Softwareentwicklung

Skriptum zur Vorlesung - 01.09.2023

Dipl.-Ing. Paul Panhofer BSc.[1]*

1 ZID, TU Wien, Taubstummengasse 11, 1040, Wien, Austria

Abstract:

MSC: p.panhofer@htlkrems.at

Keywords:

Contents

*E-mail: paul.panhofer@tuwien.ac.at

Grundlagen der objektorientierten Programmierung

December 14, 2019

1. Grundlagen der Programmierung

01

Grundlagen der Programmierung

1.1. Softwareprogramm

Softwareprogramm

Ein **Computerprogramm** ist eine den Regeln einer bestimmten **Programmiersprache** genügende Abfolge von Anweisungen, um bestimmte Aufgaben mithilfe eines Computers zu bearbeiten oder zu lösen.

Historisch gesehen hat alles mit einem bunten Gemisch aus **Anweisungen** und **Daten** innerhalb eines Betriebssystemprozesses[1] begonnen. Der **Prozess** spannte die Laufzeitumgebung für den Code auf. Programme waren zu dieser Zeit kurz und einfach.

> Die kleinste Einheit eines Programms ist eine **Anweisung**.

1.1.1 Programmbaustein: Anweisung

Ein Softwareprogramm besteht aus einer freien Abfolge von Anweisungen.

> Für Computerprogramme unterscheidet man 2 Ausprägungen von Anweisungen: **Deklarationen** und **Instruktionen**.

Auflistung: Anweisungen

- **Deklaration:** Mit der Deklaration einer Variable wird der **Datentyp**[2] und der **Bezeichner**[3] einer Variable festgelegt.

  Variablen werden zur Speicherung von Daten in Programmen verwendet.

- **Instruktion:** In der Programmierung wird der Ausdruck Instruktion als Synonym für **Befehl** verwendet. Ein Befehl ist ein definierter Einzelschritt, der von einem Computer ausgeführt werden kann. Damit können Werte verändert, Entscheidungen getroffen oder die Bildschirmausgabe adaptiert werden.

❑

[1] *Unter einem Betriebssystemprozess verstehen wir ein sich in Ausführung befindendes Programm*
[2] *Typ*
[3] *Name*

Grundlagen

1.2. Datentypen und Variablen

Variable

Variablen sind **Datencontainer** für Werte. Variablen werden zur Speicherung und Verarbeitung von Daten verwendet.

Erklärung: Variablen

- Einer Variable ist ein Teil des Speichers, destiniert zur Verwaltung von Werten, zugeordnet.
- Eine Variable besitzt dazu einen **Namen**, mit dem auf den in ihr gespeicherten Wert Bezug genommen wird und einen **Datentyp**, der die Art der Information bestimmt, die in der Variable gespeichert werden kann.
- Mit einer Variable wird ein Teil des Arbeitsspeichers verwaltet. Dem Namen der Variable wird dabei eine **Speicherzelle** zugeordnet, mit der der für sie reservierte Speicherbereich beginnt.

  Aus technischer Sicht stellt eine Variable lediglich eine **Adresse** dar, die zu einem zuvor reservierten Speicherplatz führt.
- Eine Variable muss vor ihrer Verwendung deklariert werden. Verwendet kann sie aber erst werden, wenn ihr ein Wert zugewiesen worden ist.

❑

1.2.1 Variablendeklaration/Initialisierung

Bevor eine Variable verwendet werden kann, muss sie **deklariert** werden. Dazu werden ihr ein Name und ein Typ zugeordnet.

> Eine Variable muss **deklariert** und **initialisiert** sein, bevor sie verwendet werden kann.

Erklärung: Variablendeklaration

- Stößt das Betriebssystem zur **Laufzeit** eines Programms auf eine Variablendeklaration, reserviert es für die Variable Speicherplatz im Arbeitsspeicher.
- Mit der Variablendeklaration kann einer Variable auch ein Wert zugeordnet[4] werden.

Codebeispiel: Variablendeklaration

```
1  // ----------------------------------------
2  //  Variablendeklaration/Initialisierung
3  // ----------------------------------------
4  // Deklaration einer Variable x. Der Daten-
5  // typ der Variable wird ebenfalls bestim-
6  // mit.
7  int x;
8
9  // Speichern des Wertes 10 in der Variable
10 x = 10;
11
12 // Deklaration der Variable y. Der Variable
13 // wird gleichzeitig der Wert 20 und der
14 // Datentyp int zugewiesen.
15 int y = 20;
```

❑

[4] *Variableninitialisierung*

1.2.2 Datentypen

Infolge einer **Variablendeklaration** wird einer Variable ein Name und ein Datentyp zugewiesen.

Datentyp

Ein Datentyp beschreibt eine Menge von **Werten** und **Operationen**, die auf eine Variable angewandt werden können.

Damit bestimmt der Datentyp die **Art** der Information, die in einer Variable gespeichert werden kann.

▸ **Erklärung: Datentypen**

- Jeder Wert der programmtechnisch verarbeitbar ist, kann einem Datentyp zugeordnet werden.
- C# unterscheidet dabei die folgenden einfachen Datentypen: `int`, `short`, `byte`, `long`, `double`, `float`, `char`.

▸ **Codebeispiel: Datentypen**

```
1  // ----------------------------------------
2  //  Datentypen
3  // ----------------------------------------
4  // Integer: Der Datentyp int steht fuer alle
5  // ganzen Zahlen in einem bestimmten
6  // Bereich
7
8  // Wert: 23  Datentyp: int
9  int x = 23;
10
11 // String: Der Datentyp String steht stell-
12 // vertretend fuer alle moeglichen Zeichen-
13 // ketten. Zeichenketten muessen in Anfue-
14 // hrungszeichen angegeben werden, um sie von
15 // Befehlen unterscheiden zu koennen.
16
17 // Wert: Hugo  Datentyp: String
18 string name = "Hugo";
19
20 // BOOL: Der Datentyp Bool wird zur verwal-
21 // tung von Wahrheitswerten eingesetzt.
22 // Der Datentyp hat dabei genau 2 Ausprae-
23 // gungen - true, false
24
25 // Wert: true Datentyp: Bool
26 bool flag = true;
```

❑

1.3. Kontrollstruktur

Kontrollstrukturen

Kontrollstrukturen sind **Befehle** zur Steuerung des **Programmflusses**.

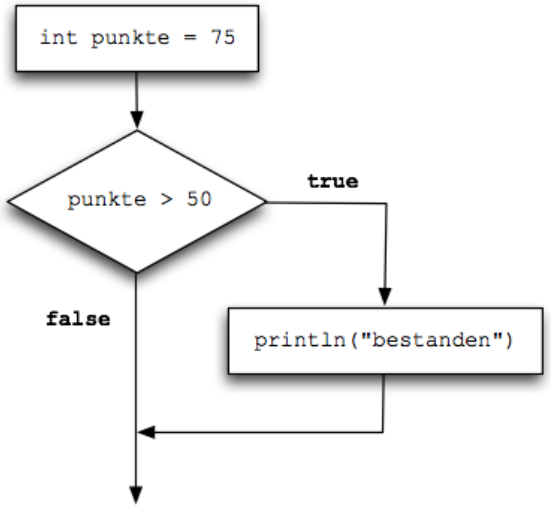
Damit bestimmen Kontrollstrukturen, die **Reihenfolge** in der die Anweisungen eines Programmes ausgeführt werden.

Es gibt 2 Formen von Kontrollstrukturen: **Schleifen** und **IF Bedingungen**.

1.3.1 Kontrollstruktur: IF Bedingung

In der Regel wird ein Programm Zeile für Zeile, Befehl für Befehl, ausgeführt. Manchmal möchte man aber eine Zeile - oder einen ganzen Block von Zeilen - nur unter einer bestimmten **Bedingung** durchführen.

Für folgendes Programm soll ermittelt werden ob ein Schüler einen Test bestanden hat oder nicht.



Der Schüler hat 75 Punkte erreicht. Das Programm prüft die Anzahl der erreichten Punkte. Falls der Schüler mehr als 50 Punkte erreicht hat, hat er die Prüfung bestanden, ansonsten ist er durchgefallen.

Mit den Einsatz einer `if Bedingung` kann der Kontrollfluss des Programms, zur Lösung der Aufgabe, leicht gesteuert werden.

▸ Codebeispiel: if Bedingung

```
// ----------------------------------------
//  SYNTAX: IF Bedingung
// ----------------------------------------
if (<condition>) {
// condition trifft zu
   <operations>
} else {
// condition trifft nicht zu
   <operations>
}

// ----------------------------------------
//  Beispiel: IF Bedingung
// ----------------------------------------
int points = 75;

// Auswertung der Bedingung
if (points > 50) {
// Falls die Bedingung zutrifft werden die
// nachfolgenden Befehle ausgefuehrt
   console.info("You passed your exam");
} else {
// Trifft die Bedingung nicht zu werden die
// Befehle in diesem Block ausgefuehrt
   console.info("You failed your exam");
}

// ----------------------------------------
//  Beispiel: IF Bedingung
// ----------------------------------------
// Ermitteln Sie den groesseren Wert 3er
// Variablen und geben Sie ihn aus.
int x = 24;
int y = -121;
int z = 53;

if (x > y) { // x > y
   if (x > z) {
      Console.WriteLine(x);
   } else {
      Console.WriteLine(z);
   }
} else { // x <= y
   if ( y > z) {
      Console.WriteLine(y);
   } else {
      Console.WriteLine(z);
   }
}
```

❑

1.3.2 Bedingungen auswerten

Bedingungen

Eine Bedingung ist ein Ausdruck, der nach Auswertung immer entweder **wahr** (`true`) oder **falsch** (`false`) ist.

Die 2 einfachsten boolschen Ausdrücke sind `true` und `false`.

Bedingungen werden auch als **boolsche Ausdrücke** bezeichnet.

▸ Codebeispiel: Bedingung auswerten

```
// ----------------------------------------
//  Bedingung Auswerten
// ----------------------------------------
// Der gewuenschte String wird immer ausge-
// geben.
if (true) { // --> wird zu true ausgew.
   Console.WriteLine("Hello world");
}


int x = 21;
if (x > 0) { // --> wird zu true ausgew.
   Console.WriteLine("value is positive");
}


int a = 7;
int b = 7;
int c = 4;

// Der == Operator prueft 2 Werte auf
// Geleichheit.
if (a == b) { // --> wird zu true ausgew.
   Console.WriteLine("values are equal");
}

if ( a != c ){ // --> wird zu true ausgew.
   Console.WriteLine("values are not equal"):
}

// Verknuepfung mehrere Bedinungen mit
// && (und) bzw. || (oder).
if( a >= b && a >= c){ // --> true
   Console.WriteLine("a ist max");
}
```

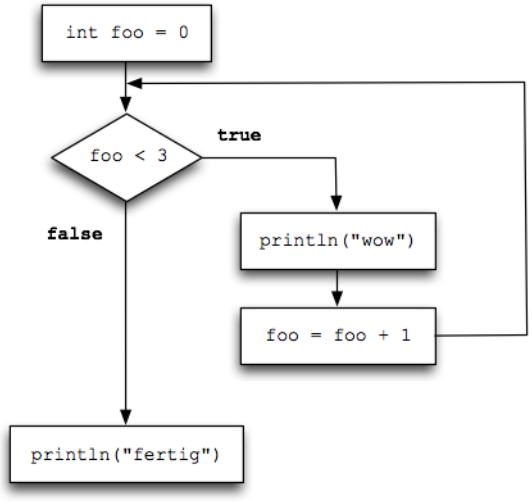
❑

1.3.3 Kontrollstruktur: WHILE

Schleife

Schleifen sind **Kontrollstrukturen**, die es ermöglichen Anweisungen bzw. Blöcke von Anweisungen zu wiederholen.

`while` Schleifen wiederholen Anweisungen solange, solange die gegebene **Bedingung** eintritt.



Codebeispiel: while Schleife

```
// ----------------------------------------
//  SYNTAX: WHILE Schleife
// ----------------------------------------
while ( <condition> ) {
   <operations>
   ...
}

// ----------------------------------------
//  Beispiel: WHILE Schleife
// ----------------------------------------
int foo = 0;

while ( foo < 3) {
   Console.WriteLine("wow");
   foo += 1;
}

Console.WriteLine("ready");
```

1.3.4 Kontrollstruktur: FOR

Die `for` Schleife ist eine Kontrollstruktur, die es ermöglicht Anweisungen bzw. Blöcke von Anweisungen zu wiederholen.

Die `for` Schleife wird vor allem dann verwendet, wenn die Gesamtzahl der **Durchläufe** bereits vor der Ausführung bekannt ist.

Codebeispiel: for Schleife

```
// ----------------------------------------
//  SYNTAX: FOR Schleife
// ----------------------------------------
// Die Syntax der for Schleife ist immer
// gleich: Eingeleitet wird die Schleife
// durch das Schluesselwort for.

// Danach folgt in runden Klammern die
// Initialisierung der Zaehlvariable.

// Die 3 Komponenten werden dabei durch ein
// Semikolon voneinander getrennt.

for( <init>; <condition>; <modifier> ) {
   <operations>
   ...
}

// ----------------------------------------
//  Beispiel: FOR Schleife
// ----------------------------------------
// Ausgabe aller Zahlen von 0 - 100
for (int i = 0; i < 101; ++i) {
   Console.WriteLine(i);
}

console.info("ready");
```

Erklärung: Funktionsweise

- **Initialisierung:** Der erste Schritt der `for` Schleife ist die Initialisierung. Dieser Schritt wird nur ein einziges Mal ausgeführt.
- **Abbruchbedingung:** Die Abbruchbedingung wird im Vorfeld definiert und dann bei jedem Durchgang überprüft. Solange die Bedingung wahr ist, wird die Schleife weiter ausgeführt.

- **Zählvariable:** Die **Zählvariable** kann zu- oder abnehmen. Der Wert wird bei jedem Durchgang modifiziert und erneut auf die Abbruchbedingung hin überprüft.
- **Wiederholung:** Die Wiederholung ist der vierte Schritt. Jede Wiederholung beginnt wieder bei der Abbruchbedingung und unterzieht diese einer erneuten Überprüfung.

❑

1.3.5 Kontrollstruktur: FOREACH

Die `foreach` Schleife ist eine Kontrollstruktur, mit der Anweisungen bzw. Blöcke von Anweisungen mehrfach wiederholt werden können.

> Die `foreach` Schleife wird in erster Linie zum Durchlaufen von **Datenstrukturen** verwendet.

▸ **Erklärung: foreach Schleife**

- Die `foreach` Schleife besitzt im Unterschied zur `for` Schleife keine **Zählvariable**[5].
- Die `foreach` Schleife definiert eine **Laufvariable**. Die Laufvariable referenziert jeweils das aktuell zu durchlaufende Objekt der Datenstuktur.
- Das bedeutet jedoch, dass wir im Gegensatz zur `for` Schleife, jeweils nur auf das gegenwärtige zu verarbeitende Element direkten Zugriff haben.

[5] *Die Zählvariable wird in der Regel als Arrayindex für den Zugriff auf die einzelnen Elemente des Arrays verwendet.*

▸ **Codebeispiel: foreach Schleife**

```
// ----------------------------------------
//  SYNTAX: foreach Schleife
// ----------------------------------------
// Die Syntax der foreach Schleife ist immer
// gleich: Eingeleitet wird die Schleife
// durch das Schluesselwort foreach.

// Danach folgt in runden Klammern die
// Initialisierung der Laufvariable. Die
// Laufvariable referenziert das jeweils
// aktuell zu durchlaufende Objekt der
// Datenstruktur.

// Mit jedem Schleifendurchlauf wird je-
// weils das sequentiell naechste Element
// der Datenstruktur referenziert.
foreach (<variable> in <collection>) {
   <operations>
   ...
}

// ----------------------------------------
//  Beispiele: foreach Schleife
// ----------------------------------------
// 1.Beispiel) Geben Sie alle Elemente des
// folgenden Arrays aus.
string[] names = new {
   "Alfred", "Hugo", "Franz", "Ali"
};

foreach (var name of names) {
   Console.WriteLine(name);
}

// Ausgabe:
// Alfred
// Hugo
// Franz
// Ali

// 2.Beispiel) Berechnen Sie die Summe
// der Werte eines Arrays
int[] numbers = new {1, 5, 87, 23, 87, 23};
int sum = 0;

foreach(var n in numbers){
   sum += n;
}
Console.WriteLine(sum);
```

❑

2. Grundlagen der objektorientierten Programmierung

2.1. Konzepte der Objektorientierung

OOP

In der objektorientierten Programmierung wird das abzubildenden **System** - Programm - auf eine Menge von **Objekten** abgebildet.

Die Logik des Programms ergibt sich aus der Interaktion der einzelnen Objekte

Mit der Objektorientierung wurde eine neues **Paradigma** in der Welt der Programmierung etabliert.

2.1.1 Objektorientierung

Die Objektorientierung ist das zur Zeit vorherrschende **Programmierparadigma**.

Objektorientierung als Paradigma ist dem **menschlichen Denken** sehr ähnlich.

Erklärung: Objektorientierung

- Die Objektorientierte Programmierung ist für Menschen leicht zu verstehen, da sie an unser natürliches menschliches Denken angelehnt ist.
- Alle vorstellbaren Dinge, die in einem Programm existieren sollen, werden durch **Objekte** beschrieben.

  Soll in einem Spiel beispielsweise ein Drache dargestellt werden, existiert dafür im Programm ein Objekt `Dragon`. Genauso existiert für ein Schwert ein Objekt `Sword` und für ein Schloss das Objekt `Castle`.
- Durch die Interaktion[6] des Benutzers mit den Objekten des Programms, wird das Programm entsprechend den Wünschen des Users adaptiert.
- Ein Programm kann damit als **System von Objekten** verstanden werden die untereinander Nachrichten austauschen.

❑

[6] *Tastendruck, Maus*

2.1.2 Objekte

Objekte werden durch folgende Größen charakterisiert: den **Eingenschaften**, dem **Zustand** und dem **Verhalten** eines Objekts.

▸ **Auflistung: Größen eines Objekts**

- **Eigenschaften:** Jedes Objekt besitzt sogenannte Eigenschaften - **Properties**. Diese Eigenschaften dienen dazu, das Objekt näher zu beschreiben.

  Für ein Flugschiff sind beispielsweise die folgenden Eigenschaften bekannt: `X`, `Y`, `Code`, `Speed`, `PullForce`, `Keywords`.

- **Zustand:** Der Zustand eines Objekts wird durch die Summe, der in den Eigenschaften des Objekts gespeicherten **Werte**, beschrieben.
  - **X:** 3, **Y:** 4
  - **Code:** Pufferfish Ship
  - **Speed:** 3
  - **PullForce:** 5
  - **Keywords:** GUNSHIP, AIRCRAFT, INDEPENDENT_DRIVE

- **Verhalten:** Das Verhalten eines Objekts, beschreibt auf welche Weise ein Objekt mit den Objekten des Programms, **interagieren** kann.

  Ein `Airship` Objekt kann beispielsweise von einem Ort zu einem anderen Ort fliegen, Drachen angreifen bzw. von Drachen angegriffen werden, `Crew` Objekte mitnehmen usw..

❑

2.1.3 Klasse

Klasse

Klassen sind **Blaupausen** für Objekte. Die Klasse bestimmt damit welche Eigenschaften und welches Verhalten ein Objekt hat.

Klassen werden auch als **Objekttypen** bezeichnet.

Klassen und **Objekte** sind die zentralen Bestandteile der objektorientierten Programmierung.

▸ **Erklärung: Klasse**

- Ein Objekt gehört immer zu einer bestimmten Klasse. Die Klasse wird als der **Objekttyp** eines Objekts bezeichnet.

  Objekte werden als **Instanzen** ihrer Klasse bezeichnet.

- Für jede Klasse kann es beliebig viele Instanzen geben. Eine Klasse gibt es genau einmal im System.

▸ **Codebeispiel: Klassendefinition**

```
1  // ---------------------------------------
2  //   Definition: Airship Klasse
3  // ---------------------------------------
4  // Klassendefinition - Klasse Airship
5  public class Airship {
6
7     // Properties - Eigenschaften
8     public int X { get; set; }
9     public int Y { get; set; }
10    public string Code { get; set; }
11    public int Speed { get; set; }
12    public int PullForce { get; set; }
13    public List<Keyword> Keywords { get; set; }
14
15    // Konstruktor
16    public Airship(){}
17
18 }
19
20 // Deklaration von Objekt t1
21 Airship a1 = new Airship();
22 // Deklaration von Objekt t2
23 Airship a2 = new Airship();
```

❑

2.2. Elemente einer Klasse

Die Definition einer Klasse folgt einer streng vorgegebenen **Syntax**.

