Softwareentwicklung

Skriptum zur Vorlesung - 01.09.2023

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

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

Abstract:

MSC: p.panhofer@htlkrems.at

Keywords:

Contents

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

Grundlagen der objektorientierten Programmierung

December 14, 2019

1. Grundlagen der Programmierung

01

Grundlagen der Programmierung

1.1. Softwareprogramm

Softwareprogramm

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

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

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

1.1.1 Programmbaustein: Anweisung

Ein Softwareprogramm besteht in der Regel aus einer freie Abfolge von Anweisungen.

> Für Programme werden 2 Ausprägungen von Anweisungen unterschieden: **Deklarationen** und **Instruktionen**.

Auflistung: Anweisungen

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

  Variablen werden zur Speicherung von Daten in Programmen verwendet.

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

❑

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

Grundlagen

1.2. Datentypen und Variablen

Variable

Variablen sind **Datencontainer** für veränderbare Werte. Variablen werden zur Speicherung von Daten in Programmen verwendet.

Eine der fundamentalen Aufgaben eines Programms ist die **Verwaltung** von Daten.

Erklärung: Variablen

- Eine Variable ist ein **Container** der Werte speichern kann.
- Eine Variable besitzt dazu einen **Namen**, mit dem auf die in ihr gespeicherten Daten Bezug genommen wird und einen **Datentyp**, der die Art der Information bestimmt, die gespeichert werden kann.
- Mit einer Variable wird ein Teil des Arbeitsspeichers verwaltet. Dem Namen der Variable wird dabei die **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 sie auch einen Wert bekommt.

❑

1.2.1 Variablendeklaration/Initialisierung

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

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

Erklärung: Variablendeklaration

- Stößt der Computer zur **Laufzeit** eines Programms auf eine Variablendeklaration, reserviert er für die Variable Speicherplatz in seinem Arbeitsspeicher.
- Mit der Variablendeklaration kann einer Variable unter Zuhilfenahme des **Zuweisungsoperators** auch ein Wert zugewiesen werden. Man spricht dann von einer **Variableninitialisierung**.

Codebeispiel: Variablendeklaration

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

❑

1.2.2 Datentypen

Infolge einer **Variablendeklaration** wird einer Varialbe zusätzlich zu einem Namen auch 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 bestimmten Datentyp zugeordnet werden.
- C# unterscheidet dabei die folgenden einfachen Datentypen: `int`, `short`, `byte`, `long`, `double`, `float`, `char`.

Codebeispiel: Datentypen

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

❑

1.3. Kontrollstruktur

Kontrollstrukturen

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

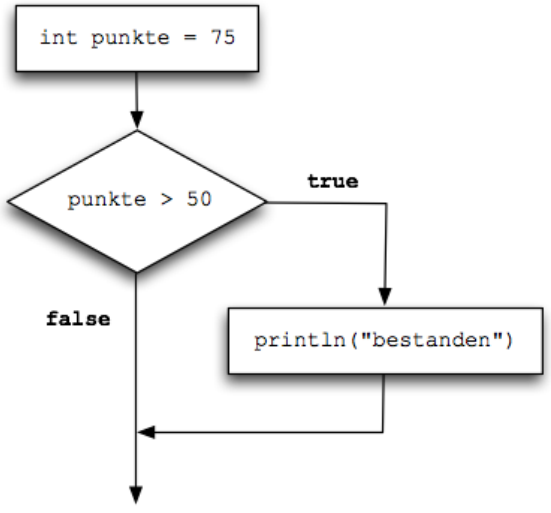
Damit bestimmen Kontrollstrukturen die **Reihenfolge** in der die Anweisungen eines Programmes ausgeführt werden.

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, von oben nach unten, ausgeführt. Manchmal möchte man aber eine Zeile - oder einen ganzen Block von Zeilen - nur unter einer bestimmten **Bedingung** durchführen.

