

# Softwareentwicklung

Skriptum zur Vorlesung - 01.09.2023

Dipl.-Ing. Paul Panhofer BSc.<sup>1\*</sup>

<sup>1</sup> ZID, TU Wien, Taubstummengasse 11, 1040, Wien, Austria

**Abstract:**

**MSC:** p.panhofer@htlkrems.at

**Keywords:**

Contents	
<b>1. Grundlagen der Programmierung</b>	6
1.1. Softwareprogramm	6
1.1.1. Programmastein: Anweisung	6
1.2. Datentypen und Variablen	7
1.2.1. Variablen Deklaration/Initialisierung	7
1.2.2. Datentypen	8
1.3. Kontrollstruktur	8
1.3.1. Kontrollstruktur: IF Bedingung	8
1.3.2. Bedingungen auswerten	9
1.3.3. Kontrollstruktur: WHILE	10
1.3.4. Kontrollstruktur: FOR	10
1.3.5. Kontrollstruktur: FOREACH	11
<b>2. Grundlagen der objektorientierten Programmierung</b>	12
2.1. Konzepte der Objektorientierung	12
2.1.1. Objektorientierung	12
2.1.2. Objekte	13
2.2. Elemente einer Klasse	14
2.2.1. Fallbeispiel: Last Aurora	14
2.2.2. Klassenelement: Konstruktur	14
2.2.3. Klassenelement: Properties	15
2.2.4. Klassenelement: Methoden	15
2.3. Speichermanagement	16
2.3.1. Arten von Datentypen	16
2.3.2. Speichermanagement: Stack	17
2.3.3. Speichermanagement: Heap	17
2.3.4. Programmefakt Zeiger	18
2.4. Vererbung	18
2.4.1. Fallbeispiel: Person	18
2.4.2. Klassenbeziehungen	19
2.4.3. Abstrakte Klassen	19
<b>3. Konzepte der objektorientierten Programmierung</b>	20
3.1. Konzept: Identität	20
3.1.1. Referenzgleichheit	20
3.1.2. Wertegleichheit	21
<b>4. Datentyp: Referenztypen</b>	22
4.1. Klassen	22
4.1.1. Klassendeklaration	22
4.1.2. Elemente einer Klasse	23

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

4.1.3. Klassenelement: Variablen	23	<b>8. Programmierung: SOLID</b>	56
4.1.4. Klassenelement: Properties	25	8.1. SOLID Prinzipien	56
4.1.5. Indexer	26	8.1.1. SOLID Prinzipien	56
<b>5. Datentyp: Collections</b>	28	8.2. Single Responsibility Prinzip	57
5.1. Datenstrukturen	28	8.2.1. Diskussion SRP	57
5.1.1. Grundlagen	28	8.3. Interface Segregation Prinzip	57
5.2. Datenstruktur: List	29	8.3.1. Interface Segregation Prinzip	57
5.2.1. Verhalten von Listen	29	8.4. Open Closed Prinzip	58
5.3. Datenstruktur: Stack	32	8.4.1. Fallbeispiel: Open Closed	58
5.3.1. Werteverarbeitung	32	Prinzip	58
5.3.2. Fallbeispiel: Stack	32	8.5. Liskovsche Substitutinsprinzip	59
5.4. Datenstruktur: Queue	35	8.5.1. Fallbeispiel: Substitutionsprinzip	59
5.4.1. Verhalten von Queues	35		
5.4.2. Fallbeispiel: Queue	35		
5.5. Datenstruktur: Dictionary	37	<b>9. Programmierung: OOP Entwurf</b>	60
5.5.1. Fallbeispiel: Array vs. Dictionary	37	9.1. Entwurfsmuster	60
5.5.2. Fallbeispiel: Dictionary	37	9.1.1. Arten von Pattern	60
5.5.3. Wertezugriff	39	9.1.2. Einsatz von Entwurfsmustern	61
<b>6. Programmierung: Strukturierung</b>	42	9.2. Erzeugermuster	61
6.1. Unterprogramme	42	9.2.1. Erzeugermuster - Singleton	61
6.1.1. Unterprogramme	42	9.2.2. Erzeugermuster - Factory	62
6.2. Objektorientierung	43	9.3. Strukturmuster	64
6.2.1. Objektorientierung	43	9.3.1. Strukturmuster - Adapter	64
6.3. Schichtenmodell	43	9.3.2. Strukturmuster - Dekorator	65
6.3.1. Prinzipien des	43	9.4. Verhaltensmuster	66
Schichtenmodells	43	9.4.1. Verhaltensmuster - Command	66
6.3.2. Fallbeispiel: Schichtenmodell	44	9.4.2. Verhaltensmuster - Strategy	68
6.4. Komponenten	46	<b>10. Architekturstil: Rest</b>	72
6.4.1. Fallbeispiel:	46	10.1. REST Prinzipien	72
Restaurantverwaltung	46	10.1.1. Architekturstil REST	72
6.5. Service	47	10.1.2. REST Prinzipien	73
6.5.1. Zusammenfassung	47	10.1.3. Addressierbarkeit	73
<b>7. Programmierung: Metriken</b>	50	10.1.4. Entkoppelung von Ressource	73
7.1. Softwaremetriken	50	und Repräsentation	73
7.1.1. Metriken	50	10.1.5. Zustandslosigkeit	73
7.1.2. Qualitätsmetriken	50	10.1.6. Hypermedia - Hateoas	74
7.2. Koppelung	51	10.1.7. Einheitliche Schnittstelle	75
7.2.1. Stufen der Koppelung	51	<b>10.2. Ressourcen</b>	76
7.2.2. Arten der Koppelung	51	10.2.1. Primärressourcen	76
7.2.3. Interaktionskoppelung	51	10.2.2. Subressourcen	76
7.2.4. Auflösen von	52	10.2.3. Listenressourcen	76
Interaktionskoppelung	52	<b>10.3. Http Methoden</b>	78
7.2.5. Fallbeispiel: Auflösen von	52	10.3.1. HTTP Methode - GET	78
Interaktionskoppelung	52	10.3.2. HTTP Methode - PUT	78
7.2.6. Vererbungskoppelung	53	10.3.3. HTTP Methode - POST	78
7.2.7. Objektkomposition	53	10.3.4. HTTP Methode - DELETE	79
7.2.8. Programmmethodik	54	10.3.5. HTTP Methode - PATCH	79
7.3. Kohäsion	55	10.3.6. HTTP Methode - OPTIONS	79
7.3.1. Kohäsion	55	<b>10.4. Web Api Entwicklung - Fallbeispiel</b>	80
7.3.2. Fallbeispiel: Servicekohäsion	55	Ordermanager	80
		10.4.1. Ressourcen einer Anwendung	80
		10.4.2. Repräsentationen von	
		Ressourcen	80

### 10.4.3. Analyse einer Repräsentation 82



# Grundlagen der objektorientierten Programmierung

December 14, 2019

## 1. Grundlagen der Programmierung

# 01

### Grundlagen der Programmierung

01. Softwareprogramm	6
02. Datentypen und Variablen	7
03. Kontrollstrukturen	8

### 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<sup>1</sup> 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**<sup>2</sup> und der **Bezeichner**<sup>3</sup> 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.



<sup>1</sup> Unter einem Betriebssystemprozess verstehen wir ein sich in Ausführung befindendes Programm

<sup>2</sup> Typ

<sup>3</sup> 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 Variablendeclaration/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: Variablendeclaration ▾

- Stößt das Betriebssystem zur **Laufzeit** eines Programms auf eine Variablendeclaration, reserviert es für die Variable Speicherplatz im Arbeitsspeicher.
- Mit der Variablendeclaration kann einer Variable auch ein Wert zugeordnet<sup>4</sup> werden.

#### ► Codebeispiel: Variablendeclaration ▾

```

1 // -----
2 // Variablendeclaration/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;

```

<sup>4</sup> Variableninitialisierung

## 1.2.2 Datentypen

Infolge einer **Variablen-deklaration** 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 Auspraeh-
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**.

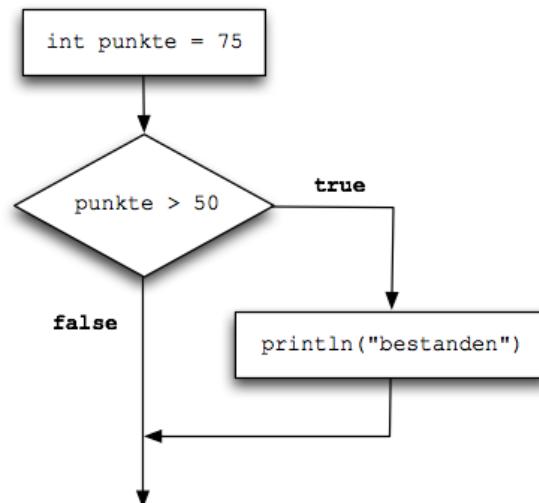
Damit bestimmen Kontrollstrukturen, die **Reihenfolge** in der die Anweisungen eines Programmes ausgeführt werden.

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 ▾

```

1  // -----
2  // SYNTAX: IF Bedingung
3  // -----
4  if (<condition>) {
5  // condition trifft zu
6  <operations>
7 } else {
8 // condition trifft nicht zu
9 <operations>
10 }

11 // -----
12 // Beispiel: IF Bedingung
13 // -----
14

15 int points = 75;

16

17 // Auswertung der Bedingung
18 if (points > 50) {
19 // Falls die Bedingung zutrifft werden die
20 // nachfolgenden Befehle ausgefuehrt
21     console.info("You passed your exam");
22 } else {
23 // Trifft die Bedingung nicht zu werden die
24 // Befehle in diesem Block ausgefuehrt
25     console.info("You failed your exam");
26 }

27 // -----
28 // Beispiel: IF Bedingung
29 // -----
30

31 // Ermitteln Sie den groesseren Wert 3er
32 // Variablen und geben Sie ihn aus.
33 int x = 24;
34 int y = -121;
35 int z = 53;

36

37 if (x > y) { // x > y
38     if (x > z) {
39         Console.WriteLine(x);
40     } else {
41         Console.WriteLine(z);
42     }
43 } else { // x <= y
44     if (y > z) {
45         Console.WriteLine(y);
46     } else {
47         Console.WriteLine(z);
48     }
49 }

```

## 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 ▾

```

1  // -----
2  // Bedingung Auswerten
3  // -----
4  // Der gewuenschte String wird immer ausge-
5  // geben.
6  if (true) { // --> wird zu true ausgew.
7      Console.WriteLine("Hello world");
8 }

9

10 int x = 21;
11 if (x > 0) { // --> wird zu true ausgew.
12     Console.WriteLine("value is positive");
13 }

14

15

16 int a = 7;
17 int b = 7;
18 int c = 4;

19

20 // Der == Operator prueft 2 Werte auf
21 // Gleichheit.
22 if (a == b) { // --> wird zu true ausgew.
23     Console.WriteLine("values are equal");
24 }

25

26

27 if (a != c){ // --> wird zu true ausgew.
28     Console.WriteLine("values are not equal");
29 }

30

31 // Verknuepfung mehrere Bedingungen mit
32 // && (und) bzw. || (oder).
33 if( a >= b && a >= c){ // --> true
34     Console.WriteLine("a ist max");
35 }

```



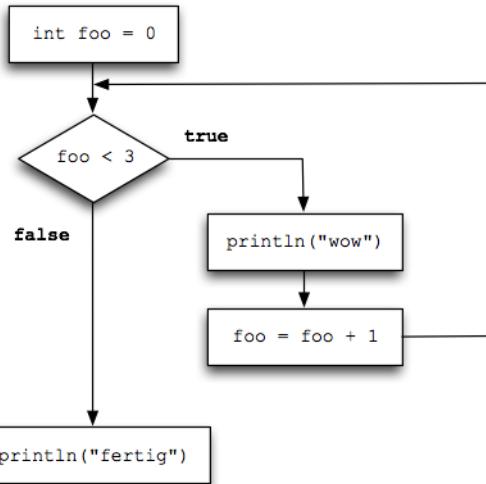
### 1.3.3 Kontrollstruktur: WHILE



#### Schleife ▾

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

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



#### ► Codebeispiel: while Schleife ▾

```

1 // -----
2 // SYNTAX: WHILE Schleife
3 // -----
4 while ( <condition> ) {
5     <operations>
6     ...
7 }
8 // -----
10 // Beispiel: WHILE Schleife
11 // -----
12 int foo = 0;
13
14 while ( foo < 3 ) {
15     Console.WriteLine("wow");
16     foo += 1;
17 }
18
19 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 ▾

```

1 // -----
2 // SYNTAX: FOR Schleife
3 // -----
4 // Die Syntax der for Schleife ist immer
5 // gleich: Eingeleitet wird die Schleife
6 // durch das Schluesselwort for.
7
8 // Danach folgt in runden Klammern die
9 // Initialisierung der Zaehlvariable.
10
11 // Die 3 Komponenten werden dabei durch ein
12 // Semikolon voneinander getrennt.
13
14 for( <init>; <condition>; <modifier> ) {
15     <operations>
16     ...
17 }
18
19 // -----
20 // Beispiel: FOR Schleife
21 // -----
22 // Ausgabe aller Zahlen von 0 - 100
23 for (int i = 0; i < 101; ++i) {
24     Console.WriteLine(i);
25 }
26
27 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**<sup>5</sup>.
- Die foreach Schleife definiert eine **Laufvariable**. Die Laufvariable referenziert jeweils das aktuell zu durchlaufende Objekt der Datenstruktur.
- Das bedeutet jedoch, dass wir im Gegensatz zur for Schleife, jeweils nur auf das gegenwärtige zu verarbeitende Element direkten Zugriff haben.

#### ► Codebeispiel: foreach Schleife ▾

```

1 // -----
2 // SYNTAX: foreach Schleife
3 // -----
4 // Die Syntax der foreach Schleife ist immer
5 // gleich: Eingeleitet wird die Schleife
6 // durch das Schluesselwort foreach.
7
8 // Danach folgt in runden Klammern die
9 // Initialisierung der Laufvariable. Die
10 // Laufvariable referenziert das jeweils
11 // aktuell zu durchlaufende Objekt der
12 // Datenstruktur.
13
14 // Mit jedem Schleifendurchlauf wird je-
15 // weils das sequentiell naechste Element
16 // der Datenstruktur referenziert.
17 foreach (<variable> in <collection>) {
18     <operations>
19     ...
20 }
21
22 // -----
23 // Beispiele: foreach Schleife
24 // -----
25 // 1. Beispiel) Geben Sie alle Elemente des
26 // folgenden Arrays aus.
27 string[] names = new {
28     "Alfred", "Hugo", "Franz", "Ali"
29 };
30
31 foreach (var name of names) {
32     Console.WriteLine(name);
33 }
34
35 // Ausgabe:
36 // Alfred
37 // Hugo
38 // Franz
39 // Ali
40
41 // 2. Beispiel) Berechnen Sie die Summe
42 // der Werte eines Arrays
43 int[] numbers = new {1, 5, 87, 23, 87, 23};
44 int sum = 0;
45
46 foreach(var n in numbers){
47     sum += n;
48 }
49 Console.WriteLine(sum);
  
```

<sup>5</sup> Die Zählvariable wird in der Regel als Arrayindex für den Zugriff auf die einzelnen Elemente des Arrays verwendet.

## 2. Grundlagen der objektorientierten Programmierung

# 03

### Grundlagen der OOP

01. Konzepte der Objektorientierung	12
02. Elemente einer Klasse	14
03. Speichermanagement	16
04. Vererbung	18

### 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<sup>6</sup> 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.



<sup>6</sup> Tastendruck, Maus

## 2.1.2 Objekte

Objekte werden durch folgende Größen charakterisiert: den **Eigenschaften**, 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 ▾

```

1  // -----
2  // Klasse: Airship.cs
3  // -----
4  // (1) Classdefinition
5  public class Airship {
6      // (2) Properties
7      public int X { get; set; }
8      public int Y { get; set; }
9      public string Code { get; set; }
10     public int Speed { get; set; }
11     public int PullForce { get; set; }
12     public List<Keyword> Keywords { get; set; }
13     public List<Compartment> Compartments
14         { get; set; }
15
16     // (3) Constructor
17     public Airship () {
18         Keywords = new ();
19         Compartments = new ();
20     }
21
22     // (4) Methodes
23     public void AddCompartment(Compartment c){
24         if(Compartments.Length < PullForce) {
25             Compartments.Add(c);
26         }
27     }
28
29     public void Move(int x, int y){
30         this.X = x; this.Y = y;
31     }
32 }
33
34 // Instanzieren eines Airship Objekts
35 Airship a = new Airship();
36 // Werte setzen
37 a.X = 4; a.Y = 10;
38 a.Code = "Dragon Spire";
39 a.Speed = 7;
40 a.PullForce = 4;

```

## 2.2.2 Klassenelement: Konstruktor



### 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**<sup>7</sup>.
- Der Konstruktor hat dabei denselben Namen wie die Klasse.