2.2.1 Fallbeispiel: Last Aurora

Folgenden Klassen dienen als Vorlage für nachfolgende Kapitel.

Codebeispiel: Last Aurora

```
// ----------------------------------------
//  Klasse: Airship.cs
// ----------------------------------------
// (1) Classdefinition
public class Airship {
   // (2) Properties
   public int X { get; set; }
   public int Y { get; set; }
   public string Code { get; set; }
   public int Speed { get; set; }
   public int PullForce { get; set; }
   public List<Keyword> Keywords { get; set; }
   public List<Compartment> Compartments
      { get; set; }

   // (3) Constructor
   public Airship () {
      Keywords = new ();
      Compartments = new ();
   }

   // (4) Methodes
   public void AddCompartment(Compartment c){
      if(Compartments.Length < PullForce) {
         Compartments.Add(c);
      }
   }

   public void Move(int x, int y){
      this.X = x; this.Y = y;
   }
}

// Instanzieren eines Airship Objekts
Airship a = new Airship();
// Werte setzten
a.X = 4; a.Y = 10;
a.Code = "Dragon Spire";
a.Speed = 7;
a.PullForce = 4;
```

❑

2.2.2 Klassenelemnt: Konstruktuor

Objektinstanzierung

Als Objektinstanzierung wird der Prozess des **Erzeugens** eines Objekts einer Klasse bezeichnet. Dazu wird für das Objekt im Speicher Raum bereitgestellt.

Der `new` Operator dient der **Speicherallokation** in C#.

Analyse: Konstruktor

- Im Zuge der **Instanzierung** eines Objekts, wird der Konstruktor der Klasse aufgerufen. Der Konstruktor dient dabei in erster Linie der **Objektinitialisierung**[7].
- Der Konstruktor hat dabei denselben Namen wie die Klasse.

Codebeispiel: Konstruktor

```
// ----------------------------------------
//  Objektinstanzierung
// ----------------------------------------
// <Klassentyp> <Objektname> = new
//       <Konstruktor>;

// Objektinstanzierung: das Airship Objekt a
// kann nach der Instanzierung verwendet
// werden.
Airship a = new Airship();
// Instanzierung eines weiteren Objekts
Airship b = new Airship();

// Wird nur ein Variable definiert ohne
// den Konstruktor aufzurufen, kann nicht
// auf die Properties des Objekts zuge-
// griffen werden, weil das Objekt noch
// gar nicht existiert.
Airship c;

// Nach dem Aufruf des Konstruktors wird
// der Variable ein Objekt im Speicher
// zugewiesen.
c = new Airship();
```

❑

[7] *Als Objektinitialisierung wird der Initialisierung des Zustands eines Objekts bezeichnet.*

2.2.3 Klassenelement: Properties

Die **Eigenschaften** eines Objekts werden durch die **Properties** des Objekts abgebildet.

▸ **Erklärung: Properties**

- In einer Klasse kann eine beliebige Zahl von Properties definiert werden. Jedes Objekt verwaltet dabei seine eigenen Properties.
- Auf den Wert der Properties kann über den **Bezeichner** des Objekts zugegriffen werden.
- Der **Zustand** eines Objekts entspricht der Summe, der in den Eigenschaften eines Objekts gespeicherten Werte.

▸ **Codebeispiel: Properties**

```
// ----------------------------------------
//  Zugriff auf Properties
// ----------------------------------------
// Bevor die Properties eines Objekts be-
// arbeitet werden koennen muss das Objekt
// instanziert werden.
Airship a1 = new Airship();
// Wertzuweisung
a1.X = 3; a1.Y = 10;
a1.Code = "Dragon Spire";
a1.Speed = 7;
a1.PullForce = 4;

Airship a2 = new Airship();
a2.X = 12; a2.Y = 8;
a2.Code = "Queen Mallon";
a2.Speed = 5;
a2.PullForce = 2;

// Pruefung der Werte ueber Unittests
// Wertezugriff
Assert.That (
   a1.Code, Is.EqualTo("Dragon Spire")
);
Assert.That(a1.Speed, Is.EqualTo(7));
Assert.That(a1.PullForce, Is.EqualTo(4));

Assert.That (
   a2.Code, Is.EqualTo("Queen Mallon")
);
Assert.That(a2.Speed, Is.EqualTo(5));
Assert.That(a2.PullForce, Is.EqualTo(2));
```

❑

2.2.4 Klassenelement: Methoden

Methoden

Methoden sind **Unterprogramme** innerhalb eines Programmes. Die Methoden eines Objekts bestimmen das **Verhalten** eines Objekts.

Methoden werden verwendet um Programme zu **strukturieren**.

▸ **Erklärung: Methoden**

- Methoden werden in Klassen definiert. Eine Methoden besteht dabei aus einer freien **Abfolge** von Anweisungen. Methoden können beliebig oft aufgerufen und wiederverwendet werden.
- Methoden spezifizieren das **Verhalten** eines Objekts. Sie beschreiben, was Objekte einer Klasse tun können.
- Methoden werden innerhalb von Klassen definiert. Dadurch haben sie Zugriff auf die Variablen eines Objekts.
- Eine Methodendefinition besteht aus 2 Teilen:
  - **Methodenkopf**
  - **Methodenrumpf**
- Der **Methodenkopf** bestimmt die grundlegende Eigenschaften einer Methode. Zum Methodenkopf gehört ein *Methodenname*[8], eine Reihe von *Parametern* und der *Rückgabetyp*[9] der Methode.
- Im **Methodenrumpf** wird das gewünschte Verhalten - die Logik - der Methodende implementiert.

[8] *Der Name einer Methode muss mit einem Buchstaben beginnen, danach können Buchstaben, Ziffern und einige Sonderzeichen folgen. Üblicherweise beginnen Namen mit einem Großbuchstaben und sind Verben.*

[9] *Methoden können einen Wert an den Aufrufer zurückgegeben werden. Fall eine Methode keinen Rückgabewert wert hat, wird dies mit dem Schlüsselwort void angezeigt.*

▸ Codebeispiel: Methodendefinition

```
// ----------------------------------------
//  Methoden
// ----------------------------------------
public class Airship {

   public int X { get; set; }
   public int Y { get; set; }
   public string Code { get; set; }
   public int Speed { get; set; }
   public int PullForce { get; set; }

   public List<Keyword> Keywords { get; set; }

   public List<Compartment> Compartments
      { get; set; }

   public Airship () {
      Keywords = new ();
      Compartments = new ();
   }

   // Methodendefinition
   /* Methodenkopf:
         @Methodenname: AddComponent
         @Parameter: Component c
         @Rueckgabewert: void
   */
   public void AddCompartment(Compartment c){
   // Methodenrumpf
      if(Compartments.Length < PullForce) {
         Compartments.Add(c);
      }
   }

   // Methodendefinition
   /* Methodenkopf:
         Methodenname: Move
         Parameter: inx x, int y
         Rueckgabewert: void
   */
   public void Move(int x, int y){
   // Methodenrumpf
      this.X = x; this.Y = y;
   }


}
```

❑

2.3. Speichermanagement

Im **Speicher** eines Rechners werden die **Daten**[10] eines Programms verwaltet.

> Je nach Art des **Datentyps** an die eine Variable gebunden ist, wird die Variable unterschiedlich im Speicher verwaltet.

2.3.1 Arten von Datentpyen

Die C# Spezifikation definiert 2 Arten von Datentpyen:

- **einfache Datentypen**
- **Referenztpyen**

Einfache Datentypen

Variablen, die an einen einfachen Datentyp gebunden sind, wird im Speicher eine festgelegte Zahl an Bits zugeordnet.

Einfache Datentypen: **bool, char, byte, short, int, long, double, float**.

Referenztypen

Für Variablen, die an einen Referenzatentyp gebunden sind, ist nicht bekannt wieviel Bits sie im Speicher zur Laufzeit belegen werden.

Referenzdatentypen: **Klassen, Records**.

Die Variablen eines C# Programms werden in 2 Strukturen verwaltet: **Heap** und dem **Stack**.

▸ Erklärung: Speichermanagement

- Im allgemeinen bezeichnen Stack und Heap **Teile des Speichers**, die einem Programm auf Betriebssystemebene zur Ausführung zugeordnet werden.
- Variablen die an einfache Datentypen gebunden sind, werden im Stack verwaltet, Variablen die an Referenzdatentypen gebunden sind, werden im Heap verwaltet.

❑

[10] *Variablen, Objekte*

```
pulbic class GameLauncer {

        public static void Main(string[] args){
                int i = 3;
                int k = 7;

                Point p1 = new Point(3, 4);
                Point p2 = new Point(5, 10);

                int z = i;
                Point p3 = p1;
        }

}
```

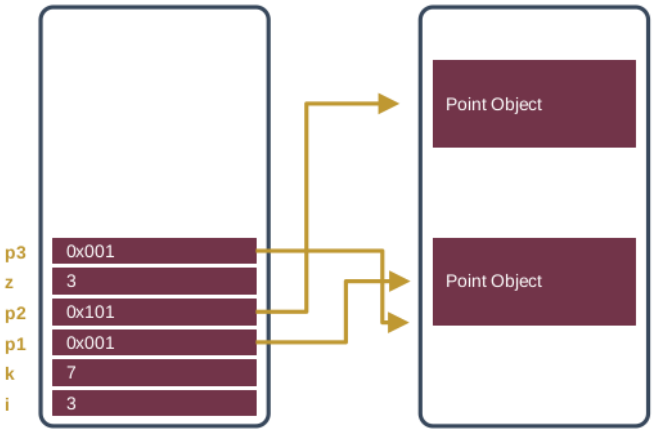


Abbildung 1. **Speicherverwaltung: Stack vs. Heap**

2.3.2 Speichermanagement: Stack

Der Stack ist ein stark **strukturierter** Teil des Speichers eines Programms. Die Werte der Programmvariablen werden am Stack gestapelt übereinander gespeichert.

▸ **Erklärung: Datenstruktur Stack** ▾

- Der Stack ist eine **Datenstruktur**, in dem Elemente nach dem **LIFO** (Last in, First out) Prinzip verwaltet.
- Der Stack kann, bedingt durch seine Struktur, sehr effizient verwaltet werden, weshalb Stackoperationen sehr schnell sind.
- Werden Daten wieder freigegeben werden, werden sie sequentiell von oben nach unten entfernt.
- Der Stack wird verwendet um Variablen die an einfache Datentypen gebunden sind, zu verwalten. Objekte können nicht auf dem Stack verwaltet werden. Für Objekte wird lediglich ein **Verweis** auf den Heap gespeichert.

❑

2.3.3 Speichermanagement: Heap

Der Heap besitzt im Gegensatz zum Stack **keine Struktur**.

> Am Heap werden die **Objekte** eines Programms zu verwaltet.

▸ **Erklärung: Datenstruktur Heap** ▾

- Der Heap ist ein **unstrukturierte Bereich** des Speichers eines Programms zur Verwaltung von Objekten.
- Während der Stack von der Größe her begrenzt ist, kann der Heap anwachsen bis die **Speichergrenze** auf Prozessebene erreicht ist.
- Im Gegegensatz zum Stack kann der Heap nicht so einfach verwaltet werden, was ihn im Einsatz langsamer macht.
- Für den **Zugriff** auf den Heap werden **Zeiger**[11][12] verwendet.
- Auf technischer Ebene ist ein Zeiger eine **Speicheradresse**, die auf einen Teil des Heaps verweist.

❑

[11] *Auf technischer Ebene ist ein Zeiger eine Speicheradresse die auf einen Teil des Heaps verweist.*
[12] *Synonym: Verweise, Referenzen*

2.3.4 Programmartefakt Zeiger

Zeiger

Ein Zeiger speichert im Gegensatz zu einer gewöhnlichen Variable keinen konkreten Wert - z.b.: `int k = 3;` - sondern einen Verweis auf eine **Speicheradresse** im Heap.

Zeiger werden auch als Objektreferenz bzw. Referenzvariable bezeichnet.

▸ **Erklärung: Programmartefakt Zeiger**

- Bei der **Definition** eines Objekts wird auf dem Stack nicht das Objekt selbst, sondern ein Verweis auf das Objekt im Heap gespeichert.
- Eine Objektvariable speichert damit nicht ein Objekt, sondern nur einen **Verweis** auf das Objekt.
- Zeiger sind dabei direkt an den **Datentyp** gebunden, wie das Objekt auf den sie verweisen.

❑

2.4. Vererbung

Vererbung ist eines der grundlegenden **Konzepte** der Objektorientierung.

Vererbung

Die Vererbung ist ein Konzept, dass eine Beziehung zwischen zwei Klassen beschreibt. Erbt eine Klasse von einer anderen Klasse, erbt sie das **Verhalten** der Basisklasse.

Auf semantischer Ebene besteht nun eine **ist ein** Beziehung, zwischen den beiden Klassen.

2.4.1 Fallbeispiel: Person

Das Konzept der Vererbung soll anhand eines Beispiels veranschaulicht werden.

▸ **Codebeispiel: Person.cs**

```
1  // ---------------------------------------
2  //  Basisklasse: Person.cs
3  // ---------------------------------------
4  public class Person {
5
6     // Properties
7     public string FirstName { get; set; }
8     public string LastName { get; set; }
9
10    // Constructor
11    public Person(
12        string firstName,
13        string lastName
14    ){
15       FirstName = firstName;
16       LastName = lastName;
17    }
18
19    //Methods
20    public String Info(){
21       return $"name: {lastName} {firstName}";
22    }
23
24 }
25 ...
26 public static void Main(String[] args){
27    Person p = new Person("Jonas","Nagi");
28    Console.WriteLn(p.Info());
29 }
```

```cpp
// ---------------------------------------
//  Subklasse: Student.cs
// ---------------------------------------
// Die Klasse Student ist eine Subklasse
// der Klasse Person. Damit erbt sie das
// Verhalten der Person Klasse

// Die Vererbungsbeziehung zwischen Klassen
// wird in der Kopfzeile der Klassendefi-
// tion etabliert.
public class Student : Person {

  public string StudentCode {
     get; set;
  }

  // Der Student Konstruktor ruft ueber das
  // base Schluesselwort automatisch den Kon-
  // struktor der Person Klasse auf.
  public Student (
     string firstName,
     string lastName,
     string studentCode
  ) : base (firstName, lastName){
     StudentCode = studentCode;
  }

}

// ---------------------------------------
//  Subklasse: Programm.cs
// ---------------------------------------
public class Programm {
   public static void Main(String[] args){
       // Ein Student Objekt hat Zugriff auf
       // das Verhalten der Person Klasse.
       Student j = new Student (
            "Jonas","Nagi", "e9725248"
       );

       // Als Datentyp kann ein Student Objekt
       // auch als Person definiert werden
       Person t = new Student (
            "Tobias","Haidi","e9845323"
       );

       Console.WriteLn(t.Info());
   }

}
```

❑

2.4.2 Klassenbeziehungen

Basisklasse

Die Basisklasse gibt ihr **Verhalten** an die Subklassen weiter. Eine Basisklasse kann eine beliebige Zahl von Subklassen haben.

Subklasse

Subklassen erben das Verhalten ihrer Basisklassen. Die Subklasse hat **Zugriff** auf die Varibalen und Methoden der Basiklasse.

Subklassen und Basisklassen stehen in einer **ist ein** Beziehung zueinander.

Erklärung: Vererbung

- Die vererbende Klasse wird auch als **Basisklasse** bezeichnet, die erbende als **Subklasse**.
- Mithilfe der Vererbung können Eigenschaften und Methoden einer übergeordneten Klasse auf andere Klassen vererbt werden.

❑

2.4.3 Abstrakte Klassen

Abstrakte Klassen sind Klassen die in ihrer Klassendefinition das Schlüsselwort `abstract` enthalten.

Erklärung: Absrakte Klassen

- Abstrakte Klassen werden gerne als **Basisklassen** für komplexe Vererbungshierarchien verwendet.
- Das besondere an abstrakten Klassen ist, dass für sie keine Instanzen erstellt werden können.

Codebeispiel: Person.cs

```cpp
// ---------------------------------------
//  Basisklasse: Person.cs
// ---------------------------------------
public abstract class Person{
   ...
}

public class Student : Person {

}
```

❑

3. Konzepte der objektorientierten Programmierung

04

Konzepte der OOP

3.1. Konzept: Identität

Die objektorientierte Programmierung unterscheidet 2 Formen der **Gleichheit** von Objekten:

- **Referenzgleichheit**
- **Wertegleichheit**

3.1.1 Referenzgleichheit

Referenzgleichheit

2 Objekte werden als referenzgleich bezeichnet, wenn 2 **Objektreferenzen** auf dasselbe Objekt im Speicher verweisen.

Referenzgleichheit wird für 2 Objekte mit dem == Operator geprüft.

Codebeispiel: Referenzgleichheit

```
1  // ---------------------------------------
2  //  Fallbeispiel: Referenzgleichheit
3  // ---------------------------------------
4  public class IdentityTest {
5     [Test]
6     public void TestReferenceEquals(){
7        Point p1 = new Point(3,4);
8        Point p2 = new Point(7,8);
9        Point p3 = p1;
10
11       Assert.That (
12          p1, Is.Not.EqualTo(p2)
13       );
14       Assert.That (
15          p1, Is.EqualTo(p3)
16       );
17
18       Assert.False (
19          p1 == p2
20       );
21       Assert.True (
22          p1 == p3
23       )
24    }
25 }
```

3.1.2 Wertegleichheit

Wertegleichheit

2 Objekte werden als wertegleich bezeichnet, wenn ihr **Zustand** identisch ist.

Wertegleichheit wird für 2 Objekte durch den Einsatz der `Equals` Methode geprüft.

Erklärung: Wertegleichheit

- Der **Zustand** eines Objekts, wird durch die Summe, der in den Eigenschaften des Objekts gespeicherten Werte beschrieben.
- Für die Prüfung auf Wertegleichheit muss in der entsprechenden Klasse die `Equals` Methode überschrieben werden.

Codebeispiel: Wertegleichheit

```
// ----------------------------------------
//  Fallbeispiel: Wertegleichheit
// ----------------------------------------
public class Point {
   public int X { get; set; }
   public int Y { get; set; }

   public override bool Equals(object o){
      // Pruefung ob ein Objekt am Heap
      // existiert
      if(ReferenceEquals(null, o))
         return false;

      // Pruefung auf Referenzgleichh.
      if(ReferenceEquals(this, obj))
         return true;

      // Pruefung ob beide Objekte den-
      // selben Typ haben
      if(obj.GetType() != this.GetType())
         return false;

      return Equals((Point)o);
   }

   protected bool Equals(Point other){
      return X == other.X && Y == other.Y;
   }
}
```

```
// ----------------------------------------
//  Fallbeispiel: Wertegleichheit
// ----------------------------------------
public class IdentityTest {

   [SetUp]
   public void Setup(){

   }

   [Test]
   public void TestEquals () {

      Point p1 = new Point(3,4);
      Point p2 = new Point(7,8);
      Point p3 = p1;
      Point p4 = new Point(7,8);

      // p1 <=> p2
      Assert.That (
         p1, Is.Not.EqualTo(p2)
      );
      Assert.False (
         p1.Equals(p2)
      );

      // p1 <=> p3
      Assert.That (
         p1, Is.EqualTo(p3)
      );
      Assert.True (
         p1.Equals(p3)
      )
      Assert.That (
         p1, Is.Same(p3)
      );

      // p1 <=> p4
      Assert.That (
         p1, Is.EqualTo(p4)
      );
      Assert.True(
         p1.Equals(p4)
      );
      Assert.That (
         p1, Is.Not.Same(p4)
      );
   }

}
```

❑

4. Datentyp: Referenztypen

05

Referenztypen

4.1. Klassen

OOP

In der objektorientierten Programmierung wird das abzubildenden **System** - Programm - auf eine Menge von **Objekten** abgebildet.

Die Logik des Programms ergibt sich aus der Interaktion der einzelnen Objekte

Mit der Objektorientierung wurde eine neues **Paradigma** in der Welt der Programmierung etabliert.

4.1.1 Klassendeklaration

Klassen dienen als Bauplan für die Abbildung von **realen Objekten** in Softwareobjekte und beschreiben die Attribute und Methoden der einzelnen Objekte.

