UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Software Entwicklung & Programmierung

Software-Architektur

PALUNO
The Ruhr Institute for Software Technology

Disclaimer

Bilder und Texte der Veranstaltungsfolien und -unterlagen sowie das gesprochene Wort innerhalb der Veranstaltung und Lehr-Lern-Videos dienen allein dem Selbst- bzw. Gruppenstudium. Jede weiterführende Nutzung ist den Teilnehmenden der Moodle-Kurse untersagt, z.B. Verbreitung an andere Studierende, in sozialen Netzwerken, dem Internet!

Darüber hinaus ist ein studentischer Mitschnitt von Webkonferenzen im Rahmen der Lehre nicht erlaubt.

Zielsetzung

Am Ende dieser Präsenzstunde könnt Ihr:

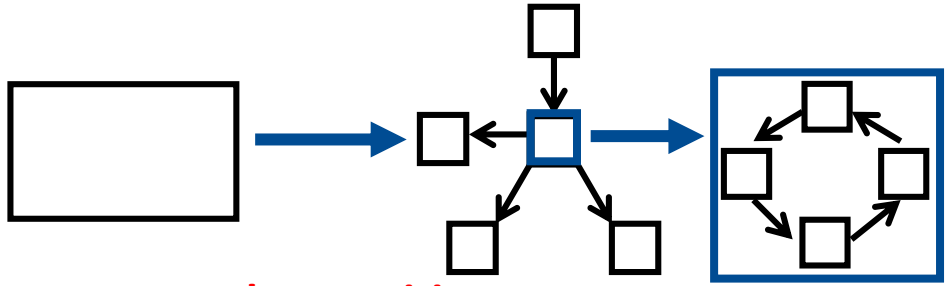
- die Grundlagen von **Software-Architektur** erläutern
- die Software-Architektur eines Software-Produkts in Form eines **UML-Klassendiagramms** dokumentieren
- eine **Dekomposition** mit Hilfe des **MVC-Patterns** durchführen

Agenda

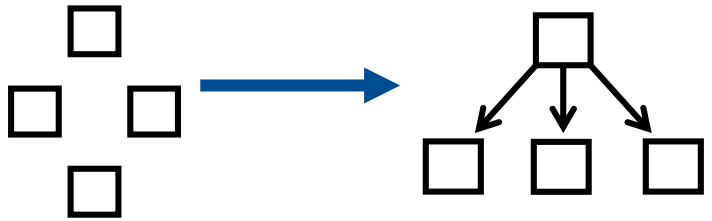
1. **Einführung Software-Architektur**
2. MVC (Model-View-Controller)
3. Dokumentation von Software Architektur mit UML-Klassendiagrammen
   - Klassen und Objekte
   - Assoziation und Abhängigkeit
   - Generalisierung
   - Interfaces
   - Packages
   - Modellierungstools

Motivation

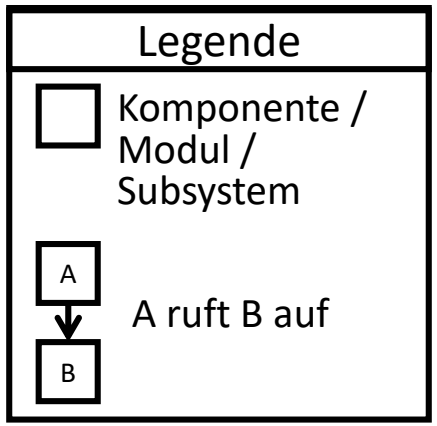
- Software-Systeme und -Prozesse können **groß** und **komplex** sein
- Um dies zu adressieren, hat sich die **Unterteilung („Separation“)** in unterschiedliche **Aspekte („Concerns“)** bewährt.
- Die unterschiedlichen Aspekte können separat behandelt und verstanden werden.
- „Separation“ in **strukturelle** „Concerns“: Das System wird **hierarchisch** in **Module (Teilsysteme oder Komponenten)** aufgeteilt
- **Ziel**: Unterteilung eines komplexen Systems in **Module** („divide and conquer“-Prinzip), z.B. Autobau, Hausbau



Dekomposition

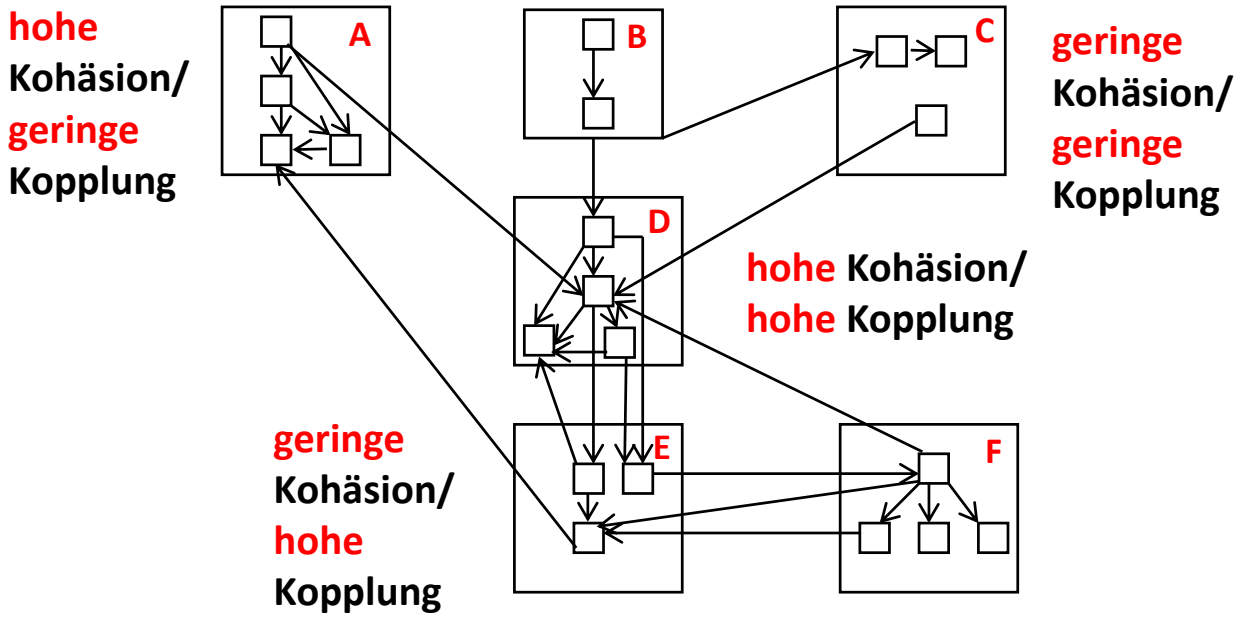


Komposition



Kohäsion und Kopplung

- **Modularität** ist eine spezielle Form der Strukturierung, bei welcher das System in **Komponenten, Module** oder **Teilsysteme** untergliedert wird
- Ein System lässt sich auf vielfältige Weise **modularisieren**. Eine gute Modularisierung zeichnet sich durch zwei wesentliche Kriterien aus: **hohe Kohäsion** und **geringe Kopplung** (sie nächste Folie).



