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. Programmierung: Strukturierung

01

Sturkturierung von Programmen

1.1. Unterprogramme

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 war die **Anweisung**.

1.1.1 Unterprogramme

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

▸ **Erklärung: Unterprogramme**

- **Unterprogramme**[2] 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**[3] der Anwendung gesunken wäre.
- Als nächstes wurden **Container** für Daten[4] 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 }
```

❑

[1] *Unter einem Betriebssystemprozess verstehen wir ein sich in Ausführung befindendes Programm*
[2] *Funktionen, Prozeduren*
[3] *Codeerwartbarkeit, Codelesbarkeit, Anpassbarkeit*
[4] *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**[5] 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**[6] zur Verfügung, die den **Zustand**[7] von Instanzen der Klasse verändert und verarbeitet.
- Variablen und Methoden stehen im kontinuierlichen Zusammenspiel.

▸ **Codebeispiel: Klassen**

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

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

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

    public Project() : base () {

    }
}
```

❑

[5] *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.*

[6] *Methoden*

[7] *Variblen*

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**

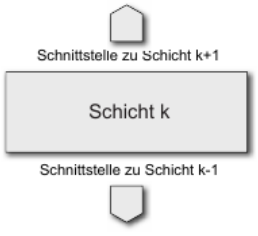
- **Teile und Herrsche:** Ein komplexes Problem wird in **unabhängige Teilprobleme** zerlegt, das jedes für sich, einfacher handhabbar ist, als das Gesamtproblem.

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

- **Unabhängigkeit:** Die einzelnen Schichten der Anwendung kommunizieren miteinander, indem die **Schnittstellenspezifikation**[8] 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**[9] für die einzelnen Schichten.