Codebeispiel: Klassendeklaration

```
1  // ----------------------------------------
2  //  SYNTAX: Klassendeklaration
3  // ----------------------------------------
4  // Klassen werden mithilfe des Schluessel-
5  // worts class gefolgt von einem eindeut-
6  // gem Bezeichner deklariert.
7
8  // [accesss modifier] class [identifier]
9
10 // Klassendeklaration
11 /*
12    access modifer: public
13    identifier: Airship
14 */
15 public class Airship {
16    ...
17
18 }
19
20 // Ein optionaler Zugriffsparameter wird
21 // dem Schluesselwort class vorangestellt.
22 // Jeder kann Instanzen dieser Klasse
23 // erstellen da public verwendet wurde.
24
25 // Der Name der Klasse folgt dem Schluessel-
26 // wort class.
```

4.1.2 Elemente einer Klasse

Eine Klasse kann eine beliebige Zahl folgender Elemente enthalten:

- **Variablen** - Felder
- **Properties** - Eigenschaften
- **Indexer**
- **Konstruktoren**
- **Methoden**

▸ Codebeispiel: Klassenelemente

```
// ---------------------------------------
//  SYNTAX: Klassenelemente
// ---------------------------------------
public class Airship {
   // Variable
   private int _id;

   // Properties
   public int X { get; set; }
   public int Y { get; set; }

   // Konstruktoren
   public Airship () {}

   public Airship (int x, int y){
      X = x;
      Y = y;
   }

   // Methoden
   public void Move (int x, int y) {
      X += x;
      Y += y;
   }
}
```

Der Zugriff auf die Elemente einer Klasse kann über **Zugriffsparameter** gesteuert werden kann.

♻ Zugriffsparameter

Zugriffsparameter legen fest in welcher Form auf die Elemente einer Klasse zugegriffen werden kann:

- **public**: Klassenelemente die als public ausgewiesen sind, sind öffentlich sichtbar. Andere Objekte haben **uneingeschränkten Zugriff** auf solche Elemente.
- **protected**: Klassenelemente die als protectd ausgewiesen sind, sind geschützt sichtbar. Objekte derselben Klasse bzw. vererbter Klassen haben Zugriff auf die Elemente solcher Objekte.
- **private**: Klassenelemente die als private ausgewiesen sind, sind für andere Objekte nicht sichtbar. Lediglich das Objekt selbst kann auf seine Klassenelemente zugreifen.

❑

4.1.3 Klassenelement: Variablen

In einer Klasse kann eine beliebige Zahl an Variablen definiert werden.

▸ Codebeispiel: Klassenelemente

```
// ---------------------------------------
//  SYNTAX: Variablendefinition
// ---------------------------------------
...
[access modifier] [const] [readonly]
   [required] [ref] [datatype] [name];


public class Airship {
   ...
   // Variablendefinition
   /*
      access modifer: private
      datatype: int
      name: _id
   */
   private readonly int _id;

}
```

♻ Variablenparameter

Bei der **Definition** von Objektvariablen können folgende Parameter gesetzt werden:

- **const**: Die Variable wird als **Konstante** ausgezeichnet. Der Wert der Variable kann zur Laufzeit nicht geändert werden.
- **readonly**: Der Wert der Variable kann zur Laufzeit nicht geändert werden. Der Variable kann jedoch im Zuge der **Objektinitialisierung** ein Wert zugewiesen werden.
- **required**: Der Variable muss im Zuge der Objektinitialisierung ein Wert zugeordnet werden.

Codebeispiel: Variablendefinition

```
// ---------------------------------------
//  Fallbeispiel: Variablendefinition
// ---------------------------------------
public class Airship {
   // Konstante
   public const string GameType = "AIRSHIP";

   // Variablen
   public readonly int Id;
   public required string Code;
   public required string Name;

   // Properties
   public int X { get; set; }
   public int Y { get; set; }

   // Konstruktor
   public Airship (
      int id, string code, string name
   ){
      Id = id;
      Code = code;
      Name = name;
   }

   public Airship (int id){ Id = id; }

   public string ToString {
      return $"Airship data: {GameType}
          {Name}"
   }
}
```

```
// ---------------------------------------
//  Fallbeispiel: Variablendefinition
// ---------------------------------------
public class AirshipTest {

   [SetUp]
   public void Setup () {}

   [Test]
   public void TestCreate () {
      // Instanzierung eines Airship
      // Objekts
      /*
      Id, Code und Name mussen zur Objekt-
      Initialisierung gesetzt werden da sie
      readonly bzw. required sind.
      */
      Airship a = new Airship (
         3, "Erazor 3","Queen Mira"
      );

      // Der GameType kann weder initialisiert
      // noch geaendert werden.
      Assert.That(
         a.GameType, Is.EqualTo("AIRSHIP")
      );

      Assert.That(
         a.Id, Is.EqualTo(3)
      );

      // Instanzierung eines Airship
      // Objekts unter Zuhilfenamhe des
      // Initialisierungsoperators
      Airship b = new Airship(5){
         Code = "Pufferfish Class",
         Name = "Queen Mary"
      };

      Assert.That(
         b.Code,
         Is.EqualTo("Pufferfish Class")
      };

      Assert.That(
         b.Id, Is.EqualTo(5)
      );
   }

}
```

❑

4.1.4 Klassenelement: Properties

Properties beschreiben die **Eigenschaften** von Objekten. In einer Klasse kann eine beliebige Zahl von Properties definiert werden.

♻ Propertyparameter

Bei der **Definition** von Properties können folgende Parameter gesetzt werden:

- **set**: Der Wert der Property kann zur Laufzeit geändert werden.
- **get**: Der Wert der Property kann zur Laufzeit geändert werden.
- **init**: Der Wert der Property kann nur im Rahmen der Objektinitialisierung gesetzt werden.
- **private set**: Der Wert der Property kann nur durch das Objekt selbst geändert werden.

Properties sind intern als **Methoden** implementiert.

▸ Codebeispiel: Variablendefinition

```
// ----------------------------------------
//  Fallbeispiel: Variablendefinition
// ----------------------------------------
public class Airship {
   // Properties
   public int X { get; set; }
   public int Y { get; set; }
   public int Id { get; init; }

   public int MinSpeed { get; init; }
   public int MaxSpeed { get; init; }
   public int Speed { get; set; }

   public string Code { get; init; }
   public string Name { get; init; }

   public int StructurePoints { get; set; }

   // Konstruktor
   public Airship () {

   }
}
```

```
// ----------------------------------------
//  Fallbeispiel: Variablendefinition
// ----------------------------------------
public class AirshipTest {

   [SetUp]
   public void Setup () {}

   [Test]
   public void TestCreate () {
      // Instanzierung eines Airship
      // Objekts
      Airship a = new Airship (){
         Id = 4,
         Code = "Firefly X11",
         Name = "Dauntless",
         MinSpeed = 2,
         MaxSpeed = 5
      }

      // Lediglich folgende Properties
      // koennen nach der Objekt-
      // initialisierung geaendert
      // werden
      a.X = 1;
      a.Y = 1;
      a.Speed = 4;
      a.StructurePoint = 10;

      Assert.That(
         a,Id,
         Is.EqualTo(4)
      };

      Assert.That(
         a.Name,
         Is.EqualTo("Dauntless")
      };

      Assert.That(
         a.X,
         Is.EqualTo(1)
      };

      Assert.That(
         a.Y,
         Is.EqualTo(1)
      };
   }
}
```

❑

4.1.5 Indexer

Indexer

Ein Indexer ist eine Property, die den direkten Zugriff auf den **Index einer Collection** erlaubt.

Ein Indexer wird wie jede andere Property einer Klasse definiert.

Codebeispiel: Indexer

```
// ---------------------------------------
//  Definition: Indexer
// ---------------------------------------
public class Airship {
   // Variables
   private Weapon[] _weapons = new Weapon[16];
   // Properties
   public string Name { get; set; }

    // Definition eines Indexers fuer das
    // _weapons Array
    public Weapon this[int i]{
        get { return _weapons[i]; }
        set { _weapon[i] = value; }
    }
}

public class GameEngineTest {
   [Test]
   public void TestIndexer(){
         Airship a = new ();

         a[0] = new Weapon("Gatling Gun");
         a[1] = new Weapon("Blaster");

         Assert.That(
            a[1],
            Is.EqualTo(new Weapon("Blaster"))
         );
         Assert.That(
            a[0],
            Is.EqualTo(
               new Weapon("Gatling Gun")
            )
         );
   }
}
```

.

.

5. Datentyp: Collections

06

Collections

5.1. Datenstrukturen

Datenstrukturen

Eine Datenstruktur ist ein Objekt, zur **Speicherung** und **Organisation** von Daten.

Die wohl einfachste Datenstruktur ist das **Array**.

5.1.1 Grundlagen

Eine Datenstruktur wird als **Behälter**[13] für andere Werte verwendet.

Erklärung: Datenstrukturen

- Je nach Datenstruktur werden die enthaltenen Werte unterschiedlich **organisiert**.

  Manche Datenstrukturen erlauben z.B.: das mehrfache Speichern gleicher Werte. Andere ordnen Objekte bereits beim Einfügen in der Datenstruktur.

- Arten von Datenstrukturen:
  - **Dictionary**
  - **List**
  - **Stack**
  - **Queue**

Analyse: Datenstrukturen

- Es ist zu beachten, dass eine Datenstruktur im Grunde nur Elemente am Stack verwaltet. Datenstrukturen arbeiten damit nicht direkt mit Objekte sondern **Objektreferenzen** vom Stack.

- Damit kann ein einzelnes Objekt mit mehreren Datenstrukturen verwaltet werden, ohne wiederholt Kopien eines einzelnen Objekts zu erstellen.

  Es werden lediglich neue Pointer am Stack angelegt die auf dasselbe Objekt verweisen.

❑

[13] *Container*

Befehl	Beschreibung	Seite
Add	Fügt ein Element am Ende der Liste ein.	29
AddRange	Fügt eine Liste von Elementen am Ende der Liste an.	29
Clear	Löschte alle Elemente der Liste.	30
Contains	Überprüft ob ein bestimmtes Element in der Liste enthalten ist.	30
Insert	Fügt ein Element an einem bestimmten Index ein.	30
InsertRange	Fügt die Elemente einer anderer Kollection ab einem bestimmten Index an.	30
Remove	Entfernt das erste Vorkommen des angegeben Elements aus der Liste.	30
RemoveAt	Entfernt das Element am angegebenem Index	30

Abbildung 2. **Listenmethoden**

5.2. Datenstruktur: List

Datenstruktur Liste

Listen verwalten eine Menge von Werten. Die Werte werden **sequentiell** auf dem Stack gespeichert und können einfach über ihre Position angesprochen werden.

Listen weisen starke Änlichkeiten zu **Arrays** auf. Im Gegensatz zum Array kann eine Liste jedoch eine beliebige Zahl von Werten verwalten.

5.2.1 Verhalten von Listen

Bei einer Liste handelt es sich um eine **geordnete** Datenstruktur auf die über einen numerischen Index zugreifen kann.

▸ Verhalten: Datenstruktur Liste

```
// ---------------------------------------
//  Syntax: Add
// ---------------------------------------
// Mit der Add Methode kann ein neues
// Element am Ende der Liste eingefuegt
// werden.
public void Add(T item){...};
```

```
// ---------------------------------------
//  Method: Add
// ---------------------------------------
public class ListTest {
   [Test]
   public void TestAdd(){
      List<Point> points = new ();
      points.Add(new Point(3,4));
      points.Add(new Point(5,6));

      Assert.That(
         points, Has.Count.EqualTo(2)
      );
   }

   [Test]
   public void TestAddRange() {
      List<Point> points = new ();
      points.Add(new Point(3,4));
      points.Add(new Point(5,6));

      List<Point> copy = new List<>();
      copy.AddRange(points);

      Assert.That(
         copy, Has.Count.EqualTo(2)
      );
    }
}
```

```
// ----------------------------------------
//  Syntax: Clear
// ----------------------------------------
// Mit der Clear Methode koennen alle Elemente
// aus der Liste geloescht werden.
public void Clear(){...};

public class ListUnitTest{
    [Test]
    public void TestClear(){
        List<Point> points = new ();

        points.Add(new Point(3,4));
        points.Add(new Point(3,2));
        points.Add(new Point(9,3));
        points.Add(new Point(2,6));

        points.Clear();

        Assert.That(
            points, Has.Count.EqualTo(0)
        );
    }
}
```

```
// ----------------------------------------
//  Syntax: Contains
// ----------------------------------------
// Mit der Contains Methode wird geprueft
// ob ein bestimmtes Element in der Liste
// enthalten ist.
public bool Contains(T elem){...};

// Hinweis: Die Contains Methode prueft auf
// Wertegleichheit. Ueberschreiben Sie die
// Equals Methode der Elementklasse
public class ListUnitTest{
     [Test]
     public void TestContains(){
          List<Point> points = new List<>();

          points.Add(new Point(4,3));
          points.Add(new Point(2,1));
          points.Add(new Point(4,9));

          Point p = new Point(2,1);

          Assert.That(
               points, Does.Contain(p)
          );
     }
}
```

```
// ----------------------------------------
//  Syntax: Insert
// ----------------------------------------
// Mit der Insert Methode kann ein Element
// an einem bestimmten Index in die Liste
// eingefuegt werden. Elemente deren Index
// groesser bzw gleich dem angegebenem
// Index ist, werden um eine Stelle nach
// hinten verschoben
public void Insert(int index, T elem){...};

public class ListUnitTest{
    [Test]
    public void TestInsert(){
        List<int> points = new List<>();

        points.Add(56);
        points.Add(3);
        points.Add(26);
        points.Add(2);

        points.Insert(1, 21);

        Assert.That(
            points, Has.Count.EqualTo(5)
        );
        Assert.AreEqual(56, points[0]);
        Assert.AreEqual(21, points[1]);
        Assert.AreEqual(3, points[2]);
    }
}
```

```
// ----------------------------------------
//  Syntax: Remove
// ----------------------------------------
// Mit der der Remove Methode wird das erste
// Vorkommen des uebergebenen Parameters aus
// der Liste entfernt.
public bool Remove(T item){...}

// Hinweis: Die Remove Methode prueft auf
// Wertegleichheit.
public class ListUnitTest{
    [Test]
    public void TestRemove(){
        List<Point> points = new List<>();

        points.Add(new Point(3,4));
        points.Add(new Point(5,2));
        points.Add(new Point(7,2));
        points.Add(new Point(2,2));
```

```
// ---------------------------------------
//  Method: Remove
// ---------------------------------------
        Assert.That(
            point, Has.Count.EqualTo(4)
        );

        Point p = new Point(5,2);

        Assert.True(points.Remove(p));
        Assert.That(
            point, Has.Count.EqualTo(3)
        );
    }
}

// ---------------------------------------
//  Syntax: RemoveAt
// ---------------------------------------
// Unter Verwendung der RemoveAt Methode
// wird ein Element an einem bestimmten
// Index geloescht.
public void RemoveAt(int index){...}

public class ListUnitTest{
    [Test]
    public void TestRemoveAt(){
        List<Point> points = new List<>();

        points.Add(new Point(3,4));
        points.Add(new Point(9,6));
        points.Add(new Point(0,4));
        points.Add(new Point(4,4));
        points.Add(new Point(3,1));
        points.Add(new Point(2,7));
        points.Add(new Point(2,1));

        Assert.That(
            point, Has.Count.EqualTo(7)
        );

        points.RemoveAt(3);
        Points.RemoveAt(1);

        Assert.That(
            point, Has.Count.EqualTo(5)
        );
    }
}
```

❑

5.3. Datenstruktur: Stack

Datenstruktur Stack

Stacks werden zum Verwalten mehrerer Werte verwendet.

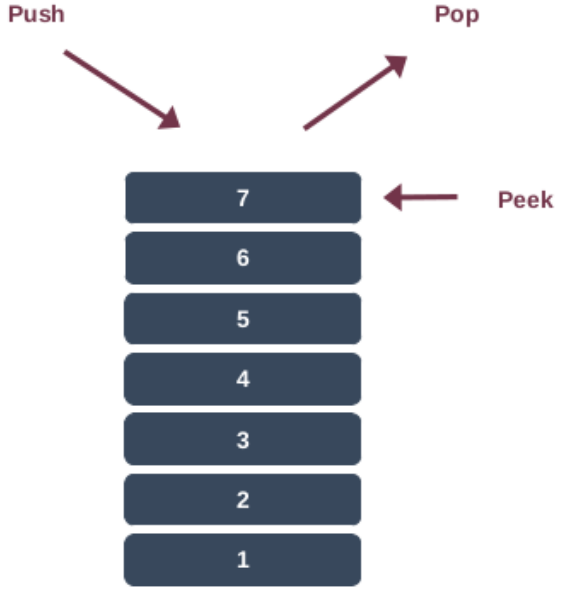
Werte werden gestapelt über bereits vorhandenen Werten eingefügt. Der zuerst eingefügte Wert steht an oberster Stelle.

5.3.1 Werteverarbeitung

Zur Verwaltung der Werte eines Stacks werden folgende Methoden verwendet.

Auflistung: Methoden

- **push:** Mit der `Push` Methode wird ein Element zum Stack **hinzugefügt**. Das Element wird dabei oben auf den Stack gelegt.
- **pop:** Mit der `Pop` Methode wird das oberste Element vom Stapel **entfernt**.
- **peek:** Die `Peek` Methode gibt eine Referenz auf das oberste Element im Stapel zurück.



5.3.2 Fallbeispiel: Stack

Verhalten: Datenstruktur Stack

```
// ---------------------------------------
//  Syntax: Push
// ---------------------------------------
// Mit der Push Methode koennen Werte zum
// Stack hinzugefuegt werden. Der Wert wird
// dabei oben am Stack eingefuegt.
public void Push(T item){...};

public class StackUnitTest{
     [Test]
     public void TestPush(){
         Stack<Point> pointStack = new ();

         pointStack.Push(new Point(3,4));
         pointStack.Push(new Point(8,3));

         Assert.That(
            pointStack, Has.Count.EqualTo(2)
         );
     }
}

// ---------------------------------------
//  Syntax: Pop
// ---------------------------------------
// Mit der Pop Methode wird das oberste
// Element vom Stapel entfernt.
public T Pop(){...};

public class StackUnitTest{
     [Test]
     public void TestPop(){
         Stack<Point> pointStack = new ();

         pointStack.Push(new Point(4,7));
         pointStack.Push(new Point(0,0));

         Point p = pointStack.Pop();
         Assert.That(
            p, Is.EqualTo(new Point(0,0))
         );
         Assert.That(
            pointStack, Has.Count.EqualTo(1)
         )
     }
}
```

Befehl	Beschreibung
Push	Die Methode fügt ein Element zum Stack hinzu. Dazu wird das Element oben auf den Stack gelegt.
Pop	Die Methode entfernt das oberste Objekt vom Stapel.
Peek	Die Methode gibt eine Referenz auf das oberste Element im Stapel zurück.
Clear	Die Methode loescht alle Objektereferenzen aus dem Stack.
Contains	Die Methode prueft ob ein bestimmtes Element in der Liste enthalten ist.
ToArray	Kopiert die im Stack verwalteten Objektreferenzen in ein Array
CopyTo	Kopiert die im Stack verwalteten Objektreferenzen in ein Array. Es werden alle Elemente ab einem bestimmten Index kopiert.