Kohäsion und Kopplung

- Die Kohäsion beschreibt wie eng die Elemente innerhalb eines Moduls zusammenhängen
  - Je höher die Kohäsion umso enger die Zusammenhänge innerhalb eines Moduls
- Die Kopplung beschreibt die Abhängigkeit der Module untereinander
  - Je höher die Abhängigkeit umso schwieriger ist es Änderungen in einem Modul vorzunehmen, da dann alle anderen Module davon betroffen sind
- Die Skalen geben jeweils unterschiedliche Ausprägungen und Stärken der Abhängigkeiten wieder
- Beispiel für unterschiedliche Ausprägungen der Kriterien:
  - Module A und B haben gute Modularität, da jeweils hohe Kohäsion und geringe Kopplung

Definition von Software-Komponenten

- Komponenten bilden abgeschlossene, strukturelle **Einheiten der Software** (Beispiel in OO-Programmierung: **Klassen**).
- Im Rahmen der Software-Architektur muss klar definiert werden, welche **Dienste** (Funktionen, Verhalten) eine Komponente **anbietet**, und welche Dienste sie von anderen Komponenten **benötigt**.
- Als Software-Architekt steht man häufig **wiederkehrenden Problemen** gegenüber, wie man eine gute Strukturierung und Modularisierung erreicht.
- Für diese Probleme haben sich über die Zeit sogenannte **Entwurfsmuster** („Design Patterns“ bzw. „Architectural Patterns“) etabliert. Diese dokumentieren grundsätzliche und **bewährte Lösungen** für **wiederkehrende Probleme**.
- Entwurfsmuster sind ein **abstrakter Wiederverwendungsansatz**, da keine konkreten Artefakte (Modelle oder Code) wiederverwendet werden, sondern Lösungsstrategien.
- Ein Beispiel für ein Entwurfsmuster ist das **Model-View-Controller-Pattern (MVC)**

Agenda

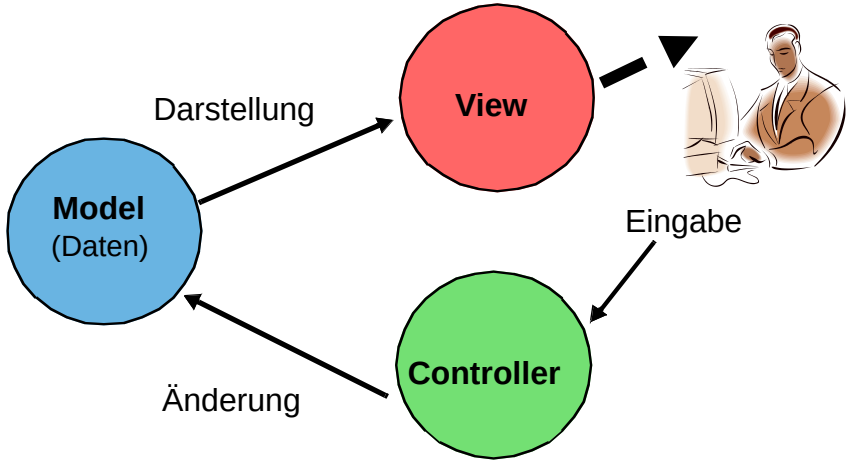
1. Einführung Software-Architektur
2. **MVC (Model-View-Controller)**
3. Dokumentation von Software Architektur mit UML-Klassendiagrammen
   - Klassen und Objekte
   - Assoziation und Abhängigkeit
   - Generalisierung
   - Interfaces
   - Packages
   - Modellierungstools

Beispiel-Muster: Model-View-Controller (MVC)

- **Kontext:**
  - Interaktive Anwendungen mit einer flexiblen Benutzerschnittstelle
- **Problem:**
  - Benutzerschnittstellen sind besonders anfällig für Änderungen. Jede Änderung einer Anwendung schlägt sich in der Regel in der Benutzerschnittstelle nieder.
  - Verschiedene Benutzer haben verschiedene Anforderungen an die Benutzerschnittstelle.
  - Zu enge Kopplung an den funktionalen Kern der Anwendung macht Änderungen schwierig und teuer.
    - Ideal wäre z.B. Portierung der Software auf ein anderes Betriebssystem. Dies erfordert nur die Anpassung der grafischen Oberfläche, aber nicht die der Datenhaltung.
- **Lösung:**
  - Aufteilung der Anwendung in drei Kernkomponenten ***Model***, ***View*** und ***Controller***
  - Der Benutzer interagiert mit dem System ausschließlich über die Controller
  - Views zeigen Daten
  - Model verwaltet die Daten

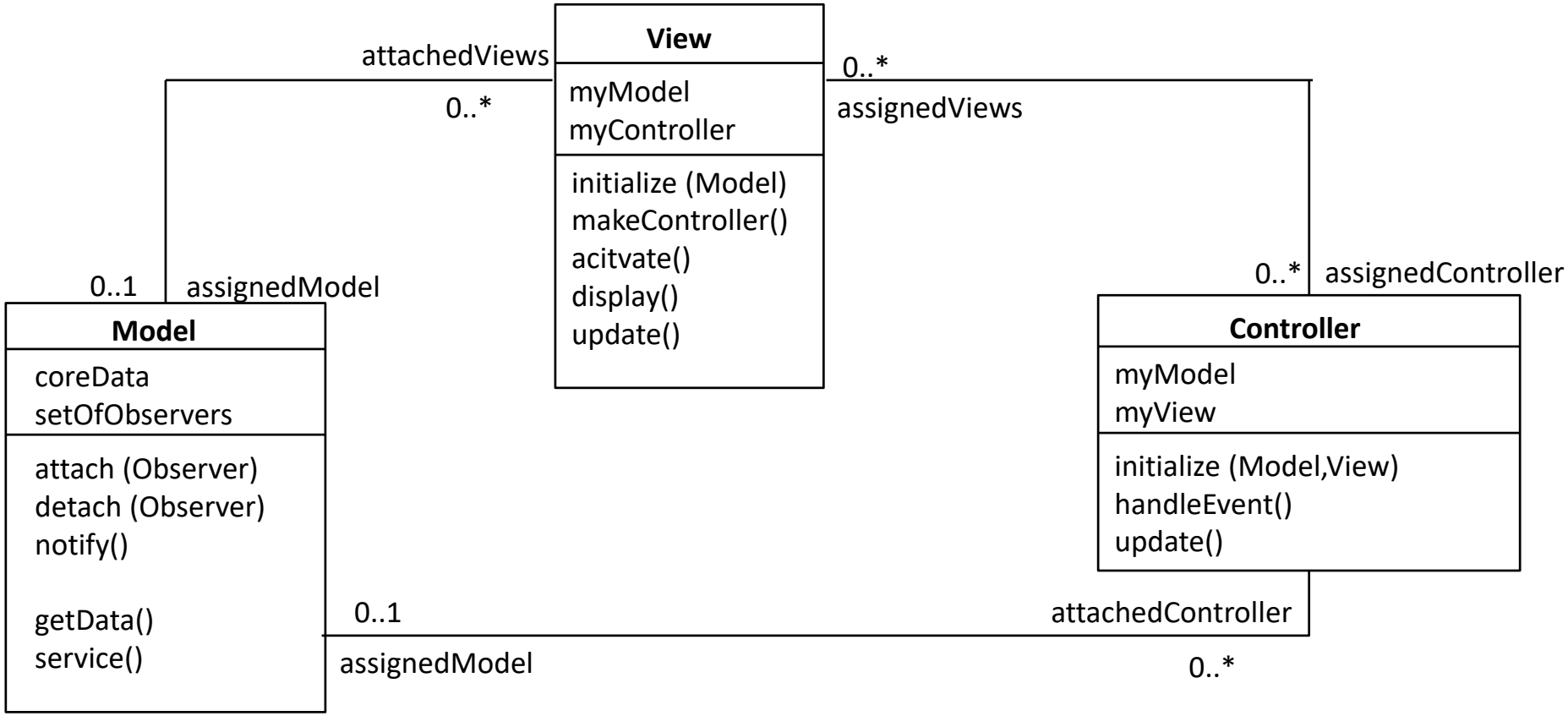
Beispiel-Muster: Model-View-Controller (MVC)

- In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task.
  - The **view** manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application.
  - The **controller** interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
  - Finally, the **model** manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).



