

# 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. Programmierung: Strukturierung</b>	4	2.2.4. Fallbeispiel: Auflösen von Interaktionskoppelung	14
1.1. Unterprogramme	4	2.2.5. Vererbungskoppelung	15
1.1.1. Unterprogramme	4	2.2.6. Objektkomposition	15
1.2. Objektorientierung	5	2.3. Kohäsion	16
1.2.1. Objektorientierung	5	2.3.1. Kohäsion	16
1.3. Schichtenmodell	5	2.3.2. Fallbeispiel: Servicekohäsion	16
1.3.1. Prinzipien des Schichtenmodells	5	<b>Index</b>	17
1.3.2. Fallbeispiel: Schichtenmodell	6		
1.4. Komponenten	8		
1.4.1. Fallbeispiel: Restaurantverwaltung	8		
1.5. Service	9		
1.5.1. Zusammenfassung	9		
<b>2. Programmierung: Metriken</b>	12		
2.1. Softwaremetriken	12		
2.1.1. Metriken	12		
2.1.2. Qualitätsmetriken	12		
2.2. Koppelung	13		
2.2.1. Koppelung	13		
2.2.2. Interaktionskoppelung	13		
2.2.3. Auflösen von Interaktionskoppelung	14		

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



# Grundlagen der objektorientierten Programmierung

December 14, 2019

## 1. Programmierung: Strukturierung

# 01

### Strukturierung von Programmen

01. Unterprogramme	4
02. Objektorientierung	5
03. Schichtenmodell	5
04. Komponenten	8
05. Service	9

## 1.1. Unterprogramme

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 war die **Anweisung**.

### 1.1.1 Unterprogramme

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

#### ► Erklärung: Unterprogramme ▼

- **Unterprogramme**<sup>2</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>3</sup> der Anwendung gesunken wäre.
- Als nächstes wurden **Container** für Daten<sup>4</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>1</sup> Unter einem Betriebssystemprozess verstehen wir ein sich in Ausführung befindendes Programm

<sup>2</sup> Funktionen, Prozeduren

<sup>3</sup> Codeerwartbarkeit, Codelesbarkeit, Anpassbarkeit

<sup>4</sup> Die Sprache C spiegelt diesen Entwicklungsstand wider: sie bietet Unterprogramme (Prozeduren und Funktionen) sowie Strukturen zur Strukturierung

## 1.2. Objektorientierung

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

### 1.2.1 Objektorientierung

Objektorientierung faßt Strukturen und Unterprogramme zu **Klassen**<sup>5</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>6</sup> zur Verfügung, die den **Zustand**<sup>7</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>5</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>6</sup> Methoden

<sup>7</sup> Variablen

## 1.3. 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.

### 1.3.1 Prinzipien des Schichtenmodells

► Prinzip: **Schichtenmodell** ▼

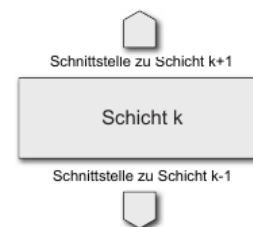
- **Teile und Herrsche**: Ein komplexes Problem wird in **unabhängige Teilprobleme** zerlegt, das jedes für sich, einfacher handhabbar ist, als das Gesamtproblem.

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

- **Unabhängigkeit**: Die einzelnen Schichten der Anwendung kommunizieren miteinander, indem die **Schnittstellenspezifikation**<sup>8</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.



- **Standardisierung**: Die Gliederung des Gesamtproblems in einzelne Schichten erleichtert die Entwicklung von **Standards**<sup>9</sup> für die einzelnen Schichten.

□

<sup>8</sup> Schnittstelle, Interface

<sup>9</sup> HTTP, FTP, usw.

### 1.3.2 Fallbeispiel: Schichtenmodell

#### ► Schnittstellenspezifikation: Modellschicht ▼

```

1 //-----
2 // AosDbContext.cs
3 //-----
4 public class AosDbContext : DbContext {
5
6     public DbSet<Trait> Traits { get; set; }
7     public DbSet<TraitItem> TItems {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         builder.Entity<TraitItem>()
45             .HasOne(ti => ti.Trait)
46             .WithMany()
47             .HasForeignKey(ti => ti.TraitId);
48     }
49 }

```

#### ► 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
32         context) {
33         Context = context;
34         Table = context.Set<TEntity>();
35     }
36
37     public TEntity Create(TEntity t) {
38         Table.Add(t);
39         Context.SaveChanges();
40
41         return t;
42     }
43
44     public List<TEntity>
45         CreateRange(List<TEntity> list) {
46         Table.AddRange(list);
47         Context.SaveChanges();
48
49         return list;
50     }
51
52     public void Update(TEntity t) {

```

