

Erfolg ist die Fähigkeit, von einem Misserfolg zum anderen zu gehen, ohne seine Begeisterung zu verlieren.  
*Churchill*

# Modellierung



von  
**Irene Rothe**

Zi. B 241  
[irene.rothe@h-brs.de](mailto:irene.rothe@h-brs.de)



Hochschule  
Bonn-Rhein-Sieg

OO Modellierung

Abitur: Du bist allwissend

Bachelor: in einigen Bereichen hast Du  
gute Grundkenntnisse

Master: Du hast ein sicheres Auftreten bei  
völliger Unwissenheit

# 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
- ✓ OOA: Analyse, Design, Programmierung
- Entwurfsmuster: Singleton, Decorator, Beobachter, Facade, Strategy, Template
- UML
- Heuristiken



# UML (Unified Modeling Language)

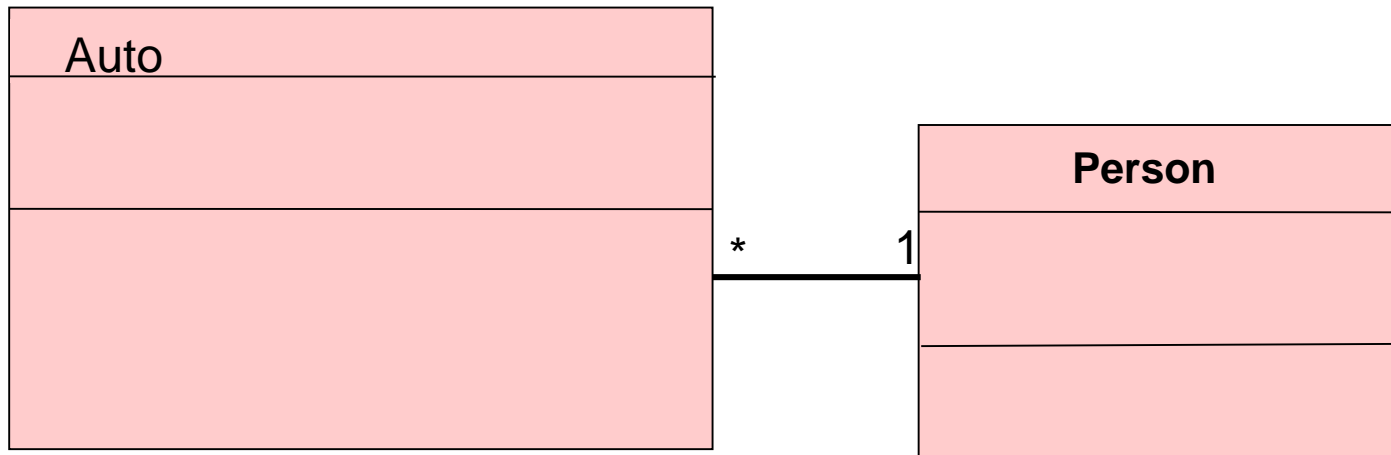
- Gesamtkonzept im Blick
- Informationen zum Code
- beschreibt Struktur der Anwendung
- beschreibt logische Zustände
- beschreibt zeitliches Verhalten
- ist unabhängig von der Programmiersprache
- Nur Notation, keine Methodik
- Ist eine Art der Dokumentation
- 2004 Standard (mit kleinen Unterschieden)
- Besteht aus vielen Teilen: funktionell (use cases), statisch (Klassenaufbau), dynamisch (Zustandsdiagramme)

→ UML zeichnen mit draw.io (benötigt z.B. google-Konto), UMLet (<https://www.umlet.com/>)



# UML: Klassendiagramm

Modellierung von Operationen im Klassenmodell unter Einbeziehung der beteiligten Objekte, Operationen (Methoden/Funktionen) und Beziehungen (Assoziationen/Relationen)



Beispiel: Krämer-Fuhrmann: uml-2014.pdf: Folie 19

# UML: Use Cases (Anwendungsfälle)

Use Case

- Akteure

Person A, Person B

- Bedingungen

...

- Hauptanwendungsfall

...

- alternativer Anwendungsfall

...

