

# Objektorientierte Analyse & Design



von

**Irene Rothe**

[irene.rothe@h-brs.de](mailto:irene.rothe@h-brs.de)



# Planung

- ✓ Einstiegsbeispiele: Swimmingpool (Waschmaschine), Schiffe versenken
- ✓ Klasse: Datei mit Eigenschaften (Attributen) und Fähigkeiten (Methoden) möglicher Objekte

✓ **OO-Basics:** Abstraktion, Kapselung, Vererbung, Polymorphie

✓ IDEs: Eclipse, javac-Editor, IntelliJ

✓ 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

# Das objektorientierte Paradigma

Der Vertrag eines Algorithmus mit seinem Nutzer ist wie ein *Verkaufsvertrag*, der für jede Eingabe eine Ausgabe garantiert.

Der Vertrag eines Objektes mit seinem Nutzer ist wie ein *Ehevertrag*, der das Verhalten über die Lebenszeit hinweg festlegt.

# Objektorientierung

- Das heißt, zusammengehörige Daten und Funktionen sollten zusammengefasst werden (in einer **Klasse**) und weitgehend vor der Welt verborgen werden (Verkapselung/Black Box-Prinzip). Bei der Ausführung entstehen **Objekte** = Instanzen von Klassen.
- Wie man entscheidet, welche Objekte günstig sind, was man nach außen zeigt und was man versteckt, ist die Aufgabe von OO-Programmierern.
- Darstellungswerkzeug: UML (Unified Modelling Language)

# Wiederholung - OO-Basics

- **Abstraktion**
- **Kapselung:** nur über Methoden können Attribute geändert werden, es interessiert nicht, wie eine Methode ihre Arbeit macht
- **Vererbung:** Erweiterung von Klassen, Vererbung von Typ und Implementation
- **Polymorphie:** Vielgestaltigkeit

# Abstraktion durch Klassen - genauer

## Fakten:

- Eine Klasse hat
  - Name
  - Eigenschaften: Attribute
  - Methoden/Funktionen
  - Zugriffsattribute: public, private, (protected)
- Aufruf von Methoden (und Attributen) über Punktoperator (oder Pfeiloperator und Zeigern bei C++)

## Vorteile:

- Grad der Komplexität soll gesenkt werden
- Grad der Wiederverwendbarkeit soll erhöht werden

**Regel/Tipp:** eine Klasse – eine Aufgabe

# Klassen

## Fakten:

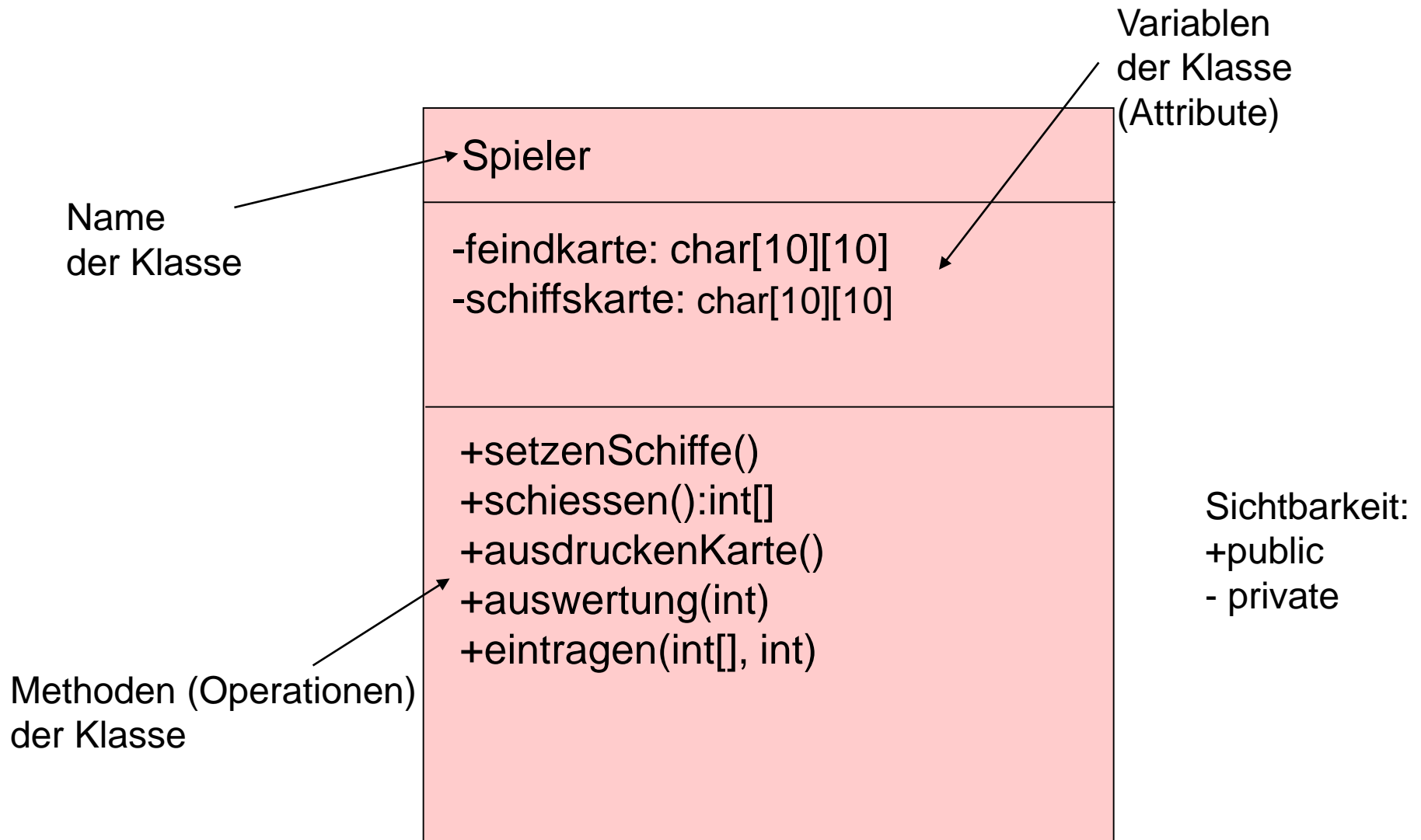
- Klassen besitzen Daten=Eigenschaften und Fähigkeiten=Methoden,
- Funktionen/Methoden sind dafür da, dass man die Daten verändern kann, Methoden beschreiben das Verhalten der Klasse/Objekt. Es gibt **getter** und **setter** Methoden.
- Klassen werden oft in eigener Textdatei abgespeichert.
- Teile der Klassen können durch Modifizierer wie **public**, **private**, **protected** spezifiziert werden.
- Klassen können hierarchisch geordnet werden (Unter-/Subklasse erben die Eigenschaften der Ober-/Superklasse),
- Objekte sind real existierende virtuelle Exemplare einer Klasse (werden erzeugt) und heißen *Instanzen einer Klasse*.

