SOFTWAREDESIGN

Brandenburgische Technische Universität Cottbus - Senftenberg

AGENDA

- Theorie
  - Definition & Aufgaben vom Softwaredesign
  - Verwendung von OOD & OOP
  - UML Klassendiagramm – Notation & Einschränkungen
  - Entwurfsprinzipien
  - Definition und Kategorisierung von Entwurfsmustern

- Praxis (Java-Programmierung)
  - Klassen- bzw. Methodenzugriff und Objektreferenzierung
  - Verwendung von Interfaces und abstrakten Klassen
  - Entwurfsmuster
    - Erzeugungsmuster
    - Verhaltensmuster
    - Strukturmuster

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

EINE DEFINITION

> Der Begriff Softwareentwurf bezeichnet eine Tätigkeit im Rahmen der Entwicklung eines Softwaresystems, die auf den Ergebnissen der Anforderungsdefinition[…] aufsetzt und sich als eigene Phase – oder aufgeteilt auf verschiedene, spezielle Entwurfsphasen (z. B. Architekturentwurf oder Komponentenentwurf) – in typischen Vorgehensmodellen der Systementwicklung wiederfindet.
>
> *Klaus Turowski, Wirtschaftsinformatik I*
> *Otto-von-Guericke-Universität Magdeburg*

In Anlehnung an [IEEE 1990, S.25] versteht man unter dem Softwareentwurf den

> Prozess der Definition der Architektur, Komponenten, Schnittstellen und anderer Merkmale eines Software(teil)systems.

Was beinhaltet die Phase Softwareentwurf ?

Peter Geßler
Peter.Gessler@b-tu.de

AUFGABEN IM SOFTWAREENTWURF

- Festlegung primärer struktureller Eigenschaften
  - Identifikation der Architekturbausteine (Module, Klassen, SW-Komponenten)
  - Architekturbausteine benennen und Randbedingungen festlegen

- Beschreibung der Außensicht von Teilsystemen
  - Festlegen von Beziehungen untereinander
  - Schnittstellen spezifizieren

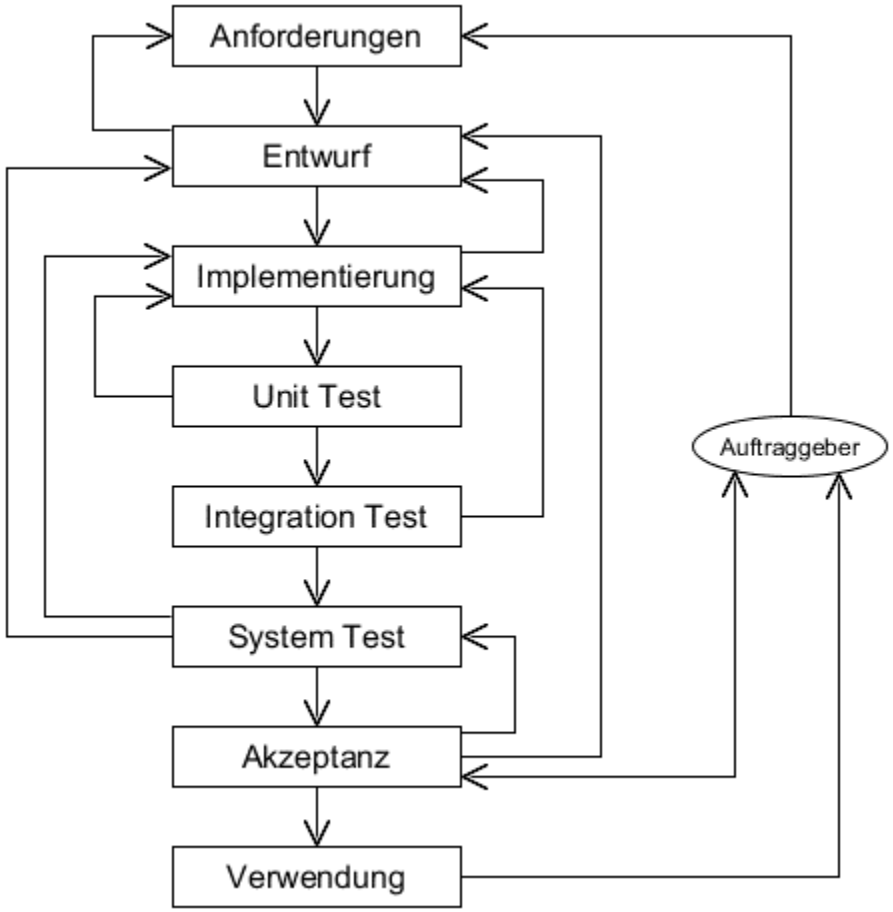
- Erstellung von Entwurfsdokumenten
  - Konzeptueller Entwurf → Repräsentation von Teilsystemen und Eigenschaften
  - High Level Entwurf → Interaktion zwischen Komponenten eines Teilsystems
  - Detail Entwurf → Beschreibung der Funktionalität einer Komponente

Ziel des Softwareentwurfs ist es, die Vorgaben der Anforderungsdefinition möglichst umfassend zu berücksichtigen, was letztlich auf die Lösung einer multikriteriellen Optimierungsaufgabe hinausläuft.

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

SOFTWAREENTWICKLUNGSPROZESS



Plan

Design

Implementation

Compile, Test & Debug

Postmortem

Support

Abb.: 1;
In Anlehnung an Watts S. Humphrey, A Discipline for Software Engineering

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

SOFTWAREENTWICKLUNGSPROZESS

Wo können wir uns Arbeit beim zweiten Durchlauf sparen?

Design

- strukturiert & kategorisiert
- Einhaltung der Entwurfsprinzipien
- Verwendung von Entwurfsmustern
- Dokumentation

Test & Integration

- automatisiertes Testen

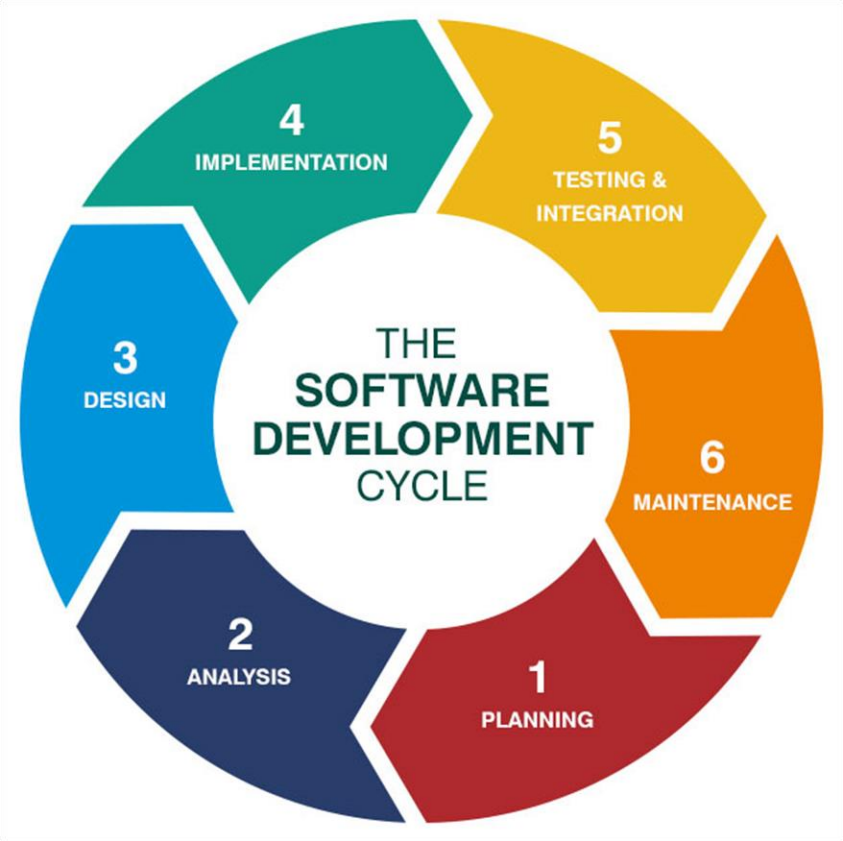


Abb.: 2
www.online.husson.edu – The Software Development Cycle

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

OBJEKTORIENTIERTER SOFTWARE-ENTWURF

Der objektorientierte Softwareentwurf stellt eine in Bezug auf das zur Strukturierung eingesetzte Paradigma spezielle Form des Softwareentwurfs dar, bei dem ein Softwaresystem in Klassen […] zerlegt wird. Der objektorientierte Softwareentwurf umfasst die Definition der Softwarearchitektur, der Klassen und der Vererbungshierachien […].

Sven Overhage, Wirtschaftsinformatik
Otto-Friedrich-Universität Bamberg

Teilaufgaben des
objektorientierten Softwareentwurfs

- Identifikation von Objekten (Klassen)
- Beschreibung dieser und ihrer Fähigkeiten
- Verknüpfung der Klassen zum Softwaresystem

Bei der Entwicklung großer Softwaresysteme wird von der Anwendung des objektorientierten Softwareentwurfs bisweilen abgeraten!

Szyperski, S.115-122, Component Software – Beyond Object-Oriented Programming

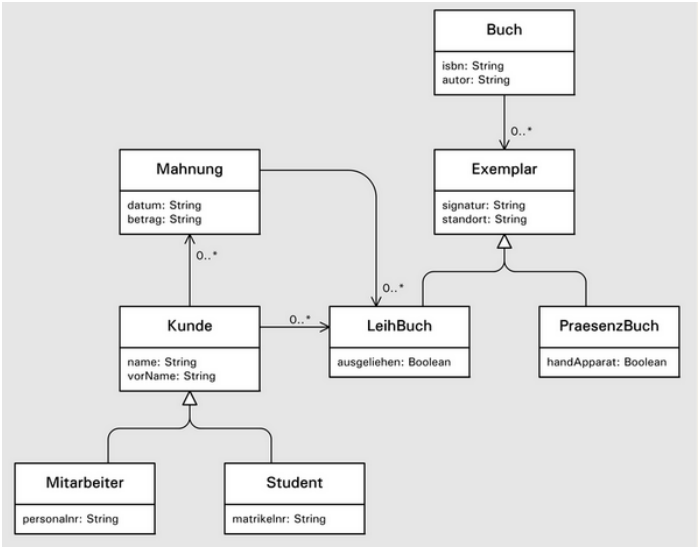


Abb.: 3; Bibliothekssystem (vereinfacht), Enzyklopädie der Wirtschaftsinformatik

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

PROGRAMMIERPARADIGMA OOP

- Erstellung von unabhängigen Komponenten (Objekte)
- Modellbaukastenprinzip
- Zusammensetzung von komplexeren Objekten durch einfache Objekte
- Vertreter: C++, C#, Java, etc.

Vorteile:

- Komplexität eines Objekts ist durch klare Strukturierung beherrschbar
- Komponenten sind größtenteils unabhängig voneinander
- Erweiterung der Funktionalität einer Komponente mittels Vererbung

Peter Geßler
Peter.Gessler@b-tu.de

WAS IST EINE SOFTWARE-ARCHITEKTUR?

Die Architektur eines Systems beschreibt ein System im Sinne einer Konstruktionszeichnung oder eines Bauplans.

Die Architektur eines Softwaresystems besteht aus (abstrakten) Komponenten und Schnittstellen, die durch eine konkrete Implementierung zu einem realen System wird.

Buch
Effektive Softwarearchitekturen,
Gernot Starke

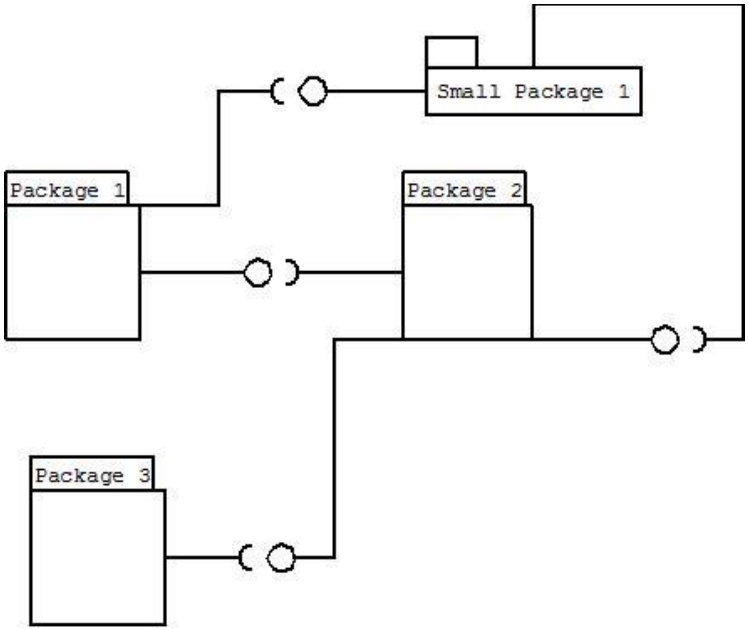


Abb.:4
Realisierung einer Software-Architektur mit Lollipop Notaion zur Darstellung der Schnittstellen.