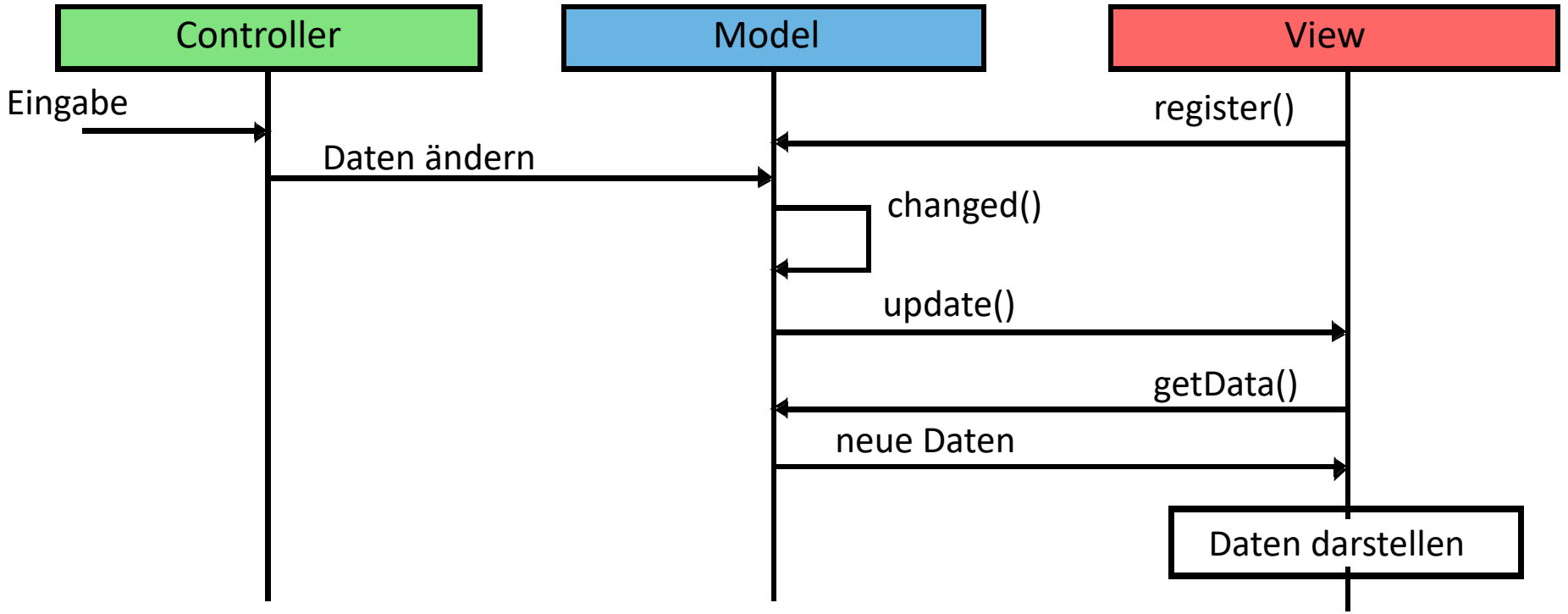
Quelle: Burbeck, Steve. "Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)."*University of Illinois in Urbana-Champaign (UIUC) Smalltalk Archive.*

Beispiel-Muster: Model-View-Controller (MVC)



Mehr zur Anwendung von MVC im Kapitel zu JavaFX

Beispiel-Muster: Model-View-Controller (MVC)



„Realisierung“ dieser Kommunikation durch

Observer-Pattern: Observable = **Model** Observer = **View**

Vorteile von MVC

- Durch **entkoppelte Datenstruktur**
  - Änderungen in einem Teilbereich hat keine negativen Auswirkungen auf die anderen
- Erleichtert arbeiten in **Teams** (Vorteil fürs SEP)
  - Bessere Lesbarkeit der **Programmstruktur**
  - Nach Klärung der Struktur **Arbeitsteilung** möglich
- Durch Entwurf der Struktur wird **Programmfehlern vorgebeugt**
- Struktur verschafft einen leichten **Einstieg** für Programmieranfänger

Code-Beispiel MVC

Model

- Typische Datenklasse
  - Properties
  - Constructor
  - Getters & Setters
  - Neben equals() lassen sich z.B. auch toString() & hashcode() nach Bedarf überschreiben

```java
public class User {

    String username;

    String password;


    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(username, user.username) &&
                Objects.equals(password, user.password);
    }

    public String getUsername() { return username; }

    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }

    public void setPassword(String password) { this.password = password; }
}
```

Code-Beispiel MVC

View

- Ausschnitt aus einer .fxml
  - Definition eines Layouts (hier AnchorPane)
  - Anordnung/Größen der children (buttons, textfields etc.) werden bestimmt
    - fx:id -> Namen der Elemente im zugehörigen Controller
  - Festlegung des zugehörigen Controllers (s. erste Zeile)
  - Für Buttons wird unter „onAction“ festgelegt, welche Methode der Controller beim Klick auf die Buttons ausführen soll

```
<AnchorPane prefHeight="500.0" prefWidth="800.0" xmlns="http://javafx.com/javafx/11.0.1" xmlns:fx="http://javafx.com/fxml/1" fx:controller="sample.Controller.Controller">
   <children>
      <Button fx:id="exitButton" layoutX="274.0" layoutY="229.0" mnemonicParsing="false" onAction="#exitButtonPressed" text="Exit" />
      <Button fx:id="loginButton" layoutX="186.0" layoutY="229.0" mnemonicParsing="false" onAction="#loginButtonPressed" text="Login" />
      <TextField fx:id="usernameField" layoutX="174.0" layoutY="145.0" />
      <PasswordField fx:id="passwordField" layoutX="174.0" layoutY="187.0" />
      <Label layoutX="143.0" layoutY="78.0" text="SEP MVC Login">
         <font>
            <Font size="31.0" />
         </font>
      </Label>
      <Text layoutX="103.0" layoutY="162.0" strokeType="OUTSIDE" strokeWidth="0.0" text="Username" />
      <Text layoutX="103.0" layoutY="205.0" strokeType="OUTSIDE" strokeWidth="0.0" text="Password" />
   </children>
</AnchorPane>
```

Code-Beispiel MVC

Controller

- Ausschnitt aus der Controllerklasse
  - Mit @FXML annotierte Elemente werden entsprechend der fx:id aus der .fxml benannt
  - Elemente werden automatisch instanziiert

```
public class Controller {

    @FXML
    Button loginButton;

    @FXML
    Button exitButton;

    @FXML
    TextField usernameField;

    @FXML
    PasswordField passwordField;


    public void loginButtonPressed(){

        String username =  usernameField.getText();
        String password = passwordField.getText();

        //todo validate login data
        User user = new User(username, password);

        usernameField.setText("");
        passwordField.setText("");

        //todo start next view

        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Login");
        alert.setHeaderText(null);
        alert.setContentText(user.getUsername()+" logged in successfully");
        alert.showAndWait();

    }
```

Agenda

1. Einführung Software-Architektur
2. MVC (Model-View-Controller)
3. **Dokumentation von Software Architektur mit UML-Klassendiagrammen**
   - Klassen und Objekte
   - Assoziation und Abhängigkeit
   - Generalisierung
   - Interfaces
   - Packages
   - Modellierungstools

Klassendiagramm - Was ist das?