```

45     Context.ChangeTracker.Clear();
46
47     Table.Update(t);
48     Context.SaveChanges();
49 }
50
51 public void UpdateRange(List<TEntity>
52     list) {
53     Table.UpdateRange(list);
54     Context.SaveChanges();
55 }
56
57 public TEntity? Read(int id) =>
58     Table.Find(id);
59
60 public List<TEntity>
61     Read(Expression<Func<TEntity, bool>>
62         filter) =>
63         Table.Where(filter).ToList();
64
65 public List<TEntity> Read(int start, int
66     count) =>
67     Table.Skip(start)
68         .Take(count)
69         .ToList();
70
71 public List<TEntity> ReadAll() =>
72     Table.ToList();
73
74 public void Delete(TEntity t) {
75     Table.Remove(t);
76     Context.SaveChanges();
77 }
78 }
79
80 ▶ Schnittstellenspezifikation: Serviceschicht ▼
81 //-----
82 // AController.cs
83 //-----
84 public class AController<TEntity> :
85     ControllerBase where TEntity : class {
86
87     private IRepository<TEntity> _repository;
88
89     private ILogger<AController<TEntity>>
90         _logger;
91
92     public AController(
93         IRepository<TEntity> repository,
94         ILogger<AController<TEntity>> logger
95     ) {
96         _repository = repository;

```

