

Interfaces und abstrakte Klassen in OO

Irene Rothe
irene.rothe@h-brs.de



Planung

- ✓ Einstiegsbeispiele: Swimmingpool (Waschmaschine), Buchaufbau, Schiffe versenken
- ✓ Klasse: Datei mit Eigenschaften (Attributen) und Fähigkeiten (Methoden) möglicher Objekte
- ✓ **OO-Basics**: Abstraktion, Kapselung, Vererbung, Polymorphie
- ✓ IDEs: Eclipse, javac-Editor, IntelliJ, Visual Studio Code

→ Abstrakte Klassen und Interfaces

→ OO-Prinzipien:

- Kapseln, was sich ändert
- Programmieren auf Schnittstelle
- Schwache Koppelung und starke Kohäsion
- Subklassen sollten ihre Superklasse vertreten können

→ UML

→ Entwurfsmuster




→ OOA

→ Heuristiken

→ Parallelprogrammierung mit Java

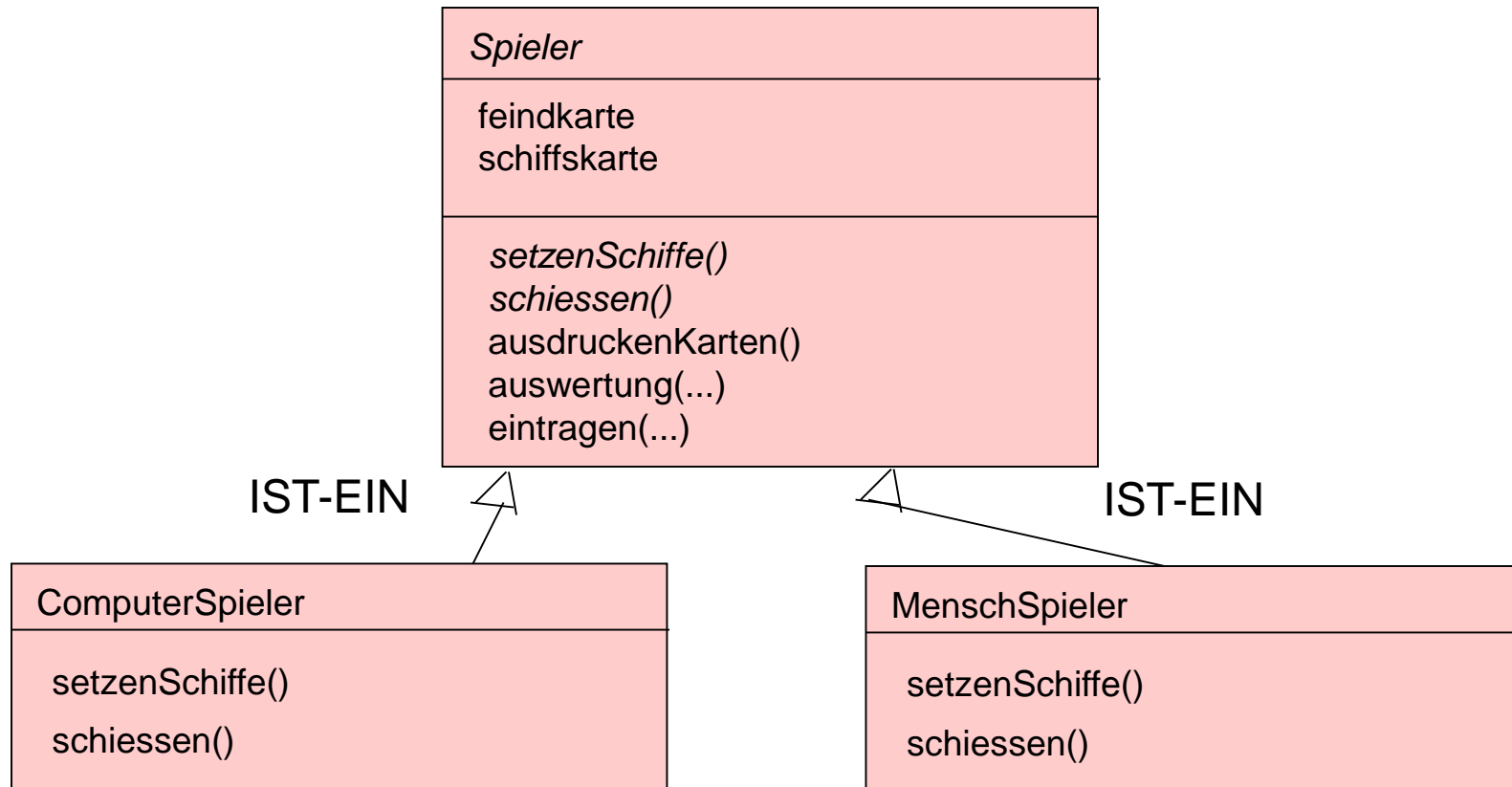


Design der Folien

-  hinterlegte Informationen sind sehr wichtig und klausurrelevant.
- Alles hinter „**Achtung**“ unbedingt beachten!
-  verwende ich, wenn Überraschende Probleme auftreten können.
- „Tipp“ benutze ich, um Ihnen einen Weg zu zeigen, wie ich damit umgehen würde.
- „Bemerkung“ in Folien beziehen sich meist auf Sonderfälle, die nicht unbedingt klausurrelevant sind, aber für Sie beim Programmieren eine Bedeutung haben könnten.