#### ► Codebeispiel: Konstruktor ▾

```

1  // -----
2  // Objektinstanzierung
3  // -----
4  // <Klassentyp> <Objektname> = new
5  //     <Konstruktor>;
6
7  // Objektinstanzierung: das Airship Objekt a
8  // kann nach der Instanzierung verwendet
9  // werden.
10 Airship a = new Airship();
11 // Instanzierung eines weiteren Objekts
12 Airship b = new Airship();
13
14 // Wird nur ein Variable definiert ohne
15 // den Konstruktor aufzurufen, kann nicht
16 // auf die Properties des Objekts zuge-
17 // griffen werden, weil das Objekt noch
18 // gar nicht existiert.
19 Airship c;
20
21 // Nach dem Aufruf des Konstruktors wird
22 // der Variable ein Objekt im Speicher
23 // zugewiesen.
24 c = new Airship();

```



<sup>7</sup> 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 ▾

```

1  // -----
2  // Zugriff auf Properties
3  // -----
4  // Bevor die Properties eines Objekts be-
5  // arbeitet werden koennen muss das Objekt
6  // instanziert werden.
7  Airship a1 = new Airship();
8  // Wertzuweisung
9  a1.X = 3; a1.Y = 10;
10 a1.Code = "Dragon Spire";
11 a1.Speed = 7;
12 a1.PullForce = 4;
13
14 Airship a2 = new Airship();
15 a2.X = 12; a2.Y = 8;
16 a2.Code = "Queen Mallon";
17 a2.Speed = 5;
18 a2.PullForce = 2;
19
20 // Pruefung der Werte ueber Unitests
21 // Wertezugriff
22 Assert.That (
23     a1.Code, Is.EqualTo("Dragon Spire")
24 );
25 Assert.That(a1.Speed, Is.EqualTo(7));
26 Assert.That(a1.PullForce, Is.EqualTo(4));
27
28 Assert.That (
29     a2.Code, Is.EqualTo("Queen Mallon")
30 );
31 Assert.That(a2.Speed, Is.EqualTo(5));
32 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 grundlegenden Eigenschaften einer Methode. Zum Methodenkopf gehört ein *Methodename*<sup>8</sup>, eine Reihe von *Parametern* und der *Rückgabetyp*<sup>9</sup> der Methode.
- Im **Methodenrumpf** wird das gewünschte Verhalten - die Logik - der Methode implementiert.



<sup>8</sup> 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.

<sup>9</sup> Methoden können einen Wert an den Aufrufer zurückgegeben werden. Fall eine Methode keinen Rückgabewert hat, wird dies mit dem Schlüsselwort *void* angezeigt.

### ► Codebeispiel: Methodendefinition ▾

```

1  // -----
2  // Methoden
3  // -----
4  public class Airship {
5
6      public int X { get; set; }
7      public int Y { get; set; }
8      public string Code { get; set; }
9      public int Speed { get; set; }
10     public int PullForce { get; set; }
11
12     public List<Keyword> Keywords { get; set; }
13
14     public List<Compartment> Compartments
15         { get; set; }
16
17     public Airship () {
18         Keywords = new ();
19         Compartments = new ();
20     }
21
22     // Methodendefinition
23     /* Methodenkopf:
24         @Methodenname: AddComponent
25         @Parameter: Component c
26         @Rueckgabewert: void
27     */
28     public void AddCompartment(Compartment c){
29     // Methodenrumpf
30         if(Compartments.Length < PullForce) {
31             Compartments.Add(c);
32         }
33     }
34
35     // Methodendefinition
36     /* Methodenkopf:
37         Methodenname: Move
38         Parameter: inx x, int y
39         Rueckgabewert: void
40     */
41     public void Move(int x, int y){
42     // Methodenrumpf
43         this.X = x; this.Y = y;
44     }
45
46
47 }
```

## 2.3. Speichermanagement ▾

Im **Speicher** eines Rechners werden die **Daten**<sup>10</sup> 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 Datentypen

Die C# Spezifikation definiert 2 Arten von Datentypen:

- **einfache Datentypen**
- **Referenztypen**



#### 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 Referenztyp 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.



<sup>10</sup> Variablen, Objekte

```

public class GameLauncher {

    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;
    }
}

```

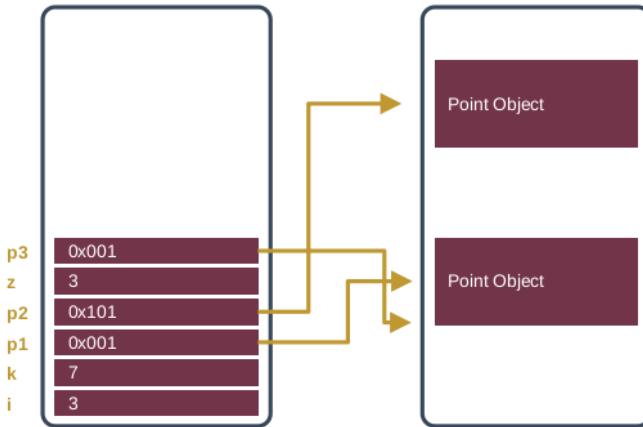


Abbildung 1. Speicherverwaltung: Stack vs. Heap

### 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 werden.
- 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 verwahren. 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 **unstrukturierter 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 Gegensatz zum Stack kann der Heap nicht so einfach verwaltet werden, was ihn im Einsatz langsamer macht.
- Für den **Zugriff** auf den Heap werden **Zeiger**<sup>1112</sup> verwendet.
- Auf technischer Ebene ist ein Zeiger eine **Speicheradresse**, die auf einen Teil des Heaps verweist.

<sup>11</sup> Auf technischer Ebene ist ein Zeiger eine Speicheradresse die auf einen Teil des Heaps verweist.

<sup>12</sup> Synonym: Verweise, Referenzen

### 2.3.4 Programmarteefakt 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: Programmarteefakt 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.WriteLine(p.Info());
29 }
```

```

1 // -----
2 // Subklasse: Student.cs
3 // -----
4 // Die Klasse Student ist eine Subklasse
5 // der Klasse Person. Damit erbt sie das
6 // Verhalten der Person Klasse
7
8 // Die Vererbungsbeziehung zwischen Klassen
9 // wird in der Kopfzeile der Klassendefi-
10 // tion etabliert.
11 public class Student : Person {
12
13     public string StudentCode {
14         get; set;
15     }
16
17     // Der Student Konstruktor ruft ueber das
18     // base Schluesselwort automatisch den Kon-
19     // struktur der Person Klasse auf.
20     public Student (
21         string firstName,
22         string lastName,
23         string studentCode
24     ) : base (firstName, lastName){
25         StudentCode = studentCode;
26     }
27
28 }
29
30 // -----
31 // Subklasse: Programm.cs
32 // -----
33 public class Programm {
34     public static void Main(String[] args){
35         // Ein Student Objekt hat Zugriff auf
36         // das Verhalten der Person Klasse.
37         Student j = new Student (
38             "Jonas", "Nagi", "e9725248"
39         );
40
41         // Als Datentyp kann ein Student Objekt
42         // auch als Person definiert werden
43         Person t = new Student (
44             "Tobias", "Haidi", "e9845323"
45         );
46
47         Console.WriteLine(t.Info());
48     }
49
50 }

```

## 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 Variablen und Methoden der Basisklasse.

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 Schluesselwort `abstract` enthalten.

#### ► Erklärung: Abstrakte 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 ▾

```

1 // -----
2 // Basisklasse: Person.cs
3 // -----
4 public abstract class Person{
5     ...
6 }
7
8 public class Student : Person {
9
10 }

```



### 3. Konzepte der objektorientierten Programmierung

# 04

Konzepte der OOP

01. Konzept: Identität

20

#### 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 ▾

```

1 // -----
2 // Fallbeispiel: Wertegleichheit
3 // -----
4 public class Point {
5     public int X { get; set; }
6     public int Y { get; set; }
7
8     public override bool Equals(object o){
9         // Prüfung ob ein Objekt am Heap
10        // existiert
11        if(ReferenceEquals(null, o))
12            return false;
13
14        // Prüfung auf Referenzgleichheit
15        if(ReferenceEquals(this, obj))
16            return true;
17
18        // Prüfung ob beide Objekte den-
19        // selben Typ haben
20        if(obj.GetType() != this.GetType())
21            return false;
22
23        return Equals((Point)o);
24    }
25
26    protected bool Equals(Point other){
27        return X == other.X && Y == other.Y;
28    }
29 }
```

```

1 // -----
2 // Fallbeispiel: Wertegleichheit
3 // -----
4 public class IdentityTest {
5
6     [SetUp]
7     public void Setup(){
8
9    }
10
11    [Test]
12    public void TestEquals () {
13
14        Point p1 = new Point(3,4);
15        Point p2 = new Point(7,8);
16        Point p3 = p1;
17        Point p4 = new Point(7,8);
18
19        // p1 <=> p2
20        Assert.That (
21            p1, Is.Not.EqualTo(p2)
22        );
23        Assert.False (
24            p1.Equals(p2)
25        );
26
27        // p1 <=> p3
28        Assert.That (
29            p1, Is.EqualTo(p3)
30        );
31        Assert.True (
32            p1.Equals(p3)
33        );
34        Assert.That (
35            p1, Is.Same(p3)
36        );
37
38        // p1 <=> p4
39        Assert.That (
40            p1, Is.EqualTo(p4)
41        );
42        Assert.True(
43            p1.Equals(p4)
44        );
45        Assert.That (
46            p1, Is.Not.Same(p4)
47        );
48    }
49
50 }
```



## 4. Datentyp: Referenztypen

# 05

### Referenztypen

01. Klassen und Objekte	22
02. Elemente einer Klasse	14
05. Speichermanagement	16

### 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 neuen **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 ▾

```

1  // -----
2  // SYNTAX: Klassenelemente
3  // -----
4  public class Airship {
5      // Variable
6      private int _id;
7
8      // Properties
9      public int X { get; set; }
10     public int Y { get; set; }
11
12     // Konstruktoren
13     public Airship () {}
14
15     public Airship (int x, int y){
16         X = x;
17         Y = y;
18     }
19
20     // Methoden
21     public void Move (int x, int y) {
22         X += x;
23         Y += y;
24     }
25 }
```

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



### Zugriffssparameter ▾

Zugriffssparameter 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 protected 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 ▾

```

1  // -----
2  // SYNTAX: Variablendefinition
3  // -----
4  ...
5  [access modifier] [const] [readonly]
6      [required] [ref] [datatype] [name];
7
8
9  public class Airship {
10     ...
11     // Variablendefinition
12     /*
13         access modifier: private
14         datatype: int
15         name: _id
16     */
17     private readonly int _id;
18
19 }
```



## 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 ▾

```

1 // -----
2 // Fallbeispiel: Variablendefinition
3 // -----
4 public class Airship {
5     // Konstante
6     public const string GameType = "AIRSHIP";
7
8     // Variablen
9     public readonly int Id;
10    public required string Code;
11    public required string Name;
12
13    // Properties
14    public int X { get; set; }
15    public int Y { get; set; }
16
17    // Konstruktor
18    public Airship (
19        int id, string code, string name
20    ){
21        Id = id;
22        Code = code;
23        Name = name;
24    }
25
26    public Airship (int id){ Id = id; }
27
28    public string ToString {
29        return $"Airship data: {GameType}
30            {Name}"
31    }
32
33    // Instanzierung eines Airship
34    // Objekts unter Zuhilfenahme des
35    // Initialisierungsoperators
36    Airship b = new Airship(5){
37        Code = "Pufferfish Class",
38        Name = "Queen Mary"
39    };
40
41    Assert.That(
42        b.Code,
43        Is.EqualTo("Pufferfish Class")
44    );
45
46    Assert.That(
47        b.Id, Is.EqualTo(5)
48    );
49
50 }

```



#### 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 ▾

```

1 // -----
2 // Fallbeispiel: Variablendefinition
3 // -----
4 public class Airship {
5     // Properties
6     public int X { get; set; }
7     public int Y { get; set; }
8     public int Id { get; init; }
9
10    public int MinSpeed { get; init; }
11    public int MaxSpeed { get; init; }
12    public int Speed { get; set; }
13
14    public string Code { get; init; }
15    public string Name { get; init; }
16
17    public int StructurePoints { get; set; }
18
19    // Konstruktor
20    public Airship () {
21
22    }
23 }
```

```

1 // -----
2 // Fallbeispiel: Variablendefinition
3 // -----
4 public class AirshipTest {
5
6     [SetUp]
7     public void Setup () {}
8
9     [Test]
10    public void TestCreate () {
11        // Instanzierung eines Airship
12        // Objekts
13        Airship a = new Airship () {
14            Id = 4,
15            Code = "Firefly X11",
16            Name = "Dauntless",
17            MinSpeed = 2,
18            MaxSpeed = 5
19        }
20
21        // Lediglich folgende Properties
22        // koennen nach der Objekt-
23        // initialisierung geaendert
24        // werden
25        a.X = 1;
26        a.Y = 1;
27        a.Speed = 4;
28        a.StructurePoint = 10;
29
30        Assert.That(
31            a.Id,
32            Is.EqualTo(4)
33        );
34
35        Assert.That(
36            a.Name,
37            Is.EqualTo("Dauntless")
38        );
39
40        Assert.That(
41            a.X,
42            Is.EqualTo(1)
43        );
44
45        Assert.That(
46            a.Y,
47            Is.EqualTo(1)
48        );
49    }
50 }
```



### 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 ▾

```

1  // -----
2  // Definition: Indexer
3  // -----
4  public class Airship {
5      // Variables
6      private Weapon[] _weapons = new Weapon[16];
7      // Properties
8      public string Name { get; set; }
9
10     // Definition eines Indexers fuer das
11     // _weapons Array
12     public Weapon this[int i]{
13         get { return _weapons[i]; }
14         set { _weapon[i] = value; }
15     }
16 }
17
18 public class GameEngineTest {
19     [Test]
20     public void TestIndexer(){
21         Airship a = new ();
22
23         a[0] = new Weapon("Gatling Gun");
24         a[1] = new Weapon("Blaster");
25
26         Assert.That(
27             a[1],
28             Is.EqualTo(new Weapon("Blaster"))
29         );
30         Assert.That(
31             a[0],
32             Is.EqualTo(
33                 new Weapon("Gatling Gun")
34             )
35         );
36     }
37 }
```





## 5. Datentyp: Collections

# 06

### Collections

01. Datenstrukturen	28
02. Datenstruktur: List	29
03. Datenstruktur: Stack	32
04. Datenstruktur: Queue	35
05. Datenstruktur: Dictionary	37

### 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**<sup>13</sup> 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 Objekten 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.



