen ermöglichen einem Klassendesigner, Typdeklarationen und Implementierungen zu trennen. Eine bestimmte Schnittstelle kann von jeder Klasse oder Struktur geerbt und implementiert werden. Ein Delegat kann für eine Methode in einer beliebigen Klasse erstellt werden, sofern die Methode zur Methodensignatur des Delegaten passt.

Merke: In beiden Fällen kann die Schnittstellenreferenz oder ein Delegat von einem Objekt verwen-

det werden, das keine Kenntnis von der Klasse hat, die die Schnittstellen- oder Delegatmethode implementiert.

Wann ist welche der beiden Varianten vorzuziehen? Verwenden Sie einen Delegaten unter folgenden Umständen:

- Ein Ereignis-Entwurfsmuster wird verwendet.
- Es ist wünschenswert, eine statische Methode einzukapseln.
- Der Aufrufer muss nicht auf andere Eigenschaften, Methoden oder Schnittstellen des Objekts zugreifen, das die Methode implementiert.
- Eine einfache Zusammensetzung ist erwünscht.
- Eine Klasse benötigt möglicherweise mehr als eine Implementierung der Methode (siehe oben).

Verwenden Sie eine Schnittstelle wenn:

- Es gibt eine Gruppe verwandter Methoden, die aufgerufen werden können.
- Eine Klasse benötigt nur eine Implementierung der Methodesignatur.
- Die Klasse, die die Schnittstelle verwendet, möchte diese Schnittstelle in andere Schnittstellen- oder Klassentypen umwandeln.
- Die implementierte Methode ist mit dem Typ oder der Identität der Klasse verknüpft, z. B. mit Vergleichsmethoden.

Ein Beispiel für die kombinierte Anwendung eines Delegaten ist `IComparable` oder die generische Version `IComparable<T>`. `IComparable` deklariert die `CompareTo`-Methode, die eine Ganzzahl zurückgibt, die eine Beziehung angibt, die kleiner, gleich oder größer als zwei Objekte desselben Typs ist. Damit kann `IComparable` Grundlage für einen Sortieralgorithmus verwendet werden. Alternativ kann aber auch eine Delegatenvergleichsmethode übergeben werden.

Obwohl die Verwendung einer Delegatenvergleichsmethode als Grundlage eines Sortieralgorithmus gültig ist, gestaltet sich die fehlende Zuordnung nicht ideal. Da die Fähigkeit zum Vergleichen zur Klasse gehört und sich der Vergleichsalgorithmus zur Laufzeit nicht ändert, ist eine Einzelmethodenschnittstelle ideal.

Vergleichen Sie dazu Link!

Praktische Implementierung

Neben dem Basiskonzept der Delegaten können in C# spezifischere Realisierungen umgesetzt werden, die die Anwendung flexibler bzw. effizienter machen.

Anonyme / Lambda Funktionen

Entwicklungshistorie von C# in Bezug auf Delegaten

Version	Delegatendefinition
< 2.0	benannte Methoden
$>= 2.0$	anonyme Methoden
ab 3.0	Lambdaausdrücke

Dabei lösen Lambdaausdrücke die anonymen Methoden als bevorzugten Weg zum Schreiben von Inlinecode ab. Allerdings bieten anonyme Methode eine Funktion, über die Lambdaausdrücke nicht verfügen. Anonyme Methoden ermöglichen das Auslassen der Parameterliste. Das bedeutet, dass eine anonyme Methode in Delegaten mit verschiedenen Signaturen konvertiert werden kann.

Anonyme Methoden

Das Erstellen anonymer Methoden verkürzt den Code, da nunmehr ein Codeblock als Delegatparameter übergeben wird.

```
// Declare a delegate pointing at an anonymous function.
Del d = delegate(int k) { /* ... */ };
```

Das folgende Codebeispiel illustriert die Verwendung. Dabei wird auch deutlich, wie eine Methodenreferenz durch einen anderen ersetzt werden kann.

```
using System;
using System.Reflection;
using System.Collections.Generic;
```

```
// Declare a delegate.
delegate void Printer(string s);

public class Program{

  static void DoWork(string k)
  {
      System.Console.WriteLine(k);
  }

  public static void Main(string[] args){
    // Anonyme Deklaration
    Printer p = delegate(string j)
    {
        Console.WriteLine(j);
    };

    p("The delegate using the anonymous method is called.");
    // Der existierende Delegat wird nun mit einer konkreten Methode
    // verknüpft
    p = DoWork;

    // alternativ könnte man auch einen neuen Delegaten anlegen
    //Printer p1 = new Printer(DoWork);
    p("The delegate using the named method is called.");
  }
}
```