## Verwendungszweck/Vorteil:

- Klassen sollten korrelieren zu realen Dingen.
- Klassen fassen Daten und Funktionen zu einer Einheit zusammen → „Füge zusammen, was zusammen gehört“ (Nachteil von C: kein direkter Bezug zwischen Daten und Funktionen) → *gekapselte* Datenstruktur

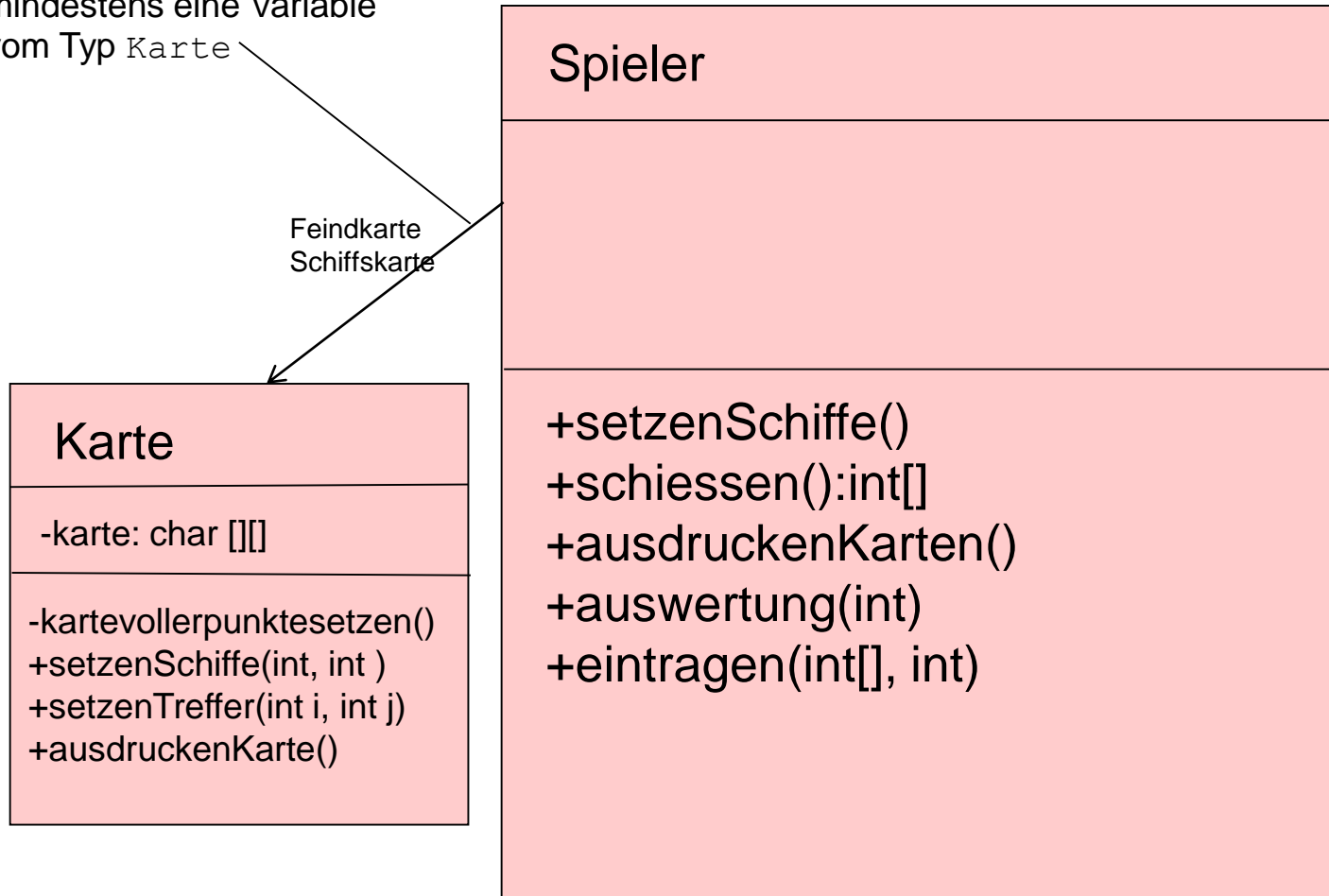


# Klassendiagramm in UML - Beispiel



# Klassendiagramm in UML - verbessertes Beispiel

Assoziation: `Spieler` hält  
mindestens eine Variable  
vom Typ `Karte`



# Datenkapselung - genauer

## Fakten:

- Klassische Programmierung hat das Problem des mangelnden Schutzkonzeptes.  
**neu:** public, privat (unauthorisierter Zugriff von außen ist nicht möglich)
- Es ist nur wichtig, zu wissen, WAS ein Objekt kann, nicht wie es das intern durchführt.

## Vorteile:

- Kompetenz und Verantwortung sind im Objekt konzentriert und nicht über das ganze Programm *“verschmiert”*