<sup>13</sup> Container

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

```

1 // -----
2 // Syntax: Add
3 // -----
4 // Mit der Add Methode kann ein neues
5 // Element am Ende der Liste eingefuegt
6 // werden.
7 public void Add(T item){...};

```

```

1 // -----
2 // Method: Add
3 // -----
4 public class ListTest {
5     [Test]
6     public void TestAdd(){
7         List<Point> points = new ();
8         points.Add(new Point(3,4));
9         points.Add(new Point(5,6));
10
11        Assert.That(
12            points, Has.Count.EqualTo(2)
13        );
14    }
15
16    [Test]
17    public void TestAddRange() {
18        List<Point> points = new ();
19        points.Add(new Point(3,4));
20        points.Add(new Point(5,6));
21
22        List<Point> copy = new List<>();
23        copy.AddRange(points);
24
25        Assert.That(
26            copy, Has.Count.EqualTo(2)
27        );
28    }
29 }

```



```

1 // -----
2 // Syntax: Clear
3 // -----
4 // Mit der Clear Methode koennen alle Elemente
5 // aus der Liste geloescht werden.
6 public void Clear(){...};

7
8 public class ListUnitTest{
9     [Test]
10    public void TestClear(){
11        List<Point> points = new ();
12
13        points.Add(new Point(3,4));
14        points.Add(new Point(3,2));
15        points.Add(new Point(9,3));
16        points.Add(new Point(2,6));
17
18        points.Clear();
19
20        Assert.That(
21            points, Has.Count.EqualTo(0)
22        );
23    }
24 }

25 // -----
26 // Syntax: Contains
27 // -----
28 // Mit der Contains Methode wird geprueft
29 // ob ein bestimmtes Element in der Liste
30 // enthalten ist.
31 public bool Contains(T elem){...};

32 // Hinweis: Die Contains Methode prueft auf
33 // Wertegleichheit. Ueberschreiben Sie die
34 // Equals Methode der Elementklasse
35 public class ListUnitTest{
36     [Test]
37     public void TestContains(){
38         List<Point> points = new List<>();
39
40         points.Add(new Point(4,3));
41         points.Add(new Point(2,1));
42         points.Add(new Point(4,9));
43
44         Point p = new Point(2,1);
45
46         Assert.That(
47             points, Does.Contain(p)
48         );
49     }
50 }
51 }

52 // -----
53 // Syntax: Insert
54 // -----
55 // Mit der Insert Methode kann ein Element
56 // an einem bestimmten Index in die Liste
57 // eingefuegt werden. Elemente deren Index
58 // groesser bzw gleich dem angegebenem
59 // Index ist, werden um eine Stelle nach
60 // hinten verschoben
61 public void Insert(int index, T elem){...};

62 public class ListUnitTest{
63     [Test]
64     public void TestInsert(){
65         List<int> points = new List<>();
66
67         points.Add(56);
68         points.Add(3);
69         points.Add(26);
70         points.Add(2);
71
72         points.Insert(1, 21);
73
74         Assert.That(
75             points, Has.Count.EqualTo(5)
76         );
77         Assert.AreEqual(56, points[0]);
78         Assert.AreEqual(21, points[1]);
79         Assert.AreEqual(3, points[2]);
80     }
81 }

82 // -----
83 // Syntax: Remove
84 // -----
85 // Mit der der Remove Methode wird das erste
86 // Vorkommen des uebergebenen Parameters aus
87 // der Liste entfernt.
88 public bool Remove(T item){...};

89 // Hinweis: Die Remove Methode prueft auf
90 // Wertegleichheit.
91 public class ListUnitTest{
92     [Test]
93     public void TestRemove(){
94         List<Point> points = new List<>();
95
96         points.Add(new Point(3,4));
97         points.Add(new Point(5,2));
98         points.Add(new Point(7,2));
99         points.Add(new Point(2,2));
100    }
101 }

```

```
1  // -----
2  // Method: Remove
3  // -----
4      Assert.That(
5          point, Has.Count.EqualTo(4)
6      );
7
8      Point p = new Point(5,2);
9
10     Assert.True(points.Remove(p));
11     Assert.That(
12         point, Has.Count.EqualTo(3)
13     );
14 }
15 }
16
17 // -----
18 // Syntax: RemoveAt
19 // -----
20 // Unter Verwendung der RemoveAt Methode
21 // wird ein Element an einem bestimmten
22 // Index geloescht.
23 public void RemoveAt(int index){...}
24
25 public class ListUnitTest{
26     [Test]
27     public void TestRemoveAt(){
28         List<Point> points = new List<>();
29
30         points.Add(new Point(3,4));
31         points.Add(new Point(9,6));
32         points.Add(new Point(0,4));
33         points.Add(new Point(4,4));
34         points.Add(new Point(3,1));
35         points.Add(new Point(2,7));
36         points.Add(new Point(2,1));
37
38         Assert.That(
39             point, Has.Count.EqualTo(7)
40         );
41
42         points.RemoveAt(3);
43         Points.RemoveAt(1);
44
45         Assert.That(
46             point, Has.Count.EqualTo(5)
47         );
48     }
49 }
```



## 5.3. Datenstruktur: Stack



### Datenstruktur Stack ▾

Stacks werden zum Verwalten mehrerer Werte verwendet.

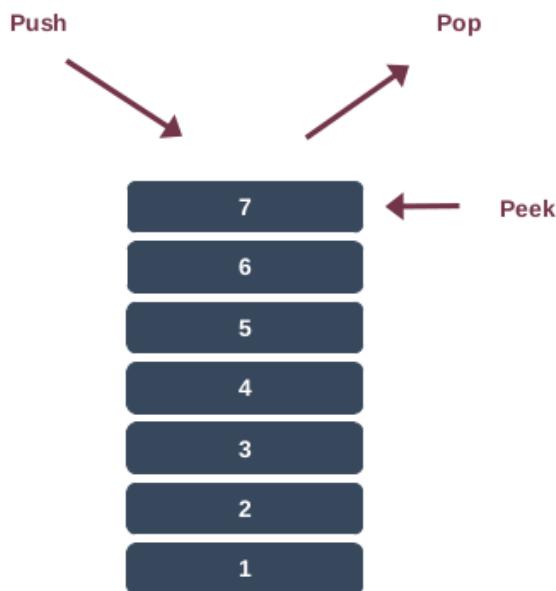
Werte werden gestapelt über bereits vorhandenen Werten eingefügt. Der zuerst eingefügte Wert steht an oberster Stelle.

### 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 ▾

```

1 // -----
2 // Syntax: Push
3 // -----
4 // Mit der Push Methode koennen Werte zum
5 // Stack hinzugefuegt werden. Der Wert wird
6 // dabei oben am Stack eingefuegt.
7 public void Push(T item){...};

8

9 public class StackUnitTests{
10     [Test]
11     public void TestPush(){
12         Stack<Point> pointStack = new ();
13
14         pointStack.Push(new Point(3,4));
15         pointStack.Push(new Point(8,3));
16
17         Assert.That(
18             pointStack, Has.Count.EqualTo(2)
19         );
20     }
21 }
22
23 // -----
24 // Syntax: Pop
25 // -----
26 // Mit der Pop Methode wird das oberste
27 // Element vom Stapel entfernt.
28 public T Pop(){...};

29
30 public class StackUnitTests{
31     [Test]
32     public void TestPop(){
33         Stack<Point> pointStack = new ();
34
35         pointStack.Push(new Point(4,7));
36         pointStack.Push(new Point(0,0));
37
38         Point p = pointStack.Pop();
39         Assert.That(
40             p, Is.EqualTo(new Point(0,0))
41         );
42         Assert.That(
43             pointStack, Has.Count.EqualTo(1)
44         )
45     }
46 }
```



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

Abbildung 3. Stackmethoden

```

1 // -----
2 // Syntax: Peek
3 // -----
4 // Die Peek Methode gibt eine Referenz auf
5 // das oberste Element im Stapel zurueck.
6 public T Peek(){...};

7
8 public class StackUnitTest{
9
10    [Test]
11    public void TestPeek(){
12        Stack<Point> pointStack = new ();
13
14        pointStack.Push(new Point(3,4));
15        pointStack.Push(new Point(3,1));
16        pointStack.Push(new Point(5,2));
17        pointStack.Push(new Point(2,8));
18        pointStack.Push(new Point(2,2));
19        pointStack.Push(new Point(2,1));
20
21        Assert.That(
22            pointStack, Has.Count.EqualTo(6)
23        );
24
25        Point p = pointStack.Peek();
26        Assert.That(
27            p, Is.EqualTo(new Point(2,1))
28        );
29    }
30 }

```

```

1 // -----
2 // Syntax: Clear
3 // -----
4 // Die Clear Methode werden die Werte des
5 // Stacks geloescht
6 public void Clear(){...};

7
8 public class StackUnitTest{
9
10    [Test]
11    public void TestClear(){
12        Stack<Point> pointStack = new ();
13
14        pointStack.Push(new Point(3,4));
15        pointStack.Push(new Point(7,1));
16        pointStack.Push(new Point(7,2));
17        pointStack.Push(new Point(1,6));
18        pointStack.Push(new Point(4,6));
19
20        Assert.That(
21            pointStack, Has.Count.EqualTo(5)
22        );
23
24        pointStack.Clear();
25        Assert.That(
26            pointStack, Has.Count.EqualTo(0)
27        );
28    }
29 }

```

```

1  // -----
2  // Syntax: Contains
3  // -----
4  // Mit der Contains Methode wird geprueft
5  // ob ein bestimmtes Element im Stack ent-
6  // halten ist.
7  public bool Contains(T item){...};

8
9  // Hinweis: Die Contains Methode prueft auf
10 // Wertegleichheit .Implementieren Sie die
11 // Equals Methode der Elementkalsse.

12
13 public class StackUnitTests{
14
15     [Test]
16     public void TestContains(){
17         Stack<Point> pointStack = ();
18
19         pointStack.Push(new Point(3,4));
20         pointStack.Push(new Point(2,1));
21         pointStack.Push(new Point(2,1));
22         pointStack.Push(new Point(2,1));
23         pointStack.Push(new Point(3,1));
24         pointStack.Push(new Point(3,1));
25         pointStack.Push(new Point(1,6));
26
27         Assert.That(
28             pointStack, Has.Count.EqualTo(7)
29         );
30
31         Point p = new Point(3,1);
32         Assert.True(
33             pointStack.Contains(p)
34         );
35     }
36
37 }
38
39 // -----
40 // Syntax: ToArray
41 // -----
42 // Die ToArray Methode kopiert die am Stack
43 // verwalteten Objektreferenzen in ein Array
44 public T[] ToArray(){...};

45
46 // Hinweis: Beachten Sie dass in Collections
47 // keine Objekte sondern Objektreferenzen
48 // verwaltet werden.

1  // -----
2  // Methode: ToArray
3  // -----
4  public class StackUnitTests{
5      [Test]
6      public void TestToArray(){
7          Stack<Point> stack = new ();
8
9          stack.Push(new Point(3,4));
10         stack.Push(new Point(2,1));
11         Assert.That(
12             stack, Has.Count.EqualTo(2)
13         );
14
15         Point[] points = stack.ToArray();
16         Assert.That(
17             stack, Has.Count.EqualTo(2)
18         );
19         Assert.That(
20             stack.Pop(), Is.Same(points[2])
21         );
22     }
23
24     // -----
25     // Syntax: CopyTo
26     // -----
27     // Kopiert die im Stack verwalteten Objekt-
28     // referenzen in ein Array. Welche Objekt-
29     // referenzen kopiert werden bestimmt der
30     // uebergebene Index
31     public void CopyTo(T[], int index){...};

32
33 public class StackUnitTests{
34     [Test]
35     public void TestCopyTo(){
36         Stack<Point> stack = new ();
37         stack.Push(new Point(3,4));
38
39         Point[] points = new
40             Point[stack.Count + 2];
41         points[0] = new Point(3,9);
42         points[1] = new Point(2,2);
43
44         stack.CopyTo(points,2);
45         Assert.That(
46             points, Has.Count.EqualTo(3)
47         );
48     }
49 }

```



Befehl	Beschreibung
<b>Enqueue</b>	Mit der Enqueue Methode wird ein neues Element in eine Queue eingefügt.
<b>Dequeue</b>	Mit der Dequeue Methode wird das als erste eingefügte Element aus einer Queue entfernt.
<b>Peek</b>	Die Peek Methode gibt eine Referenz auf das zuerst eingefügte Element einer Queue zurück.
<b>Clear</b>	Mit der Clear Methode wird eine Queue geleert.
<b>Contains</b>	Mit der Contains Methode wird geprüft ob ein bestimmtes Element in der Queue enthalten ist. Elemente werden dabei auf Wertegleichheit geprüft.
<b>ToArray</b>	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**<sup>14</sup> 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 Queueelemente 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 }
```

<sup>14</sup> First In - First Out

```

1 // -----
2 // Syntax: Dequeue
3 // -----
4 // Mit der Dequeue Methode wird aelteste
5 // Element einer Queue ausgetragen.
6 public T Dequeue(){...};

7
8 public class QueueUnitTests{
9     [Test]
10    public void TestDequeue(){
11        Queue<Point> queue = new ();
12        queue.Enqueue(new Point(7,6));
13
14        Assert.That(
15            queue, Has.Count.EqualTo(1)
16        )
17
18        Point p = queue.Dequeue();
19        Assert.That(
20            queue, Has.Count.EqualTo(0)
21        );
22        Assert.That(
23            p, Is.EqualTo(new Point(7, 6))
24        )
25    }
26 }

27 // -----
28 // Syntax: Peek
29 // -----
30 // Die Peek Methode gibt eine Referenz auf
31 // das zuerst eingefuegte Element einer
32 // Queue zurueck
33 public T Peek(){...};

34
35 public class QueueUnitTests{
36     [Test]
37     public void TestPeek(){
38         Queue<Point> queue = new ();
39
40         queue.Enqueue(new Point(3,2));
41         queue.Enqueue(new Point(6,1));
42
43         Assert.That(
44             queue, Has.Count.EqualTo(2)
45         );
46         Point p = queue.Peek();
47         Assert.That(
48             p, Is.EqualTo(new Point(3,2))
49         );
50     }
51 }
52 }
```

```

1 // -----
2 // Syntax: Contains
3 // -----
4 // Die Contains Methode prueft ob ein be-
5 // stimmtes Element in einer Queue enthalten
6 // ist. Die Elemente werden dabei auf Werte-
7 // gleichheit geprueft.
8 public bool Contains(T elem){...};

9
10 public class QueueUnitTests{
11     [Test]
12     public void TestContains(){
13         Queue<Point> queue = new Queue<>();
14
15         queue.Enqueue(new Point(3,2));
16         queue.Enqueue(new Point(6,1));
17
18         Assert.False(queue.Contains(new
19             Point(2,5)));
20     }
21
22 // -----
23 // Syntax: ToArray
24 // -----
25 // Die ToArray Methode kopiert die in einer
26 // Queue verwalteten Objektreferenzen in
27 // ein Array.
28 public T[] ToArray(){...};

29
30 public class QueueUnitTests{
31     [Test]
32     public void TestToArray(){
33         Queue<Point> queue = new Queue<>();
34         queue.Enqueue(new Point(3,2));
35         queue.Enqueue(new Point(6,1));
36
37         Assert.That(
38             queue.Count, Has.Count.EqualTo(2)
39         );
40
41         Point[] points = queue.ToArray();
42         Assert.That(
43             points.Length, Is.EqualTo(2)
44         );
45         Assert.That(
46             points[0], Is.Same(queue.Peek())
47         );
48     }
49 }
```





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

```

1 // -----
2 // Array vs. Dictionary
3 // -----
4 public class DictionaryUnitTest{
5     [Test]
6     public void CompareCollection(){
7         string[] phoneList1 = new string[100];
8
9         phoneList1[0] = "0664/8972372";
10        phoneList1[1] = "0650/3232664";
11
12        Dictionary<String, String> phoneList2
13            = new Dictionary<String, String>();
14
15        phoneList2["Haidvogl"] = "0664/89723";
16        phoneList2["Ferfeggy"] = "0650/32323";
17        phoneList2["Adler"] = "0650/56754";
18    }
19
20
21
22
23
24
25
26
27
28
29
30
31 }
```

### 5.5.2 Fallbeispiel: Dictionary

Das Verhalten eines Dictionaries wird durch folgende Methoden beschrieben.

#### Verhalten: Datenstruktur Dictionary

```

1 // -----
2 // Syntax: [] Operator
3 // -----
4 // Der Lesende und Schreibende Zugriff auf
5 // die Elemente eines Dictionaries erfolgt
6 // ueber den [] Operator
7
8 public class DictionaryUnitTest{
9     [Test]
10    public void TestReadWrite(){
11        Dictionary<string, string> pt = new
12            ();
13
14        phoneList["Haidvogl"] = "0664/89723";
15        phoneList["Ferfeggy"] = "0650/32323";
16        phoneList["Adler"] = "0650/98234";
17        phoneList["Schanndl"] = "0650/21323";
18
19        Assert.That(
20            pt, Has.Count.EqualTo(4)
21        );
22        Assert.That(
23            phoneList["Haidvogl"],
24            Is.EqualTo("0664/89723")
25        );
26
27        Assert.AreEqual(
28            phoneList["Ferfeggy"],
29            Is.EqualTo("0650/32323")
30        );
31    }
32 }
```