❑

[8] *Schnittstelle, Interface*

[9] *HTTP, FTP, usw.*

1.3.2 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();
    }
}
```

❑

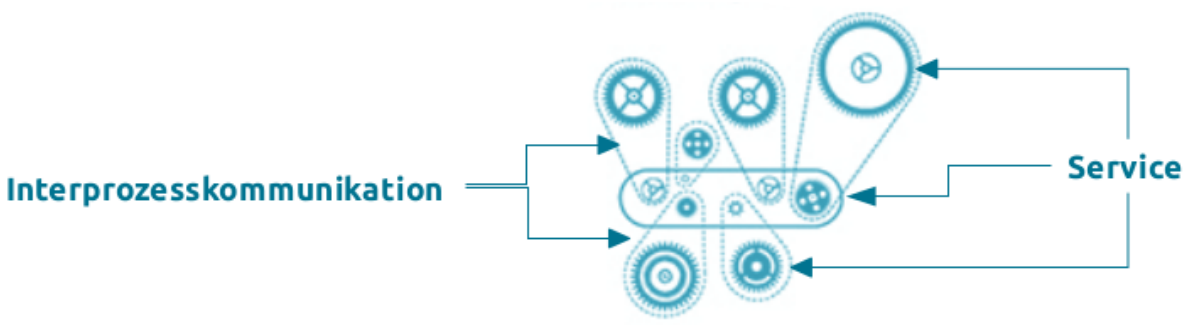


Abbildung 1. **SOA - Zusammenspiel von Services**

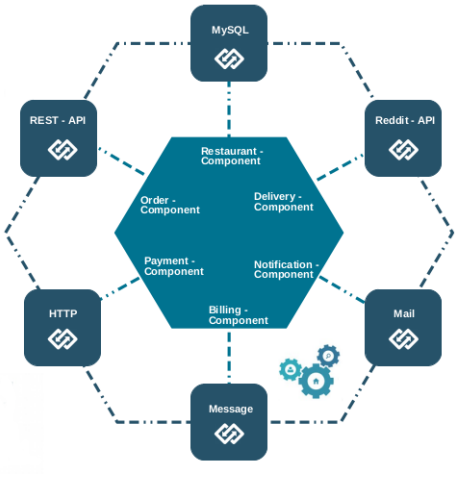
1.4. Komponenten

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.



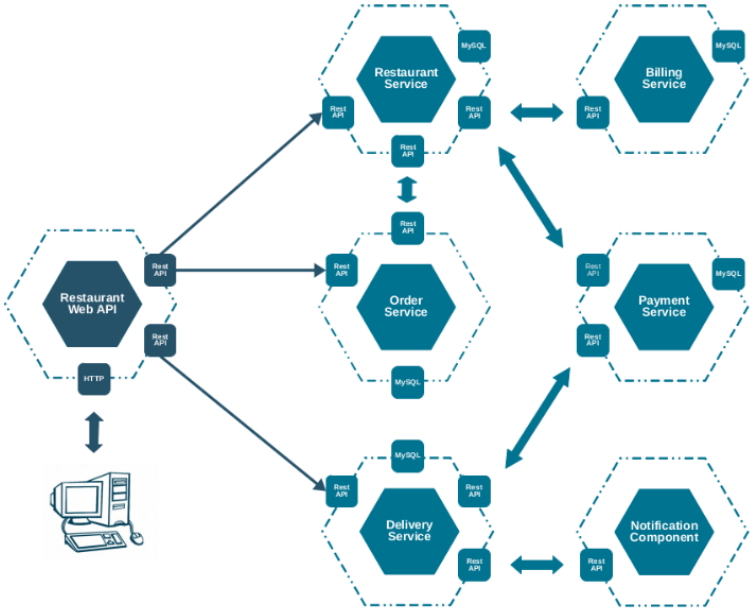
1.4.1 Fallbeispiel: Restaurantverwaltung

▸ **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 druchführen 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.
- **Komponeten** bestehen aus **Schichten**. Schichten bestehen aus **Klassen**.
- **Klassen** werden durch **Methoden** strukturiert.

❑

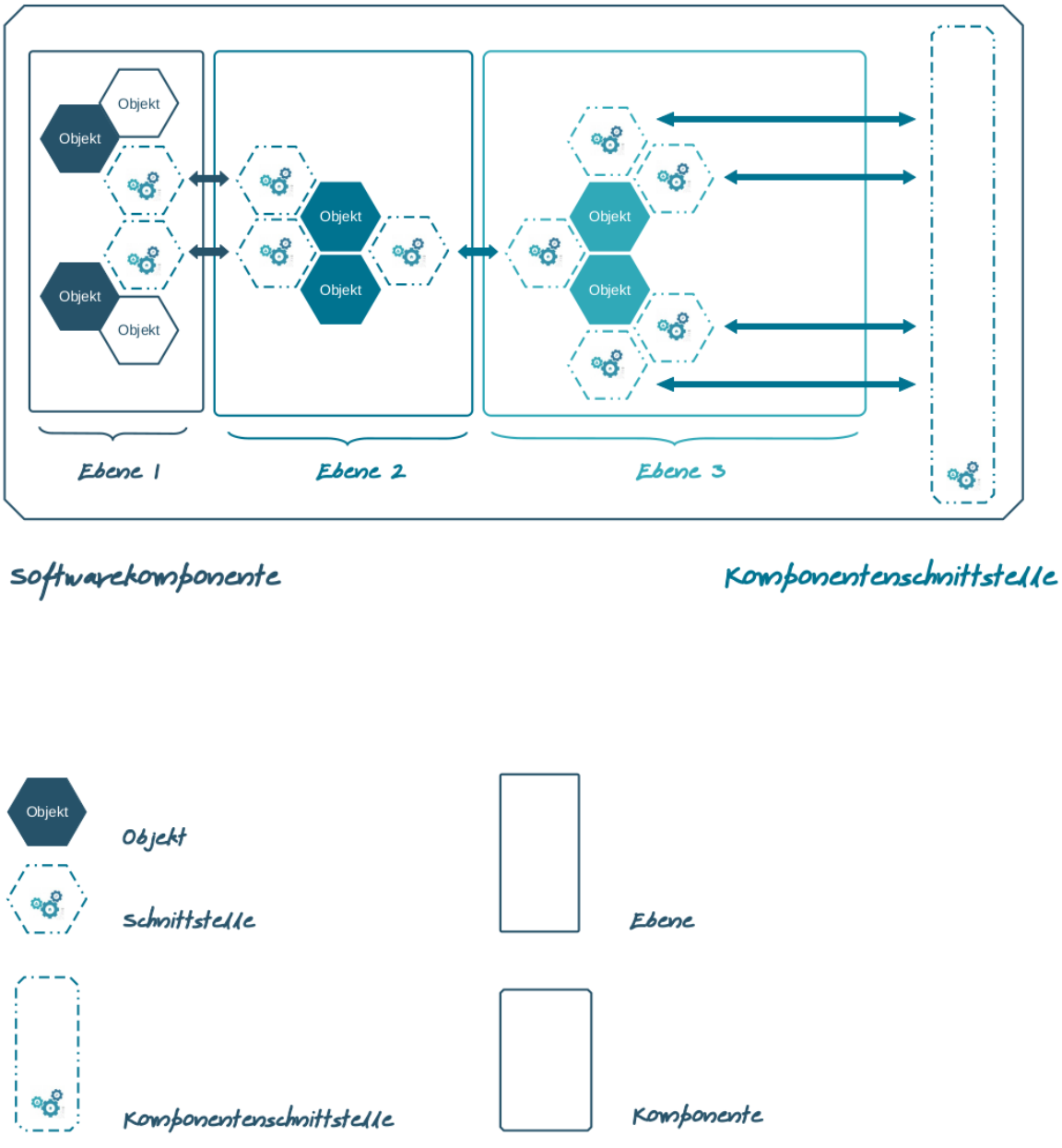


Abbildung 2. **Strukturierung einer Komponente**

.

.

2. Programmierung: Metriken

02

Programmierung: Metriken

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: 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.

❑

2.1.2 Qualitätsmetriken

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

Auflistung: Softwaremetriken

- **Koppelung:** Maß der **Abhängigkeiten** zwischen Softwareelementen[10].
- **Kohäsion:** Maß des inneren **Zusammenhalt** eines Softwareelements.

❑

[10] *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**[11], 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 =>
          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 }
```