-  hinter diesem Symbol ist ein Link fürs Anhören bzw. Gucken weiterer Infos

Schiffe versenken mit abstraktem Spieler

Motivationsbeispiel



UML (Unified Modeling Language)

Abstrakte Klasse: Fakten

Fakten:

- kann nicht zum Erzeugen von Objekten genutzt werden
- nur fürs Vererben gedacht
- wenn auch nur *eine* Methode abstrakt ist (d.h. sie existiert nur als Dummy-Rumpf), muss die Klasse auch abstrakt sein
- Unterklassen müssen *alle* abstrakten Methoden mit Inhalt füllen, wenn sie *instanzierbar* werden wollen

Verwendungszweck/Vorteil:

- drückt aus, dass Superklasse (Oberklasse) keine Ahnung von der Implementation hat (und haben will), darum soll sich die Subklasse (Unterklasse) kümmern
- Möglichkeit der Definition einer Klasse, in der beschrieben ist, was man tun *müsste*, ohne es wirklich zu tun

UML: Klassenname *kursiv*, ebenso alle abstrakten Funktionen *kursiv*

In **C++**: mindestens eine Methode muss virtuell sein, d.h. =0



Abstrakte Klasse: Aufbau

```
abstract class Klassenname {  
    ...  
    //abstrakte Methode  
    abstract void abstrakteMethode() ;  
    //konkrete Methode  
    void Methode() { }  
    . . .  
}
```