- Nachbedingungen

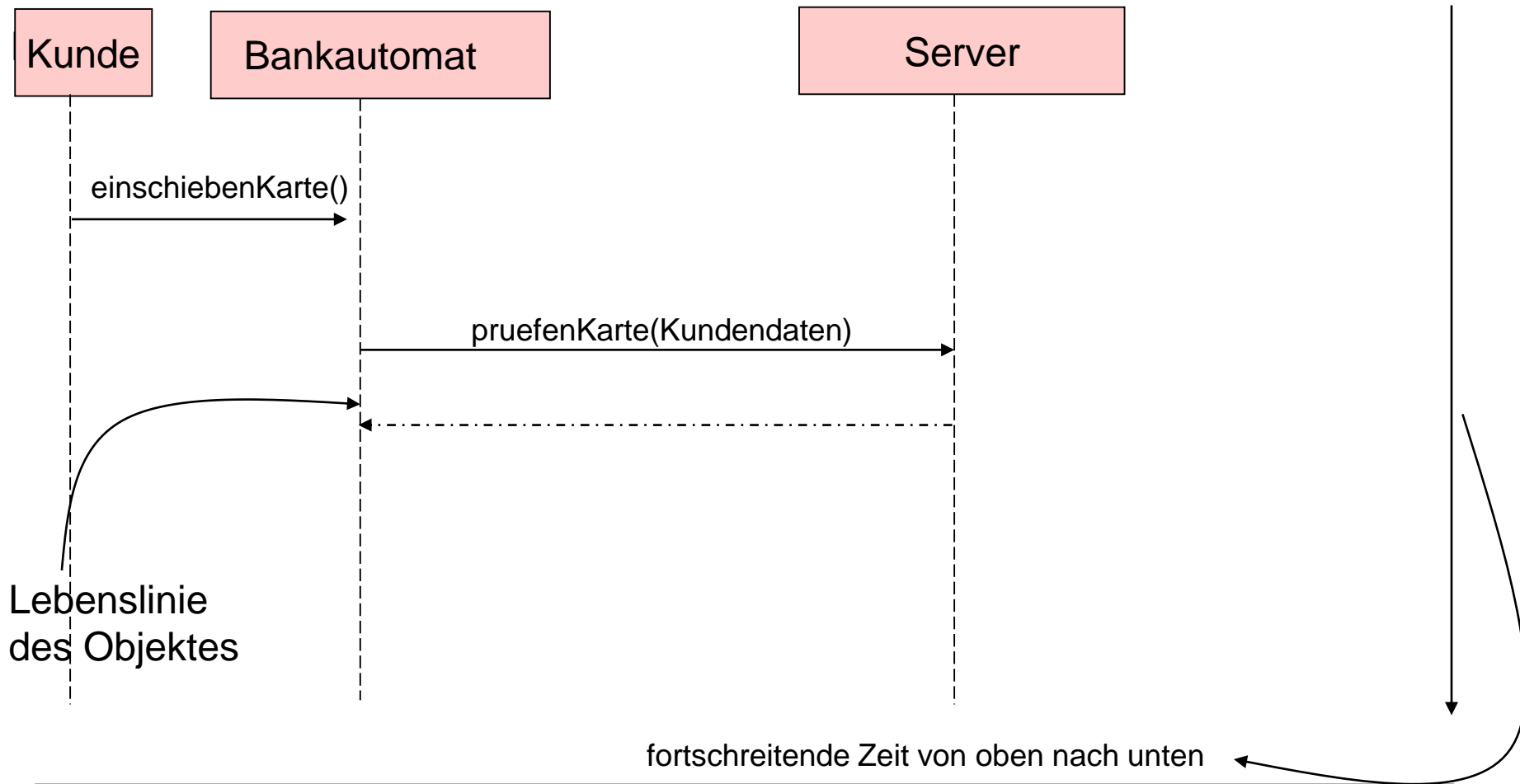
...

Ende Use Case

Beispiel: Krämer-Fuhrmann: uml-2014.pdf: Folie 13

# UML: Interaktionsdiagramm

Gibt Reihenfolge an, in welcher die Anfragen zwischen den Objekten ausgeführt werden.