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

ZIELE EINER SOFTWARE-ARCHITEKTUR

- Hilfe beim Projektmanagement
- Produktanalyse
- Visualisiert wiederverwendbare Komponenten
- Referenz für Konstruktion & Realisierung
- vereinfacht Pflege- und Weiterentwicklung
- Generierung von statischem Programmcode (bei entsprechendem Tool)

Peter Geßler
Peter.Gessler@b-tu.de

SICHTEN INNERHALB EINER ARCHITEKTUR

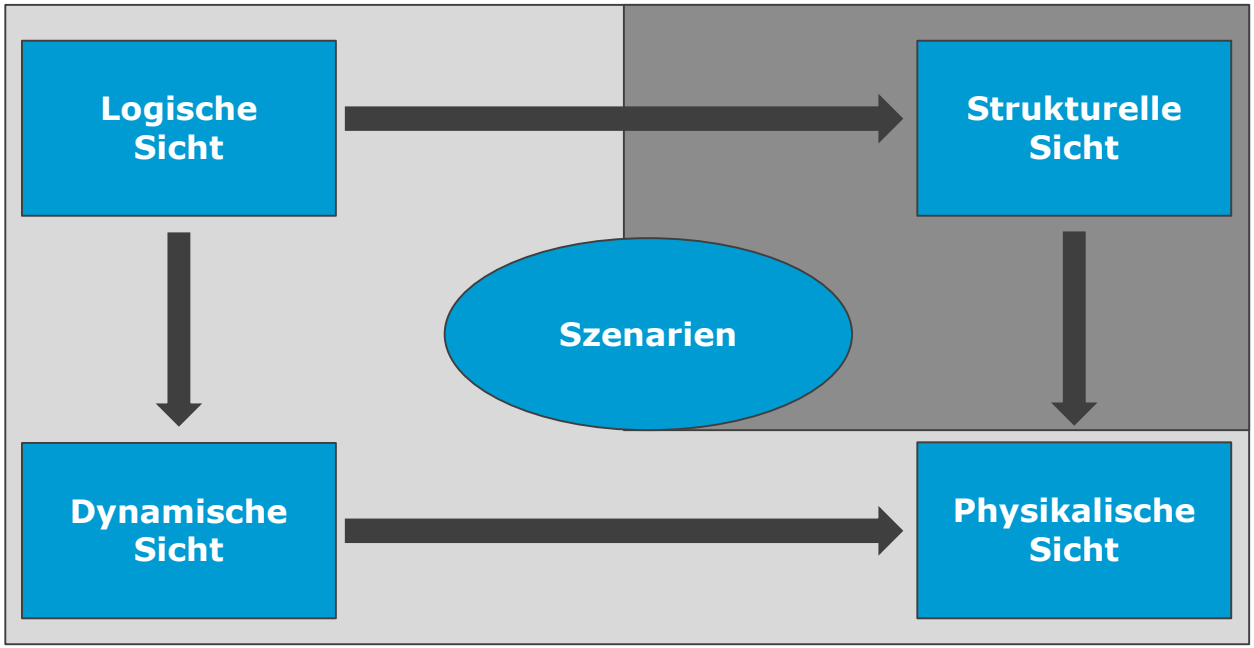


Abb.: 5
Philippe Kruchten, The 4 +1 view model of architecture, 1995

Peter Geßler
Peter.Gessler@b-tu.de

SICHTEN INNERHALB EINER ARCHITEKTUR

Logische Sicht

Aufgabe:
Beschreibung der Systemfunktionalität

Modell:
objektorientiertes Produktmodell

Zielgruppe:
Architekt, Softwareentwickler und Programmierer

Notation:
Klassen-, Sequenzdiagramme und Kommunikationsdiagramme

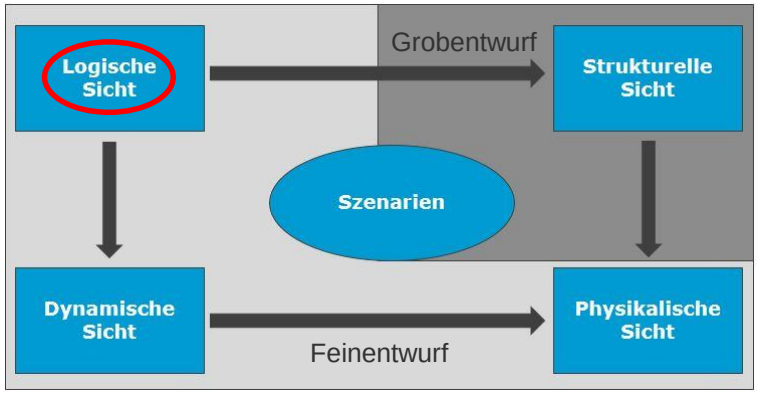


Abb.: 6
Philippe Kruchten, The 4 +1 view model of architecture, 1995

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

SICHTEN INNERHALB EINER ARCHITEKTUR

Strukturelle Sicht

Aufgabe:
Beschreibung der statischen Struktur

Modell:
Modulmodell

Zielgruppe:
Auftraggeber, Architekt und Softwareentwickler

Notation:
Paket- Komponentendiagramme, auch Blockdiagramme

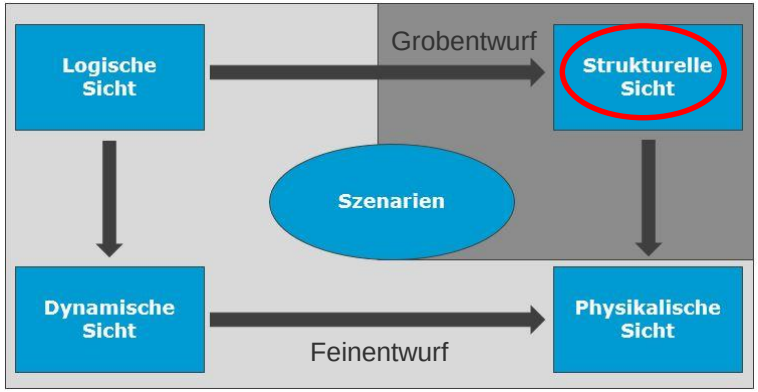


Abb.: 7
Philippe Kruchten, The 4 +1 view model of architecture, 1995

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

SICHTEN INNERHALB EINER ARCHITEKTUR

Physikalische Sicht

Aufgabe:
Software auf Hardware Komponenten abbilden

Modell:
keine allgemeinen Modelle

Zielgruppe:
Systemarchitekt, Softwareentwickler, Mitarbeiter in Wartung und Betrieb

Notation:
Deployment- und Verteilungsdiagramme sowie Konfigurationsdiagramme

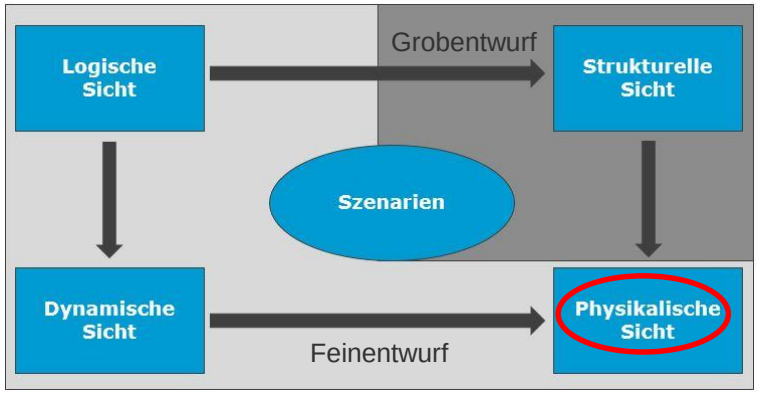


Abb.: 8
Philippe Kruchten, The 4 +1 view model of architecture, 1995

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

SICHTEN INNERHALB EINER ARCHITEKTUR

Dynamische Sicht

Aufgabe:
Abbildung von Produktmodell auf Verarbeitungsmodell

Modell:
keine allgemeinen Modelle

Zielgruppe:
Softwareentwickler und Mitarbeiter in Wartung

Notation:
Aktivitäts- und Sequenzdiagramme

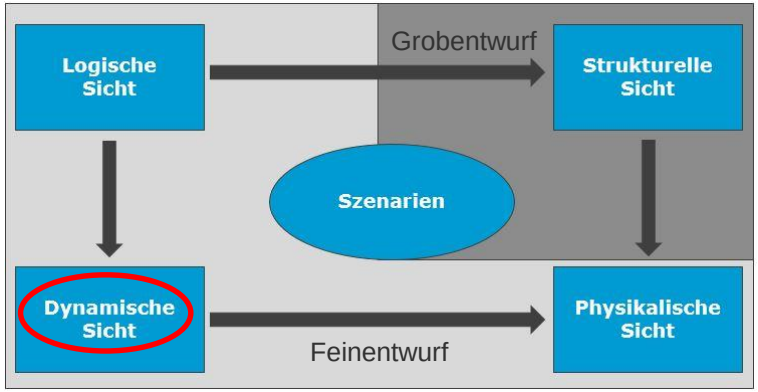


Abb.: 9
Philippe Kruchten, The 4 +1 view model of architecture, 1995

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

SICHTEN INNERHALB EINER ARCHITEKTUR

Szenarien

Aufgabe:
Spezifikation von Anwendungsfällen und Validierung der Architektur

Modell:
-

Zielgruppe:
Architekten, Softwareentwickler und Tester

Notation:
Use-Case-Diagramm
Expected-Result/s Verschriftung

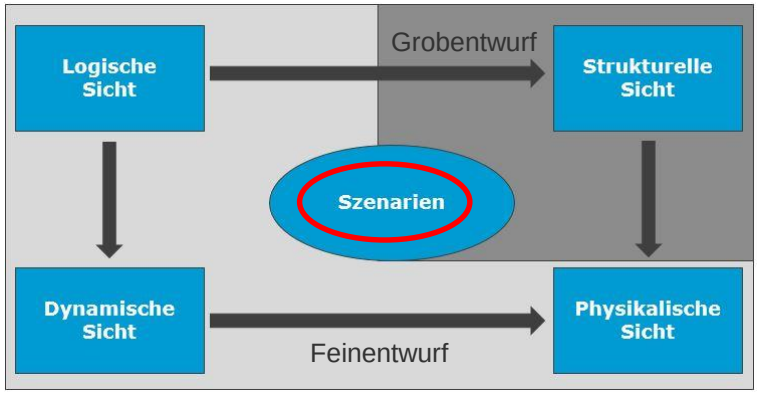


Abb.: 10
Philippe Kruchten, The 4 +1 view model of architecture, 1995

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

SOFTWARE-ARCHITEKTUR VS. SOFTWARE ENTWURF

Was ist jetzt der Unterschied?

- größere Softwareprojekten benötigen eine Differenzierung
- Architekturen dienen dem groben Entwurf einer Software
- Software Entwurf beschreibt (häufig) den Feinentwurf einer Software

Es gibt keine eindeutige Definition die eine saubere Trennung zulässt!

Nach G. Starke:

- Entwurf von Systemen durch Architektur gekennzeichnet
- Grenze zwischen Architektur und Design ist fließend

Peter Geßler
Peter.Gessler@b-tu.de

WDH.: OBJEKTORIENTIERTER SOFTWARE-ENTWURF

Der objektorientierte Softwareentwurf stellt eine in Bezug auf das zur Strukturierung eingesetzte Paradigma spezielle Form des Softwareentwurfs dar, bei dem ein Softwaresystem in Klassen […] zerlegt wird. Der objektorientierte Softwareentwurf umfasst die Definition der Softwarearchitektur, der Klassen und der Vererbungshierachien […].

Sven Overhage, Wirtschaftsinformatik
Otto-Friedrich-Universität Bamberg

Teilaufgaben des objektorientierten Softwareentwurfs

- Identifikation von Objekten (Klassen)
- Beschreibung dieser und ihrer Fähigkeiten
- Verknüpfung der Klassen zum Softwaresystem

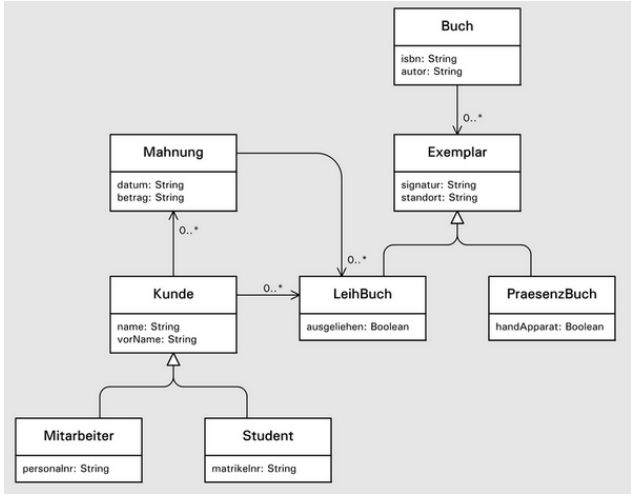


Abb.: 11; Bibliothekssystem (vereinfacht), Enzyklopädie der Wirtschaftsinformatik

Bei der Entwicklung großer Softwaresysteme wird von der Anwendung des objektorientierten Softwareentwurfs bisweilen abgeraten!