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



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

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

▸ Codebeispiel: if Bedingung

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

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

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

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

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

❑

1.3.2 Bedingungen auswerten

Bedingungen

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

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

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

▸ Codebeispiel: Bedingung auswerten

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


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


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

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

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

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

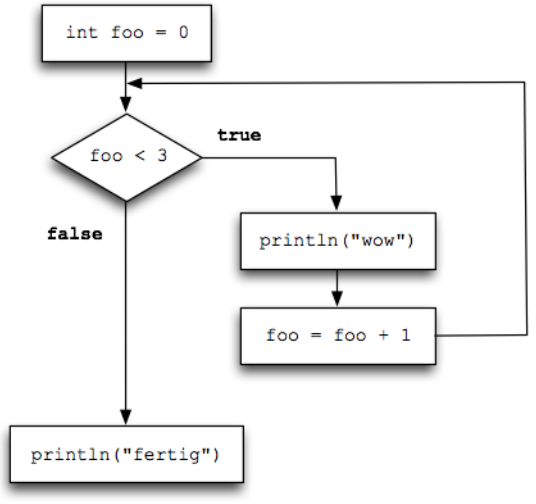
❑

1.3.3 Kontrollstruktur: WHILE

Schleife

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

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



▸ Codebeispiel: while Schleife

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

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

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

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

❑

1.3.4 Kontrollstruktur: FOR

Mit der `for` Schleife steht eine Schleifenform zur Verfügung, für die bestimmt werden kann, wie oft die Schleife ausgeführt wird.

▸ Codebeispiel: while Schleife

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

// ---------------------------------------
//  Beispiel: FOR Schleife
// ---------------------------------------



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

❑

.

.

2. Grundlagen der objektorientierten Programmierung

02

Grundlagen der OOP

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, als Programmierprinzip war eine neue **Denkweise**, die sich in der Welt der Programmentwicklung verfestigte.

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[4] 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.

❑

[4] *Tastendruck, Maus*

2.1.2 Objekte

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

▸ **Auflistung: Größen eines Objekts**

- **Eigenschaften:** Jedes Objekt besitzt sogenannte Eigenschaften[5]. Diese Eigenschaften dienen dazu, das Objekt näher zu beschreiben.

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

- **Zustand:** Der Zustand eines Objekts wird durch die Summe, der in den Eigenschaften des Objekts gespeicherten **Werte**, beschrieben.
  - **Code:**
  - **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 Siedlung zu Siedlung flieden, Drachen angreifen bzw. von Drachen angegriffen werden, `Crew` Objekte mitnehmen usw..

❑

[5] *manchmal auch als Attribute bzw. Properties bezeichnet.*

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

❑

2.2. Elemente einer Klasse

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

2.2.1 Fallbeispiel: Last Aurora

Folgenden Klassen dienen als Vorlage für nachfolgende Kapitel.

Codebeispiel: Last Aurora

```
// ----------------------------------------
//  Klasse: Truck.cs
// ----------------------------------------
// (1) Classdefinition
public class Airship {

   // (2) Properties
   public string Code { get; set; }
   public int Speed { get; set; }
   public int PullForce { get; set; }
   public List<Keyword> Keywords { get; set; }
   public List<Compartment> Compartments
      { get; set; }

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

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

// Instanzieren eines Airship Objekts
Airship a = new Airship();

// Werte setzten
a.Code = "Dragon Spire";
a.Speed = 7;
a.PullForce = 4;
```

❑

2.2.2 Klassenelemnt: Konstruktuor

Objektinstanzierung

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

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

Analyse: Konstruktor

- Im Zuge der **Instanzierung** eines Objekts, wird der Konstruktor der Klasse aufgerufen. Der Konstruktor dient dabei in erster Linie der **Objektinitialisierung**[6].
- Der Konstruktor hat dabei denselben Namen wie die Klasse.

Codebeispiel: Konstruktor

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

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

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

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

❑

[6] *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 seinen 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 des Objekts gespeicherten Werte.

▸ **Codebeispiel: Properties** ▾

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

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

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

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

❑

Grundlagen der objektorientierten Programmierung

December 14, 2019

3. Programmierung: Strukturierung

01

Sturkturierung von Programmen

3.1. Unterprogramme

Historisch gesehen hat alles mit einem bunten Gemisch aus **Anweisungen** und **Daten** innerhalb eines Betriebssystemprozesses[7] 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**.

3.1.1 Unterprogramme

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

▸ **Erklärung: Unterprogramme**

- **Unterprogramme**[8] 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**[9] der Anwendung gesunken wäre.
- Als nächstes wurden **Container** für Daten[10] 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 }
```

❑

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

3.2. Objektorientierung

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

3.2.1 Objektorientierung

Objektorientierung faßt Strukturen und Unterprogramme zu **Klassen**[11] 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**[12] zur Verfügung, die den **Zustand**[13] 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 }
```

❑

[11] *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.*

[12] *Methoden*

[13] *Variblen*

3.3. 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.