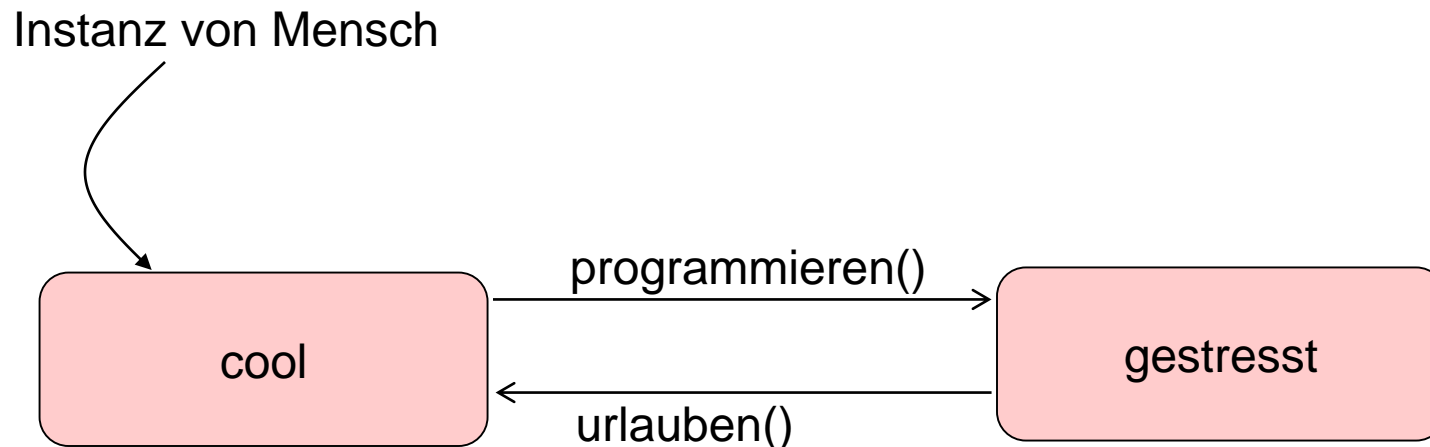
# Übung – Sequenzdiagramm für Schiffe Versenken



# UML: Zustandsdiagramme

Zustand eines Objektes zu einem bestimmten Zeitpunkt ist bestimmt durch bestimmte Attribute.

Objekte ändern ihre Zustände durch bestimmte Ereignisse (Ausführung bestimmter Operationen (in Java: Methoden)): Empfang einer Nachricht



Beispiel: Schachuhr

# Übung - Schachuhr



# Heuristiken (Faustregeln)

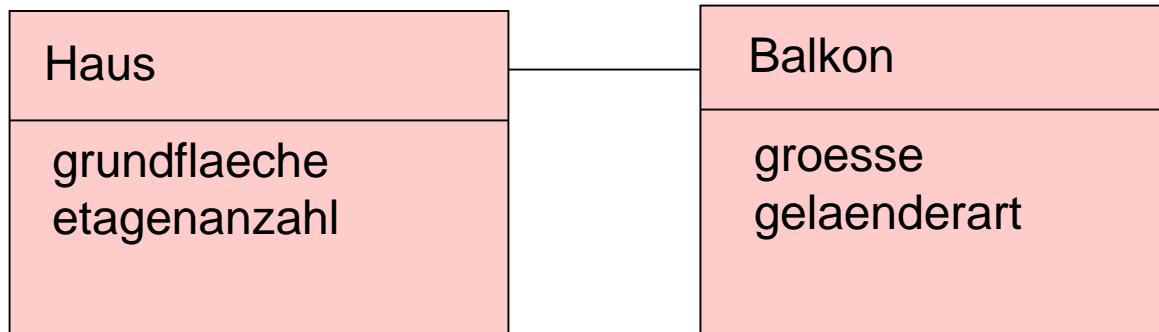
→ damit fährt man ganz gut

# Heuristik 1

Alternativen? Gibt es mehrere Alternativen während einer Modellierung, entscheidet man sich vorläufig für irgendeine.