Szyperski, S.115-122, Component Software – Beyond Object-Oriented Programming

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

GRUNDPFEILER DER OOP

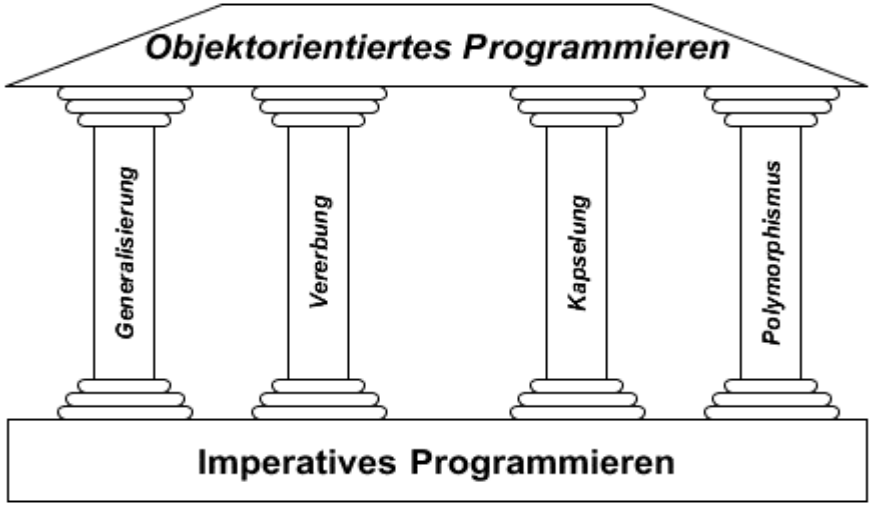


Abb.: 12; Säulen der objektorientierten Programmierung (OOP)

Generalisierung:	Basisklasse mit gemeinsamen Eigenschaften
Vererbung:	Unterklasse erhält Eigenschaften der Basisklasse
Kapselung:	Verstecken (data hiding) der Instanzvariablen
Polymorphismus:	Überschreiben von Methoden einer Basisklasse in der Unterklasse.

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

GRUNDPFEILER DER OOP - BEISPIEL

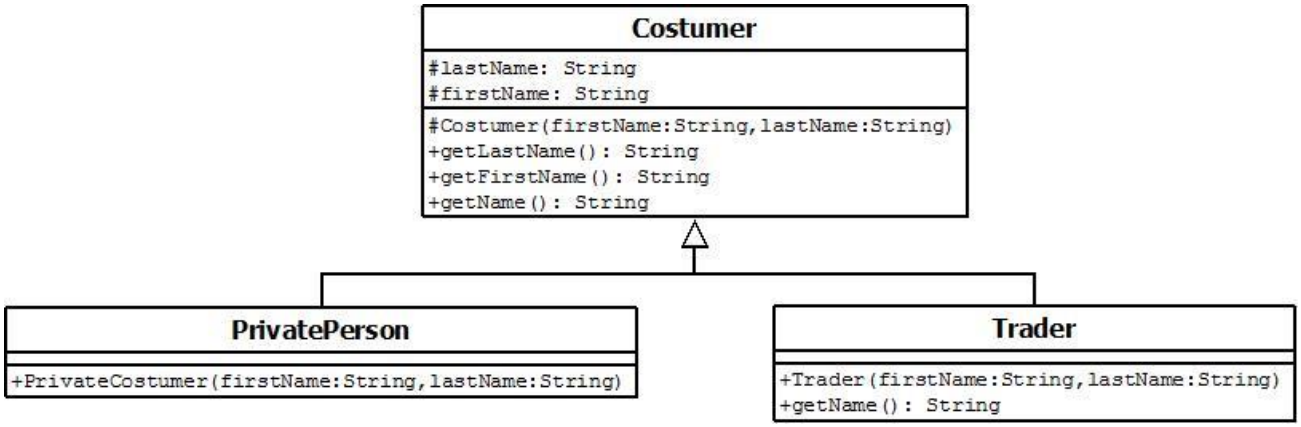


Abb.: 13; Klassendiagramm – Modellierung von Objekt Kunde

Generalisierung:	Kl. *Costumer* enthält alle Methoden die *PrivatePerson* und *Trader* gemeinsam besitzen
Vererbung:	Kl. *PrivatePerson* und *Trader* können auf die Variablen *lastName* und *firstName* zugreifen
Kapselung:	Variablen *lastName* und *firstName* sind nicht direkt von außen zu erreichen & Consumer nur innerhalb des Pakets
Polymorphismus:	Kl. *Trader* überschreibt die Methode *getName*()

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

OOP – VERERBUNG AM BEISPIEL

Costumer.java

```java
public class Costumer {

    protected String firstName;
    protected String lastName;

    protected Costumer(String firstName, String lastName) {

        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getLastName() {

        return this.lastName;
    }

    public String getFirstName() {

        return this.firstName;
    }

    public String getName() {

        return (this.firstName + " " + this.lastName);
    }
}
```

PrivatePerson.java

```java
public class PrivatePerson extends Costumer {

    public PrivatePerson(String firstName, String lastName) {

        super(firstName, lastName);
    }

}
```

Trader.java

```java
public class Trader extends Costumer {

    public Trader(String firstName, String lastName) {

        super(firstName, lastName);
    }

    public String getName() {

        return ("Trader " + this.lastName);
    }

}
```

Befehl:

extends – einfache Vererbung

this – Referenz des eigenen Objekts

super – Aufruf vom Konstruktor der Basisklasse

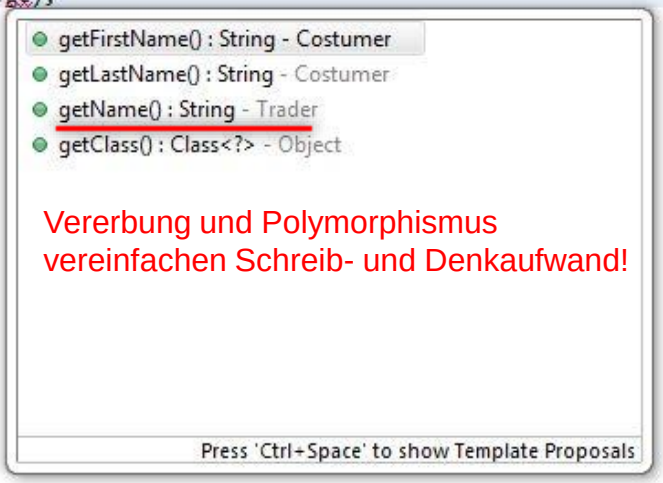
Abb.: 14; Programmcode Vererbung am Beispiel Teil 1

Peter Geßler
Peter.Gessler@b-tu.de

GRUNDPFEILER DER OOP – VERERBUNG AM BEISPIEL

Client.java

```
3  public class Client {
4
5      public static void main(String[] args) {
6
7          PrivatePerson person1 = new PrivatePerson("Max", "Mustermann");
8          Trader person2 = new Trader("Jane", "Doe");
9
0          System.out.println(person1.getFirstName()); // Output: Max
1          System.out.println(person1.getName()); // Output: Max Mustermann
2
3          System.out.println(person2.ge);
4
5      }
6
7  }
8
```



Befehl:

new – neues Objekt der Klasse erzeugen

Abb.: 15; Programmcode Vererbung am Beispiel Teil 2

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

OOP – VERERBUNG AM BEISPIEL

Client.java

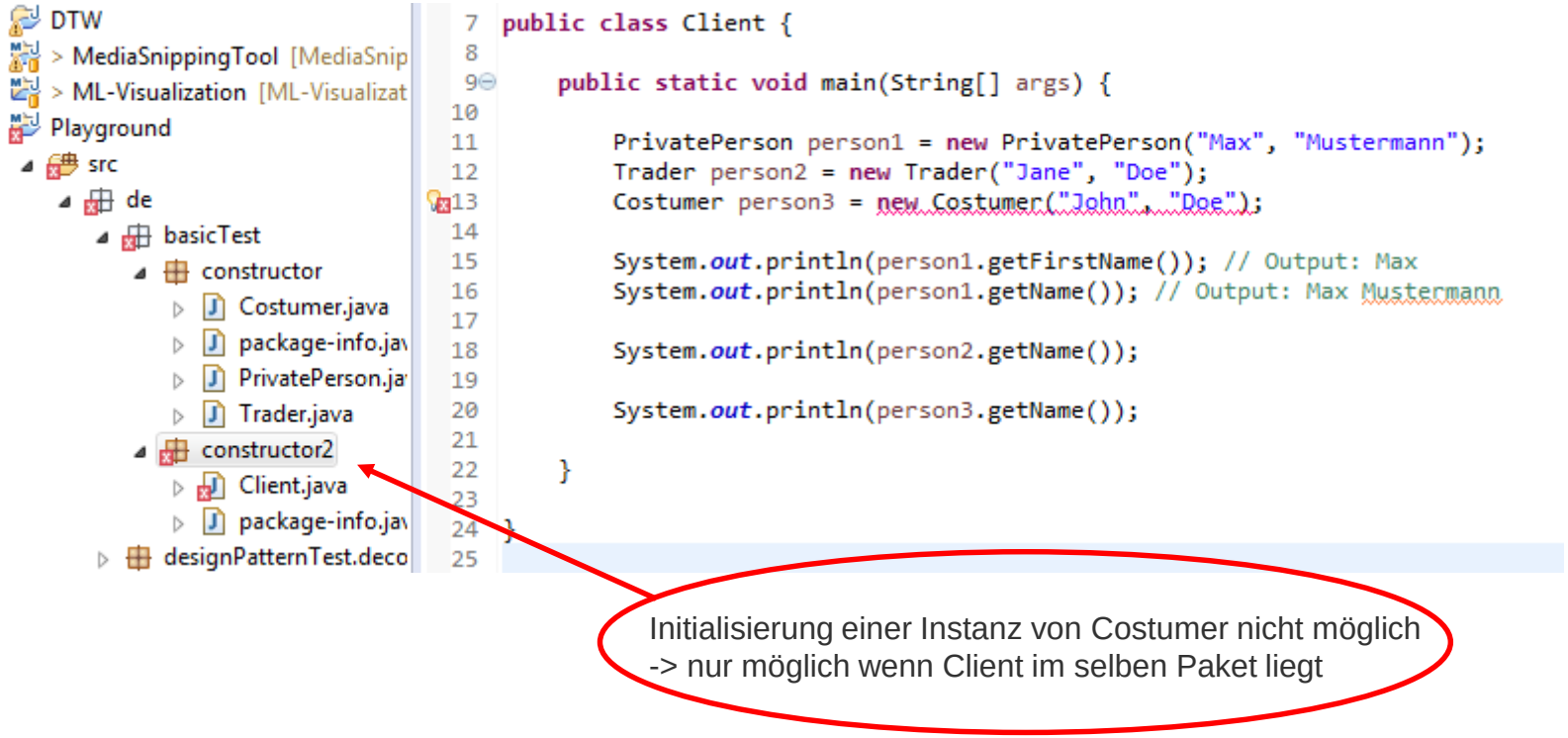


Abb.: 16; Programmcode Vererbung am Beispiel Teil 3

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

OOP – ABSTRAKTE KLASSEN & METHODEN

Szenario:

Sie entwickeln ein neuen Egoshooter in dem es natürlich unterschiedliche Feinde gibt. Darunter zählen der Reaper und der FireRaiser.
Sobald Sie einen ersten Schuss abgeben, reagieren die Objekte in ihrem Angriffsverhalten unterschiedlich.