3.3.1 Prinzipien des Schichtenmodells

▸ **Prinzip: Schichtenmodell**

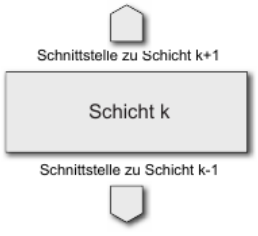
- **Teile und Herrsche:** Ein komplexes Problem wird in **unabhängige Teilprobleme** zerlegt, das jedes für sich, einfacher handhabbar ist, als das Gesamtproblem.

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

- **Unabhängigkeit:** Die einzelnen Schichten der Anwendung kommunizieren miteinander, indem die **Schnittstellenspezifikation**[14] des direkten Vorgängers bzw. Nachfolgers genutzt wird.

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

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



- **Standadisierung:** Die Gliederung des Gesamtproblems in einzelne Schichten erleichtert die Entwicklung von **Standards**[15] für die einzelnen Schichten.

❑

[14] *Schnittstelle, Interface*

[15] *HTTP, FTP, usw.*

3.3.2 Fallbeispiel: Schichtenmodell

Schnittstellenspezifikation: Modelschicht

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

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

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

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

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

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

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

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

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

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

Schnittstellenspezifikation: Domainschicht

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

    TEntity Create(TEntity t);

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

    void Update(TEntity t);

    void UpdateRange(List<TEntity> list);

    TEntity? Read(int id);

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

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

    protected readonly AosDbContext Context;

    protected readonly DbSet<TEntity> Table;

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

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

        return t;
    }

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

        return list;
    }

    public void Update(TEntity t) {
```

```
        Context.ChangeTracker.Clear();

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

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

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

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

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

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

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

▸ Schnittstellenspezifikation: Serviceschicht ▾

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

    private IRepository<TEntity> _repository;

    private ILogger<AController<TEntity>>
        _logger;

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

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

        return t;
    }

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

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

        return Ok(data);
    }

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

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

        if (data is null) return NotFound();

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

❑

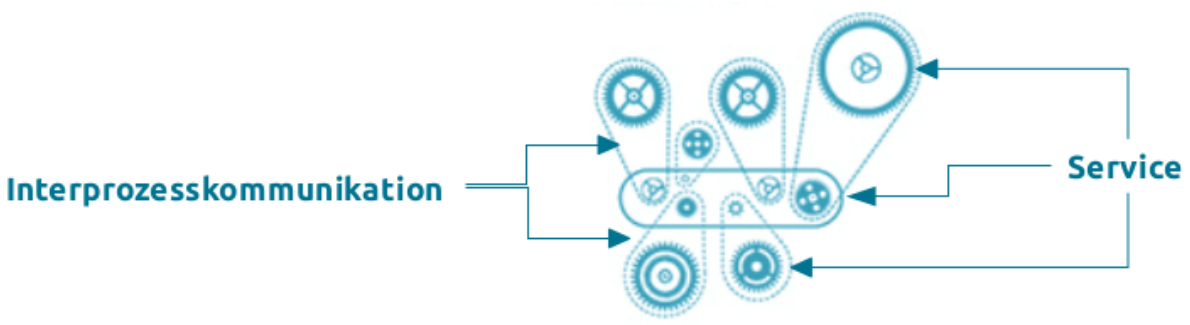


Abbildung 1. **SOA - Zusammenspiel von Services**

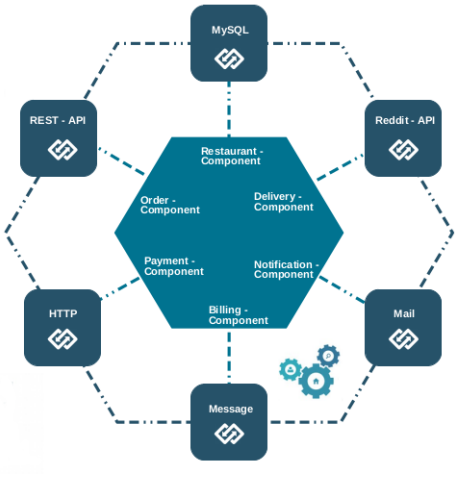
3.4. Komponenten

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.



