

# JavaAPI, OOP und CommandPattern

Programmieren Tutorium Nr.12

Aleksandr Zakharov | 27. Januar 2026

# Organisatorisches

## Übungsblatt 5

Abgabe: 29. Januar 2022 6:00 Uhr

## Abschlussaufgabe 1

- Ausgabe: 16.02.2026 um ca. 12:00 Uhr
- Abgabe: 02.03.2026 (12:00 Uhr) – 17.03.2026 (06:00 Uhr)

## Abschlussaufgabe 2

- Ausgabe: 02.03.2026 um ca. 12:00 Uhr
- Abgabe: 16.03.2026 (12:00 Uhr) – 31.03.2026 (06:00 Uhr)

Organisatorisches



Übungsblatt 4



Rekursion: Wiederholung



Java API



OOP Designprinzipien



Commands



Aufgabe: Commands



# Übungsblatt 4

- Klassenkonstanten sind `static final` nicht nur `final`

```
1 if(x == true) { return true; } else { return false; }  
2 return x; // identische Aussage zu vorher
```

- Keine eigenen Implementationen von Stack, etc.
- Hohe Verschachtelungstiefe <https://blog.codinghorror.com/flattening-arrow-code/>
- Attribute `final` setzen wenn möglich
- Benutzt Vererbung und objektorientierte Modellierung. Sonst gibt's Abzüge
- Zwei Dateien sind keine adequate Lösung für Blatt 4
- Wenn ihr um 5:00 morgens noch an der Abgabe arbeitet seid ihr zu spät dran
- `/**asdfsadsfa**/` ist kein adequater Javadoc
- Die meisten Abzüge gehen automatisch über die Tests für Funktionalität

# Programmieren gegen Interfaces

## Ein kleines Programm

```
1 public ArrayList<String> split(String input) {}
2 public String concat(ArrayList<String> input) {}
3 // Alles gut:
4 ArrayList<String> parts = split("Hello you there");
5 concat(parts); // funktioniert
```

Organisatorisches  
○

Übungsblatt 4  
○○●

Rekursion: Wiederholung  
○○

Java API  
○○○○○○

OOP Designprinzipien  
○○○○○○○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○

# Programmieren gegen Interfaces

## Ein kleines Programm

```
1 public ArrayList<String> split(String input) {}
2 public String concat(ArrayList<String> input) {}
3 // Alles gut:
4 ArrayList<String> parts = split("Hello you there");
5 concat(parts); // funktioniert
```

## Und jetzt kommt von irgendwo etwas dazu

```
1 public LinkedList<String> splitAndTransform(String input) {}
2
3 // *Sad List noises*: Kaputt
4 ArrayList<String> parts = splitAndTransform("Hello you there");
5 // Auch kaputt!
6 concat(splitAndTransform("Hello you there"));
```

# Mit Interfaces

## Ein kleines Programm

```
1 public List<String> split(String input) {}  
2 public String concat(List<String> input) {}  
3 // Alles gut:  
4 List<String> parts = split("Hello you there");  
5 concat(parts); // works
```

Organisatorisches  
○

Übungsblatt 4  
○○●

Rekursion: Wiederholung  
○○

Java API  
○○○○○○

OOP Designprinzipien  
○○○○○○○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○

# Mit Interfaces

## Ein kleines Programm

```
1 public List<String> split(String input) {}
2 public String concat(List<String> input) {}
3 // Alles gut:
4 List<String> parts = split("Hello you there");
5 concat(parts); // works
```

## Und jetzt kommt von irgendwo etwas dazu

```
1 public List<String> splitAndTransform(String input) {}
2
3 // *Happy List noises*: Geht
4 List<String> parts = splitAndTransform("Hello you there");
5 // geht auch!
6 concat(splitAndTransform("Hello you there"));
```

# Aufgabe: Rekursion

Aufgabe: Schreibt den folgenden Sortieralgorithmus als rekursive Methode:

1. Es wird eine unsortierte Liste von Ganzzahlen als Parameter übergeben.
2. Suche das kleinste Element der unsortierten Liste und entferne es aus der Liste. (Ihr könnt die Methode `int min(List<T> list)` implementieren, indem man durch die Liste geht und das zurzeit gefundene Minimum speichert)
3. Füge das entfernte kleinste Element an das Ende einer sortierten Liste.
4. Wiederhole Schritte 2-3 mit dem Rest der unsortierten Liste.
5. Gebe die fertige sortierte Liste als Lösung zurück.