Abbildung 3. **Stackmethoden**

```
// ----------------------------------------
//  Syntax: Peek
// ----------------------------------------
// Die Peek Methode gibt eine Referenz auf
// das oberste Element im Stapel zurueck.
public T Peek(){...};

public class StackUnitTest{

     [Test]
     public void TestPeek(){
         Stack<Point> pointStack = new ();

         pointStack.Push(new Point(3,4));
         pointStack.Push(new Point(3,1));
         pointStack.Push(new Point(5,2));
         pointStack.Push(new Point(2,8));
         pointStack.Push(new Point(2,2));
         pointStack.Push(new Point(2,1));

         Assert.That(
            pointStack, Has.Count.EqualTo(6)
         );

         Point p = pointStack.Peek();
         Assert.That(
              p, Is.EqualTo(new Point(2,1))
         );
     }
}
```

```
// ----------------------------------------
//  Syntax: Clear
// ----------------------------------------
// Die Clear Methode werden die Werte des
// Stacks geloescht
public void Clear(){...};

public class StackUnitTest{

     [Test]
     public void TestClear(){
         Stack<Point> pointStack = new ();

         pointStack.Push(new Point(3,4));
         pointStack.Push(new Point(7,1));
         pointStack.Push(new Point(7,2));
         pointStack.Push(new Point(1,6));
         pointStack.Push(new Point(4,6));

         Assert.That (
            pointStack, Has.Count.EqualTo(5)
         );

         pointStack.Clear();
         Assert.That (
            pointStack, Has.Count.EqualTo(0)
         );
     }
}
```

```
// ----------------------------------------
//  Syntax: Contains
// ----------------------------------------
// Mit der Contains Methode wird geprueft
// ob ein bestimmtes Element im Stack ent-
// halten ist.
public bool Contains(T item){...};

// Hinweis: Die Contains Methode prueft auf
// Wertegleichheit .Implementieren Sie die
// Equals Methode der Elementkalsse.

public class StackUnitTest{

     [Test]
     public void TestContains(){
         Stack<Point> pointStack = ();

         pointStack.Push(new Point(3,4));
         pointStack.Push(new Point(2,1));
         pointStack.Push(new Point(2,1));
         pointStack.Push(new Point(2,1));
         pointStack.Push(new Point(3,1));
         pointStack.Push(new Point(3,1));
         pointStack.Push(new Point(1,6));

         Assert.That(
            pointStock, Has.Count.EqualTo(7)
         );

         Point p = new Point(3,1);
         Assert.True(
              pointStack.Contains(p)
         );
     }

}

// ----------------------------------------
//  Syntax: ToArray
// ----------------------------------------
// Die ToArray Methode kopiert die am Stack
// verwalteten Objektreferenzen in ein Array
public T[] ToArray(){...};

// Hinweis: Beachten Sie dass in Collections
// keine Objekte sondern Objektreferenzen
// verwaltet werden.
```

```
// ----------------------------------------
//  Methode: ToArray
// ----------------------------------------
public class StackUnitTest{
     [Test]
     public void TestToArray(){
         Stack<Point> stack = new ();

         stack.Push(new Point(3,4));
         stack.Push(new Point(2,1));
         Assert.That(
            stack, Has.Count.EqualTo(2)
         );

         Point[] points = stack.ToArray();
         Assert.That(
            stack, Has.Count.EqualTo(2)
         );
         Assert.That(
             stack.Pop(), Is.Same(points[2])
         );
     }
}

// ----------------------------------------
//  Syntax: CopyTo
// ----------------------------------------
// Kopiert die im Stack verwalteten Objekt-
// referenzen in ein Array. Welche Objekt-
// referenzen kopiert werden bestimmt der
// uebergebene Index
public void CopyTo(T[], int index){...};

public class StackUnitTest{
     [Test]
     public void TestCopyTo(){
         Stack<Point> stack = new ();
         stack.Push(new Point(3,4));

         Point[] points = new
             Point[stack.Count + 2];
         points[0] = new Point(3,9);
         points[1] = new Point(2,2);

         stack.CopyTo(points,2);
         Assert.That (
            points, Has.Count.EqualTo(3)
         )
     }
}
```

❑

Befehl	Beschreibung
Enqueue	Mit der Enqueue Methode wird ein neues Element in eine Queue eingefügt.
Dequeue	Mit der Dequeue Methode wird das als erste eingefügte Element aus einer Queue entfernt.
Peek	Die Peek Methode gibt eine Referenz auf das zuerst eingefügte Element einer Queue zurück.
Clear	Mit der Clear Methode wird eine Queue geleert.
Contains	Mit der Contains Methode wird geprüft ob ein bestimmtes Element in der Queue enthalten ist. Elemente werden dabei auf Wertegleichheit geprüft.
ToArray	Die ToArray Methode kopiert die in der Queue verwalteten Objektreferenzen in ein Array.

Abbildung 4. **Queuemethoden**

5.4. Datenstruktur: Queue

Datenstruktur Queue

Queues werden zur Verwaltung mehrere Werte verwendet. Die Elemente der Queue werden in der Reihenfolge ausgelesen, in der sie in die Queue eingefügt worden sind.

5.4.1 Verhalten von Queues

Queues verwalten die in ihnen enthaltenen Elemente nach dem **FIFO** [14] Prinzip. Dabei werden folgende Methoden unterstützt:

Auflistung: Methoden

- **Enqueue:** Mit der `Enqueue` Methode wird ein Element in eine Queue **eingefügt**.
- **Dequeue:** Die `Dequeue` Methode **entfernt** das zuerst eingefügte Element aus einer Queue.
- **Peek:** Die `Peek` Methode gibt eine Referenz auf das älteste Element einer Queue zurück.

❑

5.4.2 Fallbeispiel: Queue

Folgende Methoden werden zum Verarbeiten der Queuewerte verwendet.

Verhalten: Datenstruktur Queue

```
1  // ---------------------------------------
2  //  Syntax: Enqueue
3  // ---------------------------------------
4  // Mit der Enqueue Methode wird eine neues
5  // Element in eine Queue eingefuegt
6  public void Enqueue(){...};
7
8  public class QueueUnitTest{
9      [Test]
10     public void TestEnqueue(){
11         Queue<Point> queue = new ();
12
13         queue.Enqueue(new Point(3,2));
14         queue.Enqueue(new Point(2,3));
15         queue.Enqueue(new Point(4,3));
16         queue.Enqueue(new Point(4,4));
17
18         Assert.Thas(
19               queue, Has.Count.EqualTo(4)
20         );
21     }
22 }
```

[14] *First In - First Out*

```
// ----------------------------------------
//  Syntax: Dequeue
// ----------------------------------------
// Mit der Dequeue Methode wird aelteste
// Element einer Queue ausgetragen.
public T Dequeue(){...};

public class QueueUnitTest{
     [Test]
     public void TestDequeue(){
         Queue<Point> queue = new ();
         queue.Enqueue(new Point(7,6));

         Assert.That(
            queue, Has.Count.EqualTo(1)
         )

         Point p = queue.Dequeue();
         Assert.That(
            queue, Has.Count.EqualTo(0)
         );
         Assert.That(
            p, Is.EqualTo(new Point(7, 6))
         )
     }
}

// ----------------------------------------
//  Syntax: Peek
// ----------------------------------------
// Die Peek Methode gibt eine Referenz auf
// das zuerst eingefuegte Element einer
// Queue zurueck
public T Peek(){...};

public class QueueUnitTest{
     [Test]
     public void TestPeek(){
         Queue<Point> queue = new ();

         queue.Enqueue(new Point(3,2));
         queue.Enqueue(new Point(6,1));

         Assert.That (
            queue, Has.Count.EqualTo(2)
         );
         Point p = queue.Peek();
         Assert.That(
            p, Is.EqualTo(new Point(3,2))
         );
     }
}
```

```
// ----------------------------------------
//  Syntax: Contains
// ----------------------------------------
// Die Contains Methode prueft ob ein be-
// stimmtes Element in einer Queue enthalten
// ist. Die Elemente werden dabei auf Werte-
// gleichheit geprueft.
public bool Contains(T elem){...};

public class QueueUnitTest{
     [Test]
     public void TestContains(){
         Queue<Point> queue = new Queue<>();

         queue.Enqueue(new Point(3,2));
         queue.Enqueue(new Point(6,1));

         Assert.False(queue.Contains(new
             Point(2,5)));
     }
}

// ----------------------------------------
//  Syntax: ToArray
// ----------------------------------------
// Die ToArray Methode kopiert die in einer
// Queue verwalteten Objektreferenzen in
// ein Array.
public T[] ToArray(){...};

public class QueueUnitTest{
     [Test]
     public void TestToArray(){
         Queue<Point> queue = new Queue<>();
         queue.Enqueue(new Point(3,2));
         queue.Enqueue(new Point(6,1));

         Assert.That(
            queue.Count, Has.Count.EqualTo(2)
         );

         Point[] points = queue.ToArray();
         Assert.That(
            points.Lenght, Is.EqualTo(2)
         );
         Assert.That(
             points[0], Is.Same(queue.Peek())
         );
     }
}
```

❑

Abbildung 5. **Datenstruktur Queue**

5.5. Datenstruktur: Dictionary

Datenstruktur Dictionary

Ein Dictionary ist eine Datenstruktur zur Verwaltung von **Schlüssel Werte Paaren**.

Ein Dictionary zeigt ein ähnliches Verhalten wie ein **Arrays**. Im Gegensatz zu einem Array muss der Index eines Dictionaries kein numerischer Wert sein.

5.5.1 Fallbeispiel: Array vs. Dictionary

Beim Instanzieren eines Dictionaries wird der Datentyp des Index und des zu verwaltenden Wertes definiert.

Codebeispiel: Array vs. Dictionary

```
// ----------------------------------------
//  Array vs. Dictionary
// ----------------------------------------
public class DictionaryUnitTest{
     [Test]
     public void CompareCollection(){
        string[] phoneList1 = new string[100];

        phoneList1[0] = "0664/8972372";
        phoneList1[1] = "0650/3232664";

        Dictionary<String, String> phoneList2
            = new Dictionary<String,String>();

        phoneList2["Haidvogl"] = "0664/89723";
        phoneList2["Ferfeggy"] = "0650/32323";
        phoneList2["Adler"] = "0650/56754";
     }
}
```

5.5.2 Fallbeispiel: Dictionary

Das Verhalten eines Dictionaries wird durch folgende Methoden beschrieben.

Verhalten: Datenstruktur Dictionary

```
// ----------------------------------------
//  Syntax: [] Operator
// ----------------------------------------
// Der Lesende und Schreibende Zugriff auf
// die Elemente eines Dictionaries erfolgt
// ueber den [] Operator

public class DictionaryUnitTest{
     [Test]
     public void TestReadWrite(){
          Dictionary<string, string> pt = new
               ();

          phoneList["Haidvogl"] = "0664/89723";
          phoneList["Ferfeggy"] = "0650/32323";
          phoneList["Adler"]    = "0650/98234";
          phoneList["Schanndl"] = "0650/21323";

          Assert.That (
            pt, Has.Count.EqualTo(4)
         );
          Assert.That(
               phoneList["Haidvogl"],
               Is.EqualTo("0664/89723")
          );

          Assert.AreEqual(
               phoneList["Ferfeggy"],
               Is.EqualTo("0650/32323")
          );
     }
}
```

```
// ---------------------------------------
//  Syntax: Clear
// ---------------------------------------
// Die Clear Methode loescht alle Eintraege
// aus einem Dictionary
public void Clear(){...}

public class DictionaryUnitTest{
     [Test]
     public void TestClear(){
          Dictionary<string, int> grades = new
               ();

          grades["Softwareentwicklung"] = 2;
          grades["Informationssysteme"] = 3;
          grades["Medientechnik"] = 1;

          Assert.That(
               grades,
               Has.Count.EqualTo(3)
          );

          grades.Clear();
          Assert.That(
               grades,
               Has.Count.EqualTo(0)
          );
     }
}
```

```
// ---------------------------------------
//  Syntax: ContainsKey
// ---------------------------------------
// Mit der ContainsKey Methode wird geprueft
// ob ein Dictionary einen bestimmten Wert
// als Index besitzt.
public bool ContainsKey(T key){...}

public class DictionaryUnitTest{
     [Test]
     public void TestContainsKey(){
          Dictionary<string, int> grades = new
               ();

          grades["Dezentrale Systeme"] = 2;
          grades["Medientechnik"] = 1;

          Assert.True(
             grades.ContainsKey("Medientechnik")
          );
     }
}
```

```
// ---------------------------------------
//  Syntax: ContainsValue
// ---------------------------------------
// Mit der ContainsValue Methode wird ge-
// prueft ob ein Dictionary einen bestimmten
// Wert speichert.
public bool ContainsValue(V value){...}

public class DictionaryUnitTest{
     [Test]
     public void TestContainsValue(){
          Dictionary<string, int> grades = new
               ();
          grades["Informationssysteme"] = 3;
          grades["Medientechnik"] = 1;

          Assert.True(
             grades.ContainsValue(3);
          )
     }
}

// ---------------------------------------
//  Syntax: Remove
// ---------------------------------------
// Mit der Hilfe der Remove Methode kann ein
// Schluesseleintrag aus einem Dictionary
// entfernt werden.
public bool Remove(T key){...}

public class DictionaryUnitTest{
     [Test]
     public void TestContainsKey(){
          Dictionary<string, int> grades = new
               ();
          grades["Informationssysteme"] = 3;
          grades["Medientechnik"] = 1;

          Assert.That(
               grades,
               Has.Count.EqualTo(2)
          );

          grades.Remove("Medientechnik");
          Assert.That(
               grades,
               Has.Count.EqualTo(1)
          );
     }
}
```

❑

5.5.3 Wertezugriff

Für den Zugriff auf die Werte eines Dictionary Objekts stehen die Properties `Values` und `Keys` zur Verfügung.

▸ Codebeispiel: Wertezugriff ▾

```
// ---------------------------------------
//  Dictionary Properties
// ---------------------------------------
public class DictionaryUnitTest{
     [Test]
     public void CompareCollection(){
        Dictionary<String, String> phoneList
        = new ();

        phoneList["Haidvogl"] = "0664/89723";
        phoneList["Ferfeggy"] = "0650/32323";
        phoneList["Adler"] = "0650/56754";

        foreach(var person in phoneList.Keys){
            Console.Write($"{person} ");
        }

        > Ausgabe
        Haidvogl, Ferfeggy, Adler

        foreach(var num in phoneList.Values){
           Console.Write($"{num}");
        }

        > Ausgabe
        0664/89723 0650/32323 0650/56754
     }
}
```

❑

Grundlagen der objektorientierten Programmierung

December 14, 2019

6. Programmierung: Strukturierung

01

Sturkturierung von Programmen

6.1. Unterprogramme

Historisch gesehen hat alles mit einem bunten Gemisch aus **Anweisungen** und **Daten** innerhalb eines Betriebssystemprozesses[15] begonnen. Der **Prozess** spannte die Laufzeitumgebung für den Code auf. Programme waren zu dieser Zeit kurz und einfach.

> Die kleinste Einheit eines Programms war die **Anweisung**.

6.1.1 Unterprogramme

Die zunehmende **Codekomplexität** von Softwareanwendungen verlangte nach neuen Wegen Code zu strukturieren.

▸ **Erklärung: Unterprogramme**

- **Unterprogramme**[16] entstanden als Programme umfangreicher wurden.
- Sie waren ein erster Schritt zur **Kapselung** von Code.
- Die Zahl der Anweisungen pro Anwendung konnten ansteigen, ohne dass die **Wartbarkeit**[17] der Anwendung gesunken wäre.
- Als nächstes wurden **Container** für Daten[18] entwickelt.

▸ **Codebeispiel: Unterprogramme**

```
1  struct Point3D {
2     double x,y,z;
3  };
4  main(){
5     settextstyle(BOLD_FONT,HORIZ_DIR,2);
6     x = getmaxx()/2;
7     y = getmaxy()/2;
8
9     return 0;
10 }
```

❑

[15] *Unter einem Betriebssystemprozess verstehen wir ein sich in Ausführung befindendes Programm*
[16] *Funktionen, Prozeduren*
[17] *Codeerwartbarkeit, Codelesbarkeit, Anpassbarkeit*
[18] *Die Sprache C spiegelt diesen Entwicklungsstand wider: sie bietet Unterprogramme (Prozeduren und Funktionen) sowie Strukturen zur Strukturierung*

6.2. Objektorientierung

Der nächste Schritt in der Evolution der Anwendungsprogrammierung war das objektorientierte Programmierparadigma.

6.2.1 Objektorientierung

Objektorientierung faßt Strukturen und Unterprogramme zu **Klassen**[19] zusammen. Dadurch wurde Software nochmal etwas grobgranularer, so dass sich mehrere Anweisungen innerhalb eines Prozesses verwalten ließen.

> Die kleinste Einheit eines objektorientierten Programms ist die **Klasse**.

▸ **Erklärung: Klasse**

- Eine Klasse stellt **Funktionalität**[20] zur Verfügung, die den **Zustand**[21] von Instanzen der Klasse verändert und verarbeitet.
- Variablen und Methoden stehen im kontinuierlichen Zusammenspiel.

▸ **Codebeispiel: Klassen**

```
// ----------------------------------------
//  Project.cs
// ----------------------------------------
[Table("PROJECTS")]
public class Project : AProject {

    [Key, DatabaseGenerated]
    [Column("PROJECT_ID")]
    public int Id { get; set; }

    [Required, StringLength(50)]
    [Column("TITLE")]
    public string Title { get; set;}

    public Project() : base () {

    }
}
```

❑

[19] *Die hauptsächliche **Strukturierung** von Software befindet sich heute auf dem Niveau der **1990er**, als die Objektorientierung mit C++, Delphi und dann Java ihren Siegeszug angetreten hat.*

[20] *Methoden*

[21] *Variblen*

6.3. Schichtenmodell

Das Schichtenmodell ist ein häufig angewandtes Strukturierungsprinzip für die **Architektur** von Softwaresystemen. Dabei werden einzelne logisch zusammengehörende **Aspekte** des Softwaresystems konzeptionell einer **Schicht** zugeordnet.

6.3.1 Prinzipien des Schichtenmodells

▸ **Prinzip: Schichtenmodell**

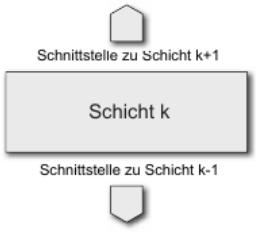
- **Teile und Herrsche:** Ein komplexes Problem wird in **unabhängige Teilprobleme** zerlegt, das jedes für sich, einfacher handhabbar ist, als das Gesamtproblem.

  Oft ist es erst durch die Fromulierung von Teilproblemen möglich, ein komplexe Probleme zu lösen.

- **Unabhängigkeit:** Die einzelnen Schichten der Anwendung kommunizieren miteinander, indem die **Schnittstellenspezifikation**[22] des direkten Vorgängers bzw. Nachfolgers genutzt wird.

  Durch die **Entkoppelung** der Spezifikation der Schicht von ihrer **Implementierung** werden Abhängigkeiten zwischen den Schichten vermieden.

- **Abschirmung:** Eine Schicht kommuniziert ausschließlich mit seinen benachbarten Schichten. Damit wird eine **Kapselung** der einzelnen Schichten erreicht, wodurch die zu bewältigende **Komplexität** sinkt.



- **Standadisierung:** Die Gliederung des Gesamtproblems in einzelne Schichten erleichtert die Entwicklung von **Standards**[23] für die einzelnen Schichten.