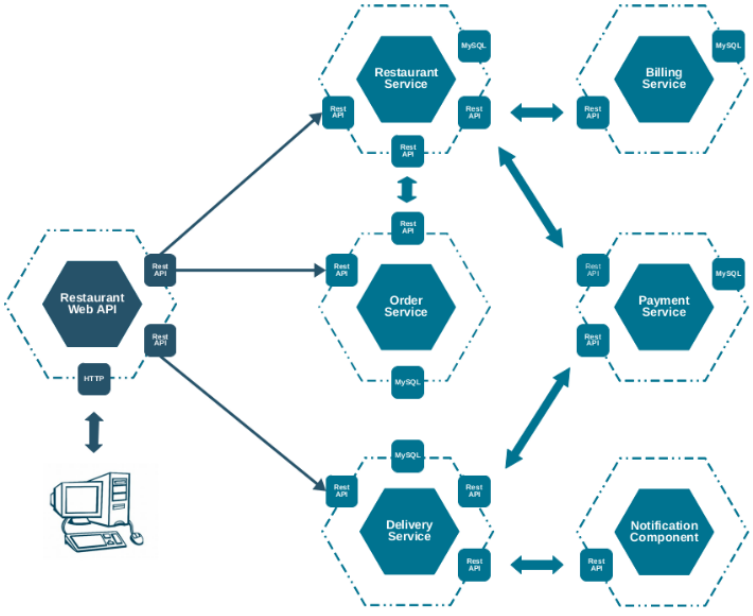
3.4.1 Fallbeispiel: Restaurantverwaltung