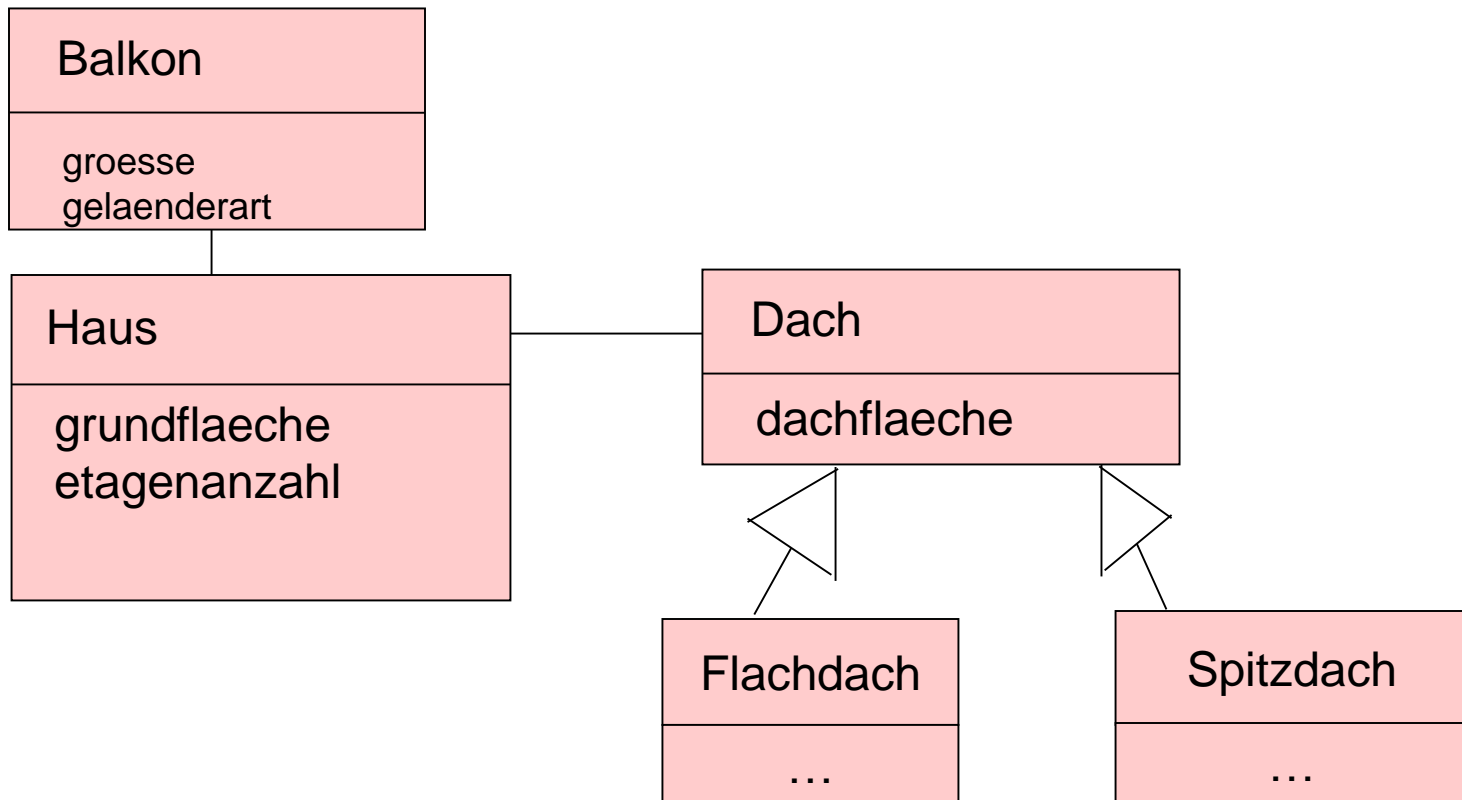
# Heuristik 2

Eine oder mehrere Klassen? Zwei verschiedene Klassen sind die bessere Wahl, wenn die Beziehung zwischen ihnen optional ist, das heißt, die eine Klasse muss ein bestimmtes Attribut nicht unbedingt haben.