❑

[22] *Schnittstelle, Interface*

[23] *HTTP, FTP, usw.*

6.3.2 Fallbeispiel: Schichtenmodell

▸ Schnittstellenspezifikation: Modelschicht

```
//----------------------------------------
//  AosDbcontext.cs
//----------------------------------------
public class AosDbContext : DbContext {

    public DbSet<Trait> Traits { get; set; }
    public DbSet<TraitItem> TItems {get;set;}

    public AosDbContext(
      DbContextOptions<AosDbContext> options)
          : base(options
    ) { }

    protected override void
         OnModelCreating(ModelBuilder builder)
         {
        builder.Entity<Attack>()
            .HasIndex(a => a.Identifier)
            .IsUnique();

        builder.Entity<Attack>()
            .HasOne(a => a.Creature)
            .WithMany()
            .HasForeignKey(a => a.CreatureId);

        builder.Entity<Attack>()
            .Property(a => a.AttackType)
            .HasConversion<string>();

        builder.Entity<Trait>()
            .HasIndex(t => t.Identifier)
            .IsUnique();

        builder.Entity<TraitItem>()
            .HasKey(ti => new {ti.CreatureId,
                ti.TraitId});

        builder.Entity<TraitItem>()
            .HasOne(ti => ti.Creature)
            .WithMany()
            .HasForeignKey(ti =>
                ti.CreatureId);

        builder.Entity<TraitItem>()
            .HasOne(ti => ti.Trait)
            .WithMany()
            .HasForeignKey(ti => ti.TraitId);
    }
}
```

▸ Schnittstellenspezifikation: Domainschicht

```
//----------------------------------------
//  IRepository.cs, AReposiotry.cs
//----------------------------------------
public interface IRepository<TEntity> where
     TEntity : class {

    TEntity Create(TEntity t);

    List<TEntity> CreateRange(List<TEntity>
        list);

    void Update(TEntity t);

    void UpdateRange(List<TEntity> list);

    TEntity? Read(int id);

    List<TEntity>
        Read(Expression<Func<TEntity, bool>>
        filter);
}

public abstract class ARepository<TEntity> :
     IRepository<TEntity> where TEntity :
     class {

    protected readonly AosDbContext Context;

    protected readonly DbSet<TEntity> Table;

    protected ARepository(AosDbContext
         context) {
        Context = context;
        Table = context.Set<TEntity>();
    }

    public TEntity Create(TEntity t) {
        Table.Add(t);
        Context.SaveChanges();

        return t;
    }

    public List<TEntity>
         CreateRange(List<TEntity> list) {
        Table.AddRange(list);
        Context.SaveChanges();

        return list;
    }

    public void Update(TEntity t) {
```

```
        Context.ChangeTracker.Clear();

        Table.Update(t);
        Context.SaveChanges();
    }

    public void UpdateRange(List<TEntity>
        list) {
        Table.UpdateRange(list);
        Context.SaveChanges();
    }

    public TEntity? Read(int id) =>
        Table.Find(id);

    public List<TEntity>
        Read(Expression<Func<TEntity, bool>>
        filter) =>
        Table.Where(filter).ToList();

    public List<TEntity> Read(int start, int
        count) =>
        Table.Skip(start)
            .Take(count)
            .ToList();

    public List<TEntity> ReadAll() =>
        Table.ToList();

    public void Delete(TEntity t) {
        Table.Remove(t);
        Context.SaveChanges();
    }
}
```

▸ Schnittstellenspezifikation: Serviceschicht ▾

```
//-----------------------------------------
//  AController.cs
//-----------------------------------------
public class AController<TEntity> :
    ControllerBase where TEntity : class {

    private IRepository<TEntity> _repository;

    private ILogger<AController<TEntity>>
        _logger;

    public AController(
      IRepository<TEntity> repository,
      ILogger<AController<TEntity>> logger
    ) {
        _repository = repository;
        _logger = logger;
    }

    [HttpPost]
    public async Task<ActionResult<TEntity>>
        Create(TEntity t) {
        await _repository.CreateAsync(t);
        _logger.LogInformation($"Created
            entity with id: {t}");

        return t;
    }

    [HttpGet("{id:int}")]
    public async Task<ActionResult<TEntity>>
        Read(int id) {
        var data = await
            _repository.ReadAsync(id);

        if (data is null) return NotFound();
        _logger.LogInformation($"reading
            entity with id {id}");

        return Ok(data);
    }

    [HttpGet]
    public async
        Task<ActionResult<List<TEntity>>>
        ReadAll(int start, int count) =>
        Ok(await
        _repository.ReadAllAsync(start,
        count));

    [HttpPut("{id:int}")]
    public async Task<ActionResult>
        Update(int id, TEntity entity) {
        var data = await
            _repository.ReadAsync(id);

        if (data is null) return NotFound();

        await _repository.UpdateAsync(entity);
        _logger.LogInformation($"updated
            entity: {entity}");
        return NoContent();
    }
}
```

❑

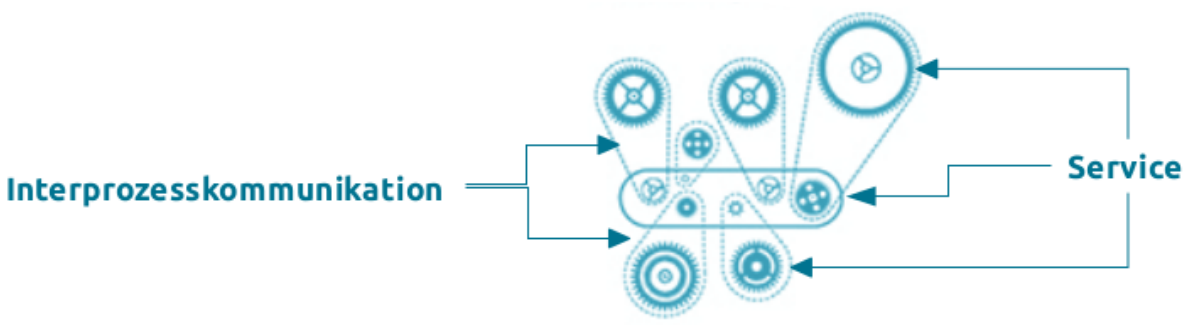


Abbildung 6. **SOA - Zusammenspiel von Services**

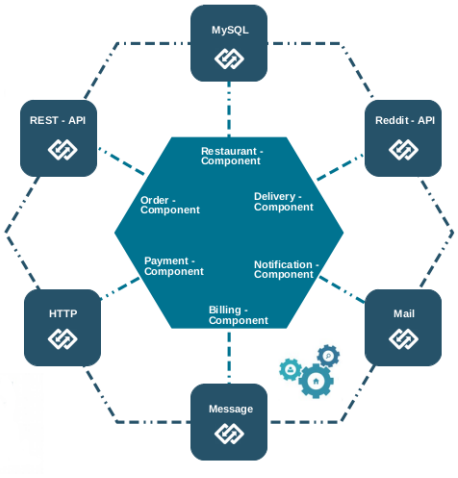
6.4. Komponenten

Bei der Entwicklung von Softwareanwendungen besteht die erste Aufgabe der Softwareentwickler darin, die voneinander unabhängigen Teile der **Anforderungsbeschreibung** voneinander zu isolieren. Wir nennen diese Teile Komponenten bzw. Module in der Softwareentwicklung.

> **Komponenten** werden in **Schichten** unterteilt. Jede Schicht wiederum besteht aus **Klassen**.

▸ **Erklärung: Komponente**

- Komponenten definieren sich als von einander **unabhängige Teile** der Anforderungsbeschreibung eines Systems.
- Für die Kommunikation stellen Komponenten **Schnittstellen** zur Verfügung.



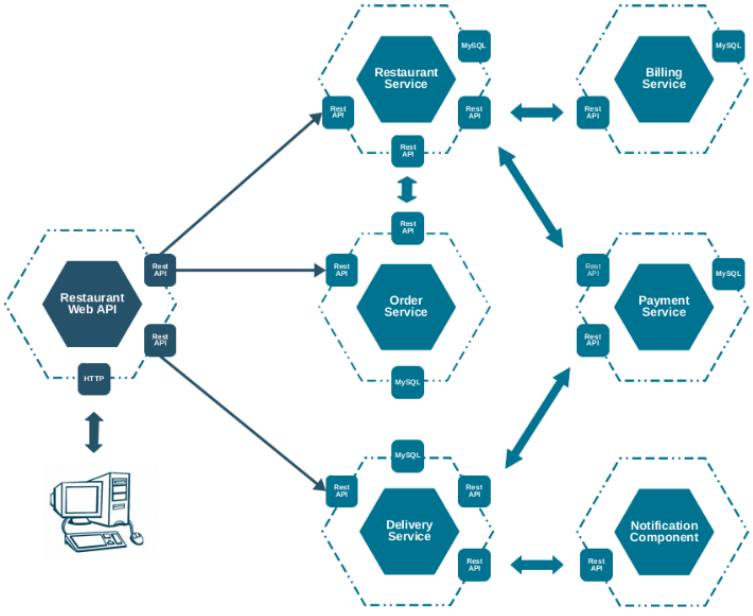
6.4.1 Fallbeispiel: Restaurantverwaltung

▸ **Fallbeispiel: Restaurantverwaltungssoftware**

- Es soll eine Restaurantverwaltungssoftware entwickelt werden.
- Als erstes isolieren wir die einzelnen **Komponenten** voneinander.
- **Komponenten** der Restaurantverwaltungssoftware:
  - **Restaurantkomponente:** Lokalbesitzer benutzen die Funktionalität der Restaurantkomponente um die Speisekarte für ihre Lokale zu verarbeiten.
  - **Orderkomponente:** Benutzer platzieren Bestellungen über eine Homepage bzw. Smartphoneanwendung Bestellungen in bestimmten Lokalen. Die Orderkomponente stellt dazu die Funktionalität zur Verfügung.
  - **Deliverykomponente:** Die Anwendung erlaubt es einer Reihe von Kurierdiensten Bestellungen auszuliefern. Die Deliverykomponente hilft bei der Verwaltung der Bestellungen.
  - **Notificationkomponente:** Die Anwendung verschickt Benachrichtigungen an die Lokale und Kunden. Die Funktionalität dafür wird von der Notificationkomponente umgesetzt.
  - **Billingkomponente:** Die Billingkomponente wird eingesetzt um die Abrechnung der Bestellung der Kunden druchführen zu können.
- Die einzelnen Komponenten können nun **unabhängig** voneinander entwickelt werden.

❑

Microservice – Architektur



6.5. Service

Service

Ein **Service** ist eine **Softwarekomponente** die in einem eigenen Betriebssystemprozess ausgeführt wird.

In einer **SOA Anwendung** bzw. in einer **Microsystemanwendung** ist das **Service** die kleinste Strukturierungseinheit der Anwendung.

▸ **Analyse: Service**

- Komplexe Softwareanwendungen verteilen ihre **Geschäftslogik** auf mehrere Service.
- Ein **Service** definiert unabhängig von seiner Implementierung eine **Schnittstelle**. Der Zugriff auf das Service erfolgt exklusiv über diese Schnittstelle.
- Die **Servicekommunikation** erfolgt über ein Technologie unabhängige Protokolle.
- Die Service einer Softwareanwendung können in unterschiedlichen Technologien implementiert werden.

❑

6.5.1 Zusammenfassung

Qualität und **Kosten** der Erstellung von Softwareanwendungen hängen entscheidend von der **Codekomplexität** ab. **Fehleranzahl** und **Robustheit** eines Codes stehen in engem Zusammenhang zur Softwarekomplexität.

Zur **Senkung** der **Codekomplexität** wurden unterschiedliche Methoden zur **Strukturierung** von Code entwickelt.

▸ **Analyse: Codestrukturierung**

- **Softwareanwendungen** bestehen aus **Services**. Ein Service ist eine **Softwarekomponente** in einem eigenen Betriebssystemprozess.
- **Komponeten** bestehen aus **Schichten**. Schichten bestehen aus **Klassen**.
- **Klassen** werden durch **Methoden** strukturiert.

❑

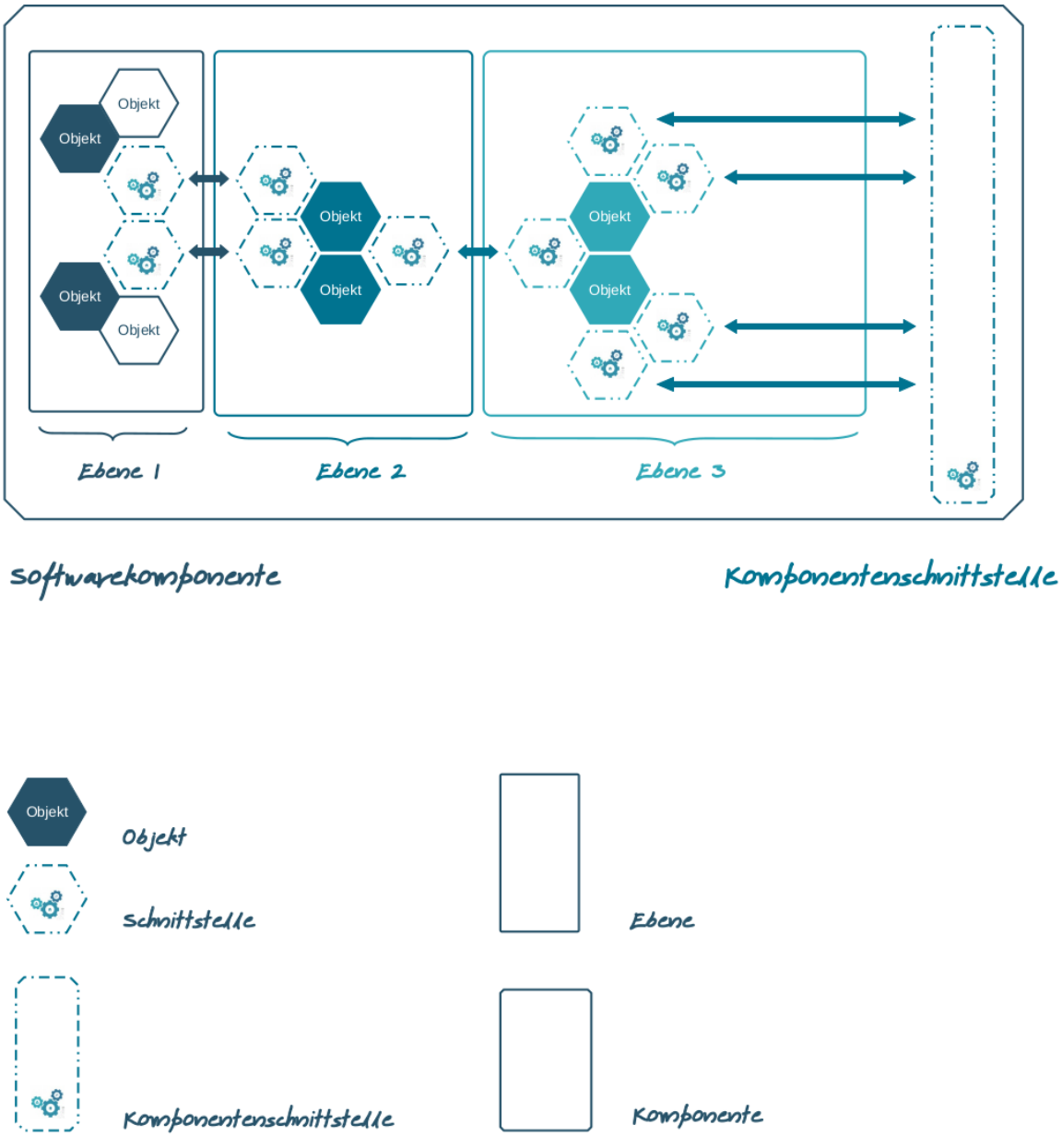


Abbildung 7. **Strukturierung einer Komponente**

.

.

7. Programmierung: Metriken

02

Programmierung: Metriken

7.1. Softwaremetriken

7.1.1 Metriken

Softwaremetrik

Eine **Softwaremetrik**, oder kurz Metrik, ist eine Funktion, die eine Eigenschaft eines Softwaresystems in einen Zahlenwert, auch **Maßzahl** genannt, abbildet.

Eine **Softwaremetrik** versucht Programmcode bzw Software im Allgemeinen mit der Hilfe einer **Maßzahl** messbar bzw. vergleichbar zu machen.

▸ **Erklärung: Softwaremetrkiken**

- Mit Softwaremetriken wird Programmcode **vergleichbar**.
- Dabei können unterschiedliche **Aspekte** von Software im Vordergrund der Messung stehen: Umfang, Aufwand, Komplexität bzw. Qualität.
- Durch die mathematische **Abbildung** einer spezifischen **Eigenschaft** der Software auf einen Zahlenwert wird ein einfacher Vergleich zwischen verschiedenen Teilen der Software ermöglicht.
- Die **Zeilenmetrik** beschreibt beispielsweise den **Umfang** eines Programms mit Hilfe der Programmzeile die für die Erstellung des Programms notwendig waren.
- Wir wollen uns hier jedoch auf Metriken beschränken die die **Qualität** des Programmcodes messen.

❑

7.1.2 Qualitätsmetriken

Wir unterscheiden 2 **Metriken** zur Beschreibung der **Qualität** von objektorientiertem Code.

▸ **Auflistung: Softwaremetriken**

- **Koppelung:** Maß der **Abhängigkeiten** zwischen Softwareelementen.
- **Kohäsion:** Maß des inneren **Zusammenhalt** eines Softwareelements.

❑

Softwaremetriken

7.2. Koppelung

Koppelung

Koppelung ist ein Maß für die **Abhängigkeit** unter **Softwareelementen**. Diese Abhängigkeit entsteht durch die Nutzung der Funktionalität des jeweils anderen Elements.

7.2.1 Stufen der Koppelung

Die Koppelung einer Codebasis wird jeweils nach der Dichte der Koppelung bewertet.

Auflistung: Stufen der Koppelung

- **Tight Coupling: Starke Koppelung** tritt auf, wenn eine Klasse direkt auf die Implementierung anderer Klassen zugreift. Änderungen an einer Klasse haben Auswirkungen auf andere Klassen. Die **Wartbarkeit** des Softwaresystems wird damit beeinträchtigt.
- **Loose Coupling: Schwache Koppelung** bedeutet, dass Klassen nur minimale Kenntnisse zu anderen Klassen haben. Der Zugriff auf die Funktionalität anderer Klassen erfolgt in erster Linie über **Schnittstellen**.

  Dies führt zu einer flexiblen und wartungsfreundlichen **Klassenstruktur** der Softwareanwendung.

❑

7.2.2 Arten der Koppelung

Für die Koppelung werden 2 unterschiedliche Arten von Koppelung unterschieden.

Auflistung: Arten der Koppelung

- **Interaktionskoppelung:** Interaktionskoppelung beschreibt das **Mass an Funktionalität**[24], das Objekte einer Klasse von Objekten anderer Klassen in Anspruch nehmen.

  Die Interaktionskoppelung ist die dominant auftrettende Form der Koppelung in Softwareprogrammen.
- **Vererbungskoppelung:** Vererbungskoppelung beschreibt das **Ausmaß der Abhängigkeit** der erbenden und der Basisklasse[25]. Vererbung, per se, ist als Konzept nicht zu verwerfen. Es ist der falsche Einsatz der Vererbung.

❑

7.2.3 Interaktionskoppelung

Interaktionskoppelung beschreibt das Mass an Funktionalität, das Objekte einer Klasse von Objekten anderer Klassen in Anspruch nehmen.

> Interaktionskoppelung tritt auf wenn Objekte einer Klasse, **Methoden** von Objekten anderer Klassen aufrufen.

[24] *Methodenaufruf*
[25] *Vererbung*

▸ Codebeispiel: Interaktionskoppelung

```
//----------------------------------------
//  Interaktionskoppelung
//----------------------------------------
public class StockMarket {
   private decimal _price;

   protected List<Portfolio> Portfolios
      { get; set; } = new();

   public decimal Price {
      set {
         _price = value;
         // Interaktionskoppelung -
         // Wird es notwending in anderer
         // Form auf die Aenderungen im
         // Kurs zu reagieren muss die
         // Codebasis der Klasse ver-
         // aendert werden
         Portfolios.ForEach(
            p => p.NotifyPriceChange(value);
         );
      }
   }
}
```

❑

7.2.4 Auflösen von Interaktionskoppelung

Durch die Trennung von **Definition** und **Implementierung** kann die Implementierung einer Klasse verändert werden, ohne dass andere Klassen davon betroffen werden.

> **Design Patterns** wurden entwickelt, um Softwarearchitekten eine Reihe von Werkzeuge zur Hand zu geben, um Interaktionskoppelung aufzulösen.

▸ Analyse: Interaktionskoppelung

- **Koppelung** zwischen Objekten kann durch die Definition und die Verwendung von Schnittstellen vermieden werden.
- Mit einer Schnittstelle wird die **Definition** einer Klasse von ihrer **Implementierung** getrennt.

❑

7.2.5 Fallbeispiel: Auflösen von Interaktionskoppelung

```
//----------------------------------------
//  Entkoppelter Code
//----------------------------------------
// Einsatz des Observerpatterns zur Entkopp-
// elung der Softwareelemente
public interface IObserver {
   void Update(decimal price);
}

public interface IObservable {
   void AddObserver(IObserver observer);
   void RemoveObserver(IObserver observer);
   void NotifyObservers();
}

public class StockMarket : IObservable {
   private decimal _currentPrice;
   private List<IObserver> _observers
      = new();

   public decimal Price {
      set {
         _currentPrice = value;
         NotifyObservers();
      }
   }

   public void AddObserver(
      IObserver observer
   ) => observers.Add(observer);

   public void RemoveObserver(
      IObserver observer
   ) => observers.Remove(observer);

   public void NotifyObservers() =>
      _observers.ForEach (
         o => o.Update(currentPrice);
      );
}

public class Portfolio : IObserver{
   public void Update(decimal newPrice) {
      // Logik zur Reaktion auf Preisaenderung
   }
}
```

❑

7.2.6 Vererbungskoppelung

Vererbungskoppelung

Vererbungskoppelung beschreibt das Ausmaß der Abhängigkeit zwischen **erbender** und **Basisklasse**.

Vererbungskoppelung kann für komplexe **Vererbungsstrukturen** auftreten.