```

15     _logger = logger;
16 }
17
18 [HttpPost]
19 public async Task<ActionResult<TEntity>>
20     Create(TEntity t) {
21     await _repository.CreateAsync(t);
22     _logger.LogInformation($"Created
23         entity with id: {t}");
24
25     return t;
26 }
27
28 [HttpGet("{id:int}")]
29 public async Task<ActionResult<TEntity>>
30     Read(int id) {
31     var data = await
32         _repository.ReadAsync(id);
33
34     if (data is null) return NotFound();
35     _logger.LogInformation($"reading
36         entity with id {id}");
37
38     return Ok(data);
39 }
40
41 [HttpGet]
42 public async
43     Task<ActionResult<List<TEntity>>>
44     ReadAll(int start, int count) =>
45     Ok(await
46         _repository.ReadAllAsync(start,
47         count));
48
49 [HttpPut("{id:int}")]
50 public async Task<ActionResult>
51     Update(int id, TEntity entity) {
52     var data = await
53         _repository.ReadAsync(id);
54
55     if (data is null) return NotFound();
56
57     await _repository.UpdateAsync(entity);
58     _logger.LogInformation($"updated
59         entity: {entity}");
60     return NoContent();
61 }
62 }

```

□

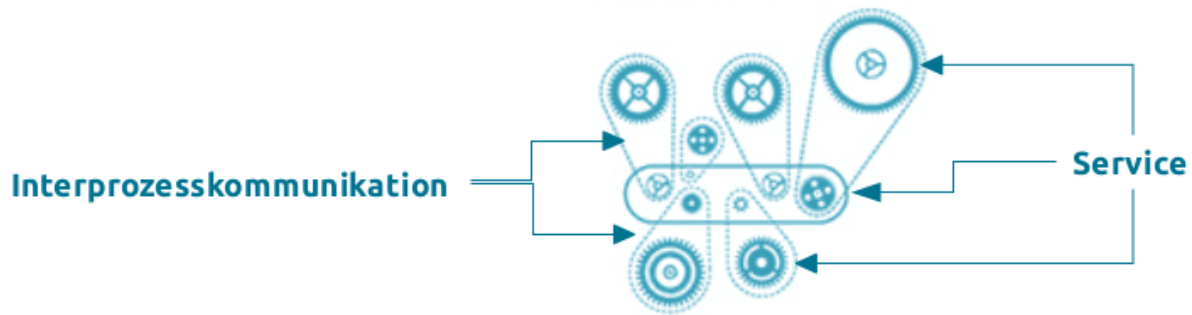


Abbildung 1. SOA - Zusammenspiel von Services

## 1.4. 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.



### 1.4.1 Fallbeispiel: Restaurantverwaltung

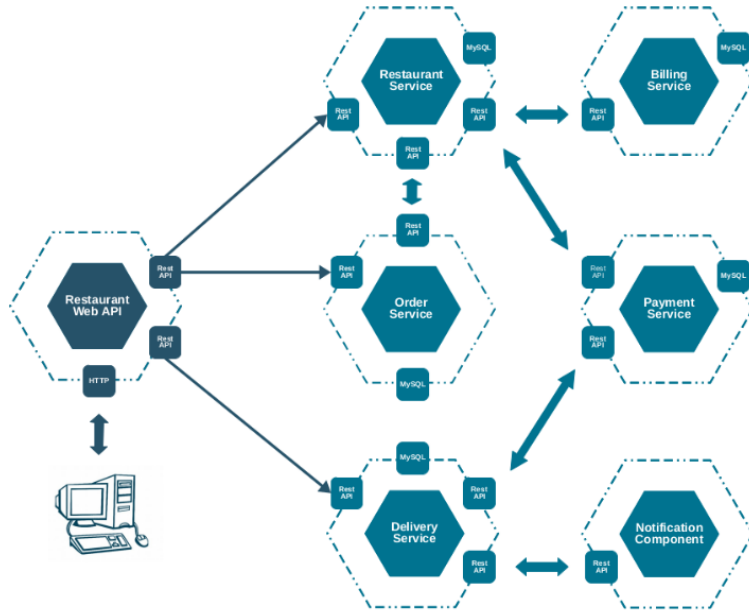
#### ► Fallbeispiel: Restaurantverwaltungssoftware ▼

- 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 durchzuführen zu können.
- Die einzelnen Komponenten können nun **unabhängig** voneinander entwickelt werden.





## Microservice - Architektur



### 1.5. 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.



#### 1.5.1 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.



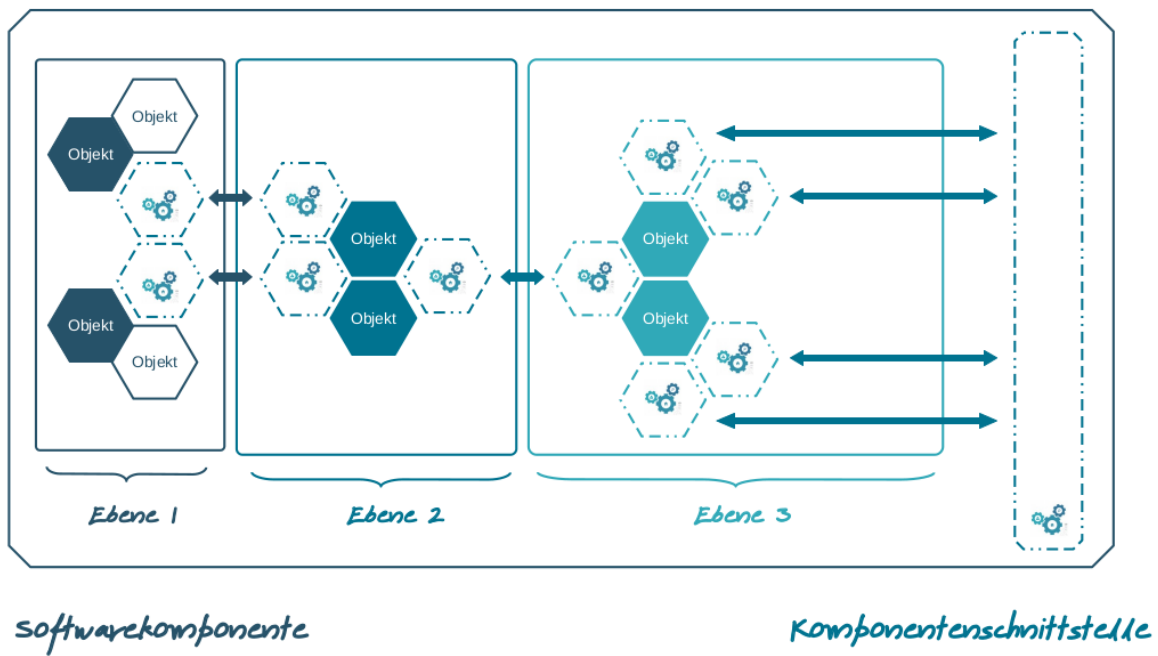


Abbildung 2. Strukturierung einer Komponente



## 2. Programmierung: Metriken

# 02

Programmierung: Metriken

01. Softwaremetriken	12
02. Koppelung	13
03. Kohäsion	16

## 2.1. Softwaremetriken

### 2.1.1 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.



### 2.1.2 Qualitätsmetriken

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

#### ► Auflistung: Softwaremetriken ▾

- **Koppelung**: Maß der **Abhängigkeiten** zwischen **Softwareelementen**<sup>10</sup>.
- **Kohäsion**: Maß des inneren **Zusammenhalt** eines **Softwareelements**.



<sup>10</sup> Objekte, Schichten, Komponenten

# Softwaremetriken



## 2.2. Koppelung

### 2.2.1 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**<sup>11</sup>, das Objekte einer Klasse von Objekten anderer Klassen in Anspruch nehmen.
- **Vererbungskoppelung:** Vererbungskoppelung beschreibt das **Ausmaß der Abhängigkeit** zwischen erbender und Basisklasse.



### 2.2.2 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 =>
8          Dice.GetInstance().Roll() <=
9              AttackValue;
10 }
11
12 public class Spearman {
13     public int AttackValue { get; set; }
14
15     public bool Attack =>
16         Dice.GetInstance().Roll() <=
17             AttackValue;
18 }
19
20 public class Bowman {
21     public int AttackValue { get; set; }
22
23     public bool Attack =>
24         Dice.GetInstance().Roll() <=
25             AttackValue;
26 }