- **Strukturdiagramm** ist Teil der Unified Modeling Language (**UML**)
- **Strukturdiagramm dient der** grafischen Darstellung von
  - **Klassen,**
  - **Schnittstellen**
  - **deren Beziehungen**
- Notation, die **unabhängig von allen Programmiersprachen** ist
- **Abstraktion** z.B. von Ausarbeitung/Implementierung einzelner Methoden
- Strukturdiagramm hilft dabei, Quellcode und Implementierungsarbeiten zu strukturieren BEVOR diese starten

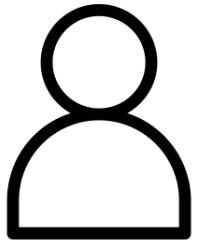
- ➔ ermöglicht eine **Aufteilung der Programmieraufgaben**

Agenda

1. Einführung Software-Architektur
2. MVC (Model-View-Controller)
3. **Dokumentation von Software Architektur mit UML-Klassendiagrammen**
   - **Klassen und Objekte**
   - Assoziation und Abhängigkeit
   - Generalisierung
   - Interfaces
   - Packages
   - Modellierungstools

Klassen und Objekte

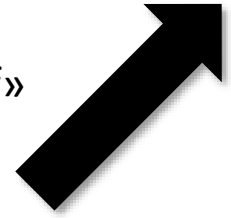
Klasse „Person“:

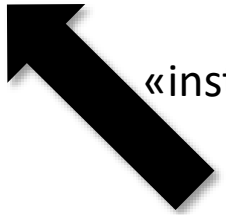


Attribute:
name
alter

Operationen:
laufen
sprechen

«instance of»





«instance of»

Objekte:

Aufruf Operation: sprechen

Attribute:
name = „Maria“
alter = 22

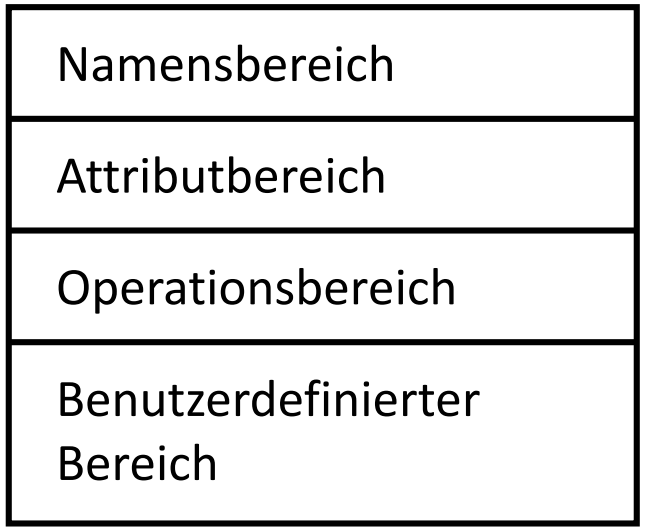
Attribute:
name = „Robert“
alter = 20

Operation: sprechen

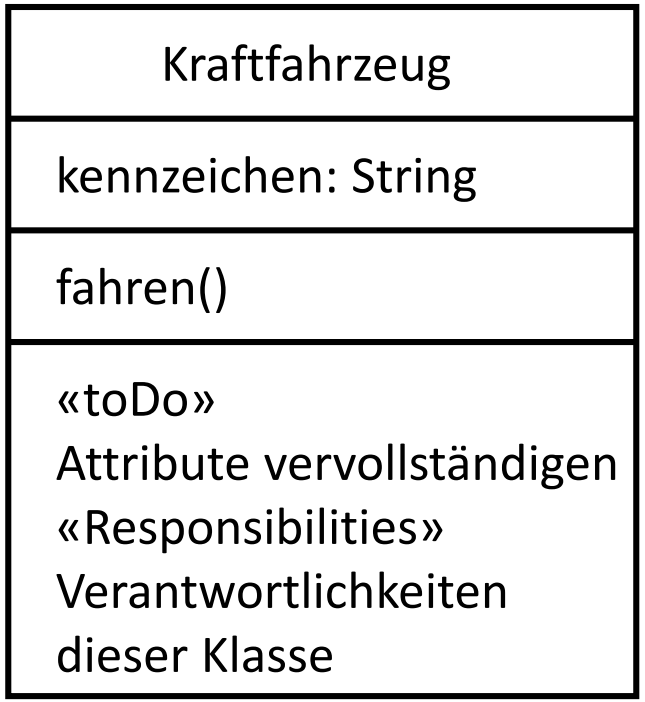
Klassendiagramm

Notation einer Klasse

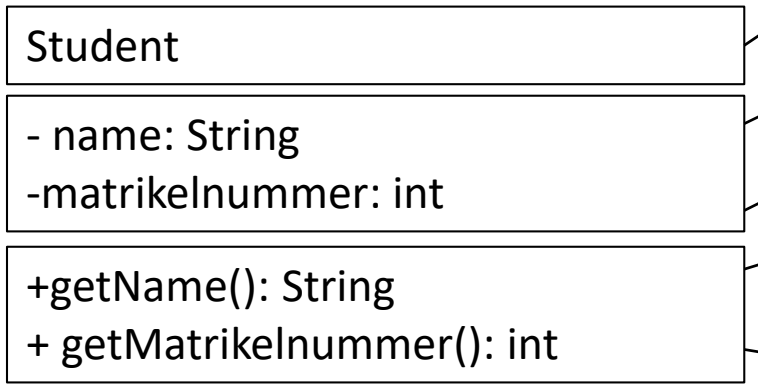
Allgemeine Notation



Beispiel



Zusammenhang: UML-Klassendiagramm und Java




```
public class Student
{
   private String name;
   private int matrikelnummer;

   public String getName()
   {
      return name;
   }
   public int getMatrikelnummer()
   {
      return matrikelnummer;
   }
}
```

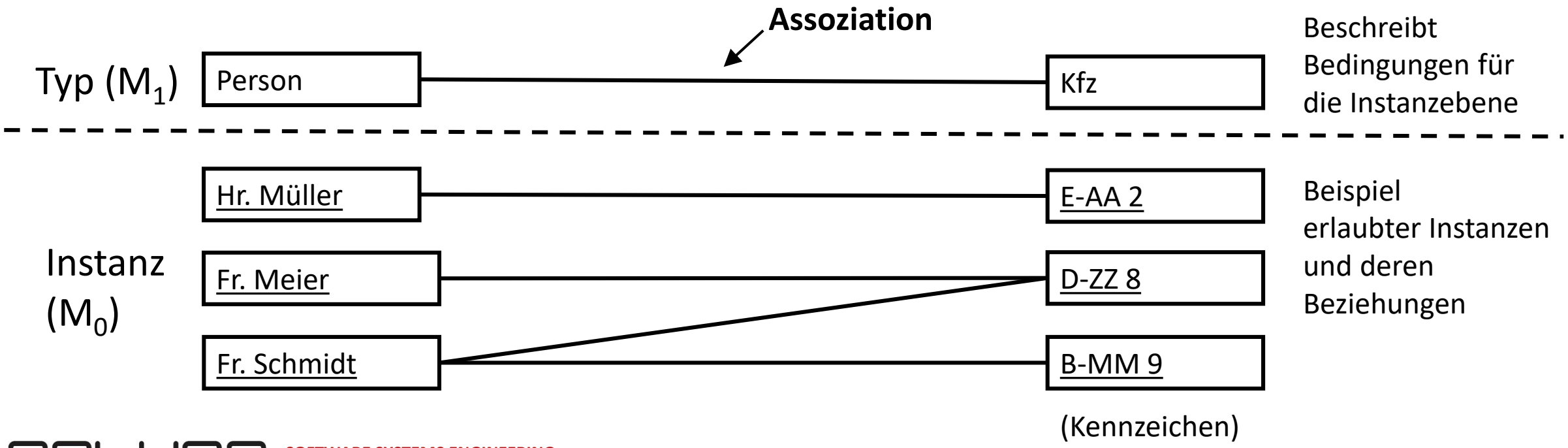
Agenda

1. Einführung Software-Architektur
2. MVC (Model-View-Controller)
3. **Dokumentation von Software Architektur mit UML-Klassendiagrammen**
   - Klassen und Objekte
   - **Assoziation und Abhängigkeit**
   - Generalisierung
   - Interfaces
   - Packages
   - Modellierungstools