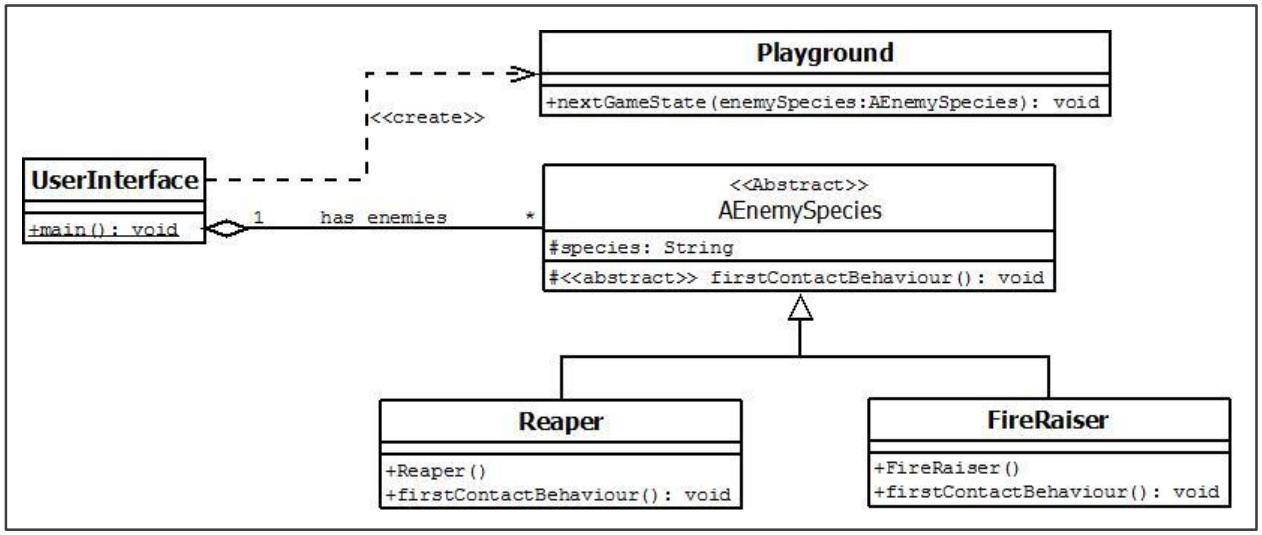


Abb.: 17; Klassendiagramm – Modellierung Nutzer-Feind Verhalten

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

OOP – ABSTRAKTE KLASSEN & METHODEN

```java
public abstract class AEnemySpecies {

    protected String species;

    protected AEnemySpecies(String speciesName) {

        this.species = speciesName;
    }

    protected abstract void firstContactBehaviour();
}
```

AEnemySpecies.java

```java
public class Reaper extends AEnemySpecies {

    public Reaper() {
        super("Reaper");
    }

    @Override
    protected void firstContactBehaviour() {

        System.out.println(species + " fährt Armkrallen "
                + "aus und schaut sich nach ihnen um.");
    }
}
```

Reaper.java

```java
public class FireRaiser extends AEnemySpecies{

    public FireRaiser() {
        super("FireRaiser");
    }

    @Override
    protected void firstContactBehaviour() {

        System.out.println(species + " legt eine Benzinspur.");
    }
}
```

FireRaiser.java

Abb.: 18; Programmcode – Abstrakte Klassen & Methoden Beispiel, Teil 1

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

OOP – ABSTRAKTE KLASSEN & METHODEN

```java
public class Playground {

    public void nextGameStep(AEnemySpecies enemySpecies) {

        enemySpecies.firstContactBehaviour();
    }
}
```

```java
public class UserInterface {

    public static void main(String[] args) {

        // create instances
        Playground playground = new Playground();
        Reaper reaper = new Reaper();
        FireRaiser fireRaiser = new FireRaiser();

        // game sequence
        playground.nextGameStep(reaper);

        System.out.println("Sie suchen einen anderen Weg");

        playground.nextGameStep(fireRaiser);
    }
}
```

Ausgabe vom Programm:

1.: Reaper fährt Krallen aus und schaut sich um.

2.: Sie suchen einen anderen Weg.

3.: FireRaiser legt eine Benzinspur.

Abb.: 19; Programmcode – Abstrakte Klassen & Methoden Beispiel, Teil 2

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

OOP – INTERFACES

Szenario:

Sie möchten jetzt nicht nur Egoshooter programmieren sondern auch andere Genres wie Abenteuerspiele. Des Weiteren möchten Sie diese verkaufen (Buyable) und auch vergleichen (Comparable) können.

Herausforderung: ein Objekt soll mehrere Typen repräsentieren

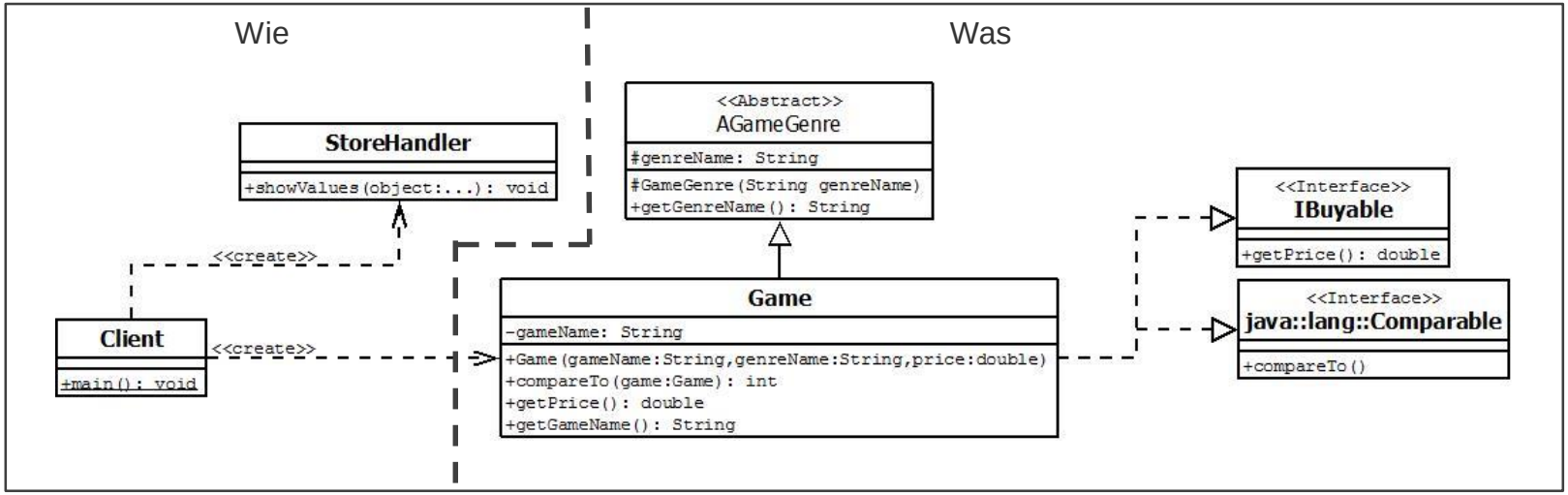


Abb.: 20; Klassendiagramm – Beispiel Interfaces

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

OOP – INTERFACES

```java
public abstract class AGameGenre {

    protected String genreName;

    protected AGameGenre(String genreName) {

        this.genreName = genreName;
    }

    public String getGenreName() {
        return genreName;
    }
}
```

AGameGenre.java

```java
public interface IBuyable {

    public double getPrice();
}
```

IBuyable.java

Game kann Objekttyp *Game*, *AGameGenre*, *IBuyable* oder *Comparable* sein

```java
public class Game extends AGameGenre implements IBuyable, Comparable<Game>{

    private String gameName;
    private double price;

    protected Game(String gameName, String genreName, double price) {
        super(genreName);

        this.gameName = gameName;
        this.price = price;
    }

    @Override
    public int compareTo(Game otherGame) {
        return Double.compare(this.price, otherGame.price);
    }

    @Override
    public double getPrice() {
        return this.price;
    }

    public String getGameName() {
        return this.gameName;
    }
}
```

Game.java

Abb.: 21; Programmcode – Beispiel Interfaces, Teil 1

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

OOP – INTERFACES

```
3  public class StoreHandler {
4
5      public void showValues (IBuyable gameObject) {
6
7          gameObject.
8      }
9  }
10
```

equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getPrice() : double - IBuyable
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
toString() : String - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
Press 'Ctrl+Space' to show Template Proposals

nur Methoden vom Interface Buyable und Kl. Object

```
3  public class StoreHandler {
4
5      public void showValues (AGameGenre gameObject) {
6
7          gameObject.
8      }
9  }
10
```

genreName : String - AGameGenre
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getGenreName() : String - AGameGenre
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
toString() : String - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
Press 'Ctrl+Space' to show Template Proposals

nur Methoden von Kl. AGameGenre und Object

```
3  public class StoreHandler {
4
5      public void showValues (Game gameObject) {
6
7          gameObject.
8      }
9  }
10
```

genreName : String - AGameGenre
compareTo(Game otherGame) : int - Game
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getGameName() : String - Game
getGenreName() : String - AGameGenre
getPrice() : double - Game
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
toString() : String - Object
wait() : void - Object
Press 'Ctrl+Space' to show Template Proposals

Methoden von allen geerbten Klassen, Schnittstellen und Object

Abb.: 22; Programmcode – Beispiel Interfaces, Teil 2

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

UML KLASSENDIAGRAMM – NOTATION & EINSCHRÄNKUNGEN

- Visualisieren der Softwarestruktur
- Beziehungen zwischen Klassen und Schnittstellen darstellen
- Unterscheiden zwischen Analyse- und Entwurfs-Klassendiagramm

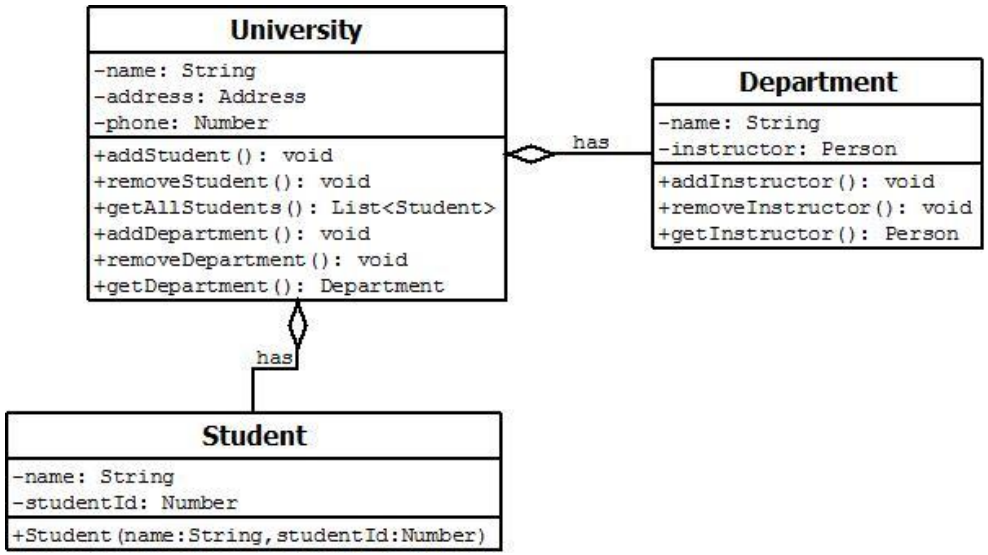


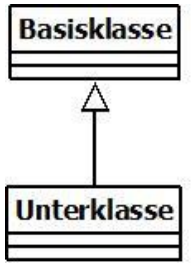
Abb.: 23; Analyseklassendiagramm (unvollständig)

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

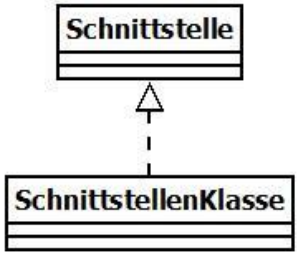
UML KLASSENDIAGRAMM – NOTATION & EINSCHRÄNKUNGEN

- zeigt nicht in welchen Zuständen sich das System befinden kann
- zeigt nicht welche Aktivitäten ausgeführt werden können
- zeigt nicht die Logik der Methoden

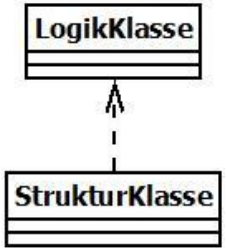
Wichtige Notationsrichtlinien:



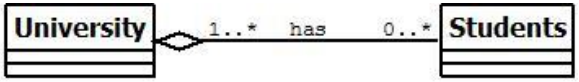
Vererbung



Schnittstellen-implementierung



erstellt, benutzt



Aggregation

Komposition

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

ENTWURFSPRINZIPIEN

- Einfachheit vor Allgemeinverwendbarkeit
  Wenn eine einfache Lösung bekannt ist, sollte diese auch verwendet werden.

- Prinzip der minimalen Verwunderung

- Vermeiden sie Wiederholungen (Don‘t Repeat Yourself)
  Keine Wiederholung von Struktur und Logik.

- Prinzip der einzelnen Verantwortlichkeit
  Eine Klasse/Methode hat die Verantwortung über eine Aufgabe. Boolsche bzw. numerische Steuer-Flags sollten vermieden werden.