▸ **Erklärung: Vererbungskoppelung**

- Vererbung ist eines der fundamentalen **Prinzipien** der **Objektorientierten** Programmierung.
- Vererbung ermöglicht das **Verhalten** einer Basisklasse auf ihre **Kindklassen** zu übertragen.
- Der Einsatz von Vererbung kann jedoch zu komplexen **Vererbungsstrukturen** führen.

  Wird es notwendig, die von der Basisklasse geerbten Methoden, in Kindklassen zur Gänze zu überschreiben verliert Vererbung seinen Sinn. In diesem Fall spricht man von Vererbungskoppelung.
- Vererbungskoppelung kann mit Hilfe von **Objektkomposition** aufgelöst werden.

▸ **Codebeispiel: Vererbungskoppelung**

```
1  //----------------------------------------
2  //  Vererbungskoppelung
3  //----------------------------------------
4  public class Duck {
5     public String Quack() => "quack";
6     public String Fly() =>
7        "flying high in the sky";
8  }
9
10 public class RedheadDuck : Duck {
11    public String Quack() => "loudly quack";
12 }
13
14 public class EntlingDuck : Duck {
15    public String Quack() => "proudly quack";
16 }
17
18 public class RubberDuck : Duck {
19    public String Quack() => "squeeze";
20    public String Fly() => "can't fly";
21 }
```

❑

7.2.7 Objektkomposition

Objektkomposition

Objektkomposition basiert in der Idee, **Objekte** bestehender Klassen in andere Klassen **einzubetten** z.B. durch Aggregation oder Referenzierung.

Zur **Auflösung der Vererbungkskoppelung** wird gerne auf das Prinzip der **Objektkomposition** zurückgegriffen.

▸ **Erklärung: Vorteile der Objektkomposition**

- Der Vorteil der Objektkomposition gegenüber der Objektvererbung liegt in der **Codeflexibilität**.
- Mit Objektkomposition kann das **Verhalten** von Objekten zur **Laufzeit** verändert werden.

▸ **Codebeispiel: Objektkomposition**

```
1  //----------------------------------------
2  //  Objektkomposition vs. Vererbungskoppelung
3  //----------------------------------------
4  public interface IQuackable {
5     String Quack();
6  }
7
8  public interface IFlyable {
9     String Fly();
10 }
11
12 pulic class DefaultQuackBehaviour :
       IQuackable {
13    public String Quack() => "quack";
14 }
15
16 public class LoudQuackBehaviour : IQuackable {
17    public String Quack() => "loudly: quack";
18 }
19
20 public class ProudQuackBehaviour : IQuackable{
21    public String Quack() =>
22       "proudly and loudly: quack";
23 }
24
25 public class SqueezeQuackBehaviour :
       IQuackable{
26    public String Quack() => "squeeze";
27 }
```

```
1  public class DefaultFlyingBehaviour :
       IFlyable {
2     public String Fly() => "flying high in the
          sky";
3  }
4
5  public class NoFlyBehaviour : IFlyable {
6     public String Fly() => "can't fly";
7  }
8
9  public class Duck{
10    public IQuackable QuackBehaviour {
11       get; set;
12    }
13
14    private IFlyable FlyBehaviour {
15       get; set;
16    }
17 }
18
19 public class DuckFactory{
20    public static Duck CreateRedheadDuck() =>
21          new Duck(){
22             QuackBehaviour = new
                   LoudQuackBehaviour(),
23             FlyBehaviour = new
                   DefaultFlyingBehaviour()
24          };
25
26    public static Duck CreateEntlingDuck() =>
27          new Duck() {
28             QuackBehavior = new
                   ProudQuackBehaviour(),
29             FlyBehavoir = new
                   DefaultFlyingBehaviour()
30          );
31
32       public static Duck RubberDuck () =>
33          new Duck() {
34             QuackBehavior = new
                   SqueezeQuackBehaviour(),
35             FlyBehavior = new NoFlyBehaviour()
36       );
37 }
```

❑

7.2.8 Programmmethodik

Folgende **Programmmethodik** ermöglicht, bei sinnvoller Anwendung, die Koppelung in Softwareprogrammen zu senken.

▸ **Auflistung: Programmmethodik** ▾

- **Verwendung von Schnittstellen**[26]**:** Mit einer Schnittstelle wird ein **Contract**[27] zwischen mehrere Klassen[28] definiert. Damit wird es möglich, Klassen einfach durch andere Klassen zu substituieren.

- **Verwendung von abstrakten Klassen:** Abstrakte Klassen ermöglichen die **Generalisierung**[29] im Zusammenspiel der Klassen eines Programms. Die Objektorientierte Softwareentwicklung nutzt generalisierte Klassen und Objekte um gemeinsames Verhalten[30] bzw. Eigenschaften[31] in **logischen Einheiten** zu bündeln.

  Gleichzeitig führt die Verwendung von abstrakten Klassen zur **Spezialisierung** der Kindklassen. Damit wird erneut die Substitution der Klassen gefördert.

- **Verwendung von Entwurfsmustern:** Entwurfsmuster helfen bie der Lösung immer wieder auftretender Probleme der Softwareentwicklung. Dazu geben Entwurfsmuster einfach eine **Klassenstruktur** vor. Der Entwurf der Klassenstruktur ist dabei darauf ausgelegt eine schwache Koppelung und eine starke Kohäsion der Klassen sicherzustellen.

- **Inversion of Control:** Inversion of Control ist ein Programmierparadigma das in der objektorientierter Programmierung Anwendung findet. Konzeptionell wird das Auflösen von **Abhängigkeiten**[32] nicht von der Klasse selbst umgesetzt, sondern an ein Framework weitergegeben.

❑

[26] *Interface*
[27] *Vertrag*
[28] *Die Klasse verpflichtet sich ein bestimmtes Verhalten zu implementieren.*
[29] *Generalisierung ist eines der Kernkonzepte der objektorientierten Programmierung.*
[30] *Methoden*
[31] *Attribute*
[32] *Referenz auf eine fremde Klasse*

7.3. Kohäsion

Kohäsion

Kohäsion ist ein Maß für den **inneren Zusammenhalt** eines **Softwareelements**.

Beim Entwurf eines **Softwaresystems** ist eine **hohe Kohäsion** anzustreben. Hohe Kohäsion begünstigt geringe Koppelung.

7.3.1 Kohäsion

Eine Klasse sollte nur Methoden bzw. Attribute enthalten, die alle zur Lösung einer gemeinsamen Aufgabe oder einem gemeinsamen **Verantwortungsbereich** gehören.

▸ **Erklärung: Kohäsion**

- Wird durch ein Softwareelement **zuviel Funktionalität** umgesetzt, ist das Element zu **generell** - seine Kohäsion nimmt ab.
- Das selbe gilt für ein Element das **zuwenig Funktionalität** implementiert und sich dadurch in die Abhängigkeit zu einer anderen Klasse begibt.

▸ **Auflistung: Arten der Kohäsion**

- **Servicekohäsion:** Die Servicekohäsion ist eine Metrik zur Beschreibung des inneren Zusammenhalts einer **Methode**.

  Methoden einer Klasse sollten sich stets auf die Lösung einer einzelnen Aufgabe/Problematik beschränken.

- **Klassenkohäsion:** Die Klassenkohäsion ist eine Metrik zur Beschreibung der inneren Zusammenhalt einer **Klasse**.

  Die Verletzung der Klassenkohäsion einer Klassen ist daran festzumachen, dass ungenutze Attribute bzw. Methoden für die Klasse definiert werden.

❑

7.3.2 Fallbeispiel: Servicekohäsion

```
1  //---------------------------------------
2  //  Servicekohaesion - schwache Kohsion
3  //---------------------------------------
4  class Vector implements Serializable{
5     public int X { get; set; }
6     public int Y { get; set; }
7
8     public float Add(Vector v){
9        this.X += v.X;
10       this.Y += v.Y;
11
12       return Math.SQRT(X * X + Y * Y);
13    }
14
15 }
```

❑

8. Programmierung: SOLID

03

SOLID Prinzipien

8.1. SOLID Prinzipien

Die SOLID Prinzipien sind eine Sammlung von **Programmierprinzipien** der Objektorientierten Programmierung.

> Die SOLID Prinzipien, gemeinsam angewandt, führen zu **schwacher Koppelung** und **starker Kohäsion** der Softwareelemente einer Softwareanwendung.

8.1.1 SOLID Prinzipien

▸ **Auflistung: SOLID Prinzipien**

Single Responsibility Prinzip

Das Single Responsibility Prinzip fordert, dass jedes Softwareelement einer Anwendung nur einen **einzelnen Aspekt** der Anwendungsspezifikation implementiert.

Open Closed Prinzip

Softwaresysteme müssen stets **erweiterbar** sein. Wird ein System erweitert, darf bestehender jedoch Code nicht verändert werden.

L. Substitutionsprinzip

Das Liskovsche Substitutionsprinzip oder **Ersetzbarkeitsprinzip** fordert, dass Instanzen einer abgeleiteten Klasse sich so zu **verhalten** haben, wie Objekte der entsprechenden Basisklasse.

Interface Segregation Prinzip

Eine **Schnittstelle** sollte stets lediglich einen einzelnen Aspekt der Funktionalität eines Systems abbilden.

Dependency Inversion

Das Dependency Inversion Prinzip führt zur **Umkehrung** der **Abhängigkeiten** zwischen Softwareelementen.

SOLID Prinzipien

8.2. Single Responsibility Prinzip

Single Responsibility Prinzip

Das Single Responsibility Prinzip fordert, dass jedes Softwareelement der Anwendung nur einen **einzelnen Aspekt** der Anwendungsspezifikation implementiert.

8.2.1 Diskussion SRP

Wird versucht, in einer Klasse **mehrere Anforderungen** einer Softwareanwendung abzubilden, führt das unweigerlich zu kompliziertem, schlecht wartbarem Code.

Analyse: Verletzung des SR Prinzips

- Die Wahrscheinlichkeit, dass solche Klassen zu einem späterem Zeitpunkt **geändert** werden müssen, steigt zusammen mit dem Risiko, sich bei solchen Änderungen **Fehler** einzuhandeln.
- Man spricht in diesem Zusammenhang auch von **Gottklassen**, da sie einen großen Teil der Funktionalität der Anwendung bündeln.
- Diese Konzentration von Funktionalität in einzelnen Klassen, führt naturgemäß zu Abhängigkeiten unter den Klassen einer Softwareanwendung.

❑

8.3. Interface Segregation Prinzip

Durch die Verwendung von Schnittstellen wird es möglich die Deklaration eines Objekts von seiner Implementierung zu trennen.

> Damit wird die **Entkoppelung** der Implementierung eines Objekts von seiner Deklaration erreicht.

8.3.1 Interface Segregation Prinzip

Interface Segregation Prinzip

Eine Schnittstelle sollte stets lediglich einen einzelnen Aspekt der Funktionalität eines Systems abbilden.

Damit wird explizit starke Kohäsion und implizit schwache Koppelung für die Softwareelemente einer Softwareanwendung gefordert.

> Komplexe Schnittstellen müssen im Kontext des IS Prinzips in mehrere Schnittstellen **aufgeteilt** werden.

Erklärung: Interface Segregation Prinzip

- Komplexe Schnittstellen ermöglichen den Zugriff auf Funktionalität die über das benötigte/erlaubte Verhalten von Softwareelementen hinausgeht.
- Damit verletzen solche Schnittstellen explizit die Prinzipien der objektorientierten Programmierung.

❑

8.4. Open Closed Prinzip

Open Closed Prinzip

Softwaresysteme müssen stets **erweiterbar** sein. Wird ein System erweitert, darf bestehender Code nicht verändert werden.

Damit wird implizit die **schwache Koppelung** von Softwareelementen gefordert.

Das Open Closed Prinzip beschreibt damit eines der wichtigsten **Prinzipien** der modernen Softwareentwicklung.

8.4.1 Fallbeispiel: Open Closed Prinzip

```
//-----------------------------------------
//  Verletzung der Open Closed Prinzips
//-----------------------------------------
public enum EColor {
   GREEN, YELLOW, RED
}

public enum EAppleType{
   GOLDEN_LADY, ROSE
}

public class Apple {
   public string Label { get; set; }
   public EColor Color { get; set; }
   public int Weight { get; set; }
   public EAppleType Type { get; set; }
   public int Price { get; set; }

}

public class AppleHandler{
   public List<Apple>
        FilterGreenApples(List<Apple> apples){
      List<Apple> filteredApples = new ();

      for(Apple a: apples){
         if(a.getColor().equals(EColor.GREEN)){
            filteredApples.add(a);
         }
      }

      return filteredApples;
   }
}
```

```
//-----------------------------------------
//  Verletzung der Open Closed Prinzips
//-----------------------------------------
// Solange nur gruene Aepfel aussortiert wer-
// rden, funktioniert der Code einwandfrei.

// Sollen nun aber zusaetzlich alle gruenen
// Aepfel gefiltert werden, die nicht mehr
// als 200g wiegen muss der bestehende Code
// veraendert werden.

// Das bedeutet aber dass Code der bereits
// getestet und ausgeliefert worden ist,
// veraendert werden muss. Es liegt damit
// eine Verletzung des Open Closed Prinzips
// vor.

// Wir wollen nun eine Loesung entwickeln,
// die offen, bestehender Code darf, aber
// nicht veraendert werden.
public interface Predicate<T>{
   bool Test(T t);
}

public WeightFilter : Predicate<Apple> {
   public int Weight { get; set; }
   public bool Test (Apple a) =>
      a.Weight >= Weight;
}

public ColorFilter : Predicate<Apple>{
   public EColor Color { get; set; }
   public boolean Test (Apple a) =>
      a.Color == Color;
}

public class AppleHandler {
   public List<Apple> Filter(
      List<Apple> apples,
      Predicate<Apple> filter
   ){
      List<Apple> filteredApples = new ();
      for(Apple a in apples){
         if(filter.Test(a)){
            filteredApples.add(a);
         }
      }
      return filteredApples;
   }
}
```

❑

8.5. Liskovsche Substitutinsprinzip

♻ L. Substitutionsprinzip

Das Liskovsche Substitutionsprinzip oder **Ersetzbarkeitsprinzip** fordert, dass Instanzen einer abgeleiteten Klasse sich so zu **verhalten** haben, wie Objekte der entsprechenden Basisklasse.

Erklärung: Substitutionsprinzip

- Ein wichtiges Prinzip der objektorientierten Programmierung ist die **Vererbung**[33]
- Vererbung beschreibt damit eine **ist ein** Beziehung[34] zwischen Kindklasse und der entsprechenden Basisklasse.

8.5.1 Fallbeispiel: Substitutionsprinzip

Eine typische Hierarchie von Klassen in einem Grafikprogramm könnte z. B. aus einer Basisklasse `GraphicalElement` und den davon abgeleiteten Unterklassen `Rectangle`, `Ellipse` bzw. `Text` bestehen.

Fallbeispiel: Substitutionsprinzip

- Beispielsweise wird man die Ableitung der Klasse `Ellipse` von der Klasse `GraphicalElement` begründen mit: Eine `Ellipse` ist ein grafisches Element.
- Die Klasse `GraphicalElement` kann dann beispielsweise eine allgemeine Methode `Draw` definieren, die von `Ellipse` Objekten ersetzt wird durch eine Methode, die speziell eine Ellipse zeichnet.
- Das Problem hierbei ist jedoch, dass das **ist-ein-Kriterium** manchmal in die Irre führt.

  Wird für das Grafikprogramm beispielsweise eine Klasse `Circle` definiert, so würde man bei naiver Anwendung des „ist-ein-Kriteriums" diese Klasse von `Ellipse`[35] ableiten.
- Diese Ableitung kann jedoch im Kontext des Grafikprogramms falsch sein.

  Grafikprogramme erlauben es üblicherweise, die grafischen Darstellung der Elemente zu ändern. Beispielsweise lässt sich bei Ellipsen die Länge der beiden Halbachsen unabhängig voneinander, ändern.
- Für einen Kreis gilt dies jedoch nicht, denn nach einer solchen Änderung wäre er kein Kreis mehr.
- Hat also die Klasse `Ellipse` die Methoden `SkaliereX` und `SkaliereY`, so würde die Klasse Kreis diese Methoden erben, obwohl dieses Verhalten für `Circle` Objekte nicht erlaubt ist.

❑

[33] *Kindklassen erben dabei das Verhalten ihrer Basisklasse.*
[34] *Ein Schüler (Kindklasse) ist eine Person (Basisklasse).*
[35] *denn ein Kreis ist eine Ellipse, nämlich eine Ellipse mit gleich langen Halbachsen*

9. Programmierung: OOP Entwurf

04

OOP Entwurfsmuster

9.1. Entwurfsmuster

Entwurfsmuster

Ein Entwurfsmuster beschreibt ein **Entwurfsproblem** der Softwareentwicklung, sowie die Klassenstruktur zu seiner **Lösung**.

Entwurfsmuster sind ein grundlegendes **Konzept** der Objektorientierten Programmierung.

▸ **Erklärung: Entwurfsmuster**

- **Entwurfsmuster** helfen bei der **Lösung** immer wieder auftretender Probleme der Softwareentwicklung.
- Entwurfsmuster werden in der objektorientierten Programmierung mittlerweile als **Standard** angesehen.

9.1.1 Arten von Pattern

Entwurfsmuster können je nach ihrem Einsatzfokus **klassifiziert** werden.

▸ **Auflistung: Arten von Entwurfsmustern**

Idiom

Idiome sind Entwurfsmuster die in die Struktur von **Programmiersprache** eingearbeitet sind.

- **Annotationen**
- **Lambda Ausdrücke**

Entwurfsmuster

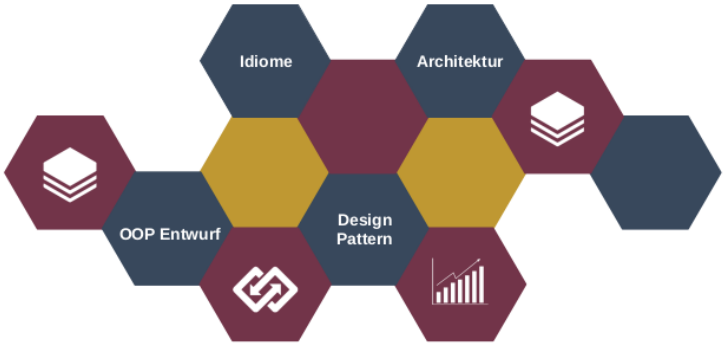
Entwurfsmuster beschreiben das **Zusammenspiel** von **Klassen**.

Architekturmuster

Architekturmuster beschreiben das **Zusammenspiel** von **Komponenten**.

❑

Entwurfsmuster



9.1.2 Einsatz von Entwurfsmustern

Entwurfsmuster abstrahieren wesentliche Konzepte der **Softwareentwicklung** und bringen sie in eine verständliche Form. Muster helfen in diesem Sinne Entwürfe zu verstehen und sie zu **dokumentieren**.

> Entwurfsmuster bestimmen die **Codestruktur** bzw. Komposition von Softwareprogrammen.

▸ **Auflistung: Musterkategorien**

- **Erzeugungsmuster:** Erzeugungsmuster unterstützen das **Erzeugen** komplexer Objekte. Der Erzeugungsprozess für Objekte wird gekapselt.
  - Singelton
  - Factorymethod
  - Builder
- **Strukturmuster:** Strukturmuster erleichtern den **Entwurf** von Software, durch die Vorgabe der Form der **Beziehungen** zwischen Klassen.
- **Verhaltensmuster:** Verhaltensmuster beschreiben die Zuständigkeiten und Interaktionen zwischen Objekten. Die Muster modellieren damit das **Verhalten** von Softwaresystemen.

❑

9.2. Erzeugermuster

Erzeugermuster unterstützen das **Erzeugen** von komplexen Objekten. Der Erzeugungsprozess von Objekten wird damit gekapselt und aus anderen Klassen ausgelagert.

9.2.1 Erzeugermuster - Singleton

Singleton

Das Singleton Entwurfsmuster definiert eine Klassenstruktur, die lediglich das Erzeugen einer **einzelnen Instanz** einer Klasse erlaubt.

Der Zugriff auf die Instanz ist **global** möglich.

▸ **Erklärung: Motivation und Kontext**

- In einer Softwareanwendung soll es für den Datenbanktreiber nur ein einzelne Instanz im System geben. Jeder Datenbankzugriff kann dann einfach über den Treiber synchronisiert werden.
- Das Singelton Entwurfsmuster erlaubt einen kontrollierten Zugriff auf die Instanz der Klasse.