Assoziation

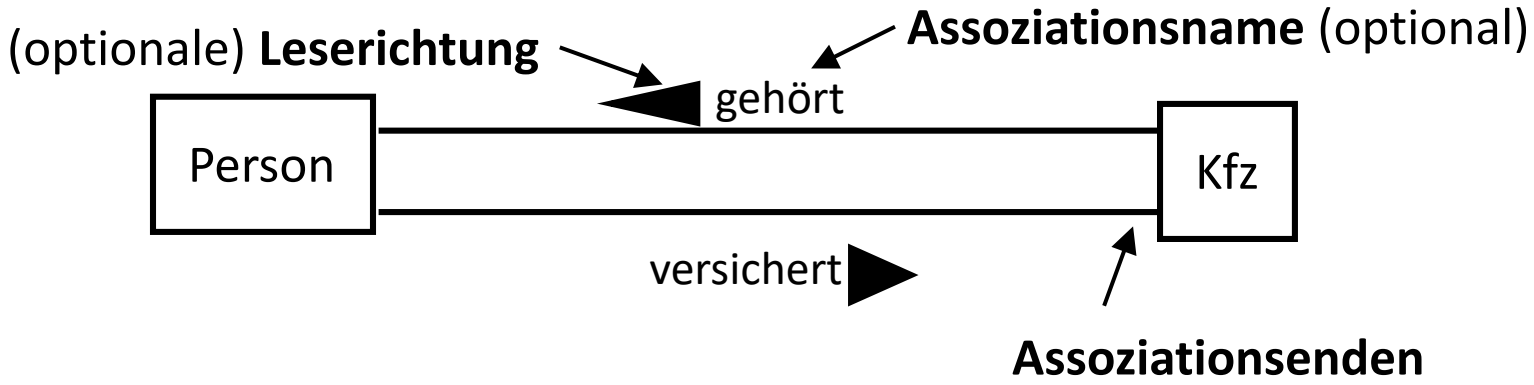
Eine Assoziation

- ist eine **Beziehung** zwischen zwei oder mehr *Klassen*
  - Daher ein *Typ,* der eine Menge von Links auf Objektebene beschreibt
- beschreibt **gemeinsame Eigenschaften** zwischen Klassen
- beschreibt die gemeinsame **Semantik und Struktur** einer Menge von **Objektbeziehungen.**



Assoziation

Modellierung



- Der Assoziationsname ist wichtig für das **Verständnis** einer Assoziation
- Die Leserichtung ist wichtig zur eindeutigen **Interpretation** der Assoziationsnamen
- An die Enden einer Assoziation können weitere **Merkmale** einer Assoziation geschrieben werden, so dass die **Semantik** einer Assoziation weiter verfeinert wird

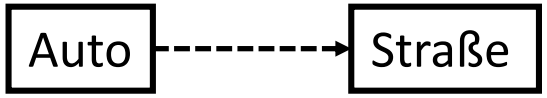
Assoziation

Assoziationsende - Multiplizität

- Multiplizität legt fest, wie viele Objekte der einen Klasse mit einem Objekt der anderen Klasse **in Beziehung stehen** können (→Aussage auf der **Instanzebene**).
  - jedes Rad gehört zu null oder einem Kraftfahrzug
  - Jedes Kraftfahrzeug kann null bis „unendlich“ viele Räder besitzen
- Notationen:
  - min..max ➔ z. B. 2..3; 3..10
  - x..x oder x ➔ genau x, z.B. 2..2, 10..10
  - min..* ➔ min bis unendlich oft, z.B. 3..*
  - 0..* oder * ➔ 0 bis unendlich oft

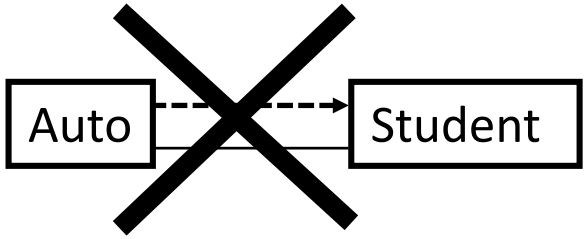
Abhängigkeit

Modellierung



- Abhängigkeit
  - wird durch einen **gestrichelten Pfeil** modelliert
  - Pfeil zeigt von der abhängigen Klasse auf die unabhängige

- **Achtung:** Assoziationen implizieren Abhängigkeit!
  - Wenn eine Assoziation zwischen zwei Klassen modelliert wurde, muss **nicht zusätzlich** noch eine Abhängigkeit modelliert werden.



Agenda

1. Einführung Software-Architektur
2. MVC (Model-View-Controller)
3. **Dokumentation von Software Architektur mit UML-Klassendiagrammen**
   - Klassen und Objekte
   - Assoziation und Abhängigkeit
   - **Generalisierung**
   - Interfaces
   - Packages
   - Modellierungstools