Peter Geßler
Peter.Gessler@b-tu.de

ENTWURFSPRINZIPIEN

- Offenes-Geschlossenes-Prinzip

  Offen für Erweiterung aber geschlossen für Veränderung. Eine neue Klasse ist sinnvoller als eine Modifizierung von vorhandenem Quellcode.

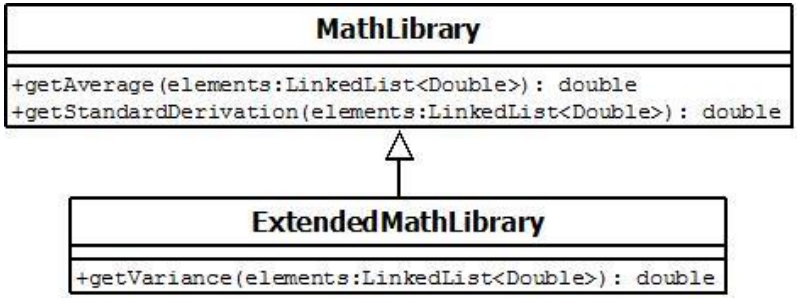
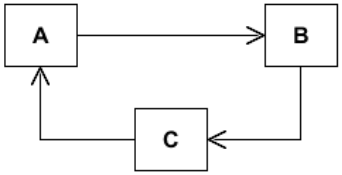
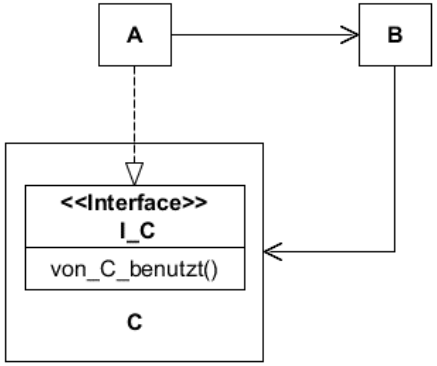


Abb.: 24; OCP

- Vermeidung zirkulärer Abhängigkeiten



Zirkuläre Abhängigkeit



Aufgelöster Zirkel

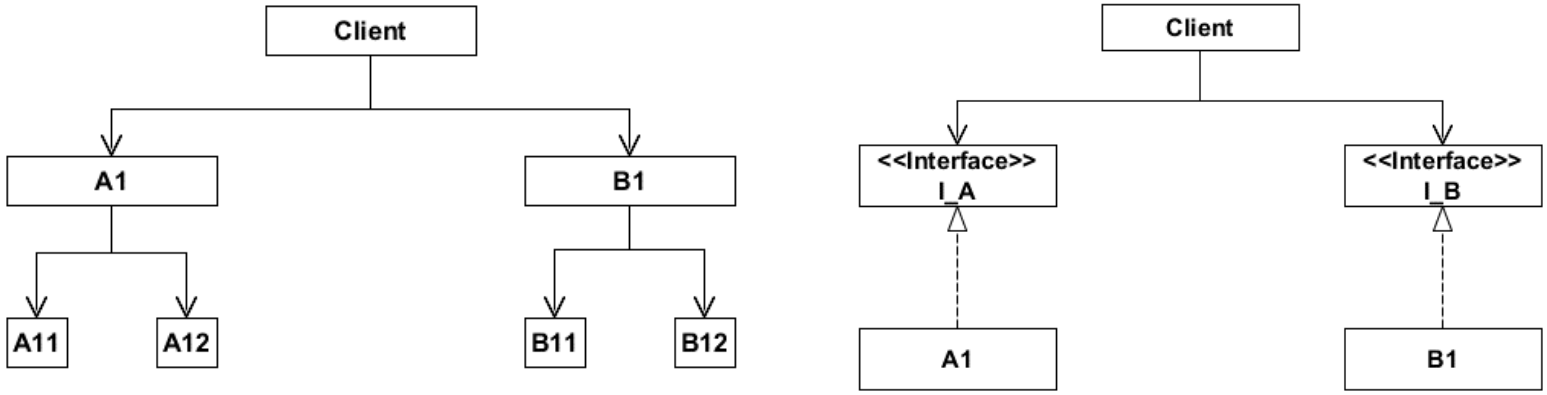
Abb.: 25; CDP

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

ENTWURFSPRINZIPIEN

- Liskov‘sches Substitutionsprinzip
  Einsatz von Unterklassen statt Oberklassen.

- Prinzip der Umkehrung von Abhängigkeiten



Schlechte Abhängigkeiten

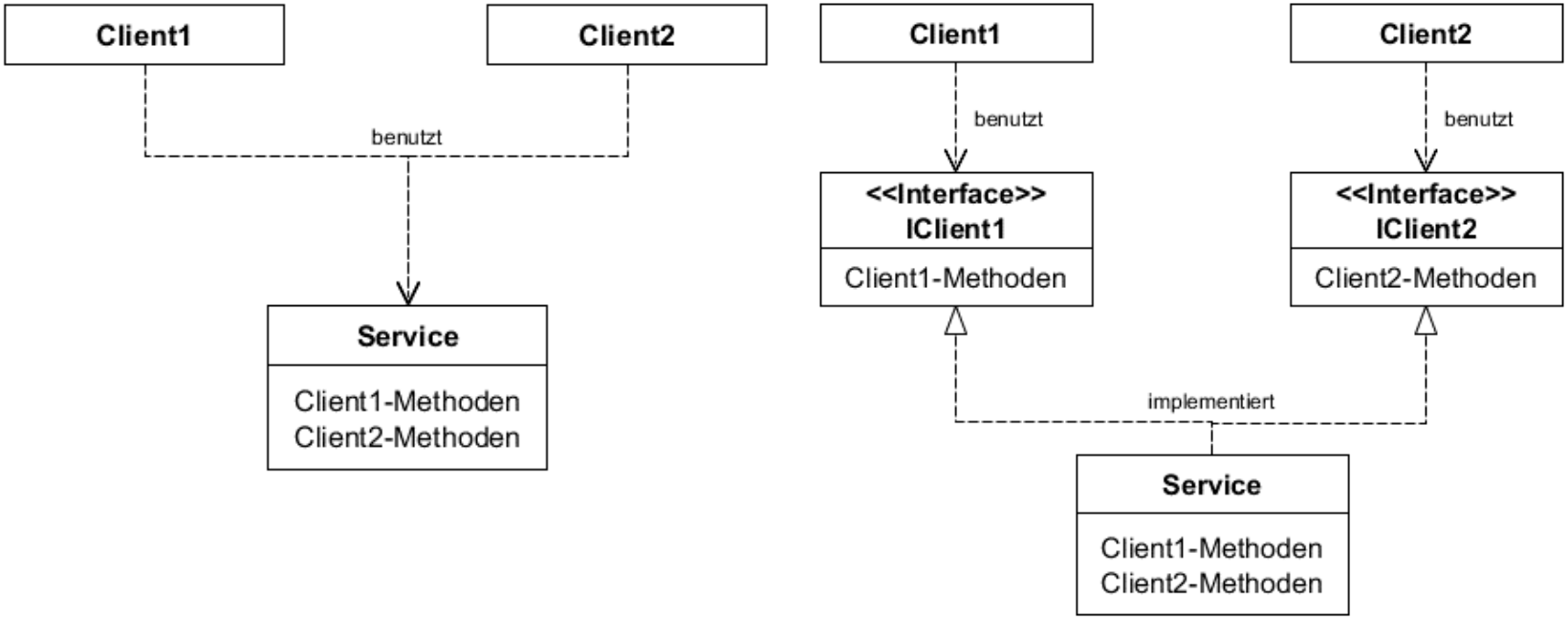
Bessere Abhängigkeiten

Abb.: 26; Umkehrung von Abhängigkeiten

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

ENTWURFSPRINZIPIEN

- Prinzip der Abtrennung von Schnittstellen



Universalschnittstelle

Spezifische Schnittstellen

Abb.: 27; Prinzip der Abtrennung von Schnittstellen

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

PRIMÄRE HEURISTIKEN IN DER OOP

1.: Eine Schnittstelle für jede Menge zusammengehöriger Dienstleistungen

2.: Assoziationen statt Vererbung bei Zugriff auf Dienstleistung

3.: Klientenklassen benutzen Schnittstelle statt Implementierungsklasse

4.: Unterklasse stellt mind. Dienstleistungen der Basisklasse zur Verfügung

5.: Jedes Objekt kann seine Dienstleistung jederzeit erbringen

6.: Monoliten- bzw. Gottklassen sind nicht zugelassen

7.: bei langer Argumentenliste ist eine Objektübergabe sinnvoller

Peter Geßler
Peter.Gessler@b-tu.de

WARUM JETZT EIGENTLICH ENTWURFSMUSTER?

Warum nicht einfach mit dem bisherigen Wissen programmieren?

Szenario:
Sie möchten ein Kassenprogramm für ihren Cafè Betrieb schreiben.
Das Programm soll lediglich die Zusammenstellung und den Preis dieser anzeigen.

Folgende Hauptsorten bieten Sie an:
Espresso 1,50 € Mocca 2,00 € Filterkaffe 1,00 €

Folgende Zusätze sind möglich:
Zucker 0,20 € Milchschaum 0,50 € Sahne 0,60 €

Wie würde das Klassendiagramm nach unserem bisherigen Wissensstand aussehen?

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

WARUM JETZT EIGENTLICH ENTWURFSMUSTER?

Warum nicht einfach mit dem bisherigen Wissen programmieren?

Szenario:
Wir modellieren unsere Freundin.

Mögliche Aktionen:
Unterhalten Kuss geben Ärgern

Zustände:
Neutral Bockig Fröhlich

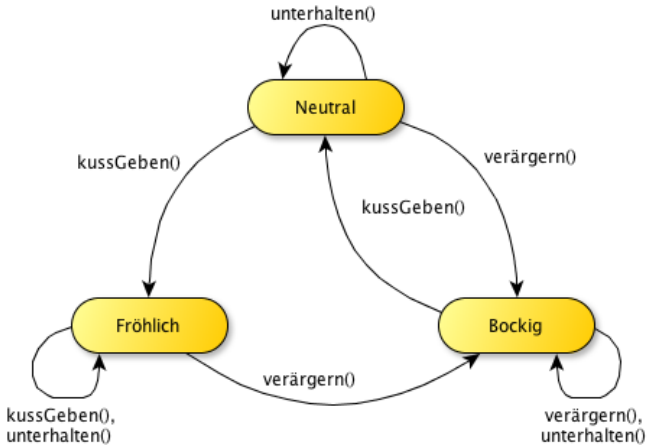


Abb.: 28
Philipp Hauer, Beispiel zu State Pattern, www.philipphauer.de

Je nach ausgeführter Aktion ändert sich ihr Zustand.

Wie würde das Klassendiagramm nach unserem bisherigen Wissensstand aussehen?

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

DEFINITION ENTWURFSMUSTER

1. Definition [Helke]
Ein Entwurfsmuster beschreibt eine häufig auftretende Struktur von miteinander kommunizierenden Komponenten, die ein allgemeines Entwurfsproblem in einen speziellen Kontext lösen.

2. Definition [Eilebrecht & Starke]
Entwurfsmuster lösen bekannte, wiederkehrende Entwurfsprobleme. Sie Fassen Design- und Architekturwissen in kompakter und wiederverwertbarer Form zusammen.

3. Definition [Starke]
Entwurfsmuster beschreiben einfache und elegante Lösungen für häufige Entwurfsprobleme.

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

HÄUFIG AUFTRETENDE ENTWURFSPROBLEME?

- Benachrichtigungsproblem

  Objekten mitteilen, dass sich mein Objektzustand geändert hat

- If-then-else Problem

  Systemverhalten ohne If-then-else Blöcke verändern

- Varianten Problem

  Objekt besitzt unterschiedliche Ausprägungen. Realisierung ohne „Klassenexplosion“.

- Single Instanz Problem

  Es darf nur eine Instanz zur Laufzeit existieren.

Peter Geßler
Peter.Gessler@b-tu.de

KATEGORISIERUNG ENTWURFSMUSTER

Erzeugungsmuster	Verhaltensmuster	Strukturmuster
Abstract Factory	Command	Adapter
Builder	Command Processor	Bridge
Factory Method	Iterator	(5) Decorator
(1) Singleton	Visitor	Fassade
Object Pool	(3) Strategy	Proxy
	Template Method	(6) Model View Controller
	(2) Observer	Flyweight
	State	(4) Composite

Es gibt mehr als 50 Entwurfsmuster!
Ein Design das viele Entwurfsmuster enthält ist noch lange kein gutes Design!
Es gilt: So „einfach“ wie möglich statt aufgebläht ohne Grund!