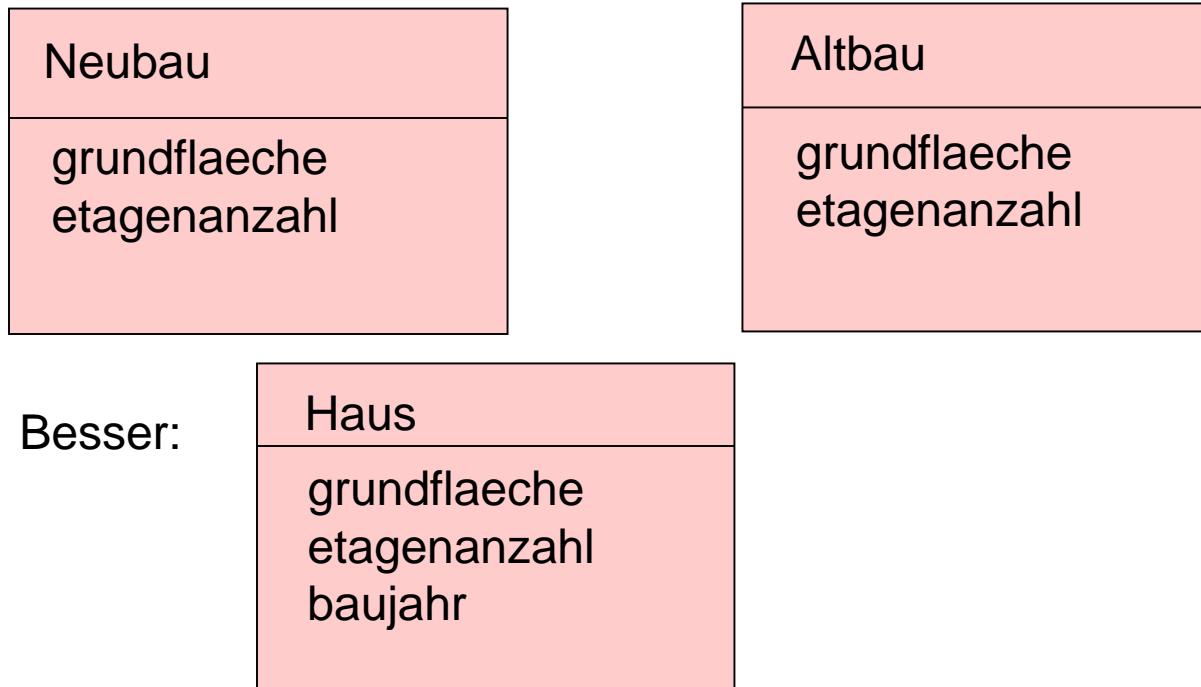
# Heuristik 3

Eine oder mehrere Klassen? Zwei verschiedene Klassen sind die bessere Wahl, wenn eine Instanz eine Verbindung zu Instanzen noch anderer Klassen eingehen kann.



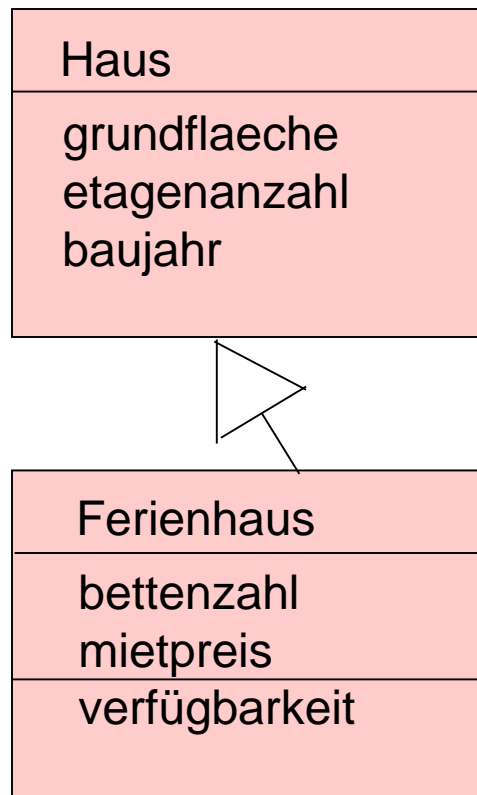
# Heuristik 4

Attribute oder Vererbung? Lassen sich Objekte mit denselben Attributen und demselben Verhalten in disjunkte Klassen aufteilen, werden sie besser in einer Klasse zusammengefasst und mit zusätzlichen Attributen unterschieden.



# Heuristik 5

Attribute oder Vererbung? Wenn sich Objekte bzgl. Attributen **und** Verhalten unterscheiden, werden sie vererbt.





# Weitere Heuristiken

6. **Geheimprinzip:** Attribute *privat* und Standardoperationen *public*

7. **Abschwächung von Kopplung** (Beziehungen zwischen Klassen):

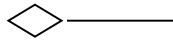
- Sind nur wenige Klassen eng gekoppelt → Zusammenfassung zu einer Klasse
- Sind mehrere Klassen eng gekoppelt → Einbau einer Vermittlerschicht

8. **Stärkung der Kohäsion:**

- Lassen sich Attribute und die Operationen einer Klasse in disjunkte Teilmengen zerlegen, dann ist die Aufteilung der Klasse zu empfehlen.
- Erbringt eine Operation mehr als eine einzige eng umrissene Funktionalität, ist die Funktionalität auf mehrere Operationen aufzuteilen.

# Heuristik: Aggregation und Komposition sind besser als Vererbung

**Aggregation (A NUTZT B):** ist, wenn eine Klasse als Teil einer anderen Klasse verwendet wird, trotzdem aber außerhalb dieser anderen Klasse existiert ,  
Komposition *ohne* das plötzliche Ende (Auto hat Räder, die auch ohne das Auto existieren),

UML-Symbol: A  B

**Komposition (A OWNS B, B ist existenzabhängig von A):** ermöglicht, das Verhalten einer Familie anderer Klassen zu verwenden + dieses Verhalten zur Laufzeit zu ändern, es wird ein **new** von der anderen Klasse in der Klasse aufgerufen (Spieler könnten zur Laufzeit mit entsprechendem Spielverhalten versorgt werden, Spieler ist vollständiger Eigentümer von Spielverhalten und ist ohne ein Spielverhalten eigentlich gar kein Spieler (ist Spieler weg, ist Spielverhalten weg; eine Karosserie existiert nur im Zusammenhang mit einem Auto))

UML-Symbol: A  B

**Wann Aggregation und wann Komposition?** → Frage: existiert das Objekt, dessen Verhalten ich nutzen möchte, außerhalb des Objektes, das das Verhalten nutzt?  
Wenn ja, dann Aggregation, ansonsten Komposition.