▸ **Fallbeispiel: Restaurantverwaltungssoftware**

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

❑

Microservice – Architektur



3.5. Service

Service

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

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

Analyse: Service

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

❑

3.5.1 Zusammenfassung

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

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

Analyse: Codestrukturierung

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

❑

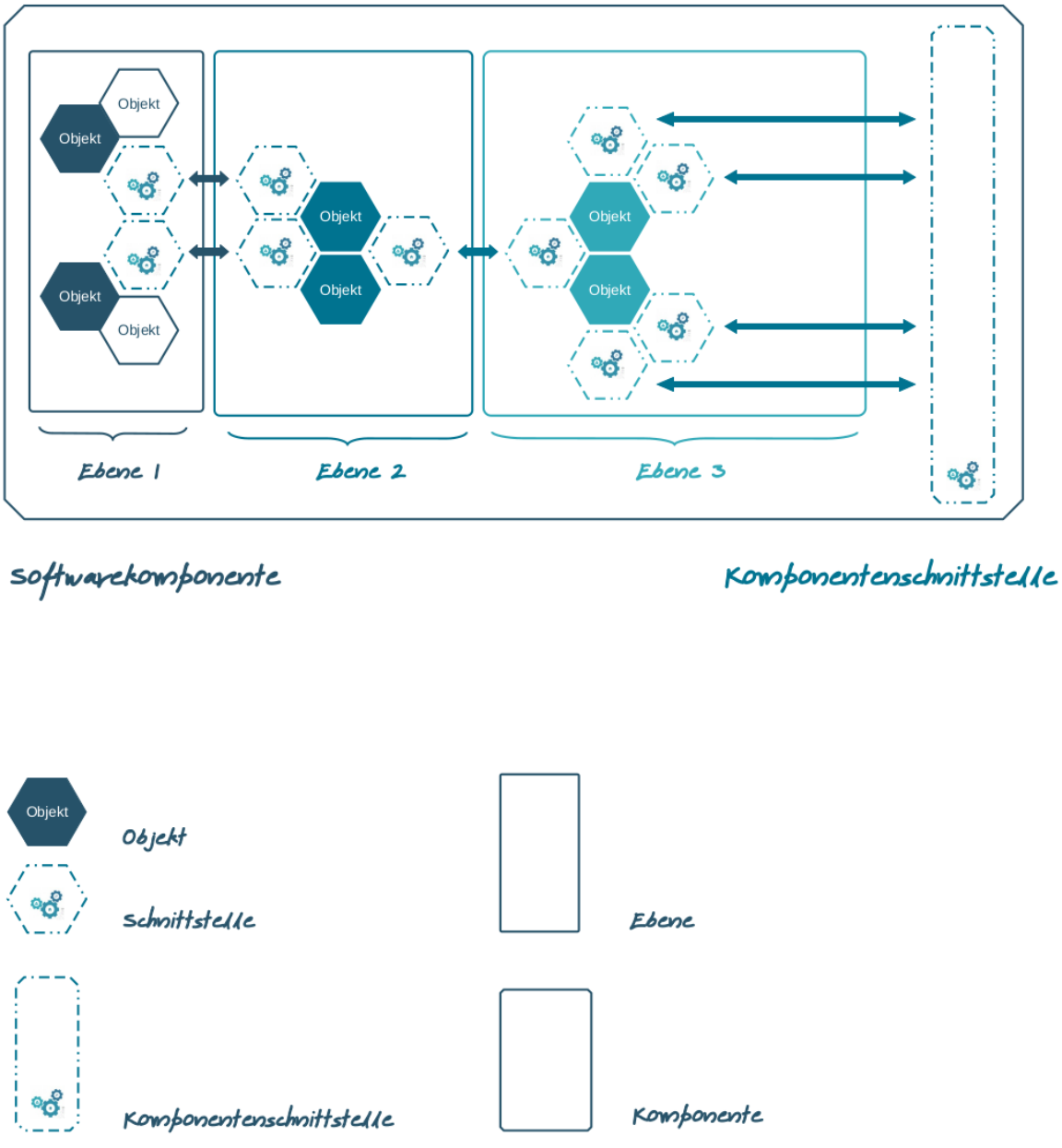


Abbildung 2. **Strukturierung einer Komponente**

.

.

4. Programmierung: Metriken

02

Programmierung: Metriken

4.1. Softwaremetriken

4.1.1 Metriken

Softwaremetrik

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

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

Erklärung: Softwaremetrkiken

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

❑

4.1.2 Qualitätsmetriken

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

Auflistung: Softwaremetriken

- **Koppelung:** Maß der **Abhängigkeiten** zwischen Softwareelementen[16].
- **Kohäsion:** Maß des inneren **Zusammenhalt** eines Softwareelements.

❑

[16] *Objekte, Schichten, Komponenten*

Softwaremetriken

4.2. Koppelung

4.2.1 Koppelung

Koppelung

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

Beim Entwurf eines **Softwaresystems** ist eine **geringe Koppelung** anzustreben.

Auflistung: Arten der Koppelung

- **Interaktionskoppelung:** Interaktionskoppelung beschreibt das **Mass an Funktionalität**[17], das Objekte einer Klasse von Objekten anderer Klassen in Anspruch nehmen.
- **Vererbungskoppelung:** Vererbungskoppelung beschreibt das **Ausmaß der Abhängigkeit** zwischen erbender und Basisklasse.

❑

4.2.2 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.

Codebeispiel: Interaktionskoppelung

```
1  //----------------------------------------
2  //  Interaktionskoppelung
3  //----------------------------------------
4  public class Swordsman {
5     public int AttackValue { get; set; }
6
7     public bool Attack =>
          Dice.GetInstance().Roll() <=
          AttackValue;
8  }
9
10 public class Spearman {
11    public int AttackValue { get; set; }
12
13    public bool Attack =>
          Dice.GetInstance().Roll() <=
          AttackValue;
14 }
15
16 public class Bowman {
17    public int AttackValue { get; set; }
18
19    public bool Attack =>
          Dice.GetInstance().Roll() <=
          AttackValue;
20 }
```

[17] *Methodenaufruf*

▸ Codebeispiel: Interaktionskoppelung

```
//----------------------------------------
//  Interaktionskoppelung
//----------------------------------------
public class GameController {

   public int DetermineHits() {
      int attackCount = 0;

      var unit1 = new Swordsman(){
         AttackValue = 5
      };
      var unit2 = new Spearman(){
         AttackValue = 4
      };
      var unit3 = new Bowman(){
         AttackValue = 6
      };

      // Interaktionskoppelung
      if(unit1.Attack()) {
         ++attackCount;
      }

      if(unit2.Attack()) {
         ++attackCount;
      }

      if(unit3.Attack()) {
         ++attackCount;
      }

      return attackCount
   }
}
```

❑

4.2.3 Auflösen von Interaktionskoppelung

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

▸ Analyse: Interaktionskoppelung

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

❑

4.2.4 Fallbeispiel: Auflösen von Interaktionskoppelung

```
//----------------------------------------
//  Entkoppelter Code
//----------------------------------------
// Schnittstellendefinition
public interface IUnit {
   bool Attack ();
}

// Klassenimplementierung
public abstract class AUnit : IUnit {
   public int AttackValue { get; set; }

   public AUnit (int attackValue) {
      AttackValue = attackValue;
   }

   public bool Attack() =>
       (Dice.GetInstance().Roll() <=
       AttackValue);
}

public class Swordsman : AUnit {
   public Swordsman : base(5){};
}

public class Spearman : AUnit {
   public Spearman : base(4){};
}

public class Bowman : AUnit {
   public Bowman : base(6){};
}

public class GameController {
   public int DetermineHits (List<IUnit>
       army) => army.Aggregate (
         0,
         (total, unit) => unit.Attack() ?
             total++ : total
      );
}

// Ausfuehrung
var army = new List(){
   new Swordsman(), new Bowmen(), new Bowmen()
};

var hits = new GameController().Attack(army);
```