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

ERZEUGUNGSMUSTER - SINGLETON

Szenario:

Sie sind inzwischen im dritten Semester angekommen und besuchen das Modul Softwarepraktikum. Ihre Aufgabe ist die Entwicklung einer Software namens „DaxParty“ (Bierbörse). Sie haben sich natürlich sofort im Internet schlau gemacht und ein vorhandenes Programm samt Quellcode gefunden.

Notiz: In der Software gibt es nur ein Bier…

Beim testen fällt ihnen auf, dass der minimale sowie maximale Preis für ein Bier, den Sie beim Start festlegen, nicht eingehalten wird.

Ihnen fällt folgende Klasse auf:

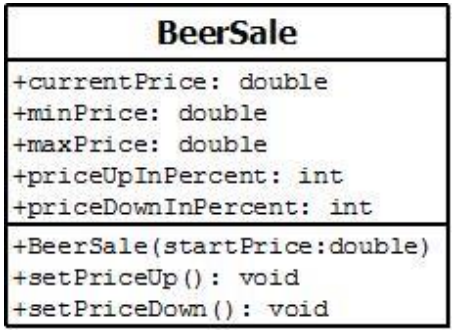


Abb.: 29; Beispiel Singleton Pattern – ugly code

Peter Geßler
Peter.Gessler@b-tu.de

ERZEUGUNGSMUSTER - SINGLETON

Fehler im Design:

- global verfügbare Variablen
- Modifizierung von externen Klassen möglich
- Plausibilitätsprüfung fehlt

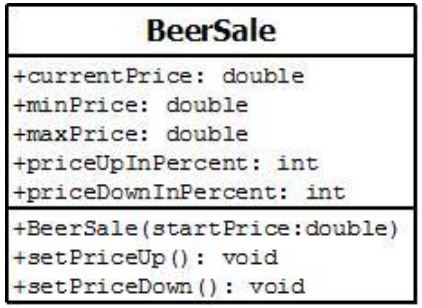


Abb.: 30; Beispiel Singleton Pattern – ugly code

Ziel:

- Werte sollen global verfügbar sein
- Veränderung von speziellen Werten nur nach Plausibilitätsprüfung
- kein direkter Variablenzugriff
- zentrale Verwaltung der Werte

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

ERZEUGUNGSMUSTER - SINGLETON

Lösung:

Abb.: 31; Beispiel Singleton – clean code

Vorteile:

- Zugriffskontrolle
- sauberer Namensraum
- Spezialisierung
- Lazy-Loading

Nachteile:

- prozedurales Programmieren
- globale Verfügbarkeit
- Intransparenz
- Objektzerstörung

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

ERZEUGUNGSMUSTER - SINGLETON

Modell:

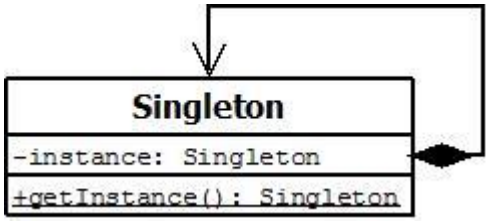


Abb.: 32; Singleton – Entwurfsmuster

Achtung: Bei Multi-Threading kann es durchaus zu Problemen kommen!

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - OBSERVER

Szenario:

Sie schreiben ein Programm für den Öl-Aktienhandel. Die Aktienkurse sind dabei vom Verhalten des Fördervolumen Parameters der OPEC abhängig. Sobald sich dieser Parameter ändert, müssen sich die Aktienkurse automatisch anpassen.

Ziel: autom. Aktualisierung der Aktienobjekte bei Parameteränderung

Bisherige Lösung:

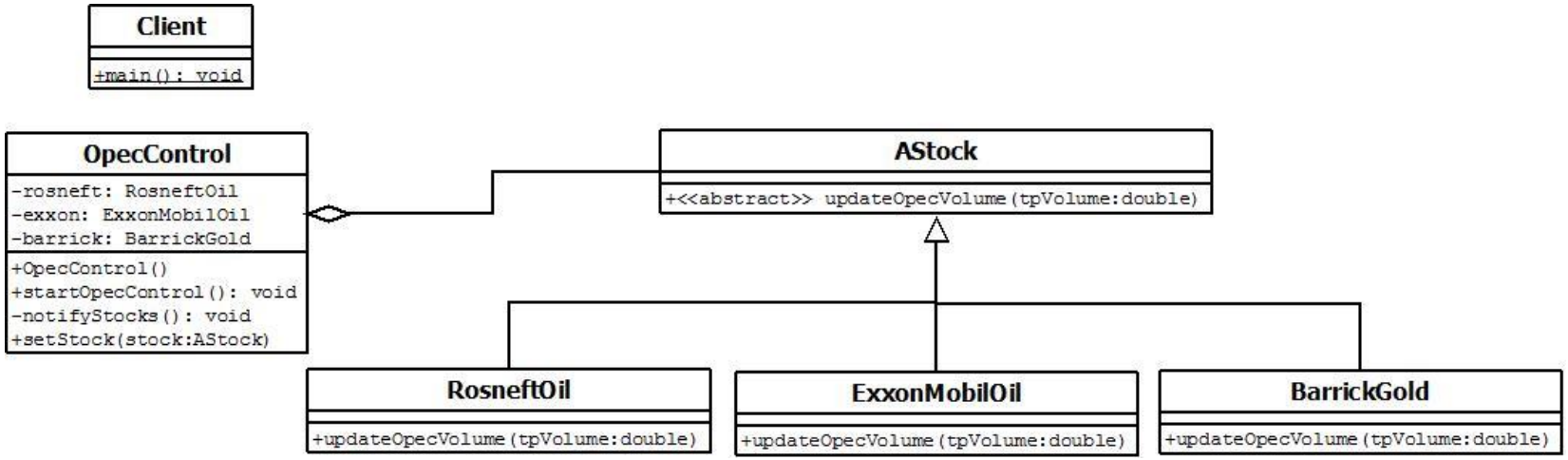


Abb.: 33; Beispiel Observer – Pattern, ugly code

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - OBSERVER

Fehler im Design:

- enge Kopplung zwischen Objekt *OpecControl* und Aktienobjekten
- Erweiterbarkeit stark eingeschränkt (notifyStockPrice() anpassen)
- keine dynamischen Änderungen möglich

Ziel:

- Entkopplung von OpecControl und Aktienobjekten
- Schnittstelle verwenden statt direkte Implementierung
- dynamische Änderungen ermöglichen durch An- / Abmeldung von Aktienobjekten

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - OBSERVER

Lösung:

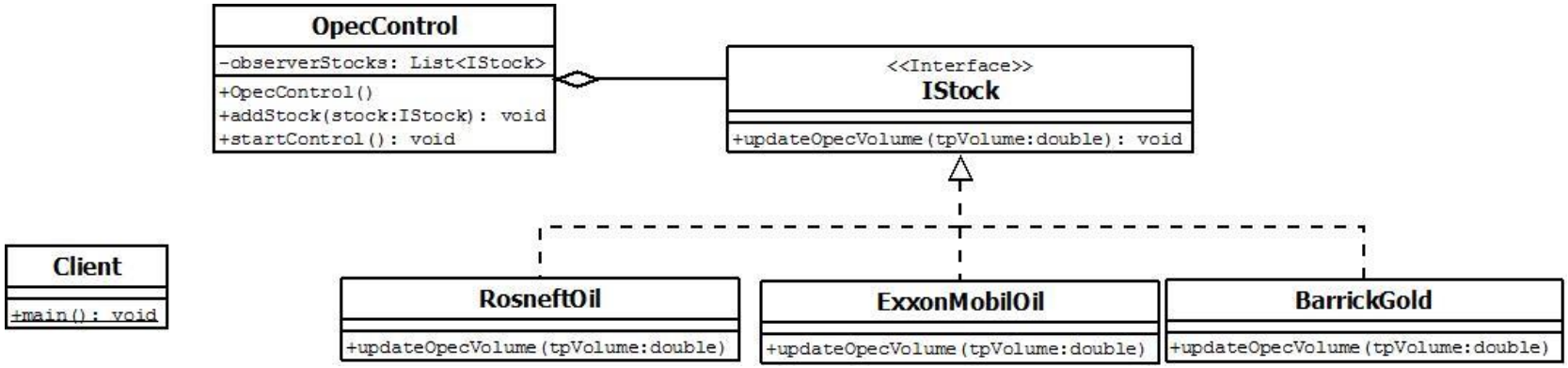


Abb.: 34;
Modellierung des Beispiels OPEC Control unter Verwendung des Observer Patterns

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - OBSERVER

Modell:

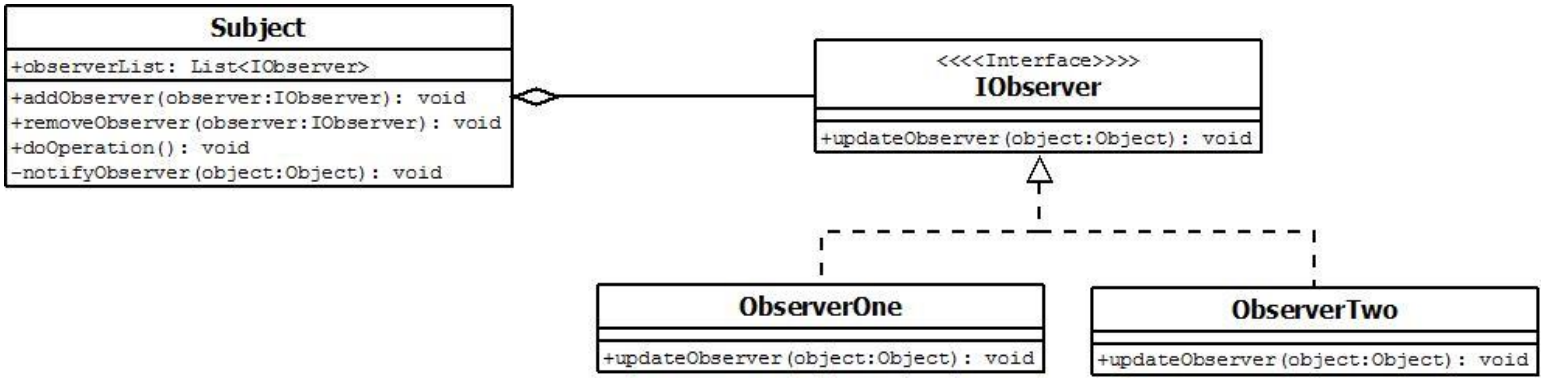


Abb.: 35; Observer - Entwurfsmuster

Die Anmeldung beim Subject kann in verschiedenen Varianten erfolgen.
Dies kommt auf den Ort der Erzeugung von ObserverOne & ObserverTwo an

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - OBSERVER

Einsatzzweck:

- Objekte über Zustandsänderung informieren
- Model-View-Controller Pattern

Varianten:

- Push Modell (Subjekt gibt Daten an Observer weiter)
- Pull Modell (Subjekt benachrichtigt Observer, Observer rufen relevante Parameter vom Subjekt ab)

Vorteile:

- Zustandskonsistenz
- Flexibilität
- Wiederverwendbarkeit
- Kompatibilität zum Schichtenmodell

Nachteile:

- Aktualisierungskaskaden
- Abmeldung von Observer

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - STRATEGY

Szenario:

Sie möchten ein Programm zur Lohnabrechnung schreiben. Ihre Mitarbeiter werden in die Steuerklassen *ledig, verheiratet* und *ledig mit Kind* eingeteilt.
In einem ersten Schritt soll das Programm, dass Nettogehalt und die Rente pro Monat ausrechnen. Des Weiteren soll eine Abfrage vom Namen und von der Steuerklasse möglich sein.