```

1 // -----
2 // Syntax: Clear
3 // -----
4 // Die Clear Methode loescht alle Eintraege
5 // aus einem Dictionary
6 public void Clear(){...}
7
8 public class DictionaryUnitTest{
9     [Test]
10    public void TestClear(){
11        Dictionary<string, int> grades = new
12            ();
13
14        grades["Softwareentwicklung"] = 2;
15        grades["Informationssysteme"] = 3;
16        grades["Medientechnik"] = 1;
17
18        Assert.That(
19            grades,
20            Has.Count.EqualTo(3)
21        );
22
23        grades.Clear();
24        Assert.That(
25            grades,
26            Has.Count.EqualTo(0)
27        );
28    }
29
30 // -----
31 // Syntax: ContainsKey
32 // -----
33 // Mit der ContainsKey Methode wird geprueft
34 // ob ein Dictionary einen bestimmten Wert
35 // als Index besitzt.
36 public bool ContainsKey(T key){...}
37
38 public class DictionaryUnitTest{
39     [Test]
40    public void TestContainsKey(){
41        Dictionary<string, int> grades = new
42            ();
43
44        grades["Dezentrale Systeme"] = 2;
45        grades["Medientechnik"] = 1;
46
47        Assert.True(
48            grades.ContainsKey("Medientechnik")
49        );
50    }
51
52 // -----
53 // Syntax: ContainsValue
54 // -----
55 // Mit der ContainsValue Methode wird ge-
56 // prueft ob ein Dictionary einen bestimmten
57 // Wert speichert.
58 public bool ContainsValue(V value){...}
59
60 public class DictionaryUnitTest{
61     [Test]
62    public void TestContainsValue(){
63        Dictionary<string, int> grades = new
64            ();
65        grades["Informationssysteme"] = 3;
66        grades["Medientechnik"] = 1;
67
68        Assert.True(
69            grades.ContainsKey(3);
70        }
71
72 // -----
73 // Syntax: Remove
74 // -----
75 // Mit der Hilfe der Remove Methode kann ein
76 // Schluessel eintrag aus einem Dictionary
77 // entfernt werden.
78 public bool Remove(T key){...}
79
80 public class DictionaryUnitTest{
81     [Test]
82    public void TestContainsKey(){
83        Dictionary<string, int> grades = new
84            ();
85        grades["Informationssysteme"] = 3;
86        grades["Medientechnik"] = 1;
87
88        Assert.That(
89            grades,
90            Has.Count.EqualTo(2)
91        );
92
93        grades.Remove("Medientechnik");
94        Assert.That(
95            grades,
96            Has.Count.EqualTo(1)
97        );
98    }
99
100 // -----
101 // Syntax: GetEnumerator
102 // -----
103 // Mit der GetEnumerator Methode kann ein
104 // Dictionary als Enumerator dargestellt
105 // werden.
106 public IEnumerator GetEnumerator(){...}
107
108 public class DictionaryUnitTest{
109     [Test]
110    public void TestGetEnumerator(){
111        Dictionary<string, int> grades = new
112            ();
113
114        grades["Informationssysteme"] = 3;
115        grades["Medientechnik"] = 1;
116
117        Assert.That(
118            grades,
119            Has.Count.EqualTo(2)
120        );
121
122        foreach (var item in grades)
123        {
124            Assert.AreEqual("Informationssysteme", item.Key);
125            Assert.AreEqual(3, item.Value);
126        }
127    }
128
129 // -----
130 // Syntax: TryGetValue
131 // -----
132 // Mit der TryGetValue Methode kann ein
133 // Wert aus einem Dictionary abgefragt
134 // werden.
135 public bool TryGetValue(T key, out V value){...}
136
137 public class DictionaryUnitTest{
138     [Test]
139    public void TestTryGetValue(){
140        Dictionary<string, int> grades = new
141            ();
142
143        grades["Informationssysteme"] = 3;
144        grades["Medientechnik"] = 1;
145
146        bool found;
147        int value;
148
149        found = grades.TryGetValue("Informationssysteme", out value);
150        Assert.True(found);
151        Assert.AreEqual(3, value);
152
153        found = grades.TryGetValue("Medientechnik", out value);
154        Assert.True(found);
155        Assert.AreEqual(1, value);
156
157        found = grades.TryGetValue("Softwareentwicklung", out value);
158        Assert.False(found);
159    }
160
161 // -----
162 // Syntax: TryGetValues
163 // -----
164 // Mit der TryGetValues Methode kann ein
165 // Wert aus einem Dictionary abgefragt
166 // werden.
167 public void TryGetValues(T key, out V[] values){...}
168
169 public class DictionaryUnitTest{
170     [Test]
171    public void TestTryGetValues(){
172        Dictionary<string, int> grades = new
173            ();
174
175        grades["Informationssysteme"] = 3;
176        grades["Medientechnik"] = 1;
177
178        int[] values;
179
180        grades.TryGetValues("Informationssysteme", out values);
181        Assert.AreEqual(1, values.Length);
182        Assert.AreEqual(3, values[0]);
183
184        grades.TryGetValues("Medientechnik", out values);
185        Assert.AreEqual(1, values.Length);
186        Assert.AreEqual(1, values[0]);
187
188        grades.TryGetValues("Softwareentwicklung", out values);
189        Assert.AreEqual(0, values.Length);
190    }
191
192 // -----
193 // Syntax: TryGetKey
194 // -----
195 // Mit der TryGetKey Methode kann ein
196 // Wert aus einem Dictionary abgefragt
197 // werden.
198 public T TryGetKey(V value){...}
199
200 public class DictionaryUnitTest{
201     [Test]
202    public void TestTryGetKey(){
203        Dictionary<string, int> grades = new
204            ();
205
206        grades["Informationssysteme"] = 3;
207        grades["Medientechnik"] = 1;
208
209        string key;
210
211        key = grades.TryGetKey(3);
212        Assert.AreEqual("Informationssysteme", key);
213
214        key = grades.TryGetKey(1);
215        Assert.AreEqual("Medientechnik", key);
216
217        key = grades.TryGetKey(5);
218        Assert.Null(key);
219    }
220
221 // -----
222 // Syntax: TryGetIndex
223 // -----
224 // Mit der TryGetIndex Methode kann ein
225 // Wert aus einem Dictionary abgefragt
226 // werden.
227 public int TryGetIndex(V value){...}
228
229 public class DictionaryUnitTest{
230     [Test]
231    public void TestTryGetIndex(){
232        Dictionary<string, int> grades = new
233            ();
234
235        grades["Informationssysteme"] = 3;
236        grades["Medientechnik"] = 1;
237
238        int index;
239
240        index = grades.TryGetIndex(3);
241        Assert.AreEqual(0, index);
242
243        index = grades.TryGetIndex(1);
244        Assert.AreEqual(1, index);
245
246        index = grades.TryGetIndex(5);
247        Assert.AreEqual(-1, index);
248    }
249
250 // -----
251 // Syntax: TryGetCount
252 // -----
253 // Mit der TryGetCount Methode kann ein
254 // Wert aus einem Dictionary abgefragt
255 // werden.
256 public int TryGetCount(V value){...}
257
258 public class DictionaryUnitTest{
259     [Test]
260    public void TestTryGetCount(){
261        Dictionary<string, int> grades = new
262            ();
263
264        grades["Informationssysteme"] = 3;
265        grades["Medientechnik"] = 1;
266
267        int count;
268
269        count = grades.TryGetCount(3);
270        Assert.AreEqual(1, count);
271
272        count = grades.TryGetCount(1);
273        Assert.AreEqual(1, count);
274
275        count = grades.TryGetCount(5);
276        Assert.AreEqual(0, count);
277    }
278
279 // -----
280 // Syntax: TryGetIndex
281 // -----
282 // Mit der TryGetIndex Methode kann ein
283 // Wert aus einem Dictionary abgefragt
284 // werden.
285 public int TryGetIndex(V value){...}
286
287 public class DictionaryUnitTest{
288     [Test]
289    public void TestTryGetIndex(){
290        Dictionary<string, int> grades = new
291            ();
292
293        grades["Informationssysteme"] = 3;
294        grades["Medientechnik"] = 1;
295
296        int index;
297
298        index = grades.TryGetIndex(3);
299        Assert.AreEqual(0, index);
300
301        index = grades.TryGetIndex(1);
302        Assert.AreEqual(1, index);
303
304        index = grades.TryGetIndex(5);
305        Assert.AreEqual(-1, index);
306    }
307
308 // -----
309 // Syntax: TryGetCount
310 // -----
311 // Mit der TryGetCount Methode kann ein
312 // Wert aus einem Dictionary abgefragt
313 // werden.
314 public int TryGetCount(V value){...}
315
316 public class DictionaryUnitTest{
317     [Test]
318    public void TestTryGetCount(){
319        Dictionary<string, int> grades = new
320            ();
321
322        grades["Informationssysteme"] = 3;
323        grades["Medientechnik"] = 1;
324
325        int count;
326
327        count = grades.TryGetCount(3);
328        Assert.AreEqual(1, count);
329
330        count = grades.TryGetCount(1);
331        Assert.AreEqual(1, count);
332
333        count = grades.TryGetCount(5);
334        Assert.AreEqual(0, count);
335    }
336
337 // -----
338 // Syntax: TryGetKey
339 // -----
340 // Mit der TryGetKey Methode kann ein
341 // Wert aus einem Dictionary abgefragt
342 // werden.
343 public T TryGetKey(V value){...}
344
345 public class DictionaryUnitTest{
346     [Test]
347    public void TestTryGetKey(){
348        Dictionary<string, int> grades = new
349            ();
350
351        grades["Informationssysteme"] = 3;
352        grades["Medientechnik"] = 1;
353
354        string key;
355
356        key = grades.TryGetKey(3);
357        Assert.AreEqual("Informationssysteme", key);
358
359        key = grades.TryGetKey(1);
360        Assert.AreEqual("Medientechnik", key);
361
362        key = grades.TryGetKey(5);
363        Assert.Null(key);
364    }
365
366 // -----
367 // Syntax: TryGetIndex
368 // -----
369 // Mit der TryGetIndex Methode kann ein
370 // Wert aus einem Dictionary abgefragt
371 // werden.
372 public int TryGetIndex(V value){...}
373
374 public class DictionaryUnitTest{
375     [Test]
376    public void TestTryGetIndex(){
377        Dictionary<string, int> grades = new
378            ();
379
380        grades["Informationssysteme"] = 3;
381        grades["Medientechnik"] = 1;
382
383        int index;
384
385        index = grades.TryGetIndex(3);
386        Assert.AreEqual(0, index);
387
388        index = grades.TryGetIndex(1);
389        Assert.AreEqual(1, index);
390
391        index = grades.TryGetIndex(5);
392        Assert.AreEqual(-1, index);
393    }
394
395 // -----
396 // Syntax: TryGetCount
397 // -----
398 // Mit der TryGetCount Methode kann ein
399 // Wert aus einem Dictionary abgefragt
400 // werden.
401 public int TryGetCount(V value){...}
402
403 public class DictionaryUnitTest{
404     [Test]
405    public void TestTryGetCount(){
406        Dictionary<string, int> grades = new
407            ();
408
409        grades["Informationssysteme"] = 3;
410        grades["Medientechnik"] = 1;
411
412        int count;
413
414        count = grades.TryGetCount(3);
415        Assert.AreEqual(1, count);
416
417        count = grades.TryGetCount(1);
418        Assert.AreEqual(1, count);
419
420        count = grades.TryGetCount(5);
421        Assert.AreEqual(0, count);
422    }
423
424 // -----
425 // Syntax: TryGetKey
426 // -----
427 // Mit der TryGetKey Methode kann ein
428 // Wert aus einem Dictionary abgefragt
429 // werden.
430 public T TryGetKey(V value){...}
431
432 public class DictionaryUnitTest{
433     [Test]
434    public void TestTryGetKey(){
435        Dictionary<string, int> grades = new
436            ();
437
438        grades["Informationssysteme"] = 3;
439        grades["Medientechnik"] = 1;
440
441        string key;
442
443        key = grades.TryGetKey(3);
444        Assert.AreEqual("Informationssysteme", key);
445
446        key = grades.TryGetKey(1);
447        Assert.AreEqual("Medientechnik", key);
448
449        key = grades.TryGetKey(5);
450        Assert.Null(key);
451    }
452
453 // -----
454 // Syntax: TryGetIndex
455 // -----
456 // Mit der TryGetIndex Methode kann ein
457 // Wert aus einem Dictionary abgefragt
458 // werden.
459 public int TryGetIndex(V value){...}
460
461 public class DictionaryUnitTest{
462     [Test]
463    public void TestTryGetIndex(){
464        Dictionary<string, int> grades = new
465            ();
466
467        grades["Informationssysteme"] = 3;
468        grades["Medientechnik"] = 1;
469
470        int index;
471
472        index = grades.TryGetIndex(3);
473        Assert.AreEqual(0, index);
474
475        index = grades.TryGetIndex(1);
476        Assert.AreEqual(1, index);
477
478        index = grades.TryGetIndex(5);
479        Assert.AreEqual(-1, index);
480    }
481
482 // -----
483 // Syntax: TryGetCount
484 // -----
485 // Mit der TryGetCount Methode kann ein
486 // Wert aus einem Dictionary abgefragt
487 // werden.
488 public int TryGetCount(V value){...}
489
490 public class DictionaryUnitTest{
491     [Test]
492    public void TestTryGetCount(){
493        Dictionary<string, int> grades = new
494            ();
495
496        grades["Informationssysteme"] = 3;
497        grades["Medientechnik"] = 1;
498
499        int count;
500
501        count = grades.TryGetCount(3);
502        Assert.AreEqual(1, count);
503
504        count = grades.TryGetCount(1);
505        Assert.AreEqual(1, count);
506
507        count = grades.TryGetCount(5);
508        Assert.AreEqual(0, count);
509    }
510
511 // -----
512 // Syntax: TryGetKey
513 // -----
514 // Mit der TryGetKey Methode kann ein
515 // Wert aus einem Dictionary abgefragt
516 // werden.
517 public T TryGetKey(V value){...}
518
519 public class DictionaryUnitTest{
520     [Test]
521    public void TestTryGetKey(){
522        Dictionary<string, int> grades = new
523            ();
524
525        grades["Informationssysteme"] = 3;
526        grades["Medientechnik"] = 1;
527
528        string key;
529
530        key = grades.TryGetKey(3);
531        Assert.AreEqual("Informationssysteme", key);
532
533        key = grades.TryGetKey(1);
534        Assert.AreEqual("Medientechnik", key);
535
536        key = grades.TryGetKey(5);
537        Assert.Null(key);
538    }
539
540 // -----
541 // Syntax: TryGetIndex
542 // -----
543 // Mit der TryGetIndex Methode kann ein
544 // Wert aus einem Dictionary abgefragt
545 // werden.
546 public int TryGetIndex(V value){...}
547
548 public class DictionaryUnitTest{
549     [Test]
550    public void TestTryGetIndex(){
551        Dictionary<string, int> grades = new
552            ();
553
554        grades["Informationssysteme"] = 3;
555        grades["Medientechnik"] = 1;
556
557        int index;
558
559        index = grades.TryGetIndex(3);
560        Assert.AreEqual(0, index);
561
562        index = grades.TryGetIndex(1);
563        Assert.AreEqual(1, index);
564
565        index = grades.TryGetIndex(5);
566        Assert.AreEqual(-1, index);
567    }
568
569 // -----
570 // Syntax: TryGetCount
571 // -----
572 // Mit der TryGetCount Methode kann ein
573 // Wert aus einem Dictionary abgefragt
574 // werden.
575 public int TryGetCount(V value){...}
576
577 public class DictionaryUnitTest{
578     [Test]
579    public void TestTryGetCount(){
580        Dictionary<string, int> grades = new
581            ();
582
583        grades["Informationssysteme"] = 3;
584        grades["Medientechnik"] = 1;
585
586        int count;
587
588        count = grades.TryGetCount(3);
589        Assert.AreEqual(1, count);
590
591        count = grades.TryGetCount(1);
592        Assert.AreEqual(1, count);
593
594        count = grades.TryGetCount(5);
595        Assert.AreEqual(0, count);
596    }
597
598 // -----
599 // Syntax: TryGetKey
600 // -----
601 // Mit der TryGetKey Methode kann ein
602 // Wert aus einem Dictionary abgefragt
603 // werden.
604 public T TryGetKey(V value){...}
605
606 public class DictionaryUnitTest{
607     [Test]
608    public void TestTryGetKey(){
609        Dictionary<string, int> grades = new
610            ();
611
612        grades["Informationssysteme"] = 3;
613        grades["Medientechnik"] = 1;
614
615        string key;
616
617        key = grades.TryGetKey(3);
618        Assert.AreEqual("Informationssysteme", key);
619
620        key = grades.TryGetKey(1);
621        Assert.AreEqual("Medientechnik", key);
622
623        key = grades.TryGetKey(5);
624        Assert.Null(key);
625    }
626
627 // -----
628 // Syntax: TryGetIndex
629 // -----
630 // Mit der TryGetIndex Methode kann ein
631 // Wert aus einem Dictionary abgefragt
632 // werden.
633 public int TryGetIndex(V value){...}
634
635 public class DictionaryUnitTest{
636     [Test]
637    public void TestTryGetIndex(){
638        Dictionary<string, int> grades = new
639            ();
640
641        grades["Informationssysteme"] = 3;
642        grades["Medientechnik"] = 1;
643
644        int index;
645
646        index = grades.TryGetIndex(3);
647        Assert.AreEqual(0, index);
648
649        index = grades.TryGetIndex(1);
650        Assert.AreEqual(1, index);
651
652        index = grades.TryGetIndex(5);
653        Assert.AreEqual(-1, index);
654    }
655
656 // -----
657 // Syntax: TryGetCount
658 // -----
659 // Mit der TryGetCount Methode kann ein
660 // Wert aus einem Dictionary abgefragt
661 // werden.
662 public int TryGetCount(V value){...}
663
664 public class DictionaryUnitTest{
665     [Test]
666    public void TestTryGetCount(){
667        Dictionary<string, int> grades = new
668            ();
669
670        grades["Informationssysteme"] = 3;
671        grades["Medientechnik"] = 1;
672
673        int count;
674
675        count = grades.TryGetCount(3);
676        Assert.AreEqual(1, count);
677
678        count = grades.TryGetCount(1);
679        Assert.AreEqual(1, count);
680
681        count = grades.TryGetCount(5);
682        Assert.AreEqual(0, count);
683    }
684
685 // -----
686 // Syntax: TryGetKey
687 // -----
688 // Mit der TryGetKey Methode kann ein
689 // Wert aus einem Dictionary abgefragt
690 // werden.
691 public T TryGetKey(V value){...}
692
693 public class DictionaryUnitTest{
694     [Test]