▸ **Codebeispiel: Fallbeispiel: Singleton**

```
1 //----------------------------------------
2 //  Entwurfsmuster: Singleton
3 //----------------------------------------
4 // Das Singelton Entwurfsmuster gibt eine
5 // bestimmte Struktur fuer die Zielklasse
6 // des Musters vor.
7
8 // Das Muster umfasst eine einzelne Klasse
```

```
//-----------------------------------------
//  Entwurfsmuster: Singleton
//-----------------------------------------
// Das Singleton Entwurfmuster definiert eine
// Klassenstruktur, die lediglich das Erzeu-
// gen einer Instanz der Klasse erlaubt.

public class Logger {
     // In der Klasse selbst wird eine Ins-
     // tanz erzeugt und an ein Feld des Kl-
     // assenobjekts gebunden.
     private readonly static Logger _instance
          = new Logger();

     public const bool LOG_TO_CONSOLE = true;

     // Damit keine Instanzen der Klasse
     // erzeugt werden koennen wird der Kon-
     // struktor private gesetzt.
     private Logger(){

     }

     // Fuer den globalen Zugriff wird eine
     // Klassenmethode zur Verfuegung gestellt
     public static final Logger GetInstance(){
        return _instance;
     }

     public void Log(String message) {
         if(LOG_TO_CONSOLE)
            Console.WriteLn(message);
     }

}

//-----------------------------------------
//  Fallbeispiel: Singleton
//-----------------------------------------
public class Programm {

    public static void Main(string[] args) {
        Logger.LOG_TO_CONSOLE = true;

        Logger logger =Logger.getInstanze();
        logger.info("Hallo Welt");
    }
}
```

❑

9.2.2 Erzeugermuster - Factory

Factory

Das Factory Entwurfsmuster dient der Entkopplung des Clients von der **konkreten Instanzierung** eines Objekts.

Erklärung: Factory

- Für komplexe Objekte wird der **Erstellungscode** eines Objekts in eine eigene Klasse ausgelagert.
- Dadurch kommt es zu einer Entkoppelung der Logik der **Objektverarbeitung** und der **Objekterzeugung**.

Codebeispiel: Factory

```
// ----------------------------------------
//   Erzeugungsmuster: Factory
// ----------------------------------------
public interface IQuackBehavior{
    string Quack();
}

public class RedheadDuck : IQuackBehavior{
    public string Quack() => "quack quack";
}

public class MarbledDuck : IQuackBehavior{
    public string Quack() => "qua qua qua";
}

public class RubberDuck : IQuackBehavior{
    public string Quack() => "squeeze";
}

public class DuckDecoy : IQuackBehavior{
    public string Quack() => "QUACK QUACK";
}
```

```
// ---------------------------------------
//  Erzeugungsmuster: Factory
// ---------------------------------------
public class DuckSimulator {
    public void Simulate(List<IQuackBehavior>
        ducks){
        foreach(IQuackBehaviour duck in ducks){
            Console.WriteLn(duck.Quack());
        }
    }
}

// Die Schnittstelle der Factory Klasse
public interface IDuckFactory{
    IQuackBehavior CreateReadHeadDuck();
    IQuackBehavior CreateMarbledDuck();
    IQuackBehavior CreateRubberDuck();
    IQuackBehavior CreateDuckDecoy();
}

public class DecoratedDuckFacotry :
    IDuckFactory{

    public IQuackBehavior CreateReadHDuck(){
        return new OutputDecorator(new
            QuackCountDecorator(new
            ReadHeadDuck()));
    }

    public IQuackBehavior CreateMarbledDuck(){
        return new OutputDecorator(new
            QuackCountDecorator(new
            MarbledDuck()));
    }

    public IQuackBehavior CreateRubberDuck(){
        return new OutputDecorator(new
            QuackCountDecorator(new
            RubberDuck()));
    }

    public IQuackBehavior CreateGoose(){
        return new OutputDecorator(new
            QuackCountDecorator(new
            HonkAdapter(new Goose()));
    }
}
```

```
// ---------------------------------------
//  Erzeugungsmuster: Factory
// ---------------------------------------
public class DuckFactory : IDuckFactory{
    public IQuackBehavior CreateReadHDuck(){
        return new ReadHeadDuck();
    }

    public IQuackBehavior CreateMarbledDuck(){
        return new MarbledDuck();
    }

    public IQuackBehavior CreateRubberDuck(){
        return new RubberDuck();
    }

    public IQuackBehavior CreateGoose(){
        return new HonkAdapter(new Goose());
    }
}

public class Programm{
    public static void Main(String[] args){
        List<IQuackBehavior> ducks = new
            List<>();
        IDuckFactory factory = new
            DecoratedDuckFactory();

        ducks.Add(factory.CreateReadHDuck());
        ducks.Add(factory.CreateMarbledDuck());
        ducks.Add(factory.CreateRubberDuck());
        ducks.Add(factory.CreateDuckDecoy());
        ducks.Add(factory.CreateGoose());

        DuckSimulator sim = new
            DuckSimulator();
        sim.Simulate(ducks);

        factory = new DuckFactory();

        ducks.Clear();

        ducks.Add(factory.CreateReadHDuck());
        ducks.Add(factory.CreateMarbledDuck());

        sim.Simulate(ducks);
    }
}
```

❑

9.3. Strukturmuster

Strukturmuster erleichtern den **Entwurf** von Softwaresystemen, durch die Vorgabe der Form der **Beziehungen** zwischen Klassen.

9.3.1 Strukturmuster - Adapter

Adapter

Mit einem Adapter kann die **Schnittstelle** eines Objekt zur Laufzeit geändert werden.

Erklärung: Motivation und Kontext

- In ein bestehendes Softwaresystem, sollen die Klassen einer externen Klassenbilothek integriert werden. Die **Schnittstellendefinitionen** beider Systeme werden in der Regel nicht kompatibel sein.

Erklärung: Eigenschaften eines Adapters

- Der Adapter fungiert als **Vermittler**, der Anfragen vom Client erhält und diese in Anfragen umwandelt, die die neuen Klassen verstehen.
- Klassen mit inkompatiblen Schnittstellen können damit in fremde Softwaresysteme integriert werden.

Codebeispiel: Entwurfsmuster: Adapter

```
//----------------------------------------
// Entwurfsmuster: Adapter
//----------------------------------------
public interface IQuackBehavior{
     string Quack();
}

public class RedheadDuck : IQuackBehavior{
    public string Quack(){
         return "... quack quack";
    }
}

public class MarbledDuck : IQuackBehavior{
    public string Quack(){
         return "... qua qua qua";
    }
}
```

```
//----------------------------------------
// Entwurfsmuster: Adapter
//----------------------------------------
public interface IHonkBehavior{
    public string Honk();
}

public class HonkAdapter : IQuackBehavior{

     private IHonkBehaviour _honkable;

     public HonkAdapter(IHonkBehavior
          honkable){
         this._honkable = honkable;
     }

     public string Quack(){
         return this._honkable.Honk();
     }

}

public class Goose : IHonkBehaviour{
    public string Honk(){
        return "... honk honk";
    }
}

public class Programm{
     public static void Main(String[] args){
          List<IQuackBehavior> ducks = new
               List<>();

          ducks.Add(new ReadHeadDuck());
          ducks.Add(new MarbledDuck());
          ducks.Add(new HonkAdapter(new
               Goose()));

          DuckSimulator sim = new
               DuckSimulator();
          sim.simulate(ducks);
     }
}

> Ausgabe

"... quack quack"
"... qua qua qua"
"... honk honk"
```

❑

9.3.2 Strukturmuster - Dekorator

Dekorator

Mit einem Dekorator kann das **Verhalten** von Objekten zur Laufzeit verändert werden.

Erklärung: Motivation und Kontext

- Oft ist es notwendig das Verhalten von **Objekten** zur Laufzeit ändern zu können.

Erklärung: Eigenschaften von Dekoratoren

- Dekorierer besitzen denselben **Datentyp**, wie die Objekte, die sie dekorieren. Damit wird der Dekorierer **stellvertretend** für das zu dekorierende Objekt verwendet.
- Der Dekorierer fügt zur Laufzeit sein **Verhalten** dem dekorierten Objekt hinzu.

Codebeispiel: Entwurfsmuster: Dekorator

```
//----------------------------------------
//  Entwurfsmuster: Dekorator
//----------------------------------------
public interface IQuackBehavior{
     string Quack();
}

public class RedheadDuck : IQuackBehavior{
    public string Quack(){
         return "... quack quack";
    }
}

public class MarbledDuck : IQuackBehavior{
    public string Quack(){
         return "... qua qua qua";
    }
}

public class DuckSimulator {
    public void Simulate(List<IQuackBehavior>
         ducks){
        foreach(IQuackBehaviour duck in ducks){
              Console.WriteLn(duck.Quack());
        }
    }
}
```

```
//----------------------------------------
//  Entwurfsmuster: Dekorator
//----------------------------------------
// Immer wenn die Quack() Methode aufgerufen
// wird soll ein interner Zaehler mitgezaehlt
// werden.

// Zusaetzlich soll vor der Ausgabe jedesmal
// noch die Zeichenkette "Output:" auszugeben.

public class OutputDecorator :IQuackBehavior{
    private IQuackBehavior _quackable;

    public OutputDecorator(IQuackBehavior q){
        this._quackable = q;
    }
    public void Quack(){
        return "Output: " + _quackable.Quack();
    }
}

public class QuackCountDecorator :
     IQuackBehavior{
    private IQuackBehavoir _quackable;
    public static int COUNTER = 0;

    public QuackCountDecorator(IQuackBehavior
         quackable){
        this._quackable = quackable;
    }

    public string Quack(){
        ++COUNTER;
        return _quackable.Quack();
    }
}

public class Programm{
     public static void Main(String[] args){
          List<IQuackBehavior> ducks = new
               List<>();

          ducks.Add(new
               QuackCountDecorator(new
               OutputDecorator(new
               ReadHeadDuck())));
          ...
     }
}
```

```
//----------------------------------------
//  Entwurfsmuster: Dekorator
//----------------------------------------
public class Programm{
    public static void Main(String[] args){
        List<IQuackBehavior> ducks = new
            List<>();

        ducks.Add(new
            QuackCountDecorator(new
            OutputDecorator(new
            ReadHeadDuck())));
        ducks.Add(new
            QuackCountDecorator(new
            OutputDecorator(new
            MarbledDuck())));

        DuckSimualtor sim = new
            DuckSimualtor();
        sim.Simulate(ducks);

        ConsoleWriteLn("quack count: " +
            QuackCountDecorator.COUNT);
    }
}

> Ausgabe:

"Output: ... quack quack"
"Output: ... qua qua qua"
"quack count: 2"
```

❑

9.4. Verhaltensmuster

Verhaltensmuster beschreiben die Zuständigkeiten und **Interaktionen** zwischen Objekten.

9.4.1 Verhaltensmuster - Command

Command

Das **Command Muster** erlaubt es eine **Methode** wie ein Objekt zu verwenden.

Damit wird es möglich Methodenobjekte in Warteschlangen zu stellen, Logbucheinträge zu führen bzw. die Auswirkungen der Methode wieder rückgängig zu machen.

▸ Codebeispiel: Command

```
//----------------------------------------
//  Schnittstelle: ICommand
//----------------------------------------
public interface ICommand{
   void execute();
   void undo();
}
//----------------------------------------
//  Klasse: ACommand
//----------------------------------------
public abstract class ACommand {
    protected Robot _robot;
    public ACommand(Robot robot) {
       _robot = robot;
    }
    public abstract void Process();
    public abstract void Undo();
}
//----------------------------------------
//  Klasse: Point
//----------------------------------------
public class Point {
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y) {
       X = x; Y = y;
    }

    public Point CalculateNeighbour(
         EDirectionType direction
    ) {
       Point p = null;

```

```
        switch (direction) {
           case EDirectionType.NORTH:
              p = new Point(
                   this._x, this._y + 1
              );
              break;
           case EDirectionType.SOUTH:
              p = new Point(
                   this._x, this._y -1
              );
              break;
           case EDirectionType.WEST:
              p = new Point(
                   this._x -1, this._y
              );
              break;
           case EDirectionType.EAST:
              p = new Point(
                   this._x + 1, this._y
              );
              break;
        }
        return p;
     }
}

//----------------------------------------
//  Klasse: Robot
//----------------------------------------
public class Robot {
   private Point Location = new (0,0);
}

//----------------------------------------
//  Klasse: MoveUpCommand
//----------------------------------------
public class MoveUpCommand : ACommand{
   public MoveUpCommand(Robot robot) :
       base(robot) { }

   public override void Process() {
      _robot.Location =
          _robot.Location.CalculateNeighbour(
                 EDirectionType.NORTH);
   }

   public override void Undo() {
      _robot.Location =
          _robot.Location.CalculateNeighbour(
                 EDirectionType.SOUTH);
   }
}
```

```
//----------------------------------------
//  Klasse: RemoteControl
//----------------------------------------
public class RemoteControl {
    private Stack<ACommand> _commands = new
        Stack<ACommand>();
    private Stack<ACommand> _history = new
        Stack<ACommand>();

    private readonly Robot _robot;

    public RemoteControl(Robot robot) {
       _robot = robot;
    }

    public void MoveUp() {
        MoveUpCommand command = new
            MoveUpCommand(_robot);

        _history.Clear();
        _commands.Push(command);
        command.Process();
    }

    public void MoveDown() {
        MoveDownCommand command = new
            MoveDownCommand(_robot);

        _history.Clear();
        _commands.Push(command);
        command.Process();
    }

    public void MoveLeft() {
        MoveLeftCommand command = new
            MoveLeftCommand(_robot);

        _history.Clear();
        _commands.Push(command);
        command.Process();
    }

    public void MoveRight() {
        MoveRightCommand command = new
            MoveRightCommand(_robot);

        _history.Clear();
        _commands.Push(command);
        command.Process();
    }

    public bool Do() {
```

```
47          if (_history.Count == 0)
48              return false;
49
50          ACommand command = _history.Pop();
51          command.Process();
52
53          _commands.Push(command);
54
55          return true;
56      }
57
58      public bool Redo() {
59          if (_commands.Count == 0)
60              return false;
61
62          ACommand command = _commands.Pop();
63          command.Undo();
64
65          _history.Push(command);
66          return true;
67      }
68  }
```

❑

9.4.2 Verhaltensmuster - Strategy

Strategy

Das **Strategie Muster** ermöglicht es das Verhalten eines Objekts zur Laufzeit zu ändern. Für das Strategie Muster wird das Verhalten einer Klasse in eine eigene Klasse ausgelagert.

Erklärung: Motivation und Kontext

- Wir haben die Aufgabe den Warenkorb eines Webshops zu programmieren. Beim Bezahlen der Waren soll der Kunde mehrere Möglichkeiten für das Überweisen des gewünschten Betrags haben.
- Der Bezahlvorgang wird als Strategie konzipiert und kann dadurch bei jedem Bestellvorgang beliebig gewählt werden.

Codebeispiel: Command

```
1  //----------------------------------------
2  //  Schnittstelle: IPaymentStrategy
3  //----------------------------------------
4  public interface IPaymentStrategy{
5      public void pay(int amount);
6  }
```

```
1  //----------------------------------------
2  //  Klasse: CreditCardStrategy
3  //----------------------------------------
4  public class CreditCardStrategy :
       IPaymentStrategy{
5
6      private CreditCardProcessor processor =
           new CreditCardProcessor();
7
8      private string _cardNumber;
9      private string _name;
10
11     public CreditCardStrategy(string name,
           string cardNumber){
12         this._name = name;
13         this._cardNumber = cardNumber;
14     }
15
16     public void pay(int amount){
17         processor.process(
18             _name, _cardNumber, amount
19         );
20     }
21 }
22
23 //----------------------------------------
24 //  Klasse: CreditCardStrategy
25 //----------------------------------------
26 public class PaypalStrategy :
       IPaymentStrategy{
27
28     private PaypalProcessor processor = new
           PaypalProcessor();
29
30     private string _email;
31     private string _pwd;
32
33     public PaypalStrategy(String email,
           String pwd){
34         this._email = email;
35         this._pwd = pwd;
36     }
37
38     public void pay(int amount){
39         processor.process(
40             _email,
41             _pwd
42         );
43     }
44 }
```

```
//----------------------------------------
//  Klasse: Item
//----------------------------------------
public class Item {

   private string _upcCode;
   private string _price;

   public UpcCode{
      get => _upcCode;
   }

   public Price{
      get => _price;
   }

   public Item(string upc, int cost){
      this._upcCode = upc;
      this._price = cost;
   }
}

//----------------------------------------
//  Klasse: ShoppingCart
//----------------------------------------
public class ShoppingCart{

   private List<Item> _items = new
       List<Item>();

   private IPaymentStrategy _paymentMethod;

   public IPaymentStrategy PaymentMethod {
       get => _paymentMethod;
       set => _paymentMethod = value;
   }

   public void AddItem(Item item){
      _items.Add(item);
   }

   public int CalculateTotal(){
      int sum = 0;
      foreach(var item in _items){
         sum += item.Price;
      }

      return sum;
   }

   public void Pay(){
      int amount = CalculateTotal();
      _paymentMethod.pay(amount);
   }
}

//----------------------------------------
//  Klasse: ShoppingCartUnitTest
//----------------------------------------
public class ShoppingCartUnitTest{

    [Test]
    public void Test(){
       ShoppingCart cart = new ShoppingCart();

       Item item1 = new Item("234", 10);
       Item item2 = new Item("567", 30);

       cart.PaymentMethod =
          new PaypalStrategy(
              "myemail@example.com","mypwd"
          );

       cart.Pay();
    }

}
```

❑

Grundlagen der objektorientierten Programmierung

December 14, 2019

10. Architekturstil: Rest

02

REST

10.1. REST Prinzipien

Rest

Rest ist eine Architekturstil zur Entwicklung **verteilter Systemen**.

REST als Architekturstil wurde konzipiert, um den Anforderungen des modernen Internets zu genügen.

10.1.1 Architekturstil REST

REST versteht sich dabei als eine **Abstraktion** des Internets.

> Das Internet versteht sich kanzeptionell als eine Sammlung von **Ressourcen**.

Erklärung: Architekturstil REST

- Primär hat eine REST[36] Anwendung die Aufgabe Ressourcen zu verwalten.
- REST stellt dabei weder eine konkrete Technologie noch ein offiziellen Standard dar. Es handelt sich vielmehr um einen Sofwarearchitekturstil, bestehend aus **Leitsätzen** und **Praktiken** für netzwerkbasierte Systeme.
- REST und HTTP werden dabei häufig in einem Atemzug genannt. Das liegt daran, dass REST typischerweise mit HTTP umgesetzt wird. In diesem Fall spricht man von RESTful HTTP.
- Dabei unterscheidet sich REST vor allem in der Forderung nach einer **einheitlichen Schnittstelle**[37] von anderen Architekturstilen.

[36] *Das Akronym REST steht für Representational State Transfer.*
[37] *HTTP Methoden*

10.1.2 REST Prinzipien

Die Sehnsucht der Entwickler nach einer einfachen Methodik bei der Entwicklung verteilter Systeme, führte zur Formulierung einer Reihe von **Grundprinzipien** für REST Anwendungen.

REST Grundprinzipien:

- **Entkoppelung von Ressourcen und Repräsentationen**
- **Addressierbarkeit**
- **Zustandslosigkeit**
- **Einheitliche Schnittstelle**
- **Hateos**

❑

10.1.3 Addressierbarkeit

Das Internet versteht sich als eine Sammlung von **Ressourcen**.

> Um die Addressierbarkeit von Ressourcen zu ermöglichen, wird Ressourcen ein eindeutiger **Schlüssel** zuwiesen.

▸ **Erklärung: Addressierbarkeit** ▾

- Durch den Einsatz von URIs[38] können Ressourcen im Internet einfach identifiziert werden.
- Damit wird es möglich Ressourcen durch die Verwendung von **HTTP Links** zu verwaltet.

❑

[38] *Unique Ressource Identifier*

10.1.4 Entkoppelung von Ressource und Repräsentation

Eine Ressource besitzt im Internet immer mehrere mögliche Repräsentationen [39].

> Fordert ein REST Service eine Ressource an, wird stets eine der **Repräsentationen** der Ressource, nie aber die Ressource selbst bereitgestellt.