```

<sup>11</sup> Methodenaufruf

### ▸ Codebeispiel: Interaktionskoppelung ▼

```

1 //-----
2 // Interaktionskoppelung
3 //-----
4 public class GameController {
5
6     public int DetermineHits() {
7         int attackCount = 0;
8
9         var unit1 = new Swordsman(){
10             AttackValue = 5
11         };
12         var unit2 = new Spearman(){
13             AttackValue = 4
14         };
15         var unit3 = new Bowman(){
16             AttackValue = 6
17         };
18
19         // Interaktionskoppelung
20         if(unit1.Attack()) {
21             ++attackCount;
22         }
23
24         if(unit2.Attack()) {
25             ++attackCount;
26         }
27
28         if(unit3.Attack()) {
29             ++attackCount;
30         }
31
32         return attackCount
33     }
34 }

```



### 2.2.3 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.



### 2.2.4 Fallbeispiel: Auflösen von Interaktionskoppelung ■

```

1 //-----
2 // Entkoppelter Code
3 //-----
4 // Schnittstellendefinition
5 public interface IUnit {
6     bool Attack ();
7 }
8
9 // Klassenimplementierung
10 public abstract class AUnit : IUnit {
11     public int AttackValue { get; set; }
12
13     public AUnit (int attackValue) {
14         AttackValue = attackValue;
15     }
16
17     public bool Attack() =>
18         (Dice.GetInstance().Roll() <=
19             AttackValue);
20 }
21
22 public class Swordsman : AUnit {
23     public Spearman : base(5){};
24 }
25
26 public class Spearman : AUnit {
27     public Spearman : base(4){};
28 }
29
30 public class Bowman : AUnit {
31     public Spearman : base(6){};
32 }
33
34 public class GameController {
35     public int DetermineHits (List<IUnit>
36         army) => army.Aggregate (
37         0,
38         (total, unit) => unit.Attack() ?
39             total++ : total
40     );
41 }
42
43 // Ausfuehrung
44 var army = new List(){
45     new Swordsman(), new Bowmen(), new Bowmen()
46 };
47
48 var hits = new GameController().Attack(army);

```



## 2.2.5 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 ▼

```
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 }
```



## 2.2.6 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 :
    IFlyable {
2     public String Fly() => "flying high in the
        sky";
3 }
4
5 public class NoFlyBehaviour : IFlyable {
6     public String Fly() => "can't fly";
7 }
8
9 public class Duck{
10     public IQuackable QuackBehaviour {
11         get; set;
12     }
13
14     private IFlyable FlyBehaviour {
15         get; set;
16     }
17 }
18
19 public class DuckFactory{
20     public static Duck CreateRedheadDuck() =>
21         new Duck(){
22             QuackBehaviour = new
23                 LoudQuackBehaviour(),
24             FlyBehaviour = new
25                 DefaultFlyingBehaviour()
26         };
27
28     public static Duck CreateEntlingDuck() =>
29         new Duck() {
30             QuackBehavior = new
31                 ProudQuackBehaviour(),
32             FlyBehavoiur = new
33                 DefaultFlyingBehaviour()
34         };
35
36     public static Duck RubberDuck () =>
37         new Duck() {
38             QuackBehavior = new
39                 SqueezeQuackBehaviour(),
40             FlyBehavior = new NoFlyBehaviour()
41         };
42 }

```

## 2.3. Kohäsion

### 2.3.1 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 ungenutzte Attribute bzw. Methoden für die Klasse definiert werden.



### 2.3.2 Fallbeispiel: Servicekohäsion

```

1 //-----
2 // Servicekohaesion - schwache Kohsion
3 //-----
4 class Vector implements Serializable{
5     public int X { get; set; }
6     public int Y { get; set; }
7
8     public float Add(Vector v){
9         this.X += v.X;
10        this.Y += v.Y;

```



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



## Index

Architekturmuster, 54

Entwurfsmuster, 54

Idiom, 54

Singleton, 55