695    public void TestTryGetKey(){
696        Dictionary<string, int> grades = new
697            ();
698
699        grades["Informationssysteme"] = 3;
700        grades["Medientechnik"] = 1;
701
702        string key;
703
704        key = grades.TryGetKey(3);
705        Assert.AreEqual("Informationssysteme", key);
706
707        key = grades.TryGetKey(1);
708        Assert.AreEqual("Medientechnik", key);
709
710        key = grades.TryGetKey(5);
711        Assert.Null(key);
712    }
713
714 // -----
715 // Syntax: TryGetIndex
716 // -----
717 // Mit der TryGetIndex Methode kann ein
718 // Wert aus einem Dictionary abgefragt
719 // werden.
720 public int TryGetIndex(V value){...}
721
722 public class DictionaryUnitTest{
723     [Test]
724    public void TestTryGetIndex(){
725        Dictionary<string, int> grades = new
726            ();
727
728        grades["Informationssysteme"] = 3;
729        grades["Medientechnik"] = 1;
730
731        int index;
732
733        index = grades.TryGetIndex(3);
734        Assert.AreEqual(0, index);
735
736        index = grades.TryGetIndex(1);
737        Assert.AreEqual(1, index);
738
739        index = grades.TryGetIndex(5);
740        Assert.AreEqual(-1, index);
741    }
742
743 // -----
744 // Syntax: TryGetCount
745 // -----
746 // Mit der TryGetCount Methode kann ein
747 // Wert aus einem Dictionary abgefragt
748 // werden.
749 public int TryGetCount(V value){...}
750
751 public class DictionaryUnitTest{
752     [Test]
753    public void TestTryGetCount(){
754        Dictionary<string, int> grades = new
755            ();
756
757        grades["Informationssysteme"] = 3;
758        grades["Medientechnik"] = 1;
759
760        int count;
761
762        count = grades.TryGetCount(3);
763        Assert.AreEqual(1, count);
764
765        count = grades.TryGetCount(1);
766        Assert.AreEqual(1, count);
767
768        count = grades.TryGetCount(5);
769        Assert.AreEqual(0, count);
770    }
771
772 // -----
773 // Syntax: TryGetKey
774 // -----
775 // Mit der TryGetKey Methode kann ein
776 // Wert aus einem Dictionary abgefragt
777 // werden.
778 public T TryGetKey(V value){...}
779
780 public class DictionaryUnitTest{
781     [Test]
782    public void TestTryGetKey(){
783        Dictionary<string, int> grades = new
784            ();
785
786        grades["Informationssysteme"] = 3;
787        grades["Medientechnik"] = 1;
788
789        string key;
790
791        key = grades.TryGetKey(3);
792        Assert.AreEqual("Informationssysteme", key);
793
794        key = grades.TryGetKey(1);
795        Assert.AreEqual("Medientechnik", key);
796
797        key = grades.TryGetKey(5);
798        Assert.Null(key);
799    }
800
801 // -----
802 // Syntax: TryGetIndex
803 // -----
804 // Mit der TryGetIndex Methode kann ein
805 // Wert aus einem Dictionary abgefragt
806 // werden.
807 public int TryGetIndex(V value){...}
808
809 public class DictionaryUnitTest{
810     [Test]
811    public void TestTryGetIndex(){
812        Dictionary<string, int> grades = new
813            ();
814
815        grades["Informationssysteme"] = 3;
816        grades["Medientechnik"] = 1;
817
818        int index;
819
820        index = grades.TryGetIndex(3);
821        Assert.AreEqual(0, index);
822
823        index = grades.TryGetIndex(1);
824        Assert.AreEqual(1, index);
825
826        index = grades.TryGetIndex(5);
827        Assert.AreEqual(-1, index);
828    }
829
830 // -----
831 // Syntax: TryGetCount
832 // -----
833 // Mit der TryGetCount Methode kann ein
834 // Wert aus einem Dictionary abgefragt
835 // werden.
836 public int TryGetCount(V value){...}
837
838 public class DictionaryUnitTest{
839     [Test]
840    public void TestTryGetCount(){
841        Dictionary<string, int> grades = new
842            ();
843
844        grades["Informationssysteme"] = 3;
845        grades["Medientechnik"] = 1;
846
847        int count;
848
849        count = grades.TryGetCount(3);
850        Assert.AreEqual(1, count);
851
852        count = grades.TryGetCount(1);
853        Assert.AreEqual(1, count);
854
855        count = grades.TryGetCount(5);
856        Assert.AreEqual(0, count);
857    }
858
859 // -----
860 // Syntax: TryGetKey
861 // -----
862 // Mit der TryGetKey Methode kann ein
863 // Wert aus einem Dictionary abgefragt
864 // werden.
865 public T TryGetKey(V value){...}
866
867 public class DictionaryUnitTest{
868     [Test]
869    public void TestTryGetKey(){
870        Dictionary<string, int> grades = new
871            ();
872
873        grades["Informationssysteme"] = 3;
874        grades["Medientechnik"] = 1;
875
876        string key;
877
878        key = grades.TryGetKey(3);
879        Assert.AreEqual("Informationssysteme", key);
880
881        key = grades.TryGetKey(1);
882        Assert.AreEqual("Medientechnik", key);
883
884        key = grades.TryGetKey(5);
885        Assert.Null(key);
886    }
887
888 // -----
889 // Syntax: TryGetIndex
890 // -----
891 // Mit der TryGetIndex Methode kann ein
892 // Wert aus einem Dictionary abgefragt
893 // werden.
894 public int TryGetIndex(V value){...}
895
896 public class DictionaryUnitTest{
897     [Test]
898    public void TestTryGetIndex(){
899        Dictionary<string, int> grades = new
900            ();
901
902        grades["Informationssysteme"] = 3;
903        grades["Medientechnik"] = 1;
904
905        int index;
906
907        index = grades.TryGetIndex(3);
908        Assert.AreEqual(0, index);
909
910        index = grades.TryGetIndex(1);
911        Assert.AreEqual(1, index);
912
913        index = grades.TryGetIndex(5);
914        Assert.AreEqual(-1, index);
915    }
916
917 // -----
918 // Syntax: TryGetCount
919 // -----
920 // Mit der TryGetCount Methode kann ein
921 // Wert aus einem Dictionary abgefragt
922 // werden.
923 public int TryGetCount(V value){...}
924
925 public class DictionaryUnitTest{
926     [Test]
927    public void TestTryGetCount(){
928        Dictionary<string, int> grades = new
929            ();
930
931        grades["Informationssysteme"] = 3;
932        grades["Medientechnik"] = 1;
933
934        int count;
935
936        count = grades.TryGetCount(3);
937        Assert.AreEqual(1, count);
938
939        count = grades.TryGetCount(1);
940        Assert.AreEqual(1, count);
941
942        count = grades.TryGetCount(5);
943        Assert.AreEqual(0, count);
944    }
945
946 // -----
947 // Syntax: TryGetKey
948 // -----
949 // Mit der TryGetKey Methode kann ein
950 // Wert aus einem Dictionary abgefragt
951 // werden.
952 public T TryGetKey(V value){...}
953
954 public class DictionaryUnitTest{
955     [Test]
956    public void TestTryGetKey(){
957        Dictionary<string, int> grades = new
958            ();
959
960        grades["Informationssysteme"] = 3;
961        grades["Medientechnik"] = 1;
962
963        string key;
964
965        key = grades.TryGetKey(3);
966        Assert.AreEqual("Informationssysteme", key);
967
968        key = grades.TryGetKey(1);
969        Assert.AreEqual("Medientechnik", key);
970
971        key = grades.TryGetKey(5);
972        Assert.Null(key);
973    }
974
975 // -----
976 // Syntax: TryGetIndex
977 // -----
978 // Mit der TryGetIndex Methode kann ein
979 // Wert aus einem Dictionary abgefragt
980 // werden.
981 public int TryGetIndex(V value){...}
982
983 public class DictionaryUnitTest{
984     [Test]
985    public void TestTryGetIndex(){
986        Dictionary<string, int> grades = new
987            ();
988
989        grades["Informationssysteme"] = 3;
990        grades["Medientechnik"] = 1;
991
992        int index;
993
994        index = grades.TryGetIndex(3);
995        Assert.AreEqual(0, index);
996
997        index = grades.TryGetIndex(1);
998        Assert.AreEqual(1, index);
999
1000        index = grades.TryGetIndex(5);
1001        Assert.AreEqual(-1, index);
1002    }
1003
1004 // -----
1005 // Syntax: TryGetCount
1006 // -----
1007 // Mit der TryGetCount Methode kann ein
1008 // Wert aus einem Dictionary abgefragt
1009 // werden.
1010 public int TryGetCount(V value){...}
1011
1012 public class DictionaryUnitTest{
1013     [Test]
1014    public void TestTryGetCount(){
1015        Dictionary<string, int> grades = new
1016            ();
1017
1018        grades["Informationssysteme"] = 3;
1019        grades["Medientechnik"] = 1;
1020
1021        int count;
1022
1023        count = grades.TryGetCount(3);
1024        Assert.AreEqual(1, count);
1025
1026        count = grades.TryGetCount(1);
1027        Assert.AreEqual(1, count);
1028
1029        count = grades.TryGetCount(5);
1030        Assert.AreEqual(0, count);
1031    }
1032
1033 // -----
1034 // Syntax: TryGetKey
1035 // -----
1036 // Mit der TryGetKey Methode kann ein
1037 // Wert aus einem Dictionary abgefragt
1038 // werden.
1039 public T TryGetKey(V value){...}
1040
1041 public class DictionaryUnitTest{
1042     [Test]
1043    public void TestTryGetKey(){
1044        Dictionary<string, int> grades = new
1045            ();
1046
1047        grades["Informationssysteme"] = 3;
1048        grades["Medientechnik"] = 1;
1049
1050        string key;
1051
1052        key = grades.TryGetKey(3);
1053        Assert.AreEqual("Informationssysteme", key);
1054
1055        key = grades.TryGetKey(1);
1056        Assert.AreEqual("Medientechnik", key);
1057
1058        key = grades.TryGetKey(5);
1059        Assert.Null(key);
1060    }
1061
1062 // -----
1063 // Syntax: TryGetIndex
1064 // -----
1065 // Mit der TryGetIndex Methode kann ein
1066 // Wert aus einem Dictionary abgefragt
1067 // werden.
1068 public int TryGetIndex(V value){...}
1069
1070 public class DictionaryUnitTest{
1071     [Test]
1072    public void TestTryGetIndex(){
1073        Dictionary<string, int> grades = new
1074            ();
1075
1076        grades["Informationssysteme"] = 3;
1077        grades["Medientechnik"] = 1;
1078
1079        int index;
1080
1081        index = grades.TryGetIndex(3);
1082        Assert.AreEqual(0, index);
1083
1084        index = grades.TryGetIndex(1);
1085        Assert.AreEqual(1, index);
1086
1087        index = grades.TryGetIndex(5);
1088        Assert.AreEqual(-1, index);
1089    }
1090
1091 // -----
1092 // Syntax: TryGetCount
1093 // -----
1094 // Mit der TryGetCount Methode kann ein
1095 // Wert aus einem Dictionary abgefragt
1096 // werden.
1097 public int TryGetCount(V value){...}
1098
1099 public class DictionaryUnitTest{
1100     [Test]
1101    public void TestTryGetCount(){
1102        Dictionary<string, int> grades = new
1103            ();
1104
1105        grades["Informationssysteme"] = 3;
1106        grades["Medientechnik"] = 1;
1107
1108        int count;
1109
1110        count = grades.TryGetCount(3);
1111        Assert.AreEqual(1, count);
1112
1113        count = grades.TryGetCount(1);
1114        Assert.AreEqual(1, count);
1115
1116        count = grades.TryGetCount(5);
1117        Assert.AreEqual(0, count);
1118    }
1119
1120 // -----
1121 // Syntax: TryGetKey
1122 // -----
1123 // Mit der TryGetKey Methode kann ein
1124 // Wert aus einem Dictionary abgefragt
1125 // werden.
1126 public T TryGetKey(V value){...}
1127
1128 public class DictionaryUnitTest{
1129     [Test]
1130    public void TestTryGetKey(){
1131        Dictionary<string, int> grades = new
1132            ();
1133
1134        grades["Informationssysteme"] = 3;
1135        grades["Medientechnik"] = 1;
1136
1137        string key;
1138
1139        key = grades.TryGetKey(3);
1140        Assert.AreEqual("Informationssysteme", key);
1141
1142        key = grades.TryGetKey(1);
1143        Assert.AreEqual("Medientechnik", key);
1144
1145        key = grades.TryGetKey(5);
1146        Assert.Null(key);
1147    }
1148
1149 // -----
1150 // Syntax: TryGetIndex
1151 // -----
1152 // Mit der TryGetIndex Methode kann ein
1153 // Wert aus einem Dictionary abgefragt
1154 // werden.
1155 public int TryGetIndex(V value){...}
1156
1157 public class DictionaryUnitTest{
1158     [Test]
1159    public void TestTryGetIndex(){
1160        Dictionary<string, int> grades = new
1161            ();
1162
1163        grades["Informationssysteme"] = 3;
1164        grades["Medientechnik"] = 1;
1165
1166        int index;
1167
1168        index = grades.TryGetIndex(3);
1169        Assert.AreEqual(0, index);
1170
1171        index = grades.TryGetIndex(1);
1172        Assert.AreEqual(1, index);
1173
1174        index = grades.TryGetIndex(5);
1175        Assert.AreEqual(-1, index);
1176    }
1177
1178 // -----
1179 // Syntax: TryGetCount
1180 // -----
1181 // Mit der TryGetCount Methode kann ein
1182 // Wert aus einem Dictionary abgefragt
1183 // werden.
1184 public int TryGetCount(V value){...}
1185
1186 public class DictionaryUnitTest{
1187     [Test]
1188    public void TestTryGetCount(){
1189        Dictionary<string, int> grades = new
1190            ();
1191
1192        grades["Informationssysteme"] = 3;
1193        grades["Medientechnik"] = 1;
1194
1195        int count;
1196
1197        count = grades.TryGetCount(3);
1198        Assert.AreEqual(1, count);
1199
1200        count = grades.TryGetCount(1);
1201        Assert.AreEqual(1, count);
1202
1203        count = grades.TryGetCount(5);
1204        Assert.AreEqual(0, count);
1205    }
1206
1207 // -----
1208 // Syntax: TryGetKey
1209 // -----
1210 // Mit der TryGetKey Methode kann ein
1211 // Wert aus einem Dictionary abgefragt
1212 // werden.
1213 public T TryGetKey(V value){...}
1214
1215 public class DictionaryUnitTest{
1216     [Test]
1217    public void TestTryGetKey(){
1218        Dictionary<string, int> grades = new
1219            ();
1220
1221        grades["Informationssysteme"] = 3;
1222        grades["Medientechnik"] = 1;
1223
1224        string key;
1225
1226        key = grades.TryGetKey(3);
1227        Assert.AreEqual("Informationssysteme", key);
1228
1229        key = grades.TryGetKey(1);
1230        Assert.AreEqual("Medientechnik", key);
1231
1232        key = grades.TryGetKey(5);
1233        Assert.Null(key);
1234    }
1235
1236 // -----
1237 // Syntax: TryGetIndex
1238 // -----
1239 // Mit der TryGetIndex Methode kann ein
1240 // Wert aus einem Dictionary abgefragt
1241 // werden.
1242 public int TryGetIndex(V value){...}
1243
1244 public class DictionaryUnitTest{
1245     [Test]
1246    public void TestTryGetIndex(){
1247        Dictionary<string, int> grades = new
1248            ();
1249
1250        grades["Informationssysteme"] = 3;
1251        grades["Medientechnik"] = 1;
1252
1253        int index;
1254
1255        index = grades.TryGetIndex(3);
1256        Assert.AreEqual(0, index);
1257
1258        index = grades.TryGetIndex(1);
1259        Assert.AreEqual(1, index);
1260
1261        index = grades.TryGetIndex(5);
1262        Assert.AreEqual(-1, index);
1263    }
1264
1265 // -----
1266 // Syntax: TryGetCount
1267 // -----
1268 // Mit der TryGetCount Methode kann ein
1269 // Wert aus einem Dictionary abgefragt
1270 // werden.
1271 public int TryGetCount(V value){...}
1272
1273 public class DictionaryUnitTest{
1274     [Test]
1275    public void TestTryGetCount(){
1276        Dictionary<string, int> grades = new
1277            ();
1278
1279        grades["Informationssysteme"] = 3;
1280        grades["Medientechnik"] = 1;
1281
1282        int count;
1283
1284        count = grades.TryGetCount(3);
1285        Assert.AreEqual(1, count);
1286
1287        count = grades.TryGetCount(1);
1288        Assert.AreEqual(1, count);
1289
1290        count = grades.TryGetCount(5);
1291        Assert.AreEqual(0, count);
1292    }
1293
1294 // -----
1295 // Syntax: TryGetKey
1296 // -----
1297 // Mit der TryGetKey Methode kann ein
1298 // Wert aus einem Dictionary abgefragt
1299 // werden.
1300 public T TryGetKey(V value){...}
1301
1302 public class DictionaryUnitTest{
1303     [Test]
1304    public void TestTryGetKey(){
1305        Dictionary<string, int> grades = new
1306            ();
1307
1308        grades["Informationssysteme"] = 3;
1309        grades["Medientechnik"] = 1;
1310
1311        string key;
1312
1313        key = grades.TryGetKey(3);
1314        Assert.AreEqual("Informationssysteme", key);
1315
1316        key = grades.TryGetKey(1);
1317        Assert.AreEqual("Medientechnik", key);
1318
1319        key = grades.TryGetKey(5);
1320        Assert.Null(key);
1321    }
1322
1323 // -----
1324 // Syntax: TryGetIndex
1325 // -----
1326 // Mit der TryGetIndex Methode kann ein
1327 // Wert aus einem Dictionary abgefragt
1328 // werden.
1329 public int TryGetIndex(V value){...}
1330
1331 public class DictionaryUnitTest{
1332     [Test]
1333    public void TestTryGetIndex(){
1334        Dictionary<string, int> grades = new
1335            ();
1336
1337        grades["Informationssysteme"] = 3;
1338        grades["Medientechnik"] = 1;
1339
1340        int index;
1341
1342        index = grades.TryGetIndex(3);
1343        Assert.AreEqual(0, index);
1344
1345        index = grades.TryGetIndex(1);
1346        Assert.AreEqual(1, index);
1347
1348        index = grades.TryGetIndex(5);
1349        Assert.AreEqual(-1, index);
1350    }
1351
1352 // -----
1353 // Syntax: TryGetCount
1354 // -----
1355 // Mit der TryGetCount Methode kann ein
1356 // Wert aus einem Dictionary abgefragt
1357 // werden.
1358 public int TryGetCount(V value){...}
1359
1360 public class DictionaryUnitTest{
1361     [Test]
1362    public void TestTryGetCount(){
1363        Dictionary<string, int> grades = new
1364            ();
1365
1366        grades["Informationssysteme"] = 3;
1367        grades["Medientechnik"] = 1;
1368
1369        int count;
1370
1371        count = grades.TryGetCount(3);
1372        Assert.AreEqual(1, count);
1373
1374        count = grades.TryGetCount(1);

```

### 5.5.3 Wertezugriff

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

#### ► Codebeispiel: Wertezugriff ▾