- Kapselung ist gut für große Projekte, wo das Hauptproblem ist, den Überblick zu behalten.
- Einfachere Testung ist möglich, da Klassen einzeln getestet werden können.

# Kapselung - Genauer

- Datenveränderung sollten nur über Methoden stattfinden können.
- Methoden nur für *die* Daten anbieten, die von außen verändert werden sollten.
- Kapseln von Algorithmen ist möglich über Methoden, die andere Methoden kompliziert aufrufen (scanf aus C?)
- Keine großen **switch**-Konstrukte sind mehr nötig. Das geht besser über Klassen und Polymorphie.

# Vererbung (IST-EIN) - Genauer

## Fakten:

- Vererbung drückt IST-EIN Beziehung aus.
- Subklasse erbt alle Attribute und Operationen der Superklassen *und* deren Implementierungen.
- In die eine Richtung der Vererbung wird *spezialisiert* in die andere *generalisiert*.

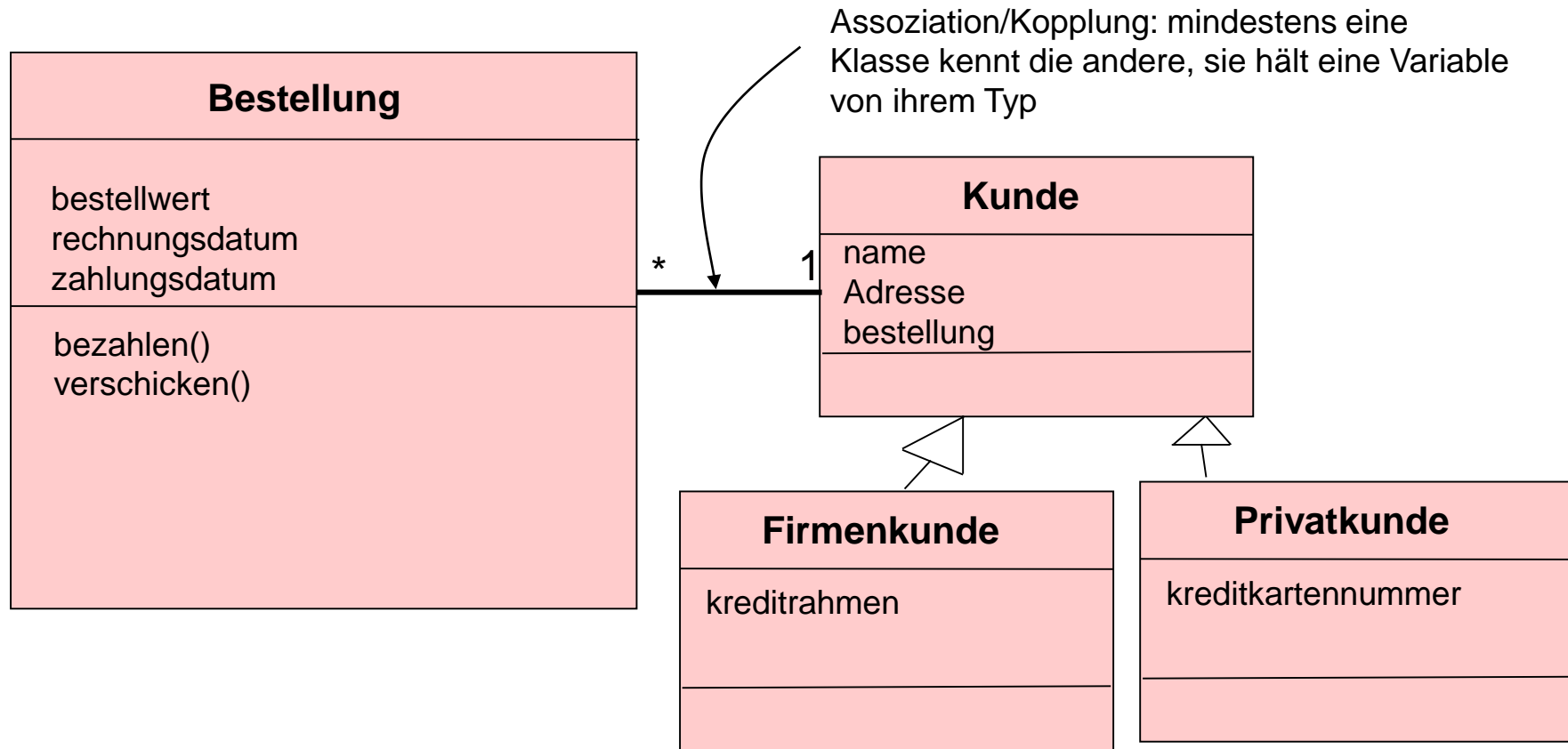
## Vorteile:

- Copy & Paste sind nicht nötig für Wiederverwendung von Programmcode.
- Doppelter Code wird einfach in eine Superklasse gesteckt.
- Falls Änderungen nötig sind, müssen diese nur an einer Stelle implementiert und dann eventuell geändert werden (und wie durch Zauberhand reagieren alle abgeleiteten Klassen darauf).

# Beziehungen in UML

Assoziation bedeutet „benutzt“-Beziehung 

Vererbung bedeutet „ist-ein“-Beziehung 



## Bemerkung zu `private`, `protected`, `public`

- `protected` in C++, so wie man sich das vorstellt
- In Java bedeutet `protected` öffentlich fürs ganze Paket, aber `private` für alle Klassen außerhalb vom Paket

# Hilfsmittel: Interfaces und abstrakte Klassen - Unterschiede

- Abstrakte Klasse haben *einige* Methoden, die nicht implementiert sind.
- Interfaces haben *nur* Methoden, die nicht implementiert sind. Eine Klasse kann *mehrere* Interfaces implementieren.

Wann sollte man was benutzen?

- Abstrakte Klasse sollte als eine Art *Schablone* verwendet werden.
- Interface (spezielle abstrakte Klasse) sollte als eine Art *Rolle* verwendet werden.



# Polymorphie (Vielgestaltigkeit) - genauer

## Fakten:

- *Gleiche* Botschaft wirkt in *verschiedenen* Zusammenhängen verschieden.
- Variablen können abhängig von ihrer Verwendung unterschiedliche Typen annehmen.
- Polymorphie tritt im Zusammenhang mit Vererbung und Schnittstellen (Interfaces) auf.
- Eine *Methode* ist polymorph, wenn sie in verschiedenen Klassen die gleiche Signatur hat, jedoch erneut implementiert ist.

## Vorteil:

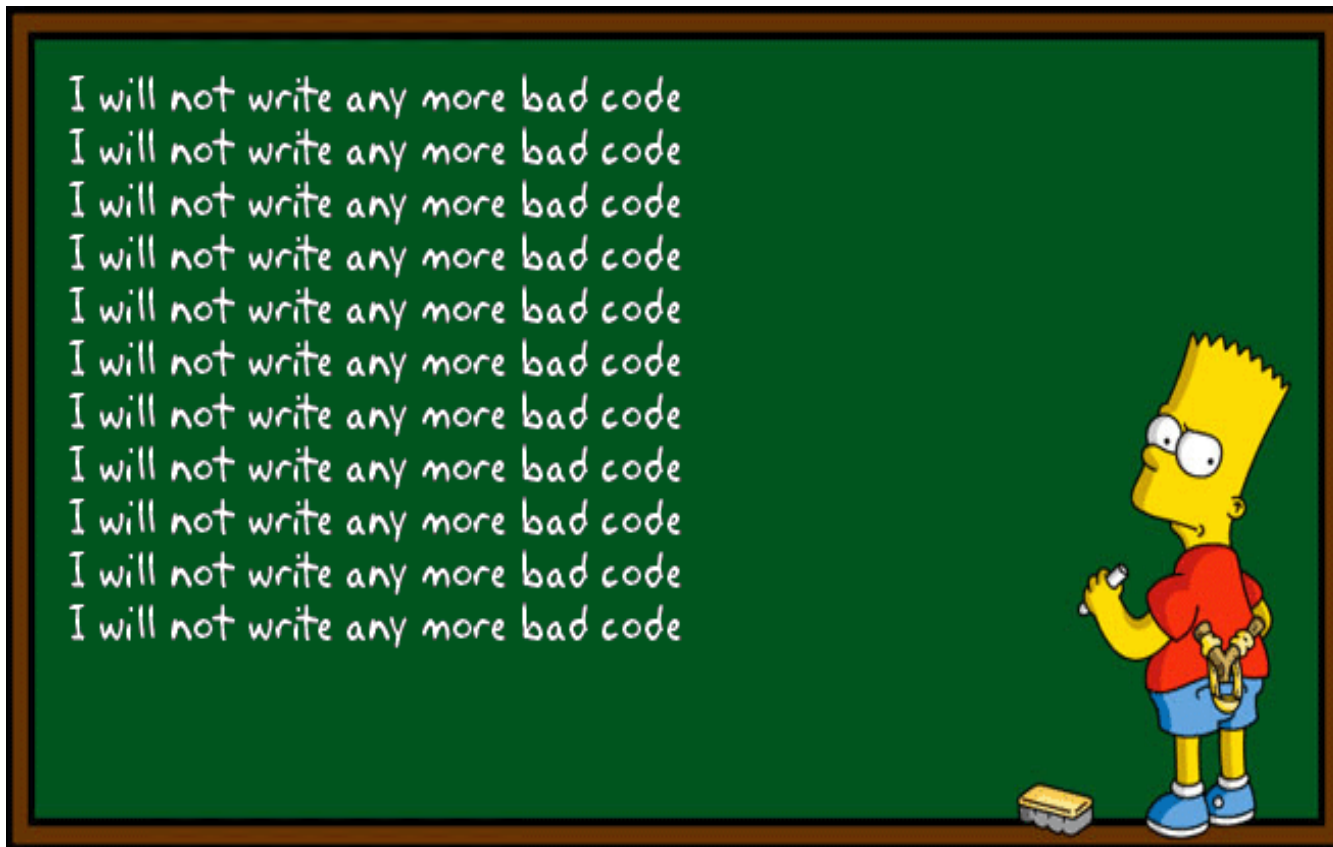
- *Dynamisches Binden* ist möglich: Gibt es in einem Vererbungsweig einer Klassenhierarchie mehrere Methoden auf unterschiedlichen Hierarchieebenen, jedoch mit gleicher Signatur, wird erst zur Laufzeit bestimmt, welche der Methoden für ein gegebenes Objekt verwendet wird (→ die ganz „unten“).