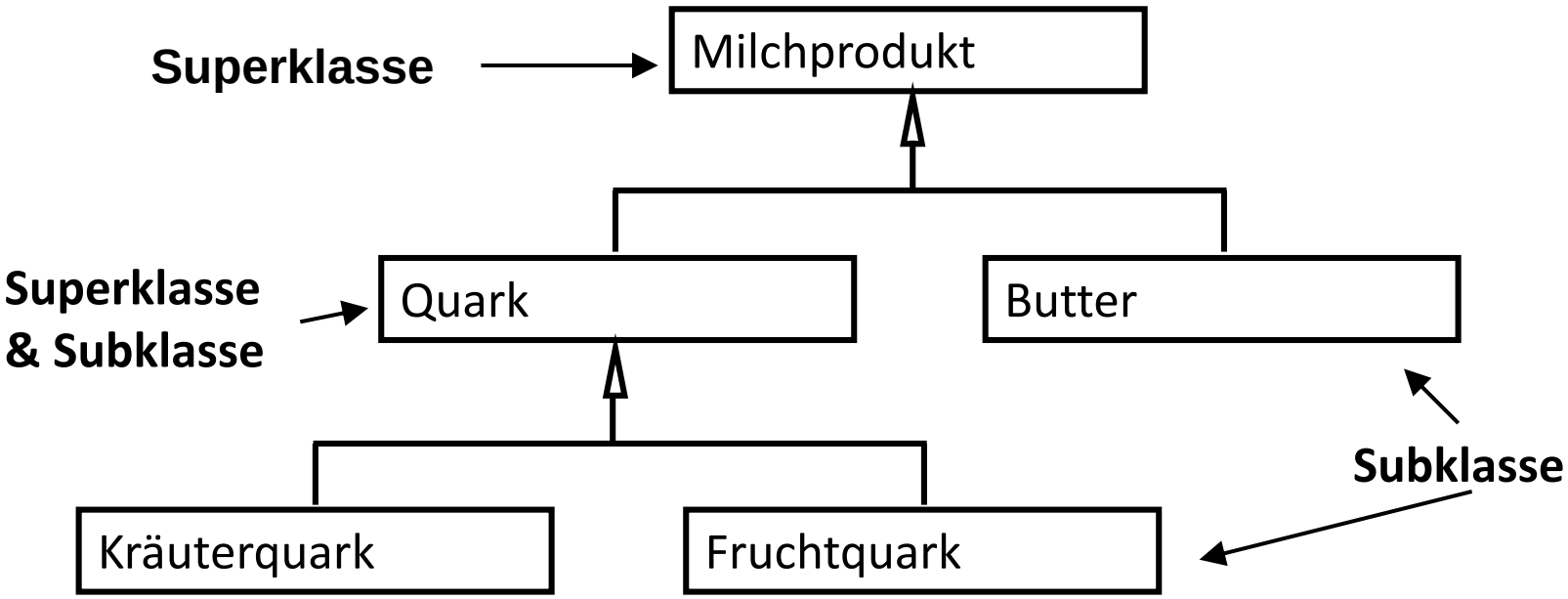
Generalisierung

- Generalisierung
  - fasst **gemeinsame** Merkmale verschiedener Klassen zusammen
  - beschreibt eine „ist ein“-Beziehung von einer **speziellen** zu einer **generellen** Klasse.
  - Gegenteil: Spezialisierung

- Realisierung/Anwendung: „**Vererbung**“
  - **speziellere Klassen erben** Eigenschaften der generelleren Klassen
  - Anwendung:
    - **Vereinfachung** der Modelle/Wiederverwendung
    - Polymorphie

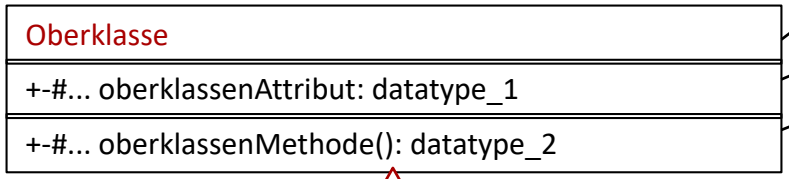
Generalisierung

Super- und Subklasse

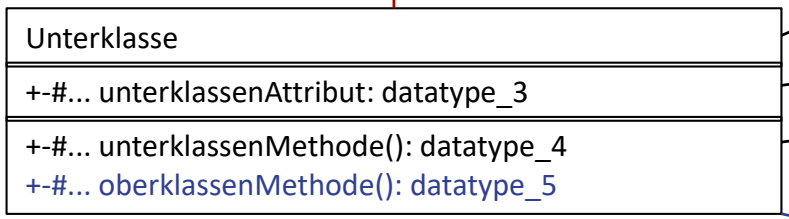


- Eine Superklasse definiert Attribute, Operationen und Assoziationen.
- Eine Subklasse erbt Eigenschaften der Superklasse.
- Eine Subklasse kann die geerbten Eigenschaften redefinieren und neue hinzufügen.

Vererbung im Klassendiagramm



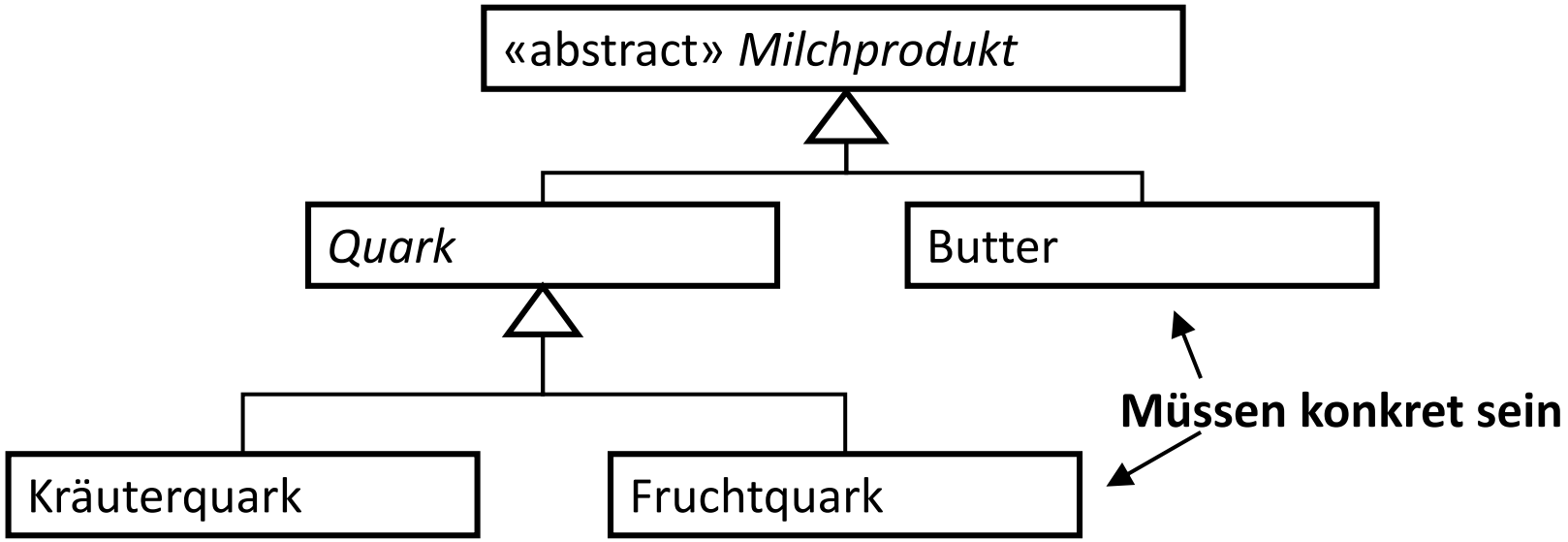

```
public class Oberklasse
{
    accessModifier_1 datatype_1 oberklassenAttribut;
    accessModifier_2 datatype_2 oberklassenMethode()
    {
        ...
    }
}
```




```
public class Unterklasse extends Oberklasse
{
    accessModifier_3 datatype_3 unterklassenAttribut;
    accessModifier_4 datatype_4 unterklassenMethode()
    {
        ...
    }
    @Override
    accessModifier_5 datatype_5 oberklassenMethode()
    {
        ...
```