▸ **Codebeispiel: JSON Repräsentation** ▾

```
1  //----------------------------------------
2  //  Repraesentation von Ressourcen
3  //----------------------------------------
4  var person = {
5     id : 4,
6     lastName : "Puntigam",
7     firstName : "Franz",
8     gender : male,
9     weight : 78,
10    height : 1.75
11 }
```

❑

10.1.5 Zustandslosigkeit

Zur Reduktion der Komplexität in der Entwicklung verteilter Systeme, hat die **Kommunikation** in einer REST Anwendung zustandslos, zu sein.

▸ **Erklärung: Zustandslosigkeit** ▾

- Konsequenterweise gibt es bei REST konformen Anwendungen keinen **Sitzungsstatus**, der serverseitig über mehrere Clientanfragen hinweg vorgehalten wird.
- Stattdessen muss der **Kommunikationszustand** im Client oder in der Repräsentation der Ressource gespeichert werden.
- Damit kann die **Koppelung**[40] zwischen Client und Server verringert werden.

❑

[39] *Eine Datenbankentität kann z.B. als xml oder json File verschickt werden*

[40] *Damit wird es möglich dass REST Anwendungen Caches, Load Balancer und andere* ***Skalierungsartefakte*** *nutzten .*

10.1.6 Hypermedia - Hateos

Hypermedia

Hypermedia bezeichnet eine nichtlineare Form von **Medien**, deren Hauptcharakteristikum die gegenseitige **Verlinkung** untereinander, ausmacht.

HTML ist dabei der wohl prominenteste Vertreter des Hypermedia Paradigmas.

Erklärung: Hateos

- Hateos ist ein Acronym und steht für *Hypermedia as the engine of application state*.
- Mit *engine* ist im Kontext von REST die **Zustandsverwaltung** für Ressourcen gemeint.
- REST Ressourcen können in unterschiedlichen **Zustände** vorliegen. Die verschiedenen Zustände einer Ressource können dabei in Form von Zustandsdiagrammen abstrahiert werden.
- In REST Anwendungen wird der Zustand einer Ressource unter Verwendung seiner **Repräsentation** verwaltet.
- Repräsentaionen können bevorstehende **Zustandsübergänge** in Form von Links an Clients weitergeben.

Erklärung: Zustandsdiagramm[41]

- Ein Zustandsdiagramm besteht aus **Knoten**, die durch **Kanten** miteinander verbunden werden.
- Knoten repräsentieren dabei die möglichen **Zustände** einer Ressource. Kanten entsprechen den möglichen Zustandsübergänge.
- Eine Kante führt demzufolge von einem Ausgangszustand zu einem Folgezustand.
- Ressourcenrepräsentationen geben die möglichen Zustandsübergänge in Form von **Links** an den Client weiter.

❑

Codebeispiel: Produktbestellung

```
//----------------------------------------
//            JSON - Order
//----------------------------------------
var order = {
   href    : "http://../orders/21",
   status  : "created",
   date    : "2014-10-03",
   total   : 2090,
   updated : "2015-01-11",
   user: {
      href:"http://../users/14",
      user-name : "Tim"
   },

   // Das actions array enthaelt alle
   // moeglichen Zustandsuebergaenge.

   // Der Client kann einen Zustands-
   // uebergang ausloesen in dem er
   // einfach dem entsprechendem
   // Link folgt!

   actions: [
      {
         href: "http://.../orders/34/cancel",
         description: "cancel order"
      },{
          href: "http://.../orders/34/process",
          description: "process order"
      },{
         href: "http://.../orders/34/update",
         description: "update order"
      }
   ],
   products : [
      {
            href :"http://../21/products/47",
            name : "Der Gladiator",
            category : "DVD"
       }
    ]
}
```

❑

[41] *Statemachine*

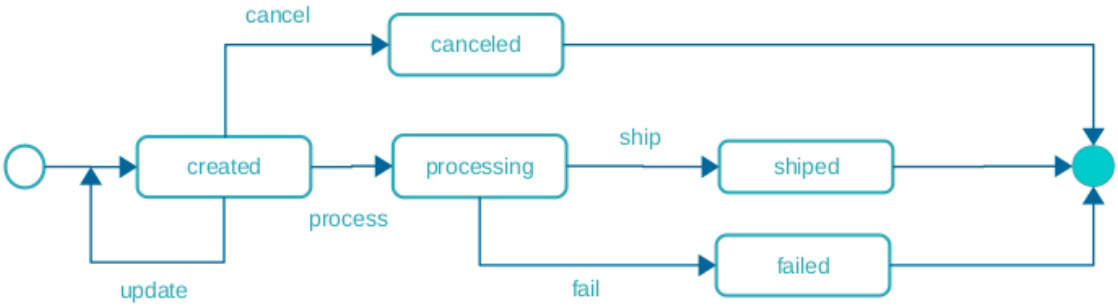


Abbildung 8. **State Machine: Produktbestellung**

10.1.7 Einheitliche Schnittstelle

REST Anwendungen verwalten Ressourcen über einen einheitlichen Satz von **Standardmethoden**.

> Zur Verwaltung von Ressourcen werden die im HTTP Protokoll definierten **HTTP Methoden** verwendet.

▸ Erklärung: REST Schnittstelle

- REST Anwendungen stellen zur Verwaltung von Ressourcen eine **einheitliche Schnittstelle** zur Verfügung.
- Damit wird der Zugriff auf Ressourcen für REST Anwendungen **standardisiert**. Die Kommunikation bzw. Orchestrierung von Servicen wird damit signifikant erleichtert.

❑

▸ Codebeispiel: Schnittstelle einer Ressource

```
1  //---------------------------------------
2  //    interface: IHttpRessource
3  //---------------------------------------
4  public interface IHttpRessource{
5     HttpResponse get();
6
7     HttpResponse put();
8
9     HttpResponse post();
10
11    HttpResponse patch();
12
13    HttpResponse delete();
14
15    HttpResponse options();
16
17 }
18
19 public interface IProduct : IHttpRessource{
20    void update();
21
22    void cancel();
23
24    void process();
25
26    void ship();
27
28    void fail();
29
30 }
```

❑

10.2. Ressourcen

Primär ist es die Aufgabe einer REST Anwendung die Ressourcen ihrer **Domäne** zu verwalten.

> Damit spielen Ressourcen eine zentrale Rolle im Entwurf von REST Anwendungen.

Rest unterscheidet mehrere **Formen** von Ressourcen:

- **Primärressourcen**
- **Subressourcen**
- **Listenressourcen**

10.2.1 Primärressourcen

Primärressource

Primärressourcen sind jene Ressourcen, die sich beim klassischen **Anwendungsentwurf** sehr früh als Kandidaten für **Entitäten** ergeben.

> Repräsentation von Primärressourcen verstehen sich in erster Linie als **Datenaggregate**.

▸ **Codebeispiel: Primärressource**

```
//----------------------------------------
//            JSON - Project
//----------------------------------------
var project = {
    href : "http://.../projects/1",
    title : "Finite Methods",
    description : "The finite ...",
    creationDate : "10.09.2022",
    projectType: "REQUEST_FUNDING_PROJECT",
    projectState: "RUNNING",
    subprojects : [
      {
         href : "http://../subprojects/2",
         title : "..."
      }, {
         href : "http://../subprojects/7",
         title: "..."
      }
    ],
    projectFundings : [
       ...
    ]
}
```

10.2.2 Subressourcen

Subressource

Ressourcen, die als Teil anderer Ressourcen auftreten, werden als Subressourcen bezeichnet.

▸ **Codebeispiel: Subressource**

```
//----------------------------------------
//  Subressource
//----------------------------------------
var project = {
    href : "http://.../projects/1",
    title : "Finite Methods ...",
    description : "....",
    creationDate : "10.09.2022",
    //subressource von project
   "project-leader" : {
        "href" : "http://.../staff/23",
        "name" : "Schuettli"
   }
}
```

10.2.3 Listenressourcen

Eine Ressource kann sich entweder auf eine einzelne Entität oder auf eine Collection von Entitäten beziehen.

> Aus REST Sicht ist auch eine **Liste von Entitäten** eine Ressource.

▸ **Codebeispiel: Listenressourcen**

```
//----------------------------------------
//  Listenressourcen Repraesentation
//----------------------------------------
var projects = {
   href : "http://.../rest/projects",
   data : [{
         href : "http://.../projects/1"
         title : "Finite Methods"
      },{
          href : "http://.../projects/2"
         title : "cloud systems research"
      },{
      ...
    }]
}
```

▸ **Erklärung: Typen von Listenressourcen** ▾

- **Paginierung**: Von Weboberfläche sind wir es gewohnt, Suchergebnisse seitenweise präsentiert zu bekommen.

  Das ist in **ROA Anwendungen** nicht anders. Auch hier möchten wir die Daten nicht im Ganzen an den Anwender schicken. Die Paginierung wird über die URI der Ressource gesteuert.

  Uri: Listenressourcen

  `.../projects?start=0&count=20`

- **Filter:** Um Listenressourcen nach bestimmten Kriterien zusammenzustellen werden **Filter** eingesetzt.

  z.B.: die Liste aller Kunden aus der *Region Nord* oder alle Produkte deren Bezeichnung mit einem A beginnt

  Uri: Listenressourcen

  `.../customers?region=Nord`

  `.../products?name=a*`

❑

.

10.3. Http Methoden

Die **Standardisierung** der Semantik und des Verhalten der Schnittstellen von ROA Anwendungen, wird durch den Einsatz der **HTTP Methoden** erreicht.

HTTP Methoden:

- **GET**
- **PUT**
- **POST**
- **PATCH**
- **DELETE**
- **HEAD**
- **OPTIONS**

10.3.1 HTTP Methode - GET

GET ist die wohl wichtigste und am häufigste verwendete HTTP Methode.

▸ **Erklärung: GET Methode**

- Die *GET* Methode gibt die Repräsentation einer Ressource an den Aufrufer der Methode zurück. Sie ist die wichtigste **Leseoperation** des HTTP Protokolls.
- Der Aufruf der Methode erfolgt gemeinsam mit der Angabe mit der *URI* der Ressource.
- Der Client wird solange in seiner Ausführung blockiert, bis er Zugriff auf die Daten erhält.

❑

10.3.2 HTTP Methode - PUT

PUT Methode

Die PUT Methode erfüllt in einer ROA Anwendung 2 Rollen:

- **Anlegen** von Ressourcen.
- **Ändern** von Ressourcen.

▸ **Erklärung: Eigenschaften von PUT**

- Die *PUT* Methode ist das Gegenstück zur *GET* Methode.
- In erster Linie wird die Methode zum Ändern vorhandener Ressourcen verwendet.
- Soll zum Anlegen einer Ressource die *PUT* Methode verwendet werden, muss die URI der Ressource bereits vom **Client** festgelegt werden.

❑

10.3.3 HTTP Methode - POST

POST Methode

Die POST Methode wird in erster Linie verwendet um Ressourcen anzulegen.

- Jedoch sehen wir auch die Verwendung von POST zum Ändern von Ressoucen.

Weil *POST* keine **semantischen Garantien** erfüllen muss, wird es in der Regel zum Anstossen von Operationen verwendet, die nicht von der Semantik der anderen HTTP Methoden abgedeckt werden.

▸ **Erklärung: POST vs PUT**

- *PUT* und *POST* können beide verwendet um Ressourcen anzulegen.
- Soll zum Anlegen einer Ressource die *POST* Methode verwendet werden, muss die URI der Ressource vom **Server** festgelegt werden.

❑

10.3.4 HTTP Methode - DELETE

delete Methode

Die DELETE Methode wird verwendet um Ressourcen zu löschen.

❑

10.3.5 HTTP Methode - PATCH

Die *PATCH* Methode wird verwendet um einzelne Teile einer Ressource zu ändern.

▸ **Erklärung: PATCH vs PUT**

- *PATCH* und *PUT* werden beide verwendet um Daten zu **ändern**.
- Der Unterschied liegt im Ausmass der Änderung:
  - **patch:** Mit der *PATCH* Methode wird ein **Teil** der Ressource geändert.
  - **put:** Mit der *PATCH* Methode wird die gesamte Ressource ersetzt.
- Mit der *PATCH* Methode ist die Intention des Clients in **semantischer** Hinsicht wesentlich klarer als mit der *PUT* Methode.

❑

10.3.6 HTTP Methode - OPTIONS

Die Methode liefert die möglichen **Kommunikationsoptionen** einer Ressource.

▸ **Erklärung: Eigenschaften von OPTIONS**

- Die *OPTIONS* Methode gibt für eine Ressourcem, ihre Schnittstelle zurück.
- Die *OPTIONS* Methode ist **idempotent**.

❑

10.4. Web Api Entwicklung - Fallbeispiel Ordermanager

Beim Entwurf einer **Web API** haben wir die Aufgabe, passende **Ressourcen** zu finden und geeignete URIs festzulegen.

10.4.1 Ressourcen einer Anwendung

Prinzipiell kann jedes Objekt, das **Ziel** eines Link sein könnte, einer Ressource sein.

▸ **Auflistung: Ressourcen**

- **Primärressourcen:** Die Schnittstelle von Primärressourcen unterstützen in der Regel alle HTTP Methoden.
  - ▸ **User**, **Order**, **Review**, **Product**
- **Subressourcen**: Subressourcen stehen immer in Abhängigkeit zu einer Primärressource.
  - ▸ **User**, **Review**, **Product**
- **Listenressourcen**: Listenresourcen repräsentieren in erster Linie eine Sammlung von Subressourcen.
  - ▸ **Userlisten**, **Produktlisten**, **Reviewlisten**, **Orderlisten**

❑

10.4.2 Repräsentationen von Ressourcen

▸ **Codebeispiel: XML Repräsentationen User**

```xml
<!-- ---------------------------------- -->
<!--      XML Repraesentation von User   -->
<!-- ---------------------------------- -->
<?xml version="2.0"?>
<user>
   <href rel="self">
      http://.../ordermanager/users/32
   </href>

   <first-name>Franz</first-name>
   <middle-name>Josef</middle-name>
   <last-name>Kurz</last-name>

   <user-name>shorty134</user-name>
   <joined-at>2011-11-11</joined-at>
</user>
```

▸ **Codebeispiel: Json Repräsentationen User**

```javascript
//---------------------------------------
//      JSON Repraesentation: User
//---------------------------------------
var user = {
    href : "http://.../ordermanager/users/32",
    first-name :"Franz",
    middle-name :"Josef",
    last-name  :"Kurz",
    user-name  :"shorty134"
    joined-at  :"2011-11-11"
}

var product = {
    href :"http://./ordermanager/products/1",
    //Stammdaten
    name :"Forbidden Stars"
    search-terms : [
      "toy", "boardgame", "40k",
      "forbidden stars"
    ],
    description :" ...",
    price : 99,

   //Subressource
   //Listenressource
    reviews : {
      href : "http://.../users/32/reviews",
        entries :[
          {
            user: {
               href :"http://.../users/32",
               user-name : "huki"
            },
            text : "Das ist ein tolles ..."
          }, {
             user: {
               href :"http://.../users/12",
               user-name : "gronkh"
             },
             text : "bla bla ..."
          }, {
             user: {
               href :"http://.../users/0",
               user-name : "macco"
             },
             text : "??? ..."
          }
      ]
   }
}
```

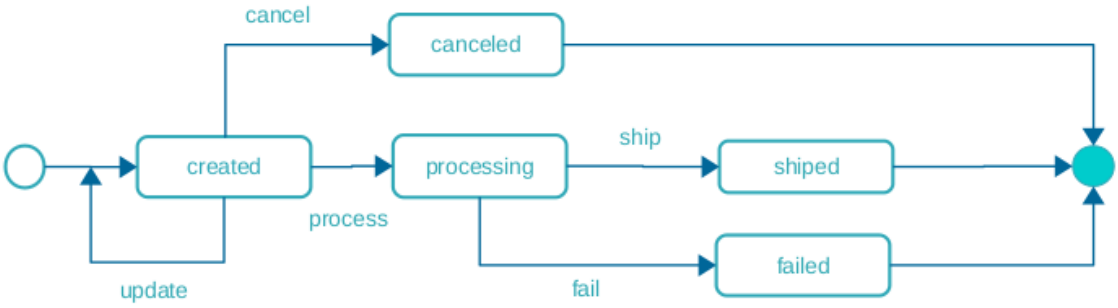


Abbildung 9. **Zustände einer Bestellung**

▸ Codebeispiel: Einzelne Bestellung

```
//----------------------------------------
//    JSON Repraesentation: Order
//----------------------------------------
var order = {
   href : "http://../orders/21",
   status : "processing",
   date   : "2014-10-03",
   total  : 2090,
   updated : "2015-01-11",
   user : {
      href :"http://../users/14",
      user-name: "Tim"
   },
   items : [
      {
         quantity: 1,
         product : {
            href : "http://../products/10",
            description : "Laptop X65",
            price : 799
      }, {
         quantity: 4,
         product : {
            href : "http://../products/13",
            description : "MP3 Player",
            price : 99
         }
      }, {
         quantity: 1,
         product : {
            href : "http://../products/7",
            description : "Boardgame",
            price : 70
         }
   ]
}
```

▸ Codebeispiel: Liste der Bestellungen

```
//----------------------------------------
//      JSON Repraesentation: orders
//----------------------------------------
var orders = {
   href :"http://../orders?start=1&count=10",
   orders : [
      {
         href : "http://../orders/32",
         date : "2014-10-03",
         price : 315,
         items : [
            "Laptop X65", "MP3 Player"
         ]
      }, {
         href : "http://../orders/2",
         date : "2015-05-03",
         price : 100,
         items : [
            "Forbidden Stars"
         ]
      }, {
         href : "http://../orders/2",
         date : "2015-05-03",
         price : 132,
         items : [
            "Chess - Lotzi"
         ]
      }, {
         href : "http://../orders/2",
         date : "2015-05-03",
         price : 525,
         items : [
            "Warhammer 40K"
         ]
      }
   ]
}
```

10.4.3 Analyse einer Repräsentation

▸ Codebeispiel: Order Repräsentation ▾

```
//----------------------------------------
//      JSON Repraesentation: order
//----------------------------------------
var order = {
   href :"http://../orders/21",
   state  : "processing",
   date   : "2014-10-03",
   total  : 2090,
   updated : "2015-01-11",
   user: {
      href :"http://../users/14",
      user-name : "Tim"
   },
   items: [
      {
         quantity : 1,
         product : {
            href :"http://../products/10",
            isbn : "3233-32HU-3",
            description : "Laptop X65",
             price : 799
          },
      }, {
         quantity : 1,
         product : {
            href :"http://../products/19",
            isbn : "3233-8763-3",
            description : "Forbidden Stars",
             price : 150
          }
      }
   }],
   actions : [
      {
         href :"http://../orders/21/cancel",
         description: "cancel order"
      },{
            href :"http://../orders/21/process",
         description: "process order"
      },{
            href:"http://../orders/21/update",
            description: "update order"
      }
   ]
}
```

▸ Analyse: Order Repräsentation ▾

- **Hypermedia:** Der Link auf sich selbst ist stets Teil der Repräsentation einer Ressource.

```
var order = {
   href :"http://../orders/21",
   ...
}
```

- **Stammdaten der Ressource:**

```
var order = {
   state  : "processing",
   date   : "2014-10-03",
   total  : 2090,
   updated : "2015-01-11",
   ...
}
```

- **Subressourcen:**

```
var order = {
   ...
   user: {
      href :"http://../users/14",
      user-name : "Tim"
   },
   items: [
      {
         quantity : 1,
         product : {
            href :"http://../products/10",
            isbn : "3233-32HU-3",
            description : "Laptop X65",
             price : 799
          },
      }, {
         ...
      }
   }],
   actions : [
      {
         href :"http://../orders/21/cancel",
         description: "cancel order"
      },
      ...
   ]
}
```

- **Zustand einer Ressource**: Eine Bestellung durchläuft eine Reihe von **Statusübergängen**.

  Der Status einer Bestellung ist Teil der Stammdaten der Ressource.

```
var order = {
   href :"http://../orders/21",
   state : "processing",
   ...
}
```

- **Zustandänderung:** In einer REST Anwendung ist die Ressource selbst für die Verwaltung ihres Zustandes verantwortlich.

  Abhängig vom Zustand der Ressource werden Methoden zur Verwaltung des Zustandes publiziert.

```
var order = {
   href :"http://../orders/21",
   state  : "processing",
   ...
   actions : [
      {
         href :"http://../orders/21/cancel",
         description: "cancel order"
      },{
           href :"http://../orders/21/process",
         description: "process order"
      },{
           href:"http://../orders/21/update",
           description: "update order"
      }
   ]
}
```

❑

.