# Gute Software...

- macht, was der Kunde möchte
  - hat keinen doppelten Code
  - ist leicht erweiterbar
  - ist wiederverwendbar
- Anwendung erprobter Prinzipien und Entwurfsmuster

→ Wer die nicht einhält, kommt direkt in die Hölle (die Hölle der Programmierer)





# OO-Prinzipien

- **Kapseln, was sich ändert** (Klassen für Erweiterung offen halten, aber für Veränderung abgeschlossen)
- **Programmieren auf Schnittstelle** (Nutzung von Interfaces und abstrakten Klassen, Programmierung auf Supertyp und nicht auf Implementation)
- **Schwache Koppelung** (Änderung einer Klasse zieht keine Änderung anderer Klassen nach sich) und **starke Kohäsion** (eine Aufgabe=eine Klasse), z.B. *Heuristik*: Komposition (HAT-EIN) besser als Vererbung (IST-EIN)
- **Subklassen sollten ihre Superklasse vertreten können** (beide Klassen sind dann *konform*)

→ Wer die nicht einhält, kommt direkt in die Hölle (die Hölle der Programmierer)



# Prinzip: Kapseln, was sich ändert - genauer

## Fakt:

- *konstanter* Code geht in Superklasse, *verändertes Verhalten* kann z.B. in ein Interface oder abstrakte Klasse gehen

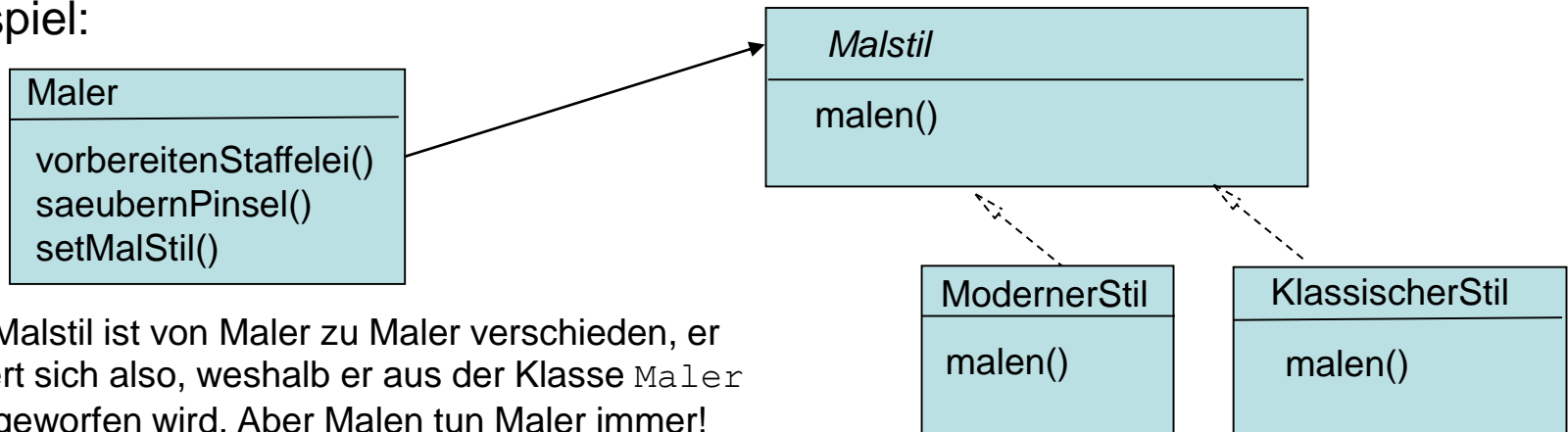
## Vorteile:

- Trennung von sich *oft ändernden* und sich *nicht so oft ändernden* Inhalten
- Klassen schützen vor **unnötigen** Veränderungen

→ **Offen für Erweiterungen:** Man kann die Klasse erweitern und sie dann ähnlich benutzen.