[11] *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
   }
}
```

❑

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

```
//----------------------------------------
//  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);
```

❑

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 **einzubetten** z.B. durch Aggregation oder Referenzierung.

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

Erklärung: Vorteile der Objektkomposition

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

Codebeispiel: Objektkomposition

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

```
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()
    );
}
```

❑

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 ungenutze Attribute bzw. Methoden für die Klasse definiert werden.

❑

2.3.2 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  }
```

❑

.

3. Programmierung: SOLID

3.1. SOLID Prinzipien

3.1.1 SOLID Prinzipien

Um die Programmierung hochwertigen Codes zu erleichtern, wurden **Prinzipien** für die **Softwareentwicklung** formuliert. Prinzipien objektorienterten Designs sind Prinzipien, die zu gutem objekorientierten Design führen.

> Die SOLID Prinzipien sind eine Sammlung von objektorientierten **Programmierprinzipien**.

▸ **Analyse: Objektorientiertes Design**

- Gutes objektorientiertes Design führt zu gut lesbarem und wartbarem Code.
- Damit wird es für mehrere Entwickler leichter gleichzeitig an der Codebasis zu arbeiten.
- Objektorientiertes Design resultiert in schwacher **Koppelung** bei gleichzeitiger starker **Kohäsion** der Softwareelemente.

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

▸ **Auflistung: SOLID Prinzipien**

Single Responsibility Prinzip

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

Open Closed Prinzip

Softwaresysteme müssen stets **erweiterbar** sein. Wird ein System erweitert, darf bestehender 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.

❑

3.1.2 Konsequenzen schlechten Codes

Die Programmierung einer Softwareanwendung ist nur ein kleiner Teil der Softwareentwicklung. Etwa 70% der Tätigkeit der Softwareentwicklung fallen in den Bereich der **Softwarewartung**[12].

> Die Wartbarkeit einer Softwareanwendung ist damit die wichtigste Metrik, zur Bestimmung der **Lebensdauer** von Softwaresystemen.

Analyse: Konsequenzen schlechten Codes

- Hohe Kosten im Rahmen der **Weiterentwicklung** der Anwendung.
- Erschwerte **Codedokumentation** und **Codelesbarkeit**.
- Schlechte **Wartbarkeit** der Anwendung.

❑

[12] *In der Softwareentwicklung bezeichnet der Begriff der Softwarewartung die Änderung einer Softwareanwendung nach dessen Auslieferung.*

3.2. 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.

3.2.1 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.

❑

3.3. 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.

3.3.1 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;
   }
}
```

❑

3.4. 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**[13]
- Vererbung beschreibt damit eine **ist ein** Beziehung[14] zwischen Kindklasse und der entsprechenden Basisklasse.

3.4.1 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`[15] 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.

❑

3.5. 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.

3.5.1 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.

❑

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

.