Lambda Funktionen

Ein Lambdaausdruck ist ein Codeblock, der wie ein Objekt behandelt wird. Er kann als Argument an eine Methode übergeben werden und er kann auch von Methodenaufrufen zurückgegeben werden.

```
(<Paramter>) => { expression or statement; }

(int a) => a * 2;             // einzelner Ausdruck - Ausdruckslambda
(int a) => { return a * 2; }; // Anweisungsblock    - Anweisungslambda
```

```
using System;
using System.Reflection;
using System.Collections.Generic;

public class Program
{
  public delegate int Del( int Value);
  public static void Main(string[] args){
      Del obj = (Value) => {
              int x=Value*2;
              return x;
      };
      Console.WriteLine(obj(5));
  }
}
```

Jeder Lambdaausdruck kann in einen Delegat-Typ konvertiert werden. Der Delegattyp, in den ein Lambdaausdruck konvertiert werden kann, wird durch die Typen seiner Parameter und Rückgabewerte definiert.

```
using System;
using System.Collections.Generic;

public static class demo
{
  public static void Main()
```

```
  {
      List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
      List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);
      foreach (var num in evenNumbers)
      {
          Console.Write("{0} ", num);
      }
      Console.WriteLine();
      Console.Read();
  }
}
```

Generische Delegaten

Delegaten können auch als Generics realisiert werden. Das folgende Beispiel wendet ein Delegate "Transformer" auf ein Array von Werten an. Dabei stellt C# sicher, dass der Typ der übergebenen Parameter in der gesamten Verarbeitungskette übernommen wird.

```
using System;
using System.Reflection;
using System.Collections.Generic;

// Schritt I - Generisches Delegat
delegate T Transformer<T>(T x);

class Utility
{
  public static void Transform<T>(ref T[] values, Transformer<T> trans)
  {
      for (int i = 0; i < values.Length; ++i)
          values[i] = trans(values[i]);
  }
}

public class Program
{
  // Schritt II - Spezifische Methode, die der Delegatensignatur entspricht
  static int Square(int x){
      Console.WriteLine("This is method Square(int x)");
      return x*x;
  }

  static double Square(double x){
      Console.WriteLine("This is method Square(double x)");
      return x*x;
  }

  static void printArray<T>(T[] values){
      foreach(T i in values)
          Console.Write(i + " ");
      Console.WriteLine();
  }

  public static void Main(string[] args){
    int[] values = { 1, 2, 3 };
    printArray<int>(values);

    Transformer <int> t = new Transformer<int>(Square);
    Utility.Transform<int>(ref values, t);
    printArray(values);
  }
```

```
}
```

Action / Func

Der generischen Idee entsprechend kann man auf die explizite Definition von eigenen Delegates vollständig verzichten. C# implementiert dafür zwei Typen vor:

```
delegate TResult Func<out TResult>();
delegate TResult Func<in T1, out TResult>(T1 arg1);
delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3);

delegate void Action();
delegate void Action<in T1>(T1 arg1);
delegate void Action<in T1, in T2>(T1 arg1, T2);
delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
```

Im folgenden Beispiel wir die Anwendung illustriert. Dabei werden 3 Delegates genutzt um die Funktionen `PrintHello` und `Square()` zu referenzieren.

Delegate-Variante	Bedeutung
myOutput	C#1.0 Version mit konkreter Methode und individuellem Delegaten (Zeile 8)
myActionOutput	Generischer Delegatentyp ohne Rückgabewert `Action`
myFuncOutput	Generischer Delegatentyp mit Rückgabewert `Func`

```
using System;
using System.Reflection;
using System.Collections.Generic;

public delegate void Output(string text);

public class Program
{
  static void PrintHello(string text)
  {
      Console.WriteLine(text);
  }

  static int Square(int x)
  {
      Console.WriteLine("This is method Square(int x)");
      return x*x;
  }

  static double Square(double x)
  {
      Console.WriteLine("This is method Square(double x)");
      return x*x;
  }

  public static void Main(string[] args){
     Output myOutput = PrintHello;
     myOutput("Das ist eine Textausgabe");
     Action<string> myActionOutput = PrintHello;
     myActionOutput("Das ist eine Action-Testausgabe!");
     Func<float, float> myFuncOutput = Square;
     Console.WriteLine(myFuncOutput(5));
  }
}
```

Natürlich lassen sich auf `Func` und `Action` auch anonyme Methoden und Lambda Expressions anwenden!

```
Func<string, string> MyLambdaAction = text => text + "modifed by Lambda";
Console.WriteLine(MyLambdaAction("Tests"));
```

Warum würde die Verwendung von Action an dieser Stelle einen Fehler generieren?

Aufgaben der Woche

- [ ] Vertiefen Sie das erlernte anhand von zusätzichen Materialien

!?alt-text