→ **Geschlossen für Änderungen:** Die Klasse kann nicht für komplett andere Dinge benutzt werden (es kann nix „umgebogen“ werden).

## Beispiel:



Der Malstil ist von Maler zu Maler verschieden, er ändert sich also, weshalb er aus der Klasse `Maler` rausgeworfen wird. Aber Malen tun Maler immer!

# Prinzip: Programmieren auf Schnittstelle - genauer

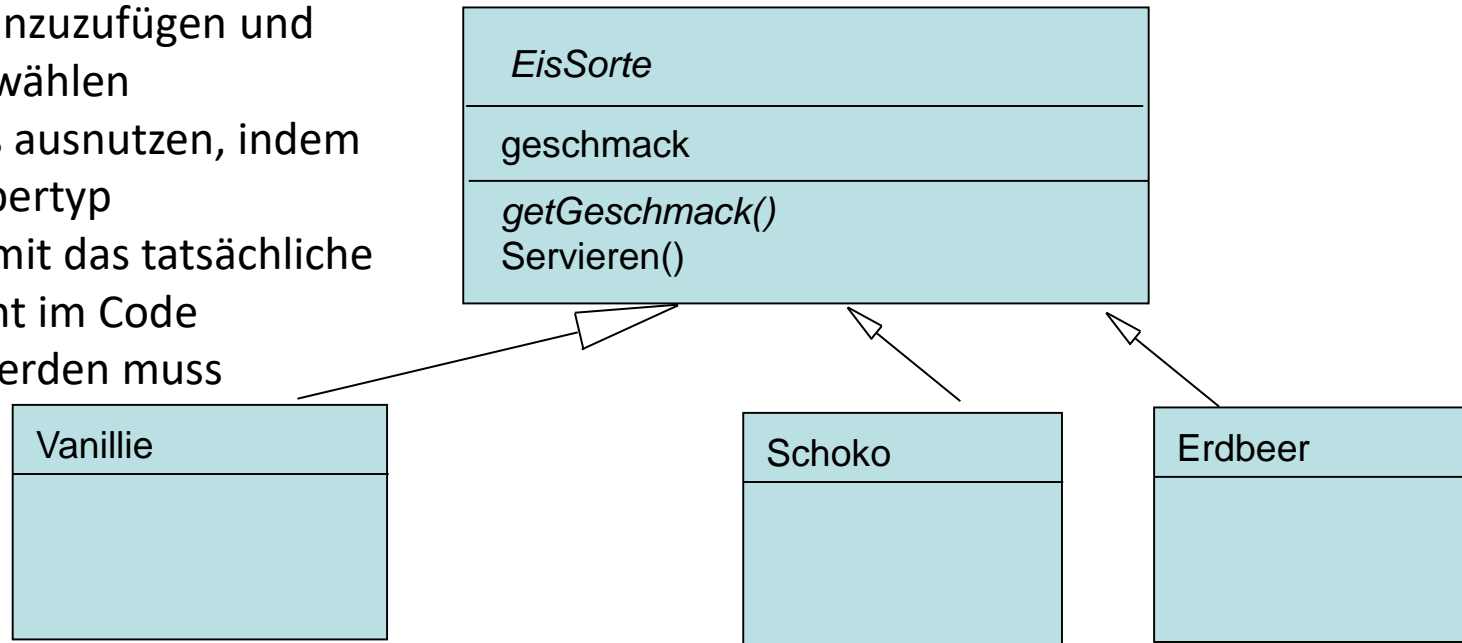
## Fakt:

- Programmierung eines Eisbechers nicht auf einen *speziellen* Erdbeereisbecher bezogen, sondern auf Schnittstelle *EisSorte*

## Vorteile:

- es ist sehr einfach, weitere Eissorten *später* hinzuzufügen und zur Laufzeit auszuwählen
- Polymorphismus ausnutzen, indem man auf einen Supertyp programmiert, damit das tatsächliche Laufzeitobjekt nicht im Code festgeschrieben werden muss

## Beispiel:



Anderes Beispiel: *Spielverhalten* in Schiffeversenken, Tierarzt spritzt alle unter Klasse Tier (abstrakt)

# Prinzip: Schwache Kopplung und starke Kohäsion - genauer 1

## Fakten:

- **Kopplung** beschreibt die Beziehungen zwischen Klassen
  - Typ der einen Klasse tritt in der anderen Klasse auf als Parametertyp in einer Operation (z.B. **new**)
  - Instanzen der einen Klasse werden in der anderen Klasse erzeugt
  - zwischen den Klassen existiert eine Generalisierung (Vererbung)
  - Interface, das von einer Klasse implementiert ist
- **Kohäsion** beschreibt den logischen Zusammenhang der Klasse und der zu realisierenden Aufgaben

## Vorteile:

- schwache Kopplung: Änderung einer Klasse zieht keine Änderung anderer Klassen nach sich
- starke Kohäsion: eine Aufgabe=eine Klasse

**Heuristik:** Komposition (HAT-EIN) ist besser als Vererbung (IST-EIN)



# Schwache Kopplung und starke Kohäsion - genauer 2

## **Starke Kohäsion:**

- System A darf keine Information und Funktionalität enthalten, die zum Zuständigkeitsbereich vom System B gehört und umgekehrt.