Ziel: Berechnung der Werte in Abhängigkeit seiner Steuerklasse.

bisherige Lösung:

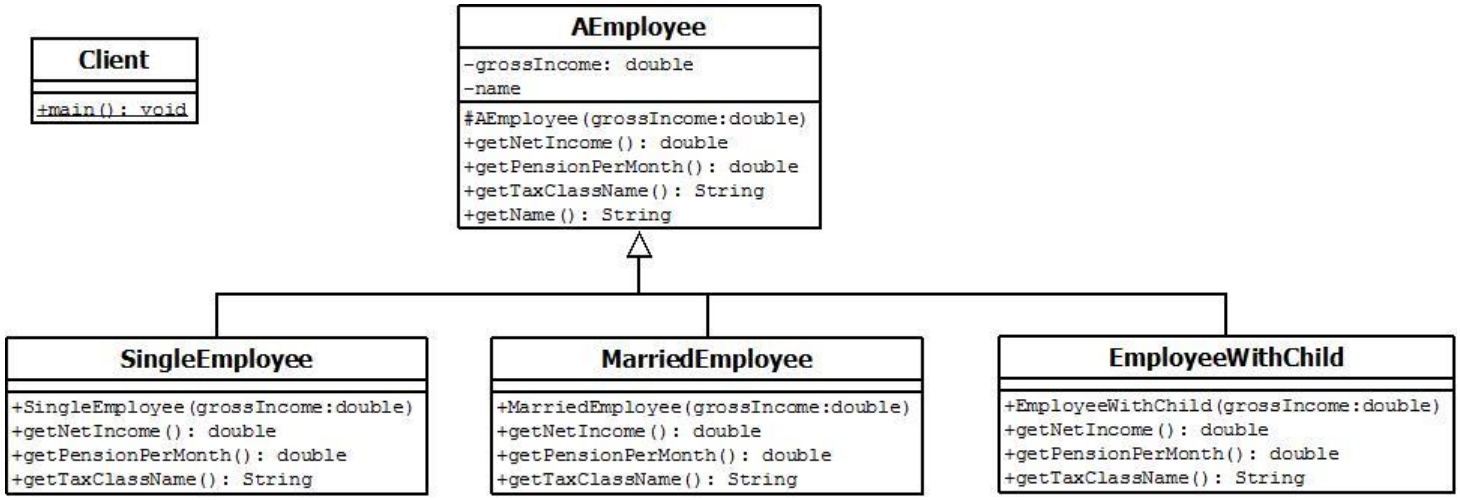


Abb.: 36; Beispiel Strategy – Pattern, ugly code

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - STRATEGY

Fehler im Design:

- Code Redundanz (doppelter Code bei gleichem Verhalten)
- Verhalten der Mitarbeiter nicht änderbar zur Laufzeit
- keine Wiederverwendbarkeit

Ziel:

- Trennung von Aspekten die sich ändern und jenen die konstant sind
- Algorithmus unabhängig vom nutzenden Client austauschen

Peter Geßler
Peter.Gessler@b-tu.de

VERHALTENSMUSTER - STRATEGY

Lösung:

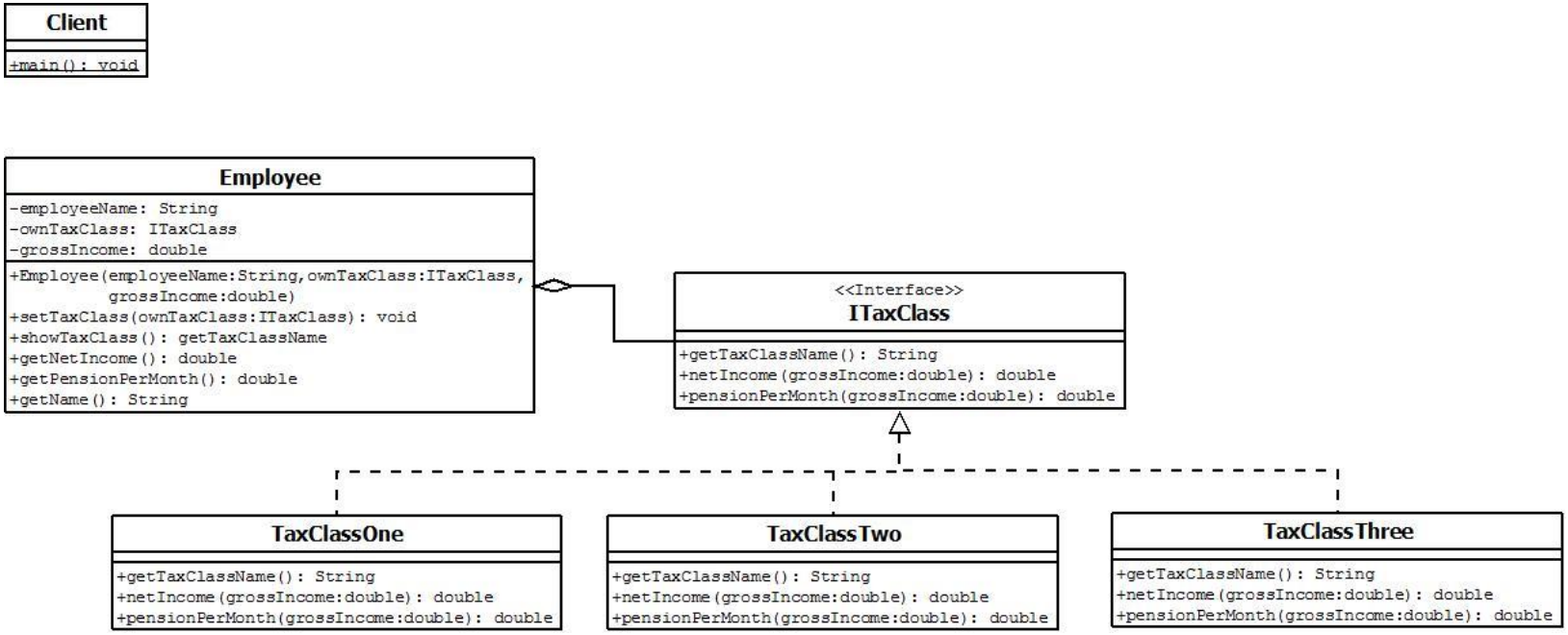


Abb.: 37; Beispiel Strategy – Pattern, clean code

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - STRATEGY

Modell:

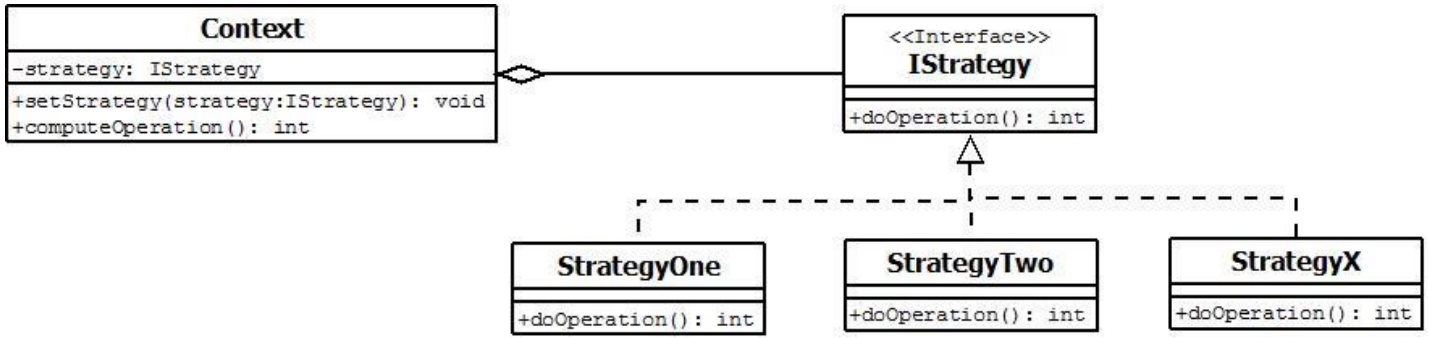


Abb.: 38; Strategy – Entwurfsmuster

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - STRATEGY

Einsatzzweck:

- Alternative zur Unterklassenbildung
- Entkopplung von Kontext und Verhalten
- verstecken von komplexen Algorithmusdetails
- If()…else if()…else… Ketten auflösen

Vorteile:

- Komposition statt Vererbung
- dynamisches Verhalten
- Vermeidung von Bedingungen

Nachteile:

- Kopplung zwischen Client und Strategieimplementierung
- evtl. unnötige Kontext-Strategie-Kommunikation
- Klassenexplosion bei zu starker Verhaltenstrennung

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – COMPOSITE

Szenario:

Das neue Verwaltungssystem soll die Personalhierarchie der Firma darstellen. Der Hierarchiebaum sieht wie folgt aus:

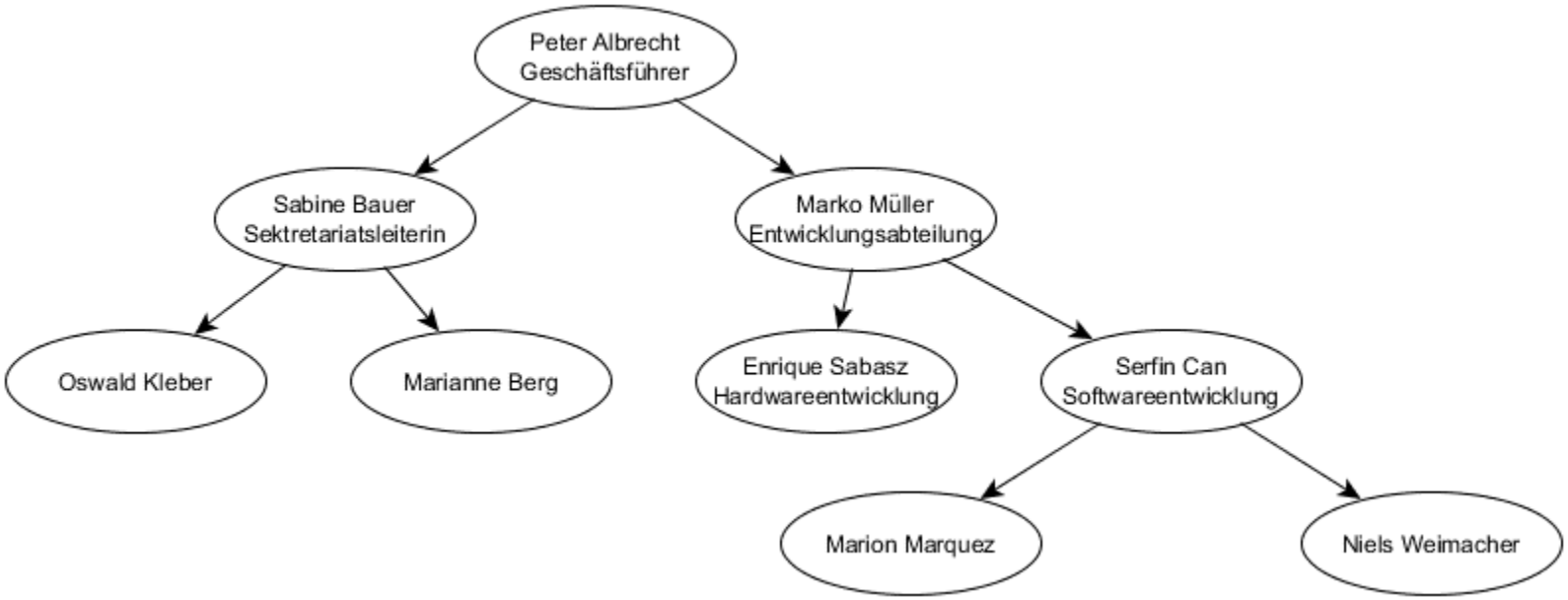


Abb.: 39; Hierachiebaum von Firmenpersonal

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – COMPOSITE

Anforderungen an das System:

- jeder Mitarbeiter und Abteilungsleiter besitzt einen Namen und eine Telefonnummer (getName(), getPhoneNumber())
- jeder Abteilungsleiter kennt seine Abteilung (getDepartment())
- Operationen um Mitarbeiter in den Baum einzuordnen, zu holen oder zu entfernen (add(), getEmployee(), remove())
- Abteilungsleiter soll Mitarbeiteranzahl in seiner Abteilung ausgeben können (getEmployeeCount())

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – COMPOSITE

Lösung vom Praktikant:

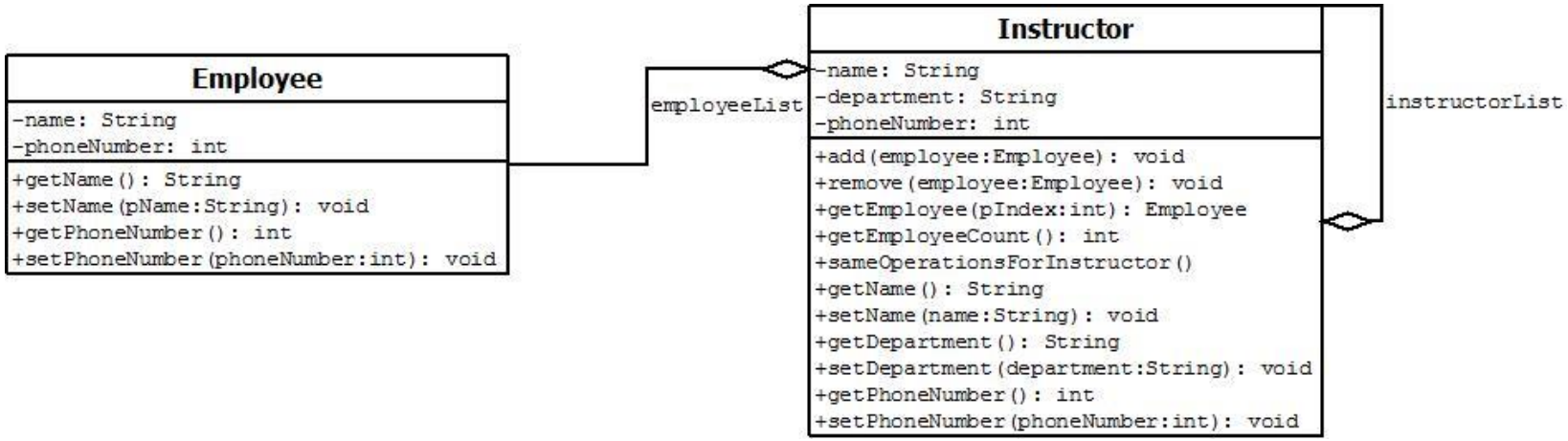


Abb.: 40; Beispiel Composite – Pattern, ugly code

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – COMPOSITE

Lösung:

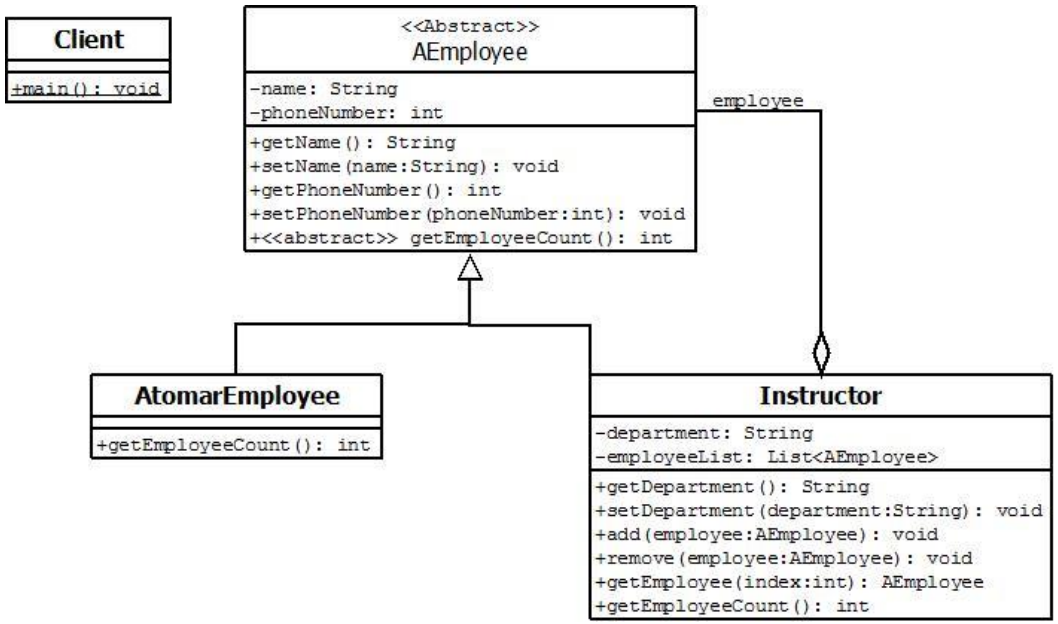


Abb.: 41; Beispiel Composite – Pattern, clean code

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – COMPOSITE

Modell:

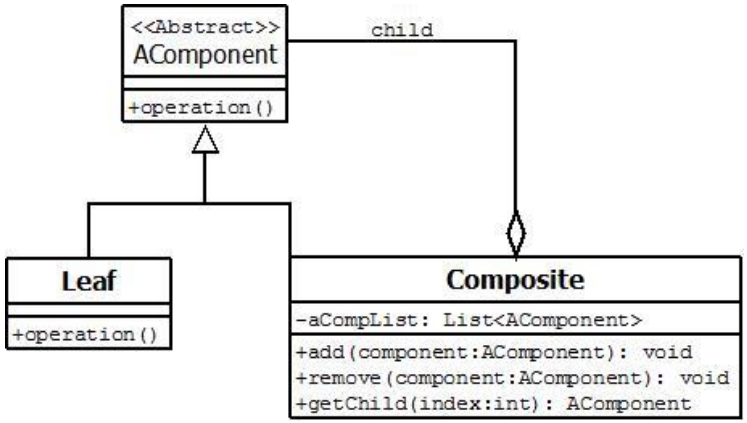


Abb.: 42; Composite – Entwurfsmuster

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

VERHALTENSMUSTER - COMPOSITE

Einsatzzweck:

- Modellierung von Hierachien

Vorteile:

- Einfache Bedienung
- Einheitliche Bedienung und Allgemeingültigkeit
- strukturelle Erweiterbarkeit

Nachteile:

- keine personelle Veränderung zur Laufzeit möglich

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – DECORATOR

Erinnern Sie sich noch an unsere Software für den CoffeeShop?

Szenario:
Sie möchten ein Kassenprogramm für ihren Cafè Betrieb schreiben.
Das Programm soll lediglich die Zusammenstellung und den Preis dieser anzeigen.

Folgende Hauptsorten bieten Sie an:
Espresso 1,50 € Mocca 2,00 € Filterkaffe 1,00 €

Folgende Zusätze sind möglich:
Zucker 0,20 € Milchschaum 0,50 € Sahne 0,60 €

Modellierung an der Tafel

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – DECORATOR

Fehler im Design:

- Klassenexplosion / enormer Zeitaufwand bei Erweiterung
- Mangelnde Flexibilität
- Mangelnde Dynamik

Ziel:

- jedes Element soll durch eine Klasse repräsentiert werden
- Dynamische Kombination zur Laufzeit ermöglichen
- Einschränkungen bereits in der Struktur festlegen
- Einhaltung des Offen/Geschlossen-Prinzips

Peter Geßler
Peter.Gessler@b-tu.de

Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – DECORATOR

Lösung:

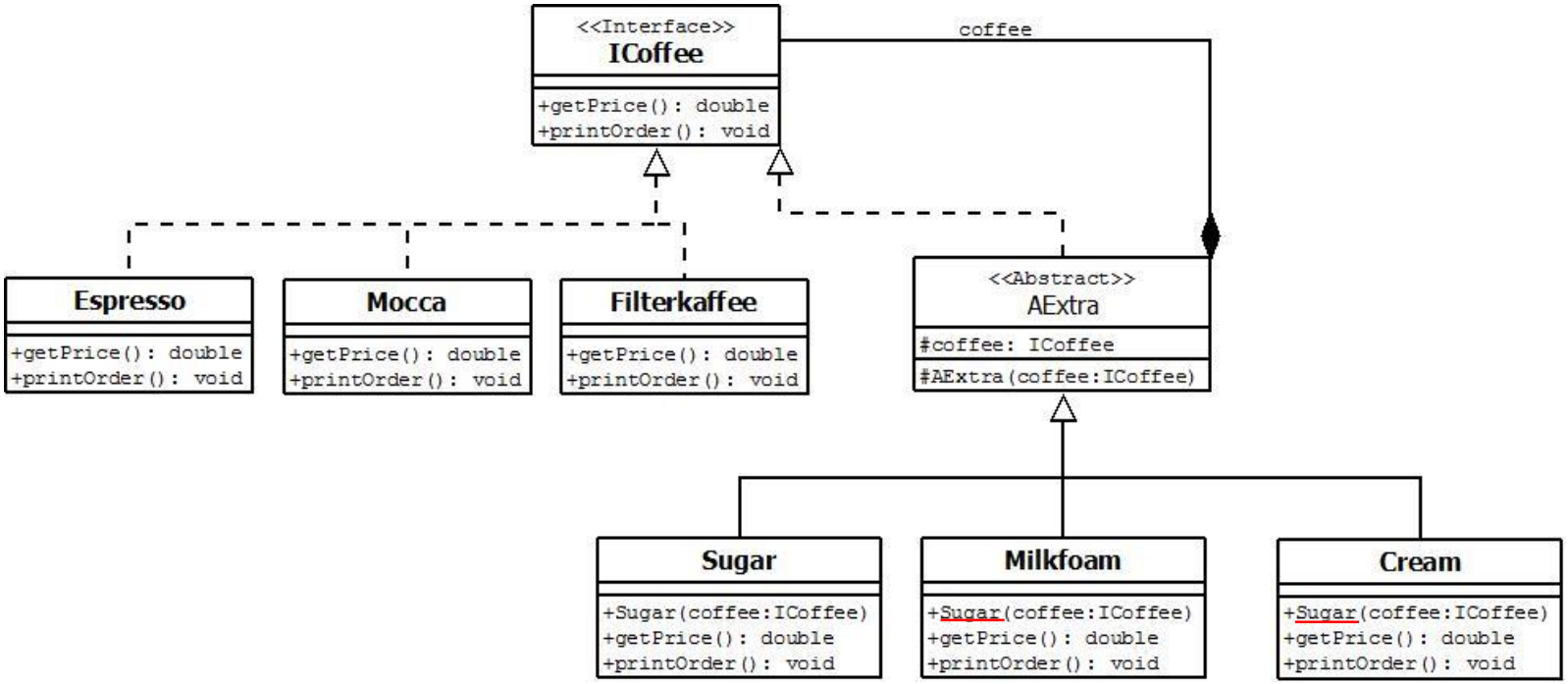


Abb.: 43; Beispiel – Decorator Pattern, clean code

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – DECORATOR

Modell:

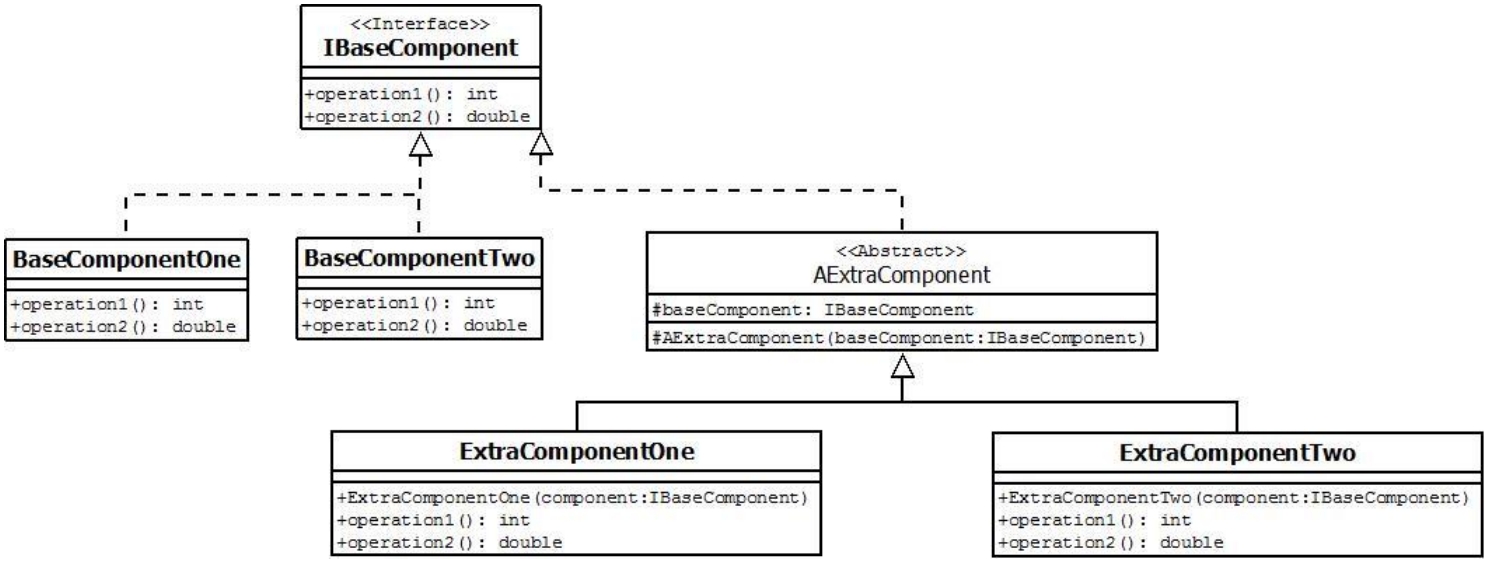


Abb.: 44; Decorator – Entwurfsmuster

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

STRUKTURMUSTER – DECORATOR

Einsatzzweck:

- Dynamische und transparente Funktionserweiterung
- Hinzufügen bzw. Entfernen von Funktionalität zur Laufzeit
- Vererbung nicht praktikabel für Funktionserweiterung

Vorteile:

- Dynamik und Flexibilität
- Transparenz
- kohäsive Klassen
- keine Vererbungs-kaskaden

Nachteile:

- erschwerte Fehlerfindung
- hohe Objektanzahl
- Komplexität bei „Decoketten“

Peter Geßler
Peter.Gessler@b-tu.de
Fakultät 1
Professur Kommunikationstechnik

SIE HABEN ES GESCHAFFT!

WILLKOMMEN ZUR DISKUSSIONS- UND FRAGENRUNDE

Peter Geßler
Peter.Gessler@b-tu.de