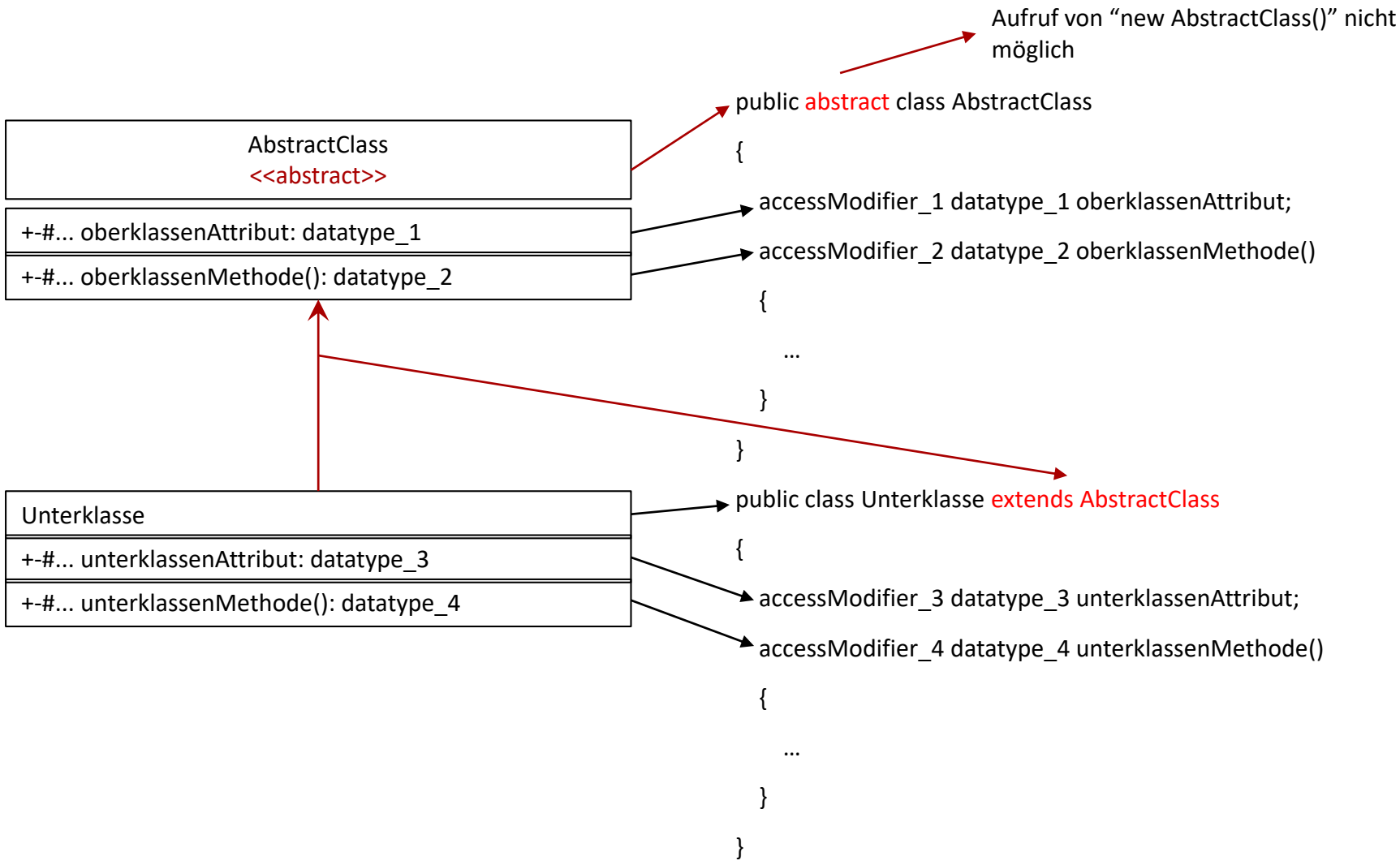
Generalisierung

Abstrakte und konkrete Klasse



- Abstrakte Klassen werden entweder durch die Angabe von «**abstract**» oder durch **kursive** Schrift kenntlich gemacht.
- Die Klassen der **untersten** Ebene der Vererbungshierarchie müssen konkret (d.h. nicht abstrakt) sein.

Abstrakte Klassen



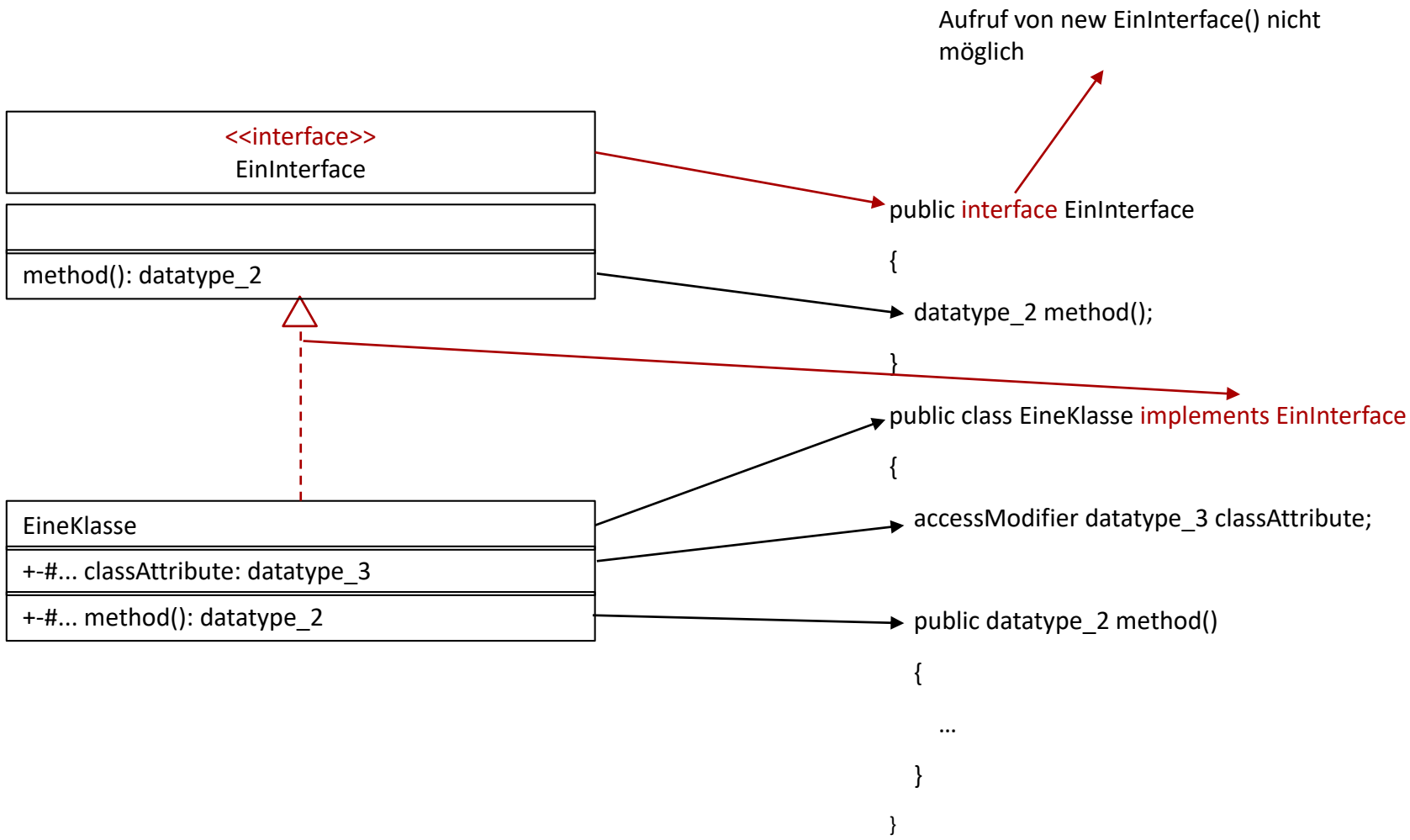
Agenda

1. Einführung Software-Architektur
2. MVC (Model-View-Controller)
3. **Dokumentation von Software Architektur mit UML-Klassendiagrammen**
   - Klassen und Objekte
   - Assoziation und Abhängigkeit
   - Generalisierung
   - **Interfaces**
   - Packages
   - Modellierungstools

Interfaces

- Geben vor, **welche Methoden** in einer Klasse implementiert werden **MÜSSEN**
- Interfaces können nur Methodenrümpfe und Konstanten enthalten
- Interfaces sind **keine Klassen**
- Analog zu abstrakten Klassen können von Interfaces keine Objekte erstellt werden