## **Schwache Kopplung:**

- es muss möglich sein, System A weitgehend auszutauschen oder zu verändern, ohne System B zu verändern.
- Änderungen von System B sollten nur möglichst einfache Änderungen in Subsystem A nach sich ziehen.

Siehe: [http://openbook.rheinwerk-verlag.de/oop/oop\\_kapitel\\_03\\_001.htm](http://openbook.rheinwerk-verlag.de/oop/oop_kapitel_03_001.htm)  
(Prinzip 1 und 2)



# Übung: Ist das gut?

```
class A {  
    //macht etwas Sinnvolles  
    void machwas() {System.out.println("ich mach ja schon");}  
}  
class B extends A{  
    //macht nix Sinnvolles  
    void machwas() {throw new RuntimeException("nix mach ich");}  
}
```

Gedacht als Überleitung für das nächste Prinzip



# Prinzip: Subklassen sollten ihre Superklasse vertreten können - genauer

## Fakt:

- beide Klassen sind dann *konform*

Beispiel: Klasse `Vogel` mit Methode `singen` und Klasse `Spatz` als Erweiterung mit `planschenInPfuetzen`

Schlechtes Beispiel: Quadrat von Rechteck ableiten

# Prinzip/Heuristik?: Verantwortlichkeit

**Beispiel:** Auto, Fahrer, Mechaniker

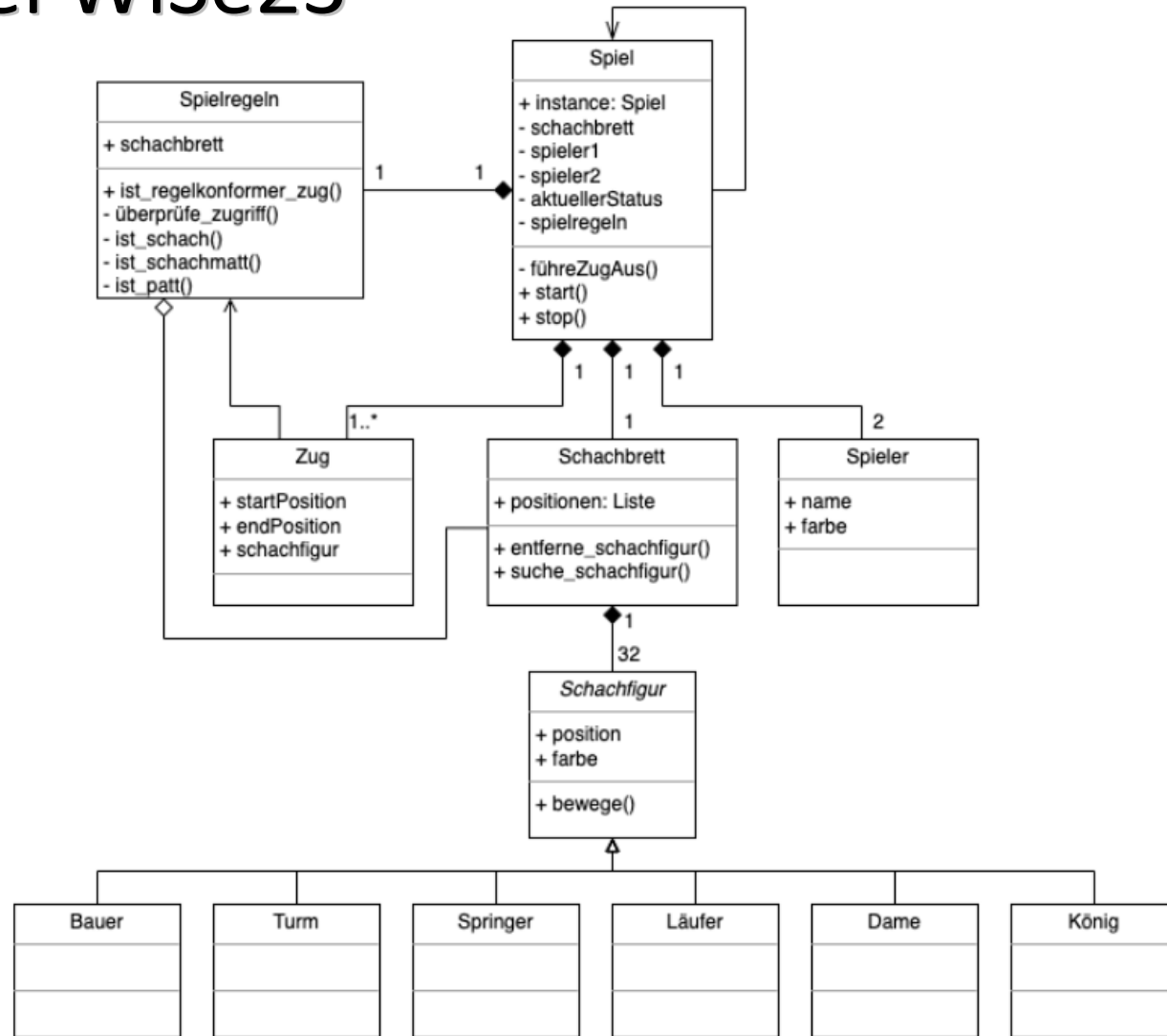
# Was bringen die OO-Prinzipien?