```
1  // -----
2  //  Dictionary Properties
3  // -----
4  public class DictionaryUnitTest{
5      [Test]
6      public void CompareCollection(){
7          Dictionary<String, String> phoneList
8          = new ();
9
10         phoneList["Haidvogl"] = "0664/89723";
11         phoneList["Ferfeggy"] = "0650/32323";
12         phoneList["Adler"] = "0650/56754";
13
14         foreach(var person in phoneList.Keys){
15             Console.WriteLine($"{person} ");
16         }
17
18         > Ausgabe
19         Haidvogl, Ferfeggy, Adler
20
21         foreach(var num in phoneList.Values){
22             Console.WriteLine($"{num}");
23         }
24
25         > Ausgabe
26         0664/89723 0650/32323 0650/56754
27     }
28 }
```





# Grundlagen der objektorientierten Programmierung

December 14, 2019

## 6. Programmierung: Strukturierung

# 01

### Strukturierung von Programmen

01. Unterprogramme	42
02. Objektorientierung	43
03. Schichtenmodell	43
04. Komponenten	46
05. Service	47

### 6.1. Unterprogramme

Historisch gesehen hat alles mit einem bunten Gemisch aus **Anweisungen** und **Daten** innerhalb eines Betriebssystemprozesses<sup>15</sup> 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**<sup>16</sup> 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**<sup>17</sup> der Anwendung gesunken wäre.
- Als nächstes wurden **Container** für Daten<sup>18</sup> 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 }
```



<sup>15</sup> Unter einem Betriebssystemprozess verstehen wir ein sich in Ausführung befindendes Programm

<sup>16</sup> Funktionen, Prozeduren

<sup>17</sup> Codeerwartbarkeit, Codelesbarkeit, Anpassbarkeit

<sup>18</sup> 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**<sup>19</sup> 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**<sup>20</sup> zur Verfügung, die den **Zustand**<sup>21</sup> von Instanzen der Klasse verändert und verarbeitet.
- Variablen und Methoden stehen im kontinuierlichen Zusammenspiel.

#### ► Codebeispiel: Klassen

```

1  // -----
2  //  Project.cs
3  // -----
4  [Table("PROJECTS")]
5  public class Project : AProject {
6
7      [Key, DatabaseGenerated]
8      [Column("PROJECT_ID")]
9      public int Id { get; set; }
10
11     [Required, StringLength(50)]
12     [Column("TITLE")]
13     public string Title { get; set; }
14
15     public Project() : base () {
16
17     }
18 }
```



<sup>19</sup> 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.

<sup>20</sup> Methoden

<sup>21</sup> Variablen

## 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

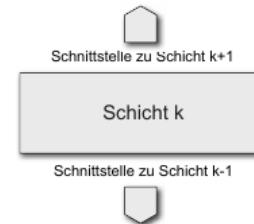
- **Teile und Herrsche:** Ein komplexes Problem wird in **unabhängige Teilprobleme** zerlegt, das jedes für sich, einfacher handhabbar ist, als das Gesamtproblem.

Oft ist es erst durch die Formulierung von Teilproblemen möglich, ein komplexes Probleme zu lösen.

- **Unabhängigkeit:** Die einzelnen Schichten der Anwendung kommunizieren miteinander, indem die **Schnittstellenspezifikation**<sup>22</sup> 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.



- **Standardsierung:** Die Gliederung des Gesamtproblems in einzelne Schichten erleichtert die Entwicklung von **Standards**<sup>23</sup> für die einzelnen Schichten.



<sup>22</sup> Schnittstelle, Interface

<sup>23</sup> HTTP, FTP, usw.

### 6.3.2 Fallbeispiel: Schichtenmodell

#### ► Schnittstellenspezifikation: Modelschicht ▼

```

1  //-----
2  //  AosDbContext.cs
3  //-----
4  public class AosDbContext : DbContext {
5
6      public DbSet<Trait> Traits { get; set; }
7      public DbSet<TraitItem> TraitItems { get; set; }
8
9      public AosDbContext(
10          DbContextOptions<AosDbContext> options)
11          : base(options)
12      ) { }
13
14  protected override void
15      OnModelCreating(ModelBuilder builder)
16      {
17          builder.Entity<Attack>()
18              .HasIndex(a => a.Identifier)
19              .IsUnique();
20
21          builder.Entity<Attack>()
22              .HasOne(a => a.Creature)
23              .WithMany()
24              .HasForeignKey(a => a.CreatureId);
25
26          builder.Entity<Attack>()
27              .Property(a => a.AttackType)
28              .HasConversion<string>();
29
30          builder.Entity<Trait>()
31              .HasIndex(t => t.Identifier)
32              .IsUnique();
33
34          builder.Entity<TraitItem>()
35              .HasKey(ti => new { ti.CreatureId,
36                  ti.TraitId });
37
38          builder.Entity<TraitItem>()
39              .HasOne(ti => ti.Creature)
40              .WithMany()
41              .HasForeignKey(ti =>
42                  ti.CreatureId);
43
44      }

```

#### ► Schnittstellenspezifikation: Domainschicht ▼

```

1  //-----
2  //  IRepository.cs, ARepository.cs
3  //-----
4  public interface IRepository<TEntity> where
5      TEntity : class {
6
7      TEntity Create(TEntity t);
8
9      List<TEntity> CreateRange(List<TEntity>
10          list);
11
12      void Update(TEntity t);
13
14      void UpdateRange(List<TEntity> list);
15
16      TEntity? Read(int id);
17
18      List<TEntity>
19          Read(Expression<Func<TEntity, bool>>
20              filter);
21
22
23  public abstract class ARepository<TEntity> :
24      IRepository<TEntity> where TEntity :
25          class {
26
27      protected readonly AosDbContext Context;
28
29      protected readonly DbSet<TEntity> Table;
30
31      protected ARepository(AosDbContext context) {
32          Context = context;
33          Table = context.Set<TEntity>();
34      }
35
36      public TEntity Create(TEntity t) {
37
38          Table.Add(t);
39          Context.SaveChanges();
40
41          return t;
42      }
43
44      public List<TEntity>
45          CreateRange(List<TEntity> list) {
46
47          Table.AddRange(list);
48          Context.SaveChanges();
49
50          return list;
51      }
52
53      public void Update(TEntity t) {
54
55      }

```



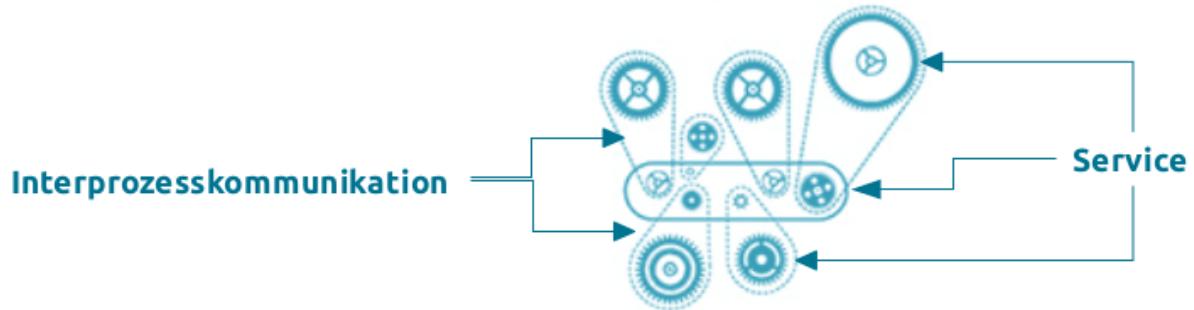


Abbildung 6. SOA - Zusammenspiel von Services

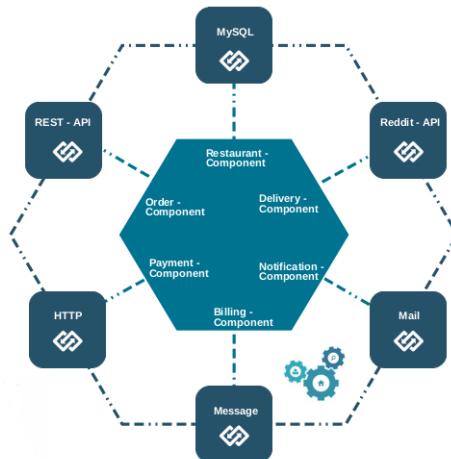
## 6.4. Komponenten

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.



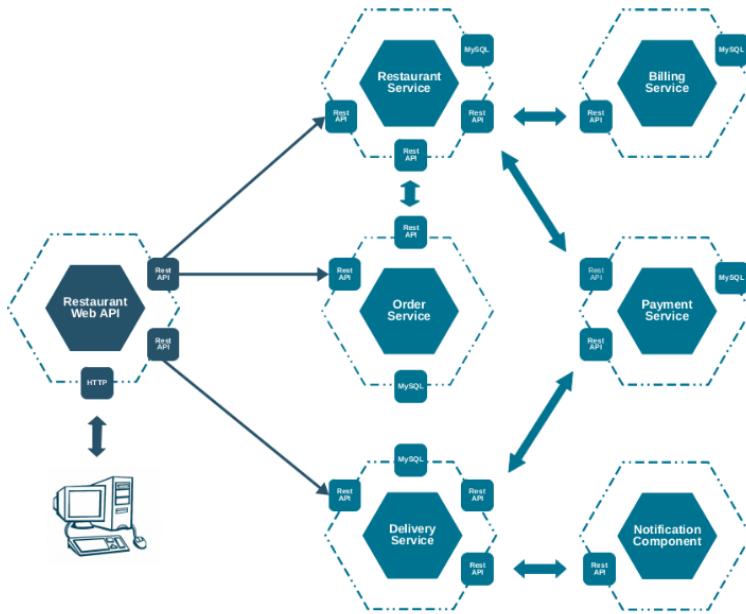
### 6.4.1 Fallbeispiel: Restaurantverwaltung

#### 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 versendet 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 durchzuführen 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 **Microsystemwendung** 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.
- **Komponenten** bestehen aus **Schichten**. Schichten bestehen aus **Klassen**.
- **Klassen** werden durch **Methoden** strukturiert.



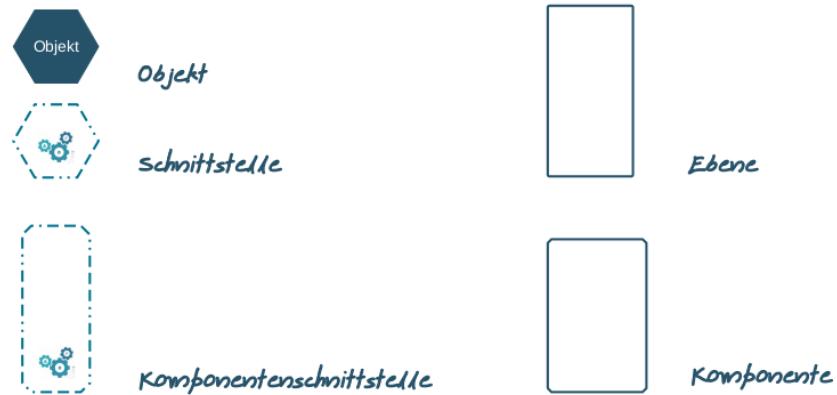
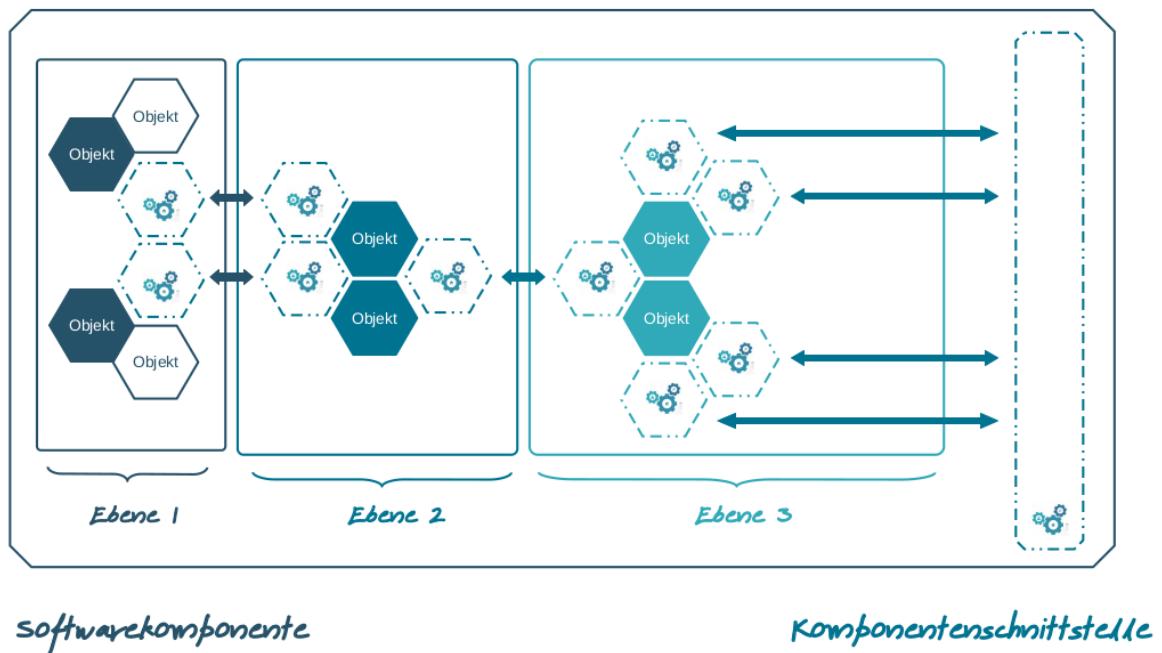


Abbildung 7. Strukturierung einer Komponente



## 7. Programmierung: Metriken

# 02

Programmierung: Metriken

01. Softwaremetriken	50
02. Koppelung	51
03. Kohäsion	55

### 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: Softwaremetriken ▾

- 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**<sup>24</sup>, das Objekte einer Klasse von Objekten anderer Klassen in Anspruch nehmen.