Interface Motivation

Attribut der
Quellklasse namens
spielverhalten vom
Typ Spielverhalten

spielverhalten *

HAT-EIN

HAT-EIN

IST-EIN

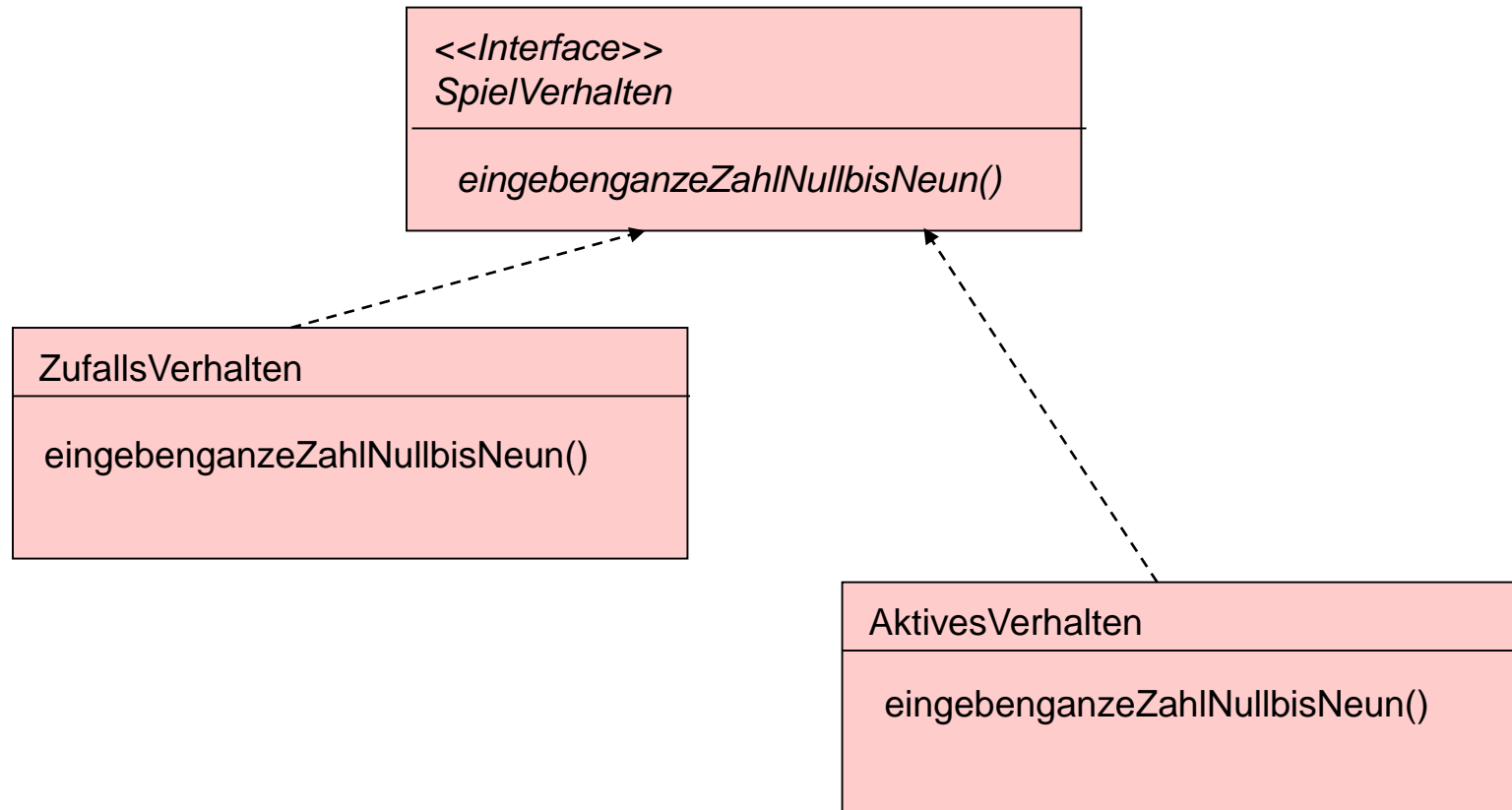
IST-EIN

Kapselungen

Bemerkung: Variable spielverhalten wird im ComputerSpieler ... zugewiesen.

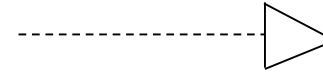


Interface: Beispiel zum Spielverhalten



UML (Unified Modeling Language)

Interface: Fakten



Fakten:

- haben keine Attribute, keine gefüllten Methoden/Funktionen
- sind nicht instanziiierbar,
- sind nicht gedacht zum Aufbau einer Klassenhierarchie sondern zur „Vererbung“ von Algorithmen,
- definieren eine Menge von *Methodenspezifikationen* (also Methoden ohne Inhalt)
- Klassen *realisieren* ein oder mehrere Interfaces

Vorteile:

- Sicherung einer *Garantie*: jede Klasse, die das Interface implementiert, stellt *garantiert* all diese Methoden zur Verfügung
- stellt sich dar als *Idee*, wovon dann verschiedene Implementierungen umgesetzt werden können
- Interface-Hierarchien anlegbar

In Java: implements

In C++: spezielle abstrakte Klassen: nur virtuelle Funktionen



Interface: Aufbau

```
interface Interfacename{  
  //Methodenspezifikationen (Dummy-Rümpfe)  
  ...  
}
```

Anwendung eines Interfaces:

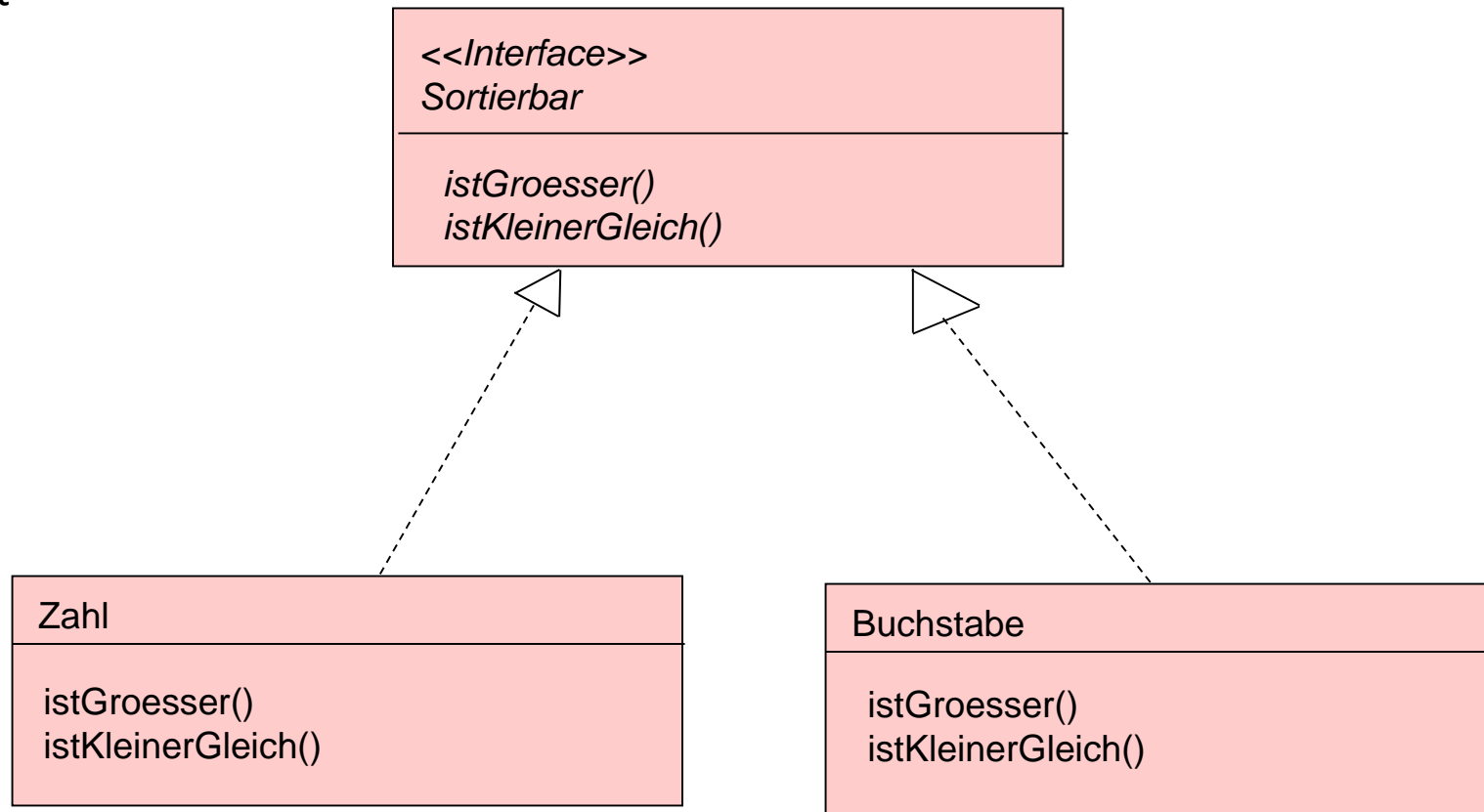
```
class Klassenname implements Interfacename{  
  //Methode aus dem Interface wird mit Code gefüllt  
  ...  
}
```

Beispiel: Sortieren

Ziel: Man möchte verschiedene Dinge mit demselben Sortieralgorithmus sortieren

Idee: ein Ding ist größer als ein anderes

Umsetzung: spezifizieren der verschiedenen Arten, wann ein Ding größer als ein anderes ist



Interfaces: eine Art der „Mehrfach“-Vererbung

In Java kann eine Klasse mehrere Interfaces definieren:

```
class Klassenname implements Schnittstelle1,  
Schnittstelle2, ... {  
...  
}
```

Eine Klasse kann nur *eine* **Superklasse** haben. Sie kann aber *mehrere* **Interfaces** implementieren.

Interface: können Konstanten enthalten

→ Konstanten-Interface

Vorteile:

- Man muss Konstanten nur einmal im System pflegen
- Möglichkeit der Austauschbarkeit
- man kann auch hier mehrere Konstanten-Interfaces implementieren
- Sichtbarmachung im UML
- Im Code wird sichtbar durch Punktoperator, woher Konstanten kommen

Beispiel:

```
interface Erdaehnlich {  
    public static final double ERDANZIEHUNG = ErdfallbeschleunigungInNProMeter = 9.81;  
}
```

Übung

CD-Player, DVD-Player, CD-Recorder und DVD-Recorder sollen in OO abgebildet werden.

Bemerkung: Recorder können auch abspielen.

Übung

UML für Mastermind