Tipp: <https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>  
`add(E e)`, `remove(Object o)`, `size()`, `addAll(Collection<? extends E> c)`

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
●○

Java API  
○○○○○

OOP Designprinzipien  
○○○○○○○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○



# Lösung

```
1 public static ArrayList<Integer> sortRecursively(ArrayList<Integer> unsorted) {
2     ArrayList<Integer> sorted = new ArrayList<>();
3     if (unsorted.isEmpty()) { // Abbruchbedingung
4         return sorted;
5     }
6     Integer min = min(unsorted); // Berechne das kleinste Element
7     unsorted.remove(min); // Entferne das kleinste Element aus der unsortierten Liste
8     sorted.add(min); // Füge das kleinste Element an das Ende der sortierten Liste
9     ArrayList<Integer> sortedRemaining = sortRecursively(unsorted); // Sortiere die Restliste
10    sorted.addAll(sortedRemaining);
11    return sorted;
12 }
```

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○●

Java API  
○○○○○○

OOP Designprinzipien  
○○○○○○○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○

# Java API

## Allgemein

- „Application Programming Interface“
- Sammlung von Funktionalitäten von häufig benötigten Klassen und Pakete
- In der „richtigen“ Welt der Programmierung
  - verwende vorhandenes möglichst effizient und vielseitig
  - keine Zeit mit Neuimplementierung von vorhandenen Dingen verschwenden

# Wichtige Links

## Javadoc

- Java 21: <https://docs.oracle.com/en/java/javase/21/docs/api/index.html>

## Oracle Dokumentation und Tutorials

- Allgemein: <https://docs.oracle.com/javase/tutorial/index.html>
- Hilfreich: Suchanfrage <Klasse> `site:https://docs.oracle.com/javase/tutorial`

# Java API

## Überblick

- java.lang (Object, String, Enum, Math, Iterable, Byte, Float, ...)
- java.util (collections, formatting, data structure manipulation)
- java.io (obviously input/output, streams, manipulating files on FS)
- java.net (networking, URL, connections)
- java.security
- java.swing, awt, javafx (graphical stuff)

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○○

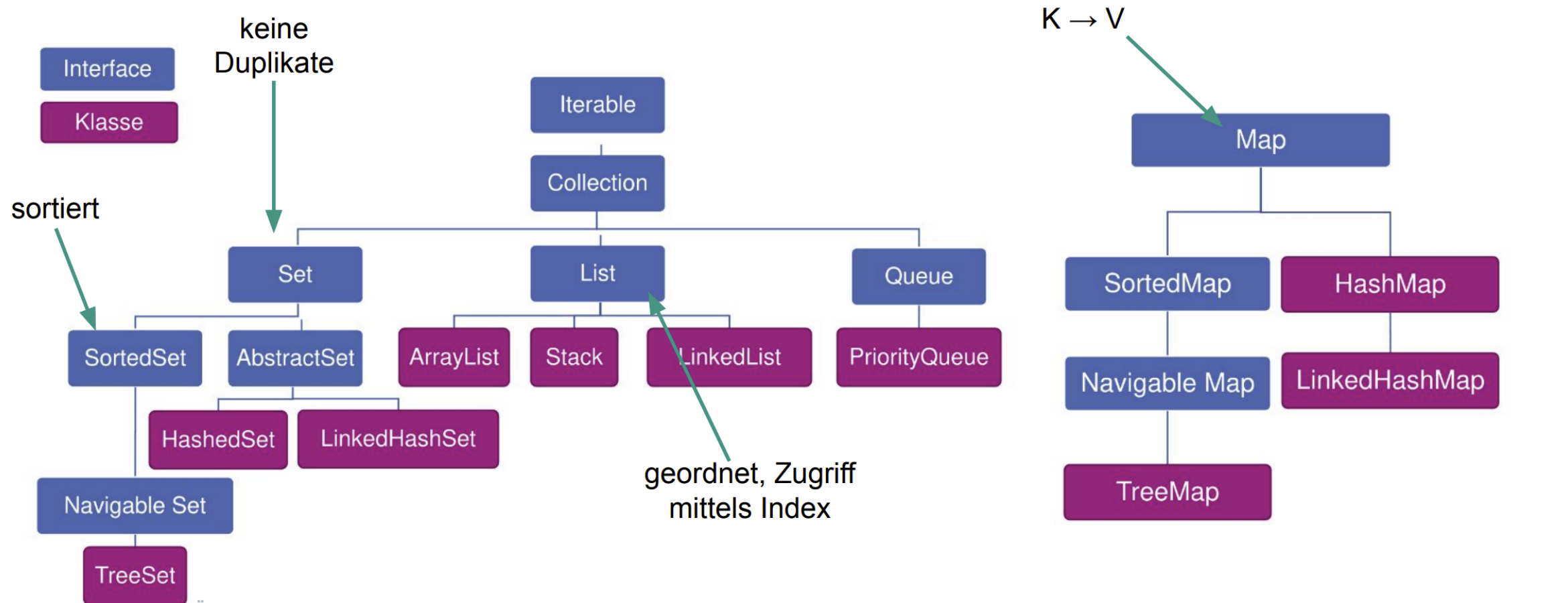
Java API  
○○●○○○

OOP Designprinzipien  
○○○○○○○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○