Die Interaktionskoppelung ist die dominant auftretende Form der Koppelung in Softwareprogrammen.

- **Vererbungskoppelung:** Vererbungskoppelung beschreibt das **Ausmaß der Abhängigkeit** der erbenden und der Basisklasse<sup>25</sup>. 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.

<sup>24</sup> Methodenaufruf

<sup>25</sup> Vererbung

#### ► Codebeispiel: Interaktionskoppelung ▾

```

1 //-----
2 // Interaktionskoppelung
3 //-----
4 public class StockMarket {
5     private decimal _price;
6
7     protected List<Portfolio> Portfolios
8         { get; set; } = new();
9
10    public decimal Price {
11        set {
12            _price = value;
13            // Interaktionskoppelung -
14            // Wird es notwendig in anderer
15            // Form auf die Änderungen im
16            // Kurs zu reagieren, muss die
17            // Codebasis der Klasse ver-
18            // ändert werden
19            Portfolios.ForEach(
20                p => p.NotifyPriceChange(value);
21            );
22        }
23    }
24 }

```



#### 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 Werkzeugen 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 ▾

```

1 //-----
2 // Entkoppelter Code
3 //-----
4 // Einsatz des Observerpatterns zur Entkopp-
5 // elung der Softwareelemente
6 public interface IObserver {
7     void Update(decimal price);
8 }
9
10 public interface IObservable {
11     void AddObserver(IObserver observer);
12     void RemoveObserver(IObserver observer);
13     void NotifyObservers();
14 }
15
16 public class StockMarket : IObservable {
17     private decimal _currentPrice;
18     private List<IObserver> _observers
19         = new();
20
21     public decimal Price {
22         set {
23             _currentPrice = value;
24             NotifyObservers();
25         }
26     }
27
28     public void AddObserver(
29         IObserver observer
30     ) => _observers.Add(observer);
31
32     public void RemoveObserver(
33         IObserver observer
34     ) => _observers.Remove(observer);
35
36     public void NotifyObservers() =>
37         _observers.ForEach (
38             o => o.Update(_currentPrice);
39         );
40 }
41
42 public class Portfolio : IObserver{
43     public void Update(decimal newPrice) {
44         // Logik zur Reaktion auf Preisaenderung
45     }
46 }

```

## 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 **ein-zubetten** z.B. durch Aggregation oder Referenzierung.

Zur **Auflösung der Vererbungskoppelung** 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 public class DefaultQuackBehaviour :
13     IQuackable {
14     public String Quack() => "quack";
15 }
16
17 public class LoudQuackBehaviour : IQuackable {
18     public String Quack() => "loudly: quack";
19 }
20
21 public class ProudQuackBehaviour : IQuackable{
22     public String Quack() =>
23         "proudly and loudly: quack";
24 }
25
26 public class SqueezeQuackBehaviour :
27     IQuackable{
28     public String Quack() => "squeeze";
29 }
```



```

1  public class DefaultFlyingBehaviour :  
2      IFlyable {  
3          public String Fly() => "flying high in the  
4              sky";  
5      }  
6  
7  public class NoFlyBehaviour : IFlyable {  
8      public String Fly() => "can't fly";  
9  }  
10  
11 public class Duck{  
12     public IQuackable QuackBehaviour {  
13         get; set;  
14     }  
15  
16     private IFlyable FlyBehaviour {  
17         get; set;  
18     }  
19 }  
20  
21 public class DuckFactory{  
22     public static Duck CreateRedheadDuck() =>  
23         new Duck(){  
24             QuackBehaviour = new  
25                 LoudQuackBehaviour(),  
26                 FlyBehaviour = new  
27                     DefaultFlyingBehaviour()  
28             };  
29  
30     public static Duck CreateEntlingDuck() =>  
31         new Duck() {  
32             QuackBehavior = new  
33                 ProudQuackBehaviour(),  
34                 FlyBehavoir = new  
35                     DefaultFlyingBehaviour()  
36             };  
37  
38     public static Duck RubberDuck () =>  
39         new Duck() {  
40             QuackBehavior = new  
41                 SqueezeQuackBehaviour(),  
42                 FlyBehavior = new NoFlyBehaviour()  
43         };  
44     }  
45 }

```

## 7.2.8 Programmmethodik

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

### ► Auflistung: Programmmethodik ▾

■ **Verwendung von Schnittstellen**<sup>26</sup>: Mit einer Schnittstelle wird ein **Contract**<sup>27</sup> zwischen mehrere Klassen<sup>28</sup> definiert. Damit wird es möglich, Klassen einfach durch andere Klassen zu substituieren.

■ **Verwendung von abstrakten Klassen**: Abstrakte Klassen ermöglichen die **Generalisierung**<sup>29</sup> im Zusammenspiel der Klassen eines Programms. Die Objektorientierte Softwareentwicklung nutzt generalisierte Klassen und Objekte um gemeinsames Verhalten<sup>30</sup> bzw. Eigenschaften<sup>31</sup> 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 bei 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**<sup>32</sup> nicht von der Klasse selbst umgesetzt, sondern an ein Framework weitergegeben.

<sup>26</sup> Interface

<sup>27</sup> Vertrag

<sup>28</sup> Die Klasse verpflichtet sich ein bestimmtes Verhalten zu implementieren.

<sup>29</sup> Generalisierung ist eines der Kernkonzepte der objektorientierten Programmierung.

<sup>30</sup> Methoden

<sup>31</sup> Attribute

<sup>32</sup> 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 ungenutzte Attribute bzw. Methoden für die Klasse definiert werden.

### 7.3.2 Fallbeispiel: Servicekohäsion

```

1  //-----
2  // Servicekohäsion - 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

01. SOLID Prinzipien	56
04. L. Substitutions Prinzip	59
05. Interface Segregation Prinzip	57
03. Open Closed Prinzip	58
02. Single Responsibility Prinzip	57

### 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

```

1 //-----
2 // Verletzung der Open Closed Prinzips
3 //-----
4 public enum EColor {
5     GREEN, YELLOW, RED
6 }
7
8 public enum EAppleType{
9     GOLDEN_LADY, ROSE
10 }
11
12 public class Apple {
13     public string Label { get; set; }
14     public EColor Color { get; set; }
15     public int Weight { get; set; }
16     public EAppleType Type { get; set; }
17     public int Price { get; set; }
18 }
19
20
21 public class AppleHandler{
22     public List<Apple>
23         FilterGreenApples(List<Apple> apples){
24             List<Apple> filteredApples = new ();
25
26             for(Apple a: apples){
27                 if(a.getColor().equals(EColor.GREEN)){
28                     filteredApples.add(a);
29                 }
30             }
31
32             return filteredApples;
33         }
34
35     // nicht veraendert werden.
36     public interface Predicate<T>{
37         bool Test(T t);
38     }
39
40     public WeightFilter : Predicate<Apple> {
41         public int Weight { get; set; }
42         public bool Test (Apple a) =>
43             a.Weight >= Weight;
44     }
45
46     public ColorFilter : Predicate<Apple>{
47         public EColor Color { get; set; }
48         public boolean Test (Apple a) =>
49             a.Color == Color;
50     }
51
52     public class AppleHandler {
53         public List<Apple> Filter(
54             List<Apple> apples,
55             Predicate<Apple> filter
56         ){
57             List<Apple> filteredApples = new ();
58             for(Apple a in apples){
59                 if(filter.Test(a)){
60                     filteredApples.add(a);
61                 }
62             }
63             return filteredApples;
64         }
65     }
66 }

```

## 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**<sup>33</sup>
- Vererbung beschreibt damit eine **ist ein** Beziehung<sup>34</sup> 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`<sup>35</sup> 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.

□

<sup>33</sup> Kindklassen erben dabei das Verhalten ihrer Basisklasse.

<sup>34</sup> Ein Schüler (Kindklasse) ist eine Person (Basisklasse).

<sup>35</sup> denn ein Kreis ist eine Ellipse, nämlich eine Ellipse mit gleich langen Halbachsen

## 9. Programmierung: OOP Entwurf

# 04

### OOP Entwurfsmuster

01. Entwurfsmuster	60
02. Erzeugungsmuster	61
03. Strukturmuster	64

### 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.
  - Singleton
  - 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 Singleton Entwurfsmuster erlaubt einen kontrollierten Zugriff auf die Instanz der Klasse.

### ► Codebeispiel: Fallbeispiel: Singleton ▾

```

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

```

```

1 //-----
2 // Entwurfsmuster: Singleton
3 //-----
4 // Das Singleton Entwurfsmuster definiert eine
5 // Klassenstruktur, die lediglich das Erzeu-
6 // gen einer Instanz der Klasse erlaubt.
7
8 public class Logger {
9     // In der Klasse selbst wird eine Ins-
10    // tanz erzeugt und an ein Feld des Kl-
11    // assenobjekts gebunden.
12    private readonly static Logger _instance
13        = new Logger();
14
15    public const bool LOG_TO_CONSOLE = true;
16
17    // Damit keine Instanzen der Klasse
18    // erzeugt werden koennen wird der Kon-
19    // struktur private gesetzt.
20    private Logger(){}
21
22
23    // Fuer den globalen Zugriff wird eine
24    // Klassenmethode zur Verfuegung gestellt
25    public static final Logger GetInstance(){
26        return _instance;
27    }
28
29    public void Log(String message) {
30        if(LOG_TO_CONSOLE)
31            Console.WriteLine(message);
32    }
33
34 }
35
36 //-----
37 // Fallbeispiel: Singleton
38 //-----
39 public class Programm {
40
41     public static void Main(string[] args) {
42         Logger.LOG_TO_CONSOLE = true;
43
44         Logger logger =Logger.getInstance();
45         logger.info("Hallo Welt");
46     }
47 }

```

## 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 ▾

```

1 // -----
2 // Erzeugermuster: Factory
3 // -----
4 public interface IQuackBehavior{
5     string Quack();
6 }
7
8 public class RedheadDuck : IQuackBehavior{
9     public string Quack() => "quack quack";
10 }
11
12 public class MarbledDuck : IQuackBehavior{
13     public string Quack() => "qua qua qua";
14 }
15
16 public class RubberDuck : IQuackBehavior{
17     public string Quack() => "squeeze";
18 }
19
20 public class DuckDecoy : IQuackBehavior{
21     public string Quack() => "QUACK QUACK";
22 }

```



```

1  // -----
2  // Erzeugungsmuster: Factory
3  // -----
4  public class DuckSimulator {
5      public void Simulate(List<IQuackBehavior>
6          ducks){
7          foreach(IQuackBehaviour duck in ducks){
8              Console.WriteLine(duck.Quack());
9          }
10     }
11
12 // Die Schnittstelle der Factory Klasse
13 public interface IDuckFactory{
14     IQuackBehavior CreateReadHeadDuck();
15     IQuackBehavior CreateMarbledDuck();
16     IQuackBehavior CreateRubberDuck();
17     IQuackBehavior CreateDuckDecoy();
18 }
19
20 public class DecoratedDuckFacotry :
21     IDuckFactory{
22     public IQuackBehavior CreateReadHDuck(){
23         return new OutputDecorator(new
24             QuackCountDecorator(new
25                 ReadHeadDuck()));
26     }
27
28     public IQuackBehavior CreateMarbledDuck(){
29         return new OutputDecorator(new
30             QuackCountDecorator(new
31                 MarbledDuck()));
32     }
33
34     public IQuackBehavior CreateRubberDuck(){
35         return new OutputDecorator(new
36             QuackCountDecorator(new
37                 RubberDuck()));
38     }
39
40     public IQuackBehavior CreateGoose(){
41         return new OutputDecorator(new
42             QuackCountDecorator(new
43                 HonkAdapter(new Goose())));
44     }
45 }
46
47 // -----
48 // Erzeugungsmuster: Factory
49 // -----
50 public class DuckFactory : IDuckFactory{
51     public IQuackBehavior CreateReadHDuck(){
52         return new ReadHeadDuck();
53     }
54
55     public IQuackBehavior CreateMarbledDuck(){
56         return new MarbledDuck();
57     }
58
59     public IQuackBehavior CreateRubberDuck(){
60         return new RubberDuck();
61     }
62
63     public IQuackBehavior CreateGoose(){
64         return new HonkAdapter(new Goose());
65     }
66 }
67
68 public class Programm{
69     public static void Main(String[] args){
70         List<IQuackBehavior> ducks = new
71             List<>();
72         IDuckFactory factory = new
73             DecoratedDuckFactory();
74
75         ducks.Add(factory.CreateReadHDuck());
76         ducks.Add(factory.CreateMarbledDuck());
77         ducks.Add(factory.CreateRubberDuck());
78         ducks.Add(factory.CreateDuckDecoy());
79         ducks.Add(factory.CreateGoose());
80
81         DuckSimulator sim = new
82             DuckSimulator();
83         sim.Simulate(ducks);
84
85         factory = new DuckFactory();
86
87         ducks.Clear();
88
89         ducks.Add(factory.CreateReadHDuck());
90         ducks.Add(factory.CreateMarbledDuck());
91
92         sim.Simulate(ducks);
93     }
94 }

```



## 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 Klassenbibliothek 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 ▾

```

1 //-----
2 // Entwurfsmuster: Adapter
3 //-----
4 public interface IQuackBehavior{
5     string Quack();
6 }
7
8 public class RedheadDuck : IQuackBehavior{
9     public string Quack(){
10         return "... quack quack";
11     }
12 }
13
14 public class MarbledDuck : IQuackBehavior{
15     public string Quack(){
16         return "... qua qua qua";
17     }
18 }
```

```

1 //-----
2 // Entwurfsmuster: Adapter
3 //-----
4 public interface IHonkBehavior{
5     public string Honk();
6 }
7
8 public class HonkAdapter : IQuackBehavior{
9
10     private IHonkBehaviour _honkable;
11
12     public HonkAdapter(IHonkBehavior honkable){
13         this._honkable = honkable;
14     }
15
16     public string Quack(){
17         return this._honkable.Honk();
18     }
19 }
20
21 public class Goose : IHonkBehaviour{
22     public string Honk(){
23         return "... honk honk";
24     }
25 }
26
27 public class Programm{
28     public static void Main(String[] args){
29         List<IQuackBehavior> ducks = new
30             List<>();
31
32         ducks.Add(new ReadHeadDuck());
33         ducks.Add(new MarbledDuck());
34         ducks.Add(new HonkAdapter(new
35             Goose()));
36
37         DuckSimulator sim = new
38             DuckSimulator();
39         sim.simulate(ducks);
40     }
41 }
42
43 > Ausgabe
44
45 "... quack quack"
46 "... qua qua qua"
47 "... 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 ▾