❑

4.2.5 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

```
//----------------------------------------
//  Vererbungskoppelung
//----------------------------------------
public class Duck {
   public String Quack() => "quack";
   public String Fly() =>
      "flying high in the sky";
}

public class RedheadDuck : Duck {
   public String Quack() => "loudly quack";
}

public class EntlingDuck : Duck {
   public String Quack() => "proudly quack";
}

public class RubberDuck : Duck {
   public String Quack() => "squeeze";
   public String Fly() => "can't fly";
}
```

4.2.6 Objektkomposition

Objektkomposition

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

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

Erklärung: Vorteile der Objektkomposition

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

Codebeispiel: Objektkomposition

```
//----------------------------------------
//  Objektkomposition vs. Vererbungskoppelung
//----------------------------------------
public interface IQuackable {
   String Quack();
}

public interface IFlyable {
   String Fly();
}

pulic class DefaultQuackBehaviour :
     IQuackable {
   public String Quack() => "quack";
}

public class LoudQuackBehaviour : IQuackable {
   public String Quack() => "loudly: quack";
}

public class ProudQuackBehaviour : IQuackable{
   public String Quack() =>
      "proudly and loudly: quack";
}

public class SqueezeQuackBehaviour :
     IQuackable{
   public String Quack() => "squeeze";
}
```

```
public class DefaultFlyingBehaviour :
    IFlyable {
   public String Fly() => "flying high in the
       sky";
}

public class NoFlyBehaviour : IFlyable {
   public String Fly() => "can't fly";
}

public class Duck{
   public IQuackable QuackBehaviour {
      get; set;
   }

   private IFlyable FlyBehaviour {
      get; set;
   }
}

public class DuckFactory{
   public static Duck CreateRedheadDuck() =>
        new Duck(){
           QuackBehaviour = new
               LoudQuackBehaviour(),
           FlyBehaviour = new
               DefaultFlyingBehaviour()
        };

   public static Duck CreateEntlingDuck() =>
        new Duck() {
           QuackBehavior = new
               ProudQuackBehaviour(),
           FlyBehavoir = new
               DefaultFlyingBehaviour()
        );

      public static Duck RubberDuck () =>
        new Duck() {
           QuackBehavior = new
               SqueezeQuackBehaviour(),
           FlyBehavior = new NoFlyBehaviour()
      );
}
```

❑

4.3. Kohäsion

4.3.1 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.

▸ **Erklärung: Kohäsion**

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

▸ **Auflistung: Arten der Kohäsion**

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

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

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

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

❑

4.3.2 Fallbeispiel: Servicekohäsion

```
//-----------------------------------------
//  Servicekohaesion - schwache Kohsion
//-----------------------------------------
class Vector implements Serializable{
   public int X { get; set; }
   public int Y { get; set; }

   public float Add(Vector v){
      this.X += v.X;
      this.Y += v.Y;
```

```
11
12        return Math.SQRT(X * X + Y * Y);
13     }
14
15  }
```

❑

.

5. Programmierung: SOLID

5.1. 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.

5.1.1 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.

Es folgt dabei dem Hollywoodprinzip: Don't call us, we call you.

5.2. 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**[18]
- Vererbung beschreibt damit eine **ist ein** Beziehung[19] zwischen Kindklasse und der entsprechenden Basisklasse.

5.2.1 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`[20] 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.

❑

5.3. 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.

5.3.1 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.

❑

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

5.4. 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.