# Aggregation und Komposition: noch mal genauer

**Assoziation:** Objekte sind unabhängig, keiner ist Besitzer des anderen Objektes.  
Beispiel: viele Dozenten haben zu tun mit vielen Studierenden, viele Studierende haben zu tun mit vielen Dozenten. Sie können kommen und gehen.

**Aggregation:** ist spezielle Assoziation mit Besitzzuordnungen.  
Beispiel: Ein Dozent kann nur zu einem Fachbereich gehören, nicht zu mehreren. Wenn aber ein FB aufgelöst würde, gäbe es den Dozenten immer noch. Also auch eine Art HAT-Beziehung.

**Komposition:** ist eine Art starke Aggregation, eine Art “mit in den Tod-reiß” Beziehung. Das kompositionierte Objekt wird gelöscht, wenn sein Besitzer gelöscht wird.  
Beispiel: Haus und seine Räume: wird das Haus abgerissen, sind auch alle Räume weg.

# Aggregation und Komposition: Beispiel

**Aggregation:** the object exists outside the other, is created outside, so it is passed as an argument (for example) to the constructor.

**Composition:** the object only exists, or only makes sense inside the other, as a part of the other.

```
// WebServer is aggregated of a HttpListener and a RequestProcessor
public class WebServer {
    private HttpListener listener;
    private RequestProcessor processor;
    public WebServer(HttpListener listener, RequestProcessor processor) {
        this.listener = listener;
        this.processor = processor;
    }
}
```

Code example for composition

```
// WebServer is an composition of HttpListener and RequestProcessor and controls their lifecycle
public class WebServer {
    private HttpListener listener;
    private RequestProcessor processor;
    public WebServer() {
        this.listener = new HttpListener(80);
        this.processor = new RequestProcessor("/www/root");
    }
}
```

# Heuristik zum Prinzip der Substituierbarkeit

Subklassen sollten ihre Superklasse vertreten können

- Verbesserung bei Nicht-Konformheit: Einführung neuer Superklasse für beide Klassen (für **Rechteck** und **Quadrat** eine Superklasse **Figur** einführen)
- Klassen werden so definiert, dass sie für Subklassenbildung offen sind, aber für unkontrollierte Zugriffe geschlossen.
- **this**: innerhalb einer Operation eine weitere Operation derselben Klasse ansprechen
- bei Überschreibung von super-Methoden: Aufruf der super-Methode mit **super** und Hinzufügen von weiterem Verhalten in der neuen überschriebenen Methode

# Entwurf mit Verträgen

- Spezifikationen der Operationen mit Vor- und Nachbedingung und Klasseninvarianten
- Systematische Behandlung von Ausnahmefällen (nur für wirkliche Ausnahmen)
- Klar geregelte Verantwortlichkeiten von Dienstnutzer und Dienstleister (wenn Dienstnutzer die Vorbedingungen erfüllt, dann garantiert der Dienstleister die Nachbedingungen ohne irgendwelche Ausnahmefälle)
- Führen Sie zu Ende, was Sie begonnen haben (z.B. öffnen/schließen von Dateien)

# Exceptions (Ausnahmen)

... sind Fehler, die z.B. in C zu Abstürzen führen, nicht reparierbare Laufzeitfehler oder Hardware-Probleme

→ Ausnahmebehandlung = „weiche“ Landung des Programms

Die JVM von Java erzeugt Exception, wie z.B.:

- Division durch 0 (nur bei int) -> ArithmeticException
- Lesen über Arraygrenzen hinweg->IndexOutOfBoundsException,
- Lesen über das Ende einer Datei hinaus->EOFException

„weiche“ Landung nötig, wenn vor Absturz noch wichtige Dinge erledigt werden sollten, wie z.B. Daten speichern...

# Ausnahmebehandlung (Exceptions)

## *try-catch*-Block :

- im **try**-Block sind die Anweisungen implementiert, bei denen ein Fehler auftreten kann
- im **catch**-Block stehen die Anweisungen, die ausgeführt werden sollen, wenn bestimmte Fehler eintreten, z.B. Ausgabe einer Fehlermeldung

```
try{//Probier mal, ob's funktioniert!
```

```
...
```

```
}
```

```
catch() {//wenns passiert, tue das hier!
```

```
...
```

```
}
```

## **Ziel:**

- Erhöhung der Übersichtlichkeit durch Trennung Teile des Codes von möglicherweise fehlerverursachendem Code
- Einfachere Fehlersuche, da in der Regel die genaue Zeilennummer und die Klasse, in der der Fehler aufgetreten ist, geliefert wird