```

1 //-----
2 // Entwurfsmuster: Dekorator
3 //-----
4 public class Programm{
5     public static void Main(String[] args){
6         List<IQuackBehavior> ducks = new
7             List<>();
8
9         ducks.Add(new
10            QuackCountDecorator(new
11                OutputDecorator(new
12                    ReadHeadDuck())));
13
14         ducks.Add(new
15            QuackCountDecorator(new
16                OutputDecorator(new
17                    MarbledDuck())));
18
19         DuckSimulator sim = new
20             DuckSimulator();
21         sim.Simulate(ducks);
22
23         ConsoleWriteLn("quack count: " +
24             QuackCountDecorator.COUNT);
25     }
26 }
27
28 > Ausgabe:
29
30 "Output: ... quack quack"
31 "Output: ... qua qua qua"
32 "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

```

1 //-----
2 // Schnittstelle: ICommand
3 //-----
4 public interface ICommand{
5     void execute();
6     void undo();
7 }
8
9 // Klasse: ACommand
10 //-----
11 public abstract class ACommand {
12     protected Robot _robot;
13     public ACommand(Robot robot) {
14         _robot = robot;
15     }
16     public abstract void Process();
17     public abstract void Undo();
18 }
19
20 // Klasse: Point
21 //-----
22 public class Point {
23     public int X { get; set; }
24     public int Y { get; set; }
25
26     public Point(int x, int y) {
27         X = x; Y = y;
28     }
29
30     public Point CalculateNeighbour(
31         EDirectionType direction
32     ) {
33         Point p = null;
34

```



```

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 :
5     IPaymentStrategy{
6
7     private CreditCardProcessor processor =
8         new CreditCardProcessor();
9
10    private string _cardNumber;
11    private string _name;
12
13    public CreditCardStrategy(string name,
14        string cardNumber){
15        this._name = name;
16        this._cardNumber = cardNumber;
17    }
18
19    public void pay(int amount){
20        processor.process(
21            _name, _cardNumber, amount
22        );
23    }
24
25 //-----
26 // Klasse: CreditCardStrategy
27 //-----
28 public class PaypalStrategy :
29     IPaymentStrategy{
30
31     private PaypalProcessor processor = new
32         PaypalProcessor();
33
34     private string _email;
35     private string _pwd;
36
37     public PaypalStrategy(String email,
38         String pwd){
39        this._email = email;
40        this._pwd = pwd;
41    }
42
43    public void pay(int amount){
44        processor.process(
45            _email,
46            _pwd
47        );
48    }
49 }
```

```

1  //-----
2  // Klasse: Item
3  //-----
4  public class Item {
5
6      private string _upcCode;
7      private string _price;
8
9      public UpcCode{
10         get => _upcCode;
11     }
12
13     public Price{
14         get => _price;
15     }
16
17     public Item(string upc, int cost){
18         this._upcCode = upc;
19         this._price = cost;
20     }
21 }
22 //-----
23 // Klasse: ShoppingCart
24 //-----
25 public class ShoppingCart{
26
27     private List<Item> _items = new
28         List<Item>();
29
30     private IPaymentStrategy _paymentMethod;
31
32     public IPaymentStrategy PaymentMethod {
33         get => _paymentMethod;
34         set => _paymentMethod = value;
35     }
36
37     public void AddItem(Item item){
38         _items.Add(item);
39     }
40
41     public int CalculateTotal(){
42         int sum = 0;
43         foreach(var item in _items){
44             sum += item.Price;
45         }
46
47         return sum;
48     }

```



```

1     public void Pay(){
2         int amount = CalculateTotal();
3         _paymentMethod.pay(amount);
4     }
5 }
6
7 //-----
8 // Klasse: ShoppingCartUnitTest
9 //-----
10 public class ShoppingCartUnitTest{
11
12     [Test]
13     public void Test(){
14         ShoppingCart cart = new ShoppingCart();
15
16         Item item1 = new Item("234", 10);
17         Item item2 = new Item("567", 30);
18
19         cart.PaymentMethod =
20             new PaypalStrategy(
21                 "myemail@example.com", "mypwd"
22             );
23
24         cart.Pay();
25     }
26 }
27

```



# Grundlagen der objektorientierten Programmierung

December 14, 2019

## 10. Architekturstil: Rest

# 02

REST

01. Rest Prinzipien	72
02. Ressourcen	76
03. Http Methoden	78
04. Fallbeispiel: Ordermanager	80

### 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<sup>36</sup> 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**<sup>37</sup> von anderen Architekturstilen.



<sup>36</sup> Das Akronym REST steht für *Representational State Transfer*.

<sup>37</sup> 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** zugesetzt.

#### ► Erklärung: Addressierbarkeit ▾

- Durch den Einsatz von URLs<sup>38</sup> können Ressourcen im Internet einfach identifiziert werden.
- Damit wird es möglich Ressourcen durch die Verwendung von **HTTP Links** zu verwaltet.

<sup>38</sup> Unique Resource Identifier

### 10.1.4 Entkoppelung von Ressource und Repräsentation

Eine Ressource besitzt im Internet immer mehrere mögliche Repräsentationen<sup>39</sup>.

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 // Repräsentation 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 vor gehalten wird.
- Stattdessen muss der **Kommunikationszustand** im Client oder in der Repräsentation der Ressource gespeichert werden.
- Damit kann die **Koppelung**<sup>40</sup> zwischen Client und Server verringert werden.

<sup>39</sup> Eine Datenbankentität kann z.B. als xml oder json File verschickt werden

<sup>40</sup> Damit wird es möglich dass REST Anwendungen Caches, Load Balancer und andere **Skalierungs artefakte** nutzen .

## 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.
- Repräsentationen können bevorstehende **Zustandsübergänge** in Form von Links an Clients weitergeben.

### ► Erklärung: Zustandsdiagramm<sup>41</sup> ▾

- 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 ▾

```

1 //-----
2 //          JSON - Order
3 //-----
4 var order = {
5     href      : "http://..../orders/21",
6     status    : "created",
7     date     : "2014-10-03",
8     total    : 2090,
9     updated  : "2015-01-11",
10    user: {
11        href:"http://..../users/14",
12        user-name : "Tim"
13    },
14
15    // Das actions array enthaelt alle
16    // moeglichen Zustandsuebergaenge.
17
18    // Der Client kann einen Zustands-
19    // uebergang ausloesen in dem er
20    // einfach dem entsprechendem
21    // Link folgt!
22
23    actions: [
24        {
25            href: "http://.../orders/34/cancel",
26            description: "cancel order"
27        },{
28            href: "http://.../orders/34/process",
29            description: "process order"
30        },{
31            href: "http://.../orders/34/update",
32            description: "update order"
33        }
34    ],
35    products : [
36        {
37            href :"http://..../21/products/47",
38            name : "Der Gladiator",
39            category : "DVD"
40        }
41    ]
42 }
```



<sup>41</sup> *Statemachine*

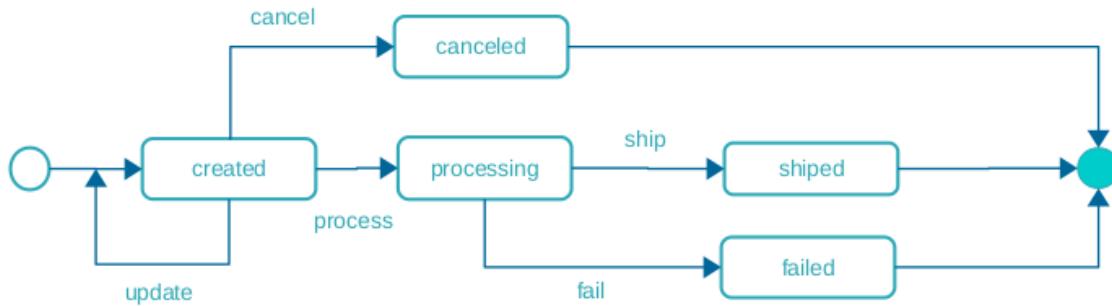


Abbildung 8. State Machine: Produktbestellung

### 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     HttpResponseMessage get();
6
7     HttpResponseMessage put();
8
9     HttpResponseMessage post();
10
11    HttpResponseMessage patch();
12
13    HttpResponseMessage delete();
14
15    HttpResponseMessage 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 ▾

```

1 //-----
2 //      JSON - Project
3 //-----
4 var project = {
5     href : "http://.../projects/1",
6     title : "Finite Methods",
7     description : "The finite ...",
8     creationDate : "10.09.2022",
9     projectType: "REQUEST_FUNDING_PROJECT",
10    projectState: "RUNNING",
11    subprojects : [
12        {
13            href : "http://.../subprojects/2",
14            title : "..."
15        },
16        {
17            href : "http://.../subprojects/7",
18            title: "..."
19        }
20    ],
21    projectFundings : [
22        ...
23    ]
}

```

## 10.2.2 Subressourcen



#### Subressource ▾

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

#### ► Codebeispiel: Subressource ▾

```

1 //-----
2 //      Subressource
3 //-----
4 var project = {
5     href : "http://.../projects/1",
6     title : "Finite Methods ...",
7     description : "....",
8     creationDate : "10.09.2022",
9     //subressource von project
10    "project-leader" : {
11        "href" : "http://.../staff/23",
12        "name" : "Schuetzli"
13    }
14 }

```

### 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 ▾

```

1 //-----
2 //      Listenressourcen Repraesentation
3 //-----
4 var projects = [
5     href : "http://.../rest/projects",
6     data : [
7         {
8             href : "http://.../projects/1"
9             title : "Finite Methods"
10        },
11        {
12            href : "http://.../projects/2"
13            title : "cloud systems research"
14        },
15        ...
16    ]
17 ]

```

► 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 Ressourcen.

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 Ressourcen, 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:** Listenressourcen repräsentieren in erster Linie eine Sammlung von Subressourcen.
- **Userlisten, Produktlisten, Reviewlisten, Orderlisten**

### 10.4.2 Repräsentationen von Ressourcen

#### ► Codebeispiel: XML Repräsentationen User

```

1  <!-- ----- -->
2  <!--      XML Repräsentation von User      -->
3  <!-- ----- -->
4  <?xml version="2.0"?>
5  <user>
6      <href rel="self">
7          http://.../ordermanager/users/32
8      </href>
9
10     <first-name>Franz</first-name>
11     <middle-name>Josef</middle-name>
12     <last-name>Kurz</last-name>
13
14     <user-name>shorty134</user-name>
15     <joined-at>2011-11-11</joined-at>
16 </user>

```

#### ► Codebeispiel: Json Repräsentationen User

```

1  //-----
2  //      JSON Repräsentation: User
3  //-----
4  var user = {
5      href : "http://.../ordermanager/users/32",
6      first-name : "Franz",
7      middle-name : "Josef",
8      last-name : "Kurz",
9      user-name : "shorty134"
10     joined-at : "2011-11-11"
11 }
12
13 var product = {
14     href : "http://.../ordermanager/products/1",
15     //Stammdaten
16     name : "Forbidden Stars"
17     search-terms : [
18         "toy", "boardgame", "40k",
19         "forbidden stars"
20     ],
21     description : "...",
22     price : 99,
23
24     //Subressource
25     //Listenressource
26     reviews : {
27         href : "http://.../users/32/reviews",
28         entries : [
29             {
30                 user: {
31                     href : "http://.../users/32",
32                     user-name : "huki"
33                 },
34                 text : "Das ist ein tolles ..."
35             },
36             {
37                 user: {
38                     href : "http://.../users/12",
39                     user-name : "gronkh"
40                 },
41                 text : "bla bla ..."
42             },
43             {
44                 user: {
45                     href : "http://.../users/0",
46                     user-name : "macco"
47                 },
48                 text : "??? ..."
49             }
50         }
51     }
52 }

```

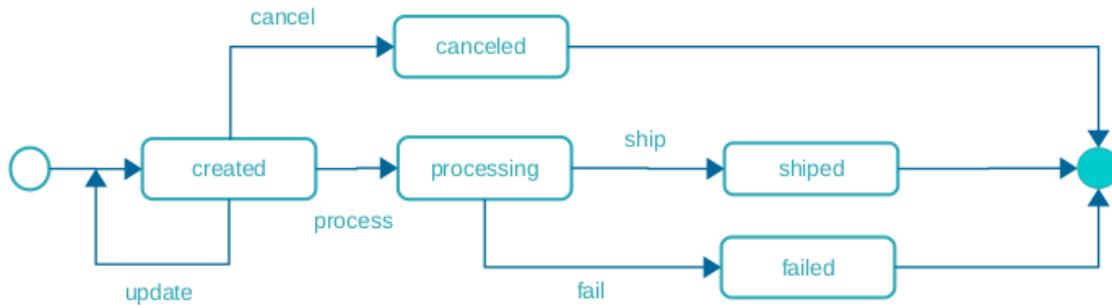


Abbildung 9. Zustände einer Bestellung

► Codebeispiel: Einzelne Bestellung ▾

```

1 //-----
2 // JSON Repraesentation: Order
3 //-----
4 var order = {
5   href : "http://..../orders/21",
6   status : "processing",
7   date : "2014-10-03",
8   total : 2090,
9   updated : "2015-01-11",
10  user : {
11    href :"http://..../users/14",
12    user-name: "Tim"
13  },
14  items : [
15    {
16      quantity: 1,
17      product : {
18        href : "http://..../products/10",
19        description : "Laptop X65",
20        price : 799
21      },
22      quantity: 4,
23      product : {
24        href : "http://..../products/13",
25        description : "MP3 Player",
26        price : 99
27      }
28    },
29    {
30      quantity: 1,
31      product : {
32        href : "http://..../products/7",
33        description : "Boardgame",
34        price : 70
35      }
36    ]
37 }
  
```

► Codebeispiel: Liste der Bestellungen ▾

```

1 //-----
2 // JSON Repraesentation: orders
3 //-----
4 var orders = [
5   {
6     href :"http://..../orders/32",
7     orders : [
8       {
9         href : "http://..../orders/32",
10        date : "2014-10-03",
11        price : 315,
12        items : [
13          "Laptop X65", "MP3 Player"
14        ]
15      },
16      {
17        href : "http://..../orders/2",
18        date : "2015-05-03",
19        price : 100,
20        items : [
21          "Forbidden Stars"
22        ]
23      },
24      {
25        href : "http://..../orders/2",
26        date : "2015-05-03",
27        price : 132,
28        items : [
29          "Chess - Lotzi"
30        ]
31      },
32      {
33        href : "http://..../orders/2",
34        date : "2015-05-03",
35        price : 525,
36        items : [
37          "Warhammer 40K"
38        ]
39      }
40    ]
41  ]
  
```

### 10.4.3 Analyse einer Repräsentation

#### ► Codebeispiel: Order Repräsentation ▾

```

1  //-----
2  //      JSON Repräsentation: order
3  //-----
4  var order = {
5      href : "http://..../orders/21",
6      state : "processing",
7      date : "2014-10-03",
8      total : 2090,
9      updated : "2015-01-11",
10     user: {
11         href : "http://..../users/14",
12         user-name : "Tim"
13     },
14     items: [
15         {
16             quantity : 1,
17             product : {
18                 href : "http://..../products/10",
19                 isbn : "3233-32HU-3",
20                 description : "Laptop X65",
21                 price : 799
22             },
23             {
24                 quantity : 1,
25                 product : {
26                     href : "http://..../products/19",
27                     isbn : "3233-8763-3",
28                     description : "Forbidden Stars",
29                     price : 150
30                 }
31             }
32         ],
33     actions : [
34         {
35             href : "http://..../orders/21/cancel",
36             description: "cancel order"
37         },
38         {
39             href : "http://..../orders/21/process",
40             description: "process order"
41         },
42         {
43             href : "http://..../orders/21/update",
44             description: "update order"
45         }
46     ]
47 }

```

#### ► Analyse: Order Repräsentation ▾

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

```

1  var order = {
2      href : "http://..../orders/21",
3      ...
4  }

```

- **Stammdaten der Ressource:**

```

1  var order = {
2      state : "processing",
3      date : "2014-10-03",
4      total : 2090,
5      updated : "2015-01-11",
6      ...
7  }

```

- **Subressourcen:**

```

1  var order = {
2      ...
3      user: {
4          href : "http://..../users/14",
5          user-name : "Tim"
6      },
7      items: [
8          {
8             quantity : 1,
9             product : {
10                 href : "http://..../products/10",
11                 isbn : "3233-32HU-3",
12                 description : "Laptop X65",
13                 price : 799
14             },
15             {
16                 ...
17             }
18         ],
19     actions : [
20         {
21             href : "http://..../orders/21/cancel",
22             description: "cancel order"
23         },
24         ...
25     ]
26 }

```

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

Der Status einer Bestellung ist Teil der Stammdaten der Ressource.

```
1  var order = {  
2      href :"http://.../orders/21",  
3      state : "processing",  
4      ...  
5  }
```

- **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.

```
1  var order = {  
2      href :"http://.../orders/21",  
3      state : "processing",  
4      ...  
5      actions : [  
6          {  
7              href :"http://.../orders/21/cancel",  
8              description: "cancel order"  
9          }, {  
10              href :"http://.../orders/21/process",  
11              description: "process order"  
12          }, {  
13              href :"http://.../orders/21/update",  
14              description: "update order"  
15          }  
16      ]  
17  }
```