Interfaces

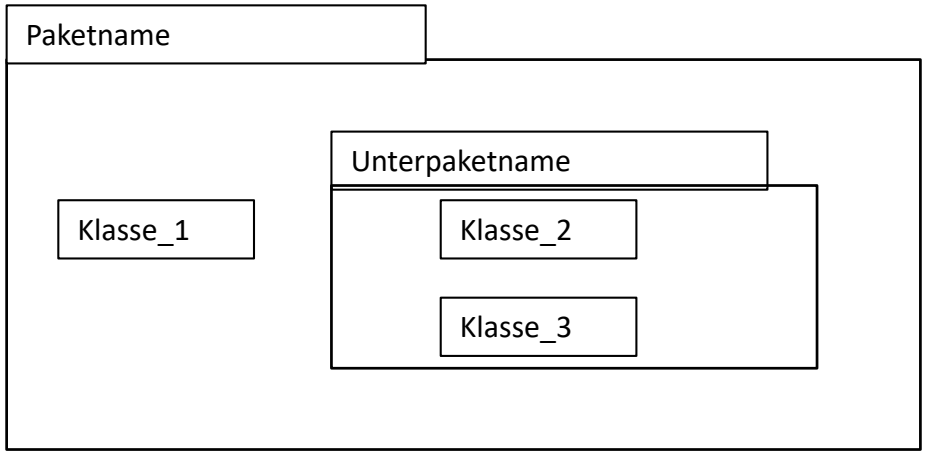


Agenda

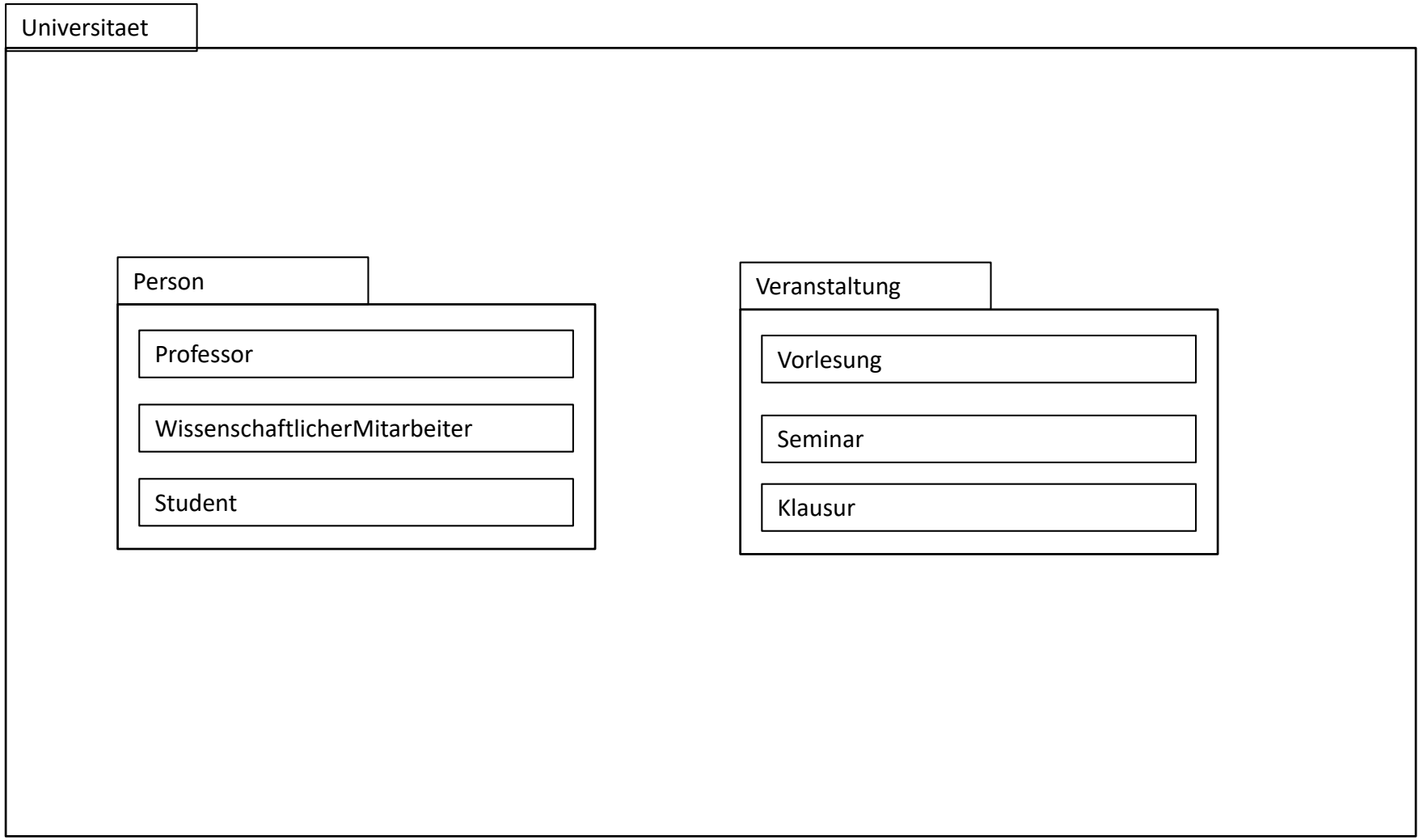
1. Einführung Software-Architektur
2. MVC (Model-View-Controller)
3. **Dokumentation von Software Architektur mit UML-Klassendiagrammen**
   - Klassen und Objekte
   - Assoziation und Abhängigkeit
   - Generalisierung
   - Interfaces
   - **Packages**
   - Modellierungstools

Packages

- Jedes Package (zu deutsch: “Paket”) kann mehrere Klassen, Interfaces oder weitere Packages enthalten
- Ermöglichen **hierarchische Gliederung der Klassen**



Beispiel



Agenda

1. Einführung Software-Architektur
2. MVC (Model-View-Controller)
3. **Dokumentation von Software Architektur mit UML-Klassendiagrammen**
   - Klassen und Objekte
   - Assoziation und Abhängigkeit
   - Generalisierung
   - Interfaces
   - Packages
   - **Modellierungstools**

Modellierungstools für UML-Klassendiagramme

- Zur grafischen Modellierung von UML Klassendiagramen empfehlen wir **Microsoft Visio** oder die Webseite: **https://www.draw.io/**

- Eine Studentenlizenz für **Microsoft Visio** sowie das Programm zum Herunterladen erhaltet Ihr kostenlos über **DreamSpark Premium**:
  - Anmeldung mit eurer Nutzerkennung der UDE
  - Weitere Infos findet ihr unter: https://www.uni-due.de/zim/services/software/msdnaa/

- Weitere Folien zu UML Klassendiagrammen findet Ihr als Referenz im Anhang dieses Foliensatzes

Quellen

- Object Management Group: Unified Modeling Language Specification, Version 2.5, Juni 2015; Abrufbar unter www. omg.org.
- Tom Pender: UML Bible; John Wiley & Sons; Indianapolis 2003.
- Hans-Erik Erikson et al: UML2 Toolkit; Addison-Wesley, München 2004.
- Mario Jeckle et. al.: UML2 Glasklar; Hanser, München 2001.
- Laurent Doldi: UML2 Illustrated. TMSO 2003.
- Martin Fowler: UML Distilled, 3rd edition; Addison-Wesley, Boston 2004.
- http://openbook.rheinwerk-verlag.de/oo/oo_06_moduleundarchitektur_001.htm
- [Parnas 1972] D.L. Parnas: On the Criteria to Be Used in Decomposing Systems into Modules.” CACM 15(12):1053-1058, 1972.
- [Buschmann et al. 2001] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.: Pattern Oriented Software Architecture. John Wiley & Sons, New York, USA, 2001.

Quellen

- http://www.datenbanken-verstehen.de/lexikon/model-view-controller-pattern/

Verwendete Grafiken

- Grafiken von https://thenounproject.com/
  - Account by Gregor Cresnar
  - Account by Yaroslav Samoylov

UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Vielen Dank für Eure Aufmerksamkeit

PALUNO
The Ruhr Institute for Software Technology