# Exceptions Vorteile

- Fehler können dort behandelt werden, wo es Sinn macht
- Fehler kann man nur noch explizit ignorieren
- Fehlerbehandlung wird vereinheitlicht

# Eigene Exceptions

```
//Meine erste eigene Ausnahme
class NegativeWurzelException extends Exception {
    NegativeWurzelException() {
        super("Wurzel einer negativen Zahl ist nicht berechenbar");
    }
}

class EigeneAusnahmeJava{
    public static void main(String[] args){
        ...
        try{...}catch (NegativeWurzelException e){}
        ...
    }
    public static double wurzel(int n, double x) throws NegativeWurzelException{
        if(x>=0){
            ...
        }
        else{
            throw new NegativeWurzelException();
        }
    }
}
```



# Frameworks

- ... bestehen aus vorgegebenen Pattern und Strukturen, in die man eigenen Code reinschreibt (siehe Observer-Muster)
- ... sind keine fertigen Programme, sondern müssen an vorgesehenen Stellen ergänzt werden.
- um dies zu tun, werden geeignete Klassen implementiert und registriert, welche dann von dem Framework genutzt und kontrolliert werden können.
- Beispiel: Bereitstellung der Funktion eines Buttons. Nun kann eine Listener-Methode registriert werden, welche das Framework immer beim Drücken dieses Buttons aufruft.

# Vorgehen beim Programmieren

## Design patterns:

- lokale, sich auf wenige Klassen beziehende Entwurfserfahrungen der Wiederverwendung zugänglich machen
- konstruktive Vorlage
- Beschreibung einer Familie von Lösungen für ein Entwurfsproblem
- es gibt erzeugende Muster, strukturelle Muster, Verhaltensmuster

**Frameworks:** Wiederverwendung von ganzen Entwürfen mit teilweiser Implementierung (z.B. MVC: Model-View-Controller für GUIs)

## Arbeit mit Entwurfsmustern:

1. Durchlesen der Beschreibung
2. komplettes Verstehen der Klassen und ihrer gegenseitigen Abhängigkeiten
3. Blick auf Beispiel Quellcode, eventuelle Verwendung von Teilen davon

# Neu: DevOps 1

= Zusammenführung von Development und IT-Operations

... ist ein Mindset

**Es gibt 4 Arten von Arbeiten/Aufgaben:**

- Businessaufgaben
- Interne Aufgaben
- Änderungsaufgaben
- Ungeplante Arbeit (Notfälle)

# Neu: DevOps 2

Dafür braucht es **3 Wege**, die man gehen muss:

1. Arbeitsfluss von Entwicklung über IT-Operationen zum Kunden, Engpässe suchen (kleine Gruppen mit Arbeitsintervallen, keine fehlerhaften Produkte weiterreichen, auf globale Ziele hin optimieren, kontinuierliche Builds inklusive der Umgebungen)
2. Schnelles und konstantes Feedback („Stoppen des Fließbands“, Messpunkte, gemeinsame Ziele, automatische Tests und Builds)
3. Kultur (Vertrauen, Verletzlichkeit zeigen), Experimentieren, Risiken eingehen, aus Fehlschlägen lernen, Wiederholung und Übung

# Neu: DevOps 3

... ist ein Mindset

Das bedeutet fürs Team:

- man geht davon aus, dass alle Teammitglieder gut arbeiten wollen, gerne arbeiten, ein gutes Produkt liefern wollen
- effizientere Zusammenarbeit durch gegenseitiges Verständnis für die Arbeit der anderen
- Förderung von Lern- und Experimentierfreude
- hohe Zufriedenheit der Mitarbeitenden
- man sollte nicht der Idiot sein, der NICHT um Hilfe

Bemerkung: kann man überall im Leben anwenden



# Neu: DevOps 4

- Skripte und Tools für die agile Entwicklung
- Automatisierung steht weit oben, z.B. Build- und Release-Prozesse
- Automatisiertes Testen
- Verwaltung über z.B. Github
- Infrastructure wird wie Code behandelt (automatisiert, versioniert...)
- Dokumentation, Schulungsmaterial...Euer Fingerabdruck!