- schlechte Software fällt auseinander bei Veränderungen („Spaghetticode“, „fortschreitendes Verrotten“), gute Software lässt sich leicht ändern.
- **bessere Verständlichkeit:** jede Klasse sollte nur eine Sache/Konzept machen/repräsentieren (z.B. **Karte**-Klasse extra und nicht im **Spieler**) und tun, was ihre Namen anzeigen (wenn ein Objekt *Hund* heißt, sollte es bellen können)
- ungenutzte Eigenschaften sind nutzlos
- Vermeidung von doppeltem Code durch Kapselung, dadurch **einfacher wartbar**
- Verhalten von Klassen **zur Laufzeit änderbar**

# Refactoring

→ Code den OO-Prinzipien anpassen

# Beispiel WiSe23



# Wie setzt man die OO-Prinzipien um?

- Man hinterfragt immer wieder seinen aktuellen Klassenentwurf danach, ob so viele wie mögliche OO-Prinzipien umgesetzt wurden
- Refactorisiere, solange Prinzipien verletzt
- **Entwurfsmuster** sind bewährte Ideen für konkrete Probleme, wo viele Menschen nachgedacht haben, sie ganzheitlich perfekt zu lösen.

# Was ist gute Software?

- Kundenorientierte Programmierung, das heißt der Kunde ist happy
- Objektorientierte Programmierung, das heißt, es existiert kein doppelter Code, jedes Objekt ist ziemlich eigenständig und leicht erweiterbar
- Entwurfsmuster werden genutzt

## Gute Software erstellen in 3 Schritten:

1. Software soll machen, was der Kunde wünscht
2. Anwendung elementarer OO-Prinzipien
3. Design in Richtung Wiederverwendbarkeit und Wartung



# Wie gehe ich an eine Aufgabe objektorientiert ran?

Vom Wunsch zum Ziel:

## 1. OO Analyse (OOA)

- Welche Objekte gibt es? → **Klassen**
- Durch welche Attribute werden sie jeweils beschrieben? → **Attribute** in den Klassen
- Was kann man mit diesen Objekten tun? → **Methoden** in den Klassen
- Haben die Objekte Gemeinsamkeiten, die man in eine **Superklasse** verschieben kann?
- Sollten die Superklassen instanziiierbar sein? → **abstrakte** Klassen
- Welche Beziehungen gibt es zwischen den Klassen?

(HAT-, KENNT-, IST-Beziehung)

- Haben die Objekte austauschbare Verhaltensarten? → **Interfaces** (änderbare Sachen da hinein)
- Weitere Beschreibungsmöglichkeiten: Anwendungsfalldiagramm, Entity-Relationship-Modell, Klassendiagramm, Sequenzdiagramm

## 2. OO Design (OOD)

- Welche Hilfsmittel benötigt man, um die Klasse aufzubauen?
- Welche Datentypen sind einzusetzen?
- Worauf kann zurück gegriffen werden?

Anwendung des Prinzips DRY (Don't Repeat Yourself)

## 3. OO Programmierung (OOP)

Entwicklung der Klassen → Test → bei Schwierigkeiten zurück zu OOD und OOA



Außerdem kann das OOA (Objektorientierte Analyse)-Modell enthalten:

- Anwendungsfalldiagramme (Use cases)
- Klassendiagramme
- Sequenzdiagramme

...

# Design: CRC-Karten

Class Responsibility Collaboration-Cards:

- helfen beim Brainstorming zu objektorientiertem Design
- Karteikarten wie folgt erstellen:

Name der Klasse	
Verantwortlichkeiten der Klasse (Single-Responsibility-Prinzip beachten!)	Namen der Klassen mit denen diese Klasse zusammenarbeiten muss, um ihre Verantwortlichkeiten zu erfüllen

# Literatur

- „Entwurfsmuster von Kopf bis Fuß“, O'Reilly
- „Python von Kopf bis Fuß“, O'Reilly
- „Objektorientierte Analyse und Design“, O'Reilly
- S. Bauer: „Eclipse für C/C++ Programmierer“
- Freemann, Steve: Pryce, Nat: „Growing-Object-Oriented Software, Guided by Tests“, Addison-Wesley 2009
- Bernhard Lahres, Gregor Rayman: Objektorientierte Programmierung ([http://openbook.rheinwerk-verlag.de/oop/oop\\_kapitel\\_03\\_001.htm](http://openbook.rheinwerk-verlag.de/oop/oop_kapitel_03_001.htm))
- Post: Besser coden, Rheinwerk, 2021