5.4.1 Fallbeispiel: Open Closed Prinzip

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

public enum EAppleType{
   GOLDEN_LADY, ROSE
}

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

}

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

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

      return filteredApples;
   }
}
```

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

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

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

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

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

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

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

❑

5.5. Single Responsibility Prinzip

Single Responsibility Prinzip

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

Damit wird explizit die **starke Kohäsion** von Softwareelemnten gefordert.

> Folgt man dem objektorientiertem Paradigma, ist die Codebasis auf viele, von ihrem Codeumfang her, **kleine Klassen** aufgeteilt.

5.5.1 Verletzung des Single Responsibilty Prinzips

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.
- Damit wird ein System von Softwareelementen geschaffen die insgesamt start gekoppelt, sebst aber eine schwache Kohäsion haben.

❑

6. Programmierung: OOP Entwurf

04

OOP Entwurfsmuster

6.1. 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.

6.1.1 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

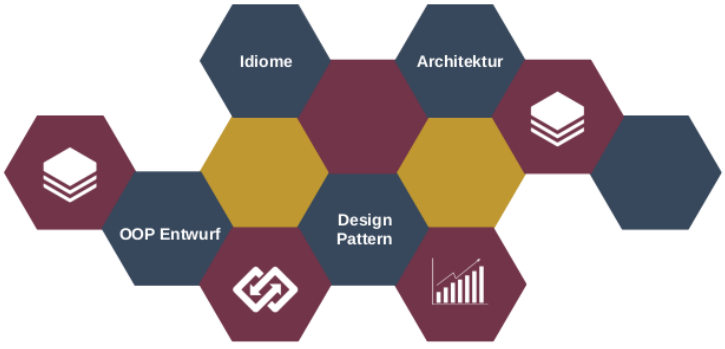
Entwurfsmuster beschreiben das **Zusammenspiel** von **Klassen**.

Architekturmuster

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

❑

Entwurfsmuster



6.1.2 Einsatz von Entwurfsmustern

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

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

▸ **Auflistung: Musterkategorien**

- **Erzeugungsmuster:** Erzeugungsmuster unterstützen das **Erzeugen** komplexer Objekte. Der Erzeugungsprozess für Objekte wird gekapselt.
  - Singelton
  - Factorymethod
- **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**

❑

6.2. Erzeugermuster

Erzeugermuster unterstützen das **Erzeugen** von komplexen Objekten. Der Erzeugungsprozess für Objekte kann gekapselt werden.

6.2.1 Erzeugermuster - Singleton

Singleton

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

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

▸ **Erklärung: Motivation und Kontext**

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

▸ **Codebeispiel: Fallbeispiel: Singleton**

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

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

public class Logger{
     // In der Klasse selbst wird eine
     // Instanz erzeugt und an ein Feld des
     // Klassenobjekts gebunden.
     private static Logger instance = new
          Logger();

     public const bool LOG_TO_CONSOLE = true;

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

     }

     // Fuer den globalen Zugriff wird eine
     // Klassenmethode geschrieben.
     public static final Logger getInstance(){
        return instance;
     }

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

}

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

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

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

❑

6.2.2 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** des Objekts in eine eigene Klasse ausgelagert.
- Dadurch kommt es zu einer **Entkoppelung** der Logik für die Objektverarbeitung und der **Objekterzeugung**.

Codebeispiel: Factory

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

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

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

public class RubberDuck : IQuackBehavior{
    public string Quack(){
        return "... squeeze";
    }
}

public class DuckDecoy : IQuackBehavior{
    public string Quack(){
        return "... QUACK QUACK";
    }
}
```

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

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

public class DecoratedDuckFacotry :
    IDuckFactory{

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

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

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

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

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

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

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

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

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

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

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

        factory = new DuckFactory();

        ducks.Clear();

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

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

❑

6.3. Strukturmuster

Strukturmuster beschreiben die **Struktur** komplexer Objekte zur Laufzeit.

6.3.1 Strukturmuster - Adapter

Adapter

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

Erklärung: Motivation und Kontext

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

Erklärung: Eigenschaften eines Adapters

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

Codebeispiel: Entwurfsmuster: Adapter

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

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

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

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

public class HonkAdapter : IQuackBehavior{

     private IHonkBehaviour _honkable;

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

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

}

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

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

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

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

> Ausgabe

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

❑

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

Codebeispiel: Entwurfsmuster: Dekorator

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

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

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

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

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

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

public class OutputDecorator :IQuackBehavior{
    private IQuackBehavior _quackable;

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

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

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

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

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

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

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

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

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

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

> Ausgabe:

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

❑

6.4. Verhaltensmuster

Mit **Verhaltensmustern** können komplexe **Interaktionen** zwischen Objekten modelliert werden.