→ IT+Business müssen eins sein

→ IT hilft Business erfolgreich zu sein

Buch: „Projekt Phoenix: Der Roman über IT und DevOps - Neue Erfolgsstrategien für Ihre Firma“, Gene Kim , Kevin Behr, et al., O’Reilly, 2015

Buch: „DevOps Cookbook“ DeBois, Wills, Orzen





# Objektorientiert zu prozedural 1

**Prozedural:** break down programming task into a collection of variables, data structures and functions, in der Regel Trennung von Daten und Methoden auf diesen

**Object oriented:** break down programming task into objects that connect behavior and data by using interfaces, it makes it possible that objects can operate on *own* data

→ Das kann aber übertrieben werden

→ Wenn unnötig, wird viel static genommen wird oder Utility-classes

**Fazit:** nicht alles muss OO sein, man braucht nicht immer getter und setter, utility class kann auch gut sein

**Man sollte sich die Frage stellen: mache ich etwas, weil man es eben so macht oder weil es sinnvoll ist?**



# Objektorientiert zu prozedural 2

- OO wurde erfunden, um Komplexität zu beherrschen, Codedoppelungen zu vermeiden, Aufteilung der Entwicklungsarbeit auf mehrere einfach möglich zu machen.
- OO ist keine Konkurrenz zur prozeduralen Programmierung, sondern eine Erweiterung bzw. Ergänzung
- Man begibt sich in die prozedurale Programmierung sobald Logik von Daten getrennt werden
- OO enthält immer auch prozedurale Programmierung
- Nachteile OO: Mehraufwand bei Modellpflege, unnötige Komplexität
- Nachteile prozedurale Entwicklung: schlechte Wartbarkeit, Designbarkeit, Wiederverwendbarkeit

**Wartbarkeit** generell: was zusammengehört...bring zusammen, Verantwortlichkeiten trennen, Abhängigkeit reduzieren

# Das Marie Kondo Prinzip

If it doesn't spark joy, get rid of it.

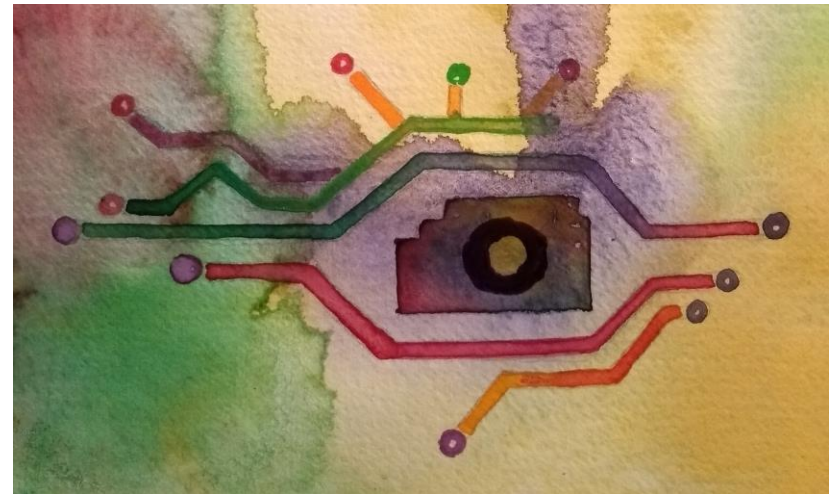
- Wenn etwas keinen Spaß macht, ändere es!
- If a class/method/line doesn't spark joy, get rid of it.
- Weniger ist immer mehr
- Joy: jede Zeile Code macht das, was sie soll (ohne 5 Bedingungen abzufragen oder 4 Fälle gleichzeitig abhandelt)

Frage nach Verstoß gegen single responsibility principle, ansonsten refactoring, open close principle? → siehe SOLID (single resp., open-close, Liskov substitution (Ersetzbarkeit in Oberklassen), Interface segregation, Dependency inversion (Abstraktion, keine Konkretisierung), nicht mehr als 2 Ebenen tief, zu viele ifs in zu vielen Schleifen → no feeling of joy

Quelle: <https://www.karim-geiger.de/blog/das-marie-kondo-software-design-principle>

# Empfehlung fürs Lernen

Stelle einem Chatbot, z.B. <https://chat.openai.com>, Deine Fragen und gucke, ob die Antwort passt



# Literatur

- M. Grand „Patterns in Java“ Wiley Verlag, 1998
- McLaughlin, Pollice, West: „Objektorientierte Analyse und Design“ O'Reilly, 2007
- Gamma, Helm, Johnson, Vlissides: „Entwurfsmuster“, Addison-Wesley, 2004
- <https://jaxenter.de/java-prozedural-es-muss-nicht-immer-objektorientierung-sein-20709>
- <https://www.karim-geiger.de/blog/das-marie-kondo-software-design-principle>
- Gene Kim, et al.: „Projekt Phoenix: Der Roman über IT und DevOps - Neue Erfolgsstrategien für Ihre Firma“, O'Reilly, 2015