6.4.1 Verhaltensmuster - Command

Command

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

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

▸ Codebeispiel: Command

```
//----------------------------------------
//  Schnittstelle: ICommand
//----------------------------------------
// ICommand deklariert die Schnittstelle
// der Befehlsobjekte. Ein Befehlsobjekt kann
// durch das Aufrufen der execute() Methode
// aufgerufen werden.
public interface ICommand{
   void execute();
   void undo();
}

//----------------------------------------
//  Klasse: ACommand
//----------------------------------------
public abstract class ACommand {

    protected Robot _robot;

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

    public abstract void Process();
    public abstract void Undo();

}

//----------------------------------------
//  Klasse: Point
//----------------------------------------
public class Point {
    private int _x;
    private int _y;
```

```
35
36      public int X {
37         get => _x;
38         set => _x = value;
39      }
40
41      public int Y {
42         get => _y;
43         set => _y = value;
44      }
45
46      public Point(int x, int y) {
47         _x = x;
48         _y = y;
49      }
50
51      public Point CalculateNeighbour(
52            EDirectionType direction
53      ) {
54         Point p = null;
55
56         switch (direction) {
57            case EDirectionType.NORTH:
58               p = new Point(
59                     this._x, this._y + 1
60               );
61               break;
62
63            case EDirectionType.SOUTH:
64               p = new Point(
65                     this._x, this._y -1
66               );
67               break;
68
69            case EDirectionType.WEST:
70               p = new Point(
71                     this._x -1, this._y
72               );
73               break;
74
75            case EDirectionType.EAST:
76               p = new Point(
77                     this._x + 1, this._y
78               );
79               break;
80         }
81
82         return p;
83      }
84   }
```

```
1   //----------------------------------------
2   //  Klasse: Robot
3   //----------------------------------------
4   public class Robot {
5
6      private Point _location = null;
7
8      public Point Location {
9         get => _location;
10        set => _location = value;
11     }
12  }
13
14  //----------------------------------------
15  //  Klasse: MoveUpCommand
16  //----------------------------------------
17  public class MoveUpCommand : ACommand{
18     public MoveUpCommand(Robot robot) :
          base(robot) { }
19
20     public override void Process() {
21        _robot.Location =
22           _robot.Location.CalculateNeighbour(
23                 EDirectionType.NORTH);
24     }
25
26     public override void Undo() {
27        _robot.Location =
28           _robot.Location.CalculateNeighbour(
29                 EDirectionType.SOUTH);
30     }
31  }
32
33  //----------------------------------------
34  //  Klasse: MoveDownCommand
35  //----------------------------------------
36  public class MoveDownCommand : ACommand{
37      public MoveDownCommand(Robot robot) :
            base(robot) { }
38
39      public override void Process() {
40         _robot.Location =
41            _robot.Location.CalculateNeighbour(
42               EDirectionType.SOUTH);
43         }
44
45      public override void Undo() {
46          _robot.Location =
47             _robot.Location.CalculateNeighbour(
48                EDirectionType.NORTH);
49      }
50  }
```

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

    private Stack<ACommand> _commands = new
        Stack<ACommand>();
    private Stack<ACommand> _history = new
        Stack<ACommand>();

    private Robot _robot;

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

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

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

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

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

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

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

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

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

    public bool Do() {
        if (_history.Count == 0)
            return false;

        ACommand command = _history.Pop();
        command.Process();

        _commands.Push(command);

        return true;
    }

    public bool Redo() {
       if (_commands.Count == 0)
           return false;

       ACommand command = _commands.Pop();
       command.Undo();

       _history.Push(command);
       return true;
    }
}
```

❑

6.4.2 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**

```
//----------------------------------------
// Schnittstelle: IPaymentStrategy
//----------------------------------------
public interface IPaymentStrategy{
```

```
    public void pay(int amount);
}
```

```
//----------------------------------------
//  Klasse: CreditCardStrategy
//----------------------------------------
public class CreditCardStrategy :
    IPaymentStrategy{

   private CreditCardProcessor processor =
       new CreditCardProcessor();

   private string _cardNumber;
   private string _name;

   public CreditCardStrategy(string name,
       string cardNumber){
      this._name = name;
      this._cardNumber = cardNumber;
   }

   public void pay(int amount){
       processor.process(
           _name, _cardNumber, amount
       );
   }
}

//----------------------------------------
//  Klasse: CreditCardStrategy
//----------------------------------------
public class PaypalStrategy :
    IPaymentStrategy{

    private PaypalProcessor processor = new
        PaypalProcessor();

    private string _email;
    private string _pwd;

    public PaypalStrategy(String email,
        String pwd){
       this._email = email;
       this._pwd = pwd;
    }

    public void pay(int amount){
       processor.process(
           _email,
           _pwd
       );
   }
}
```

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

   private string _upcCode;
   private string _price;

   public UpcCode{
      get => _upcCode;
   }

   public Price{
      get => _price;
   }

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

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

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

   private IPaymentStrategy _paymentMethod;

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

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

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

      return sum;
   }

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

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

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

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

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

       cart.Pay();
    }

}
```

❑

.

.