# java.util - Wichtige Interfaces / Klassen



Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○○

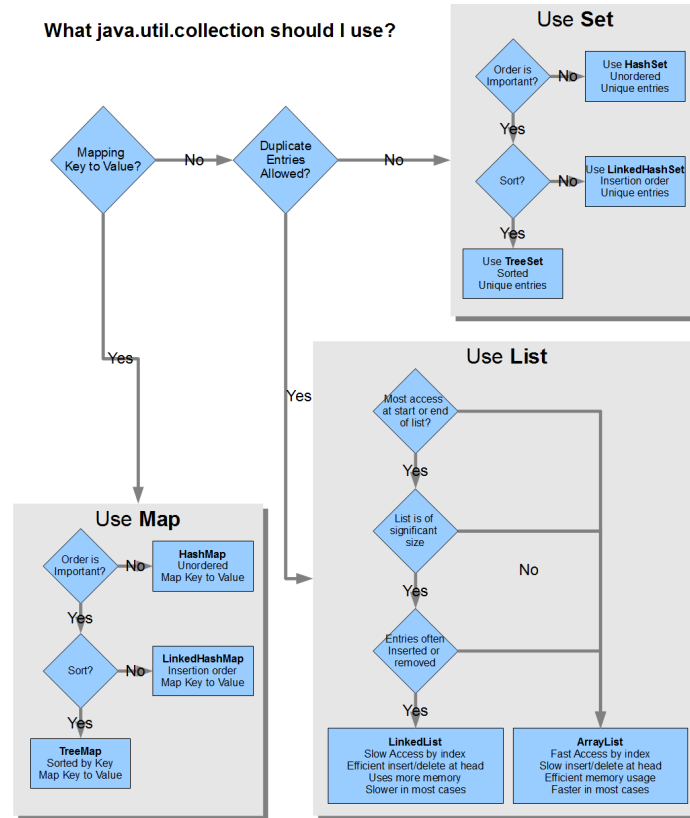
Java API  
○○○●○○

OOP Designprinzipien  
○○○○○○○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○

# Java API - Collection



[Quelle: <https://i.stack.imgur.com/aSDsG.png> - Collection]

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○○

Java API  
○○○○●○

OOP Designprinzipien  
○○○○○○○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○

# Java API - Collection

## Map<K, V>

- (key, value) Paare
- Funktionen
  - put(K key, V value)
  - get(K key)
  - remove(K key)
- implementiert durch
  - HashMap (sehr schnell für gehashte Werte)
  - TreeMap (ein Baum ...)

# OOP Designprinzipien

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○○

Java API  
○○○○○○

OOP Designprinzipien  
●○○○○○○○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○



# Warum Designprinzipien?

- Viele Möglichkeiten, eine Idee umzusetzen
- Manche sind besser, manche schlechter
  - z.B. Neue Features, Library, ...
- In Softwaretechnik (2. Semester): Etablierte Design-Patterns
  - Grundlagen schon hier

# OOP Designprinzipien

- Prinzip 1 (Datenkapselung)
  - Minimiere die Zugriffsmöglichkeiten auf Klassen und Attribute
- Prinzip 2
  - Bevorzuge Komposition gegenüber Vererbung
- Prinzip 3
  - Programmiere gegen Schnittstellen und nicht gegen eine Implementierung
- Prinzip 4 (Open-Closed Principle)
  - Software-Komponenten sollten offen für Erweiterung, aber geschlossen für Änderung sein
- SOLID-Prinzipien

# 1. Datenkapselung

- Wann immer möglich: private mit Gettern/Settern verwenden

Schlecht

```
public class Monopoly {  
    public Player currentPlayer;  
    ...  
}
```

Besser

```
public class Monopoly {  
    private Player currentPlayer;  
    ...  
    public Player getCurrentPlayer() {...}  
    public void setCurrentPlayer() {...}  
    // oder nextPlayer() oder so  
    ...  
}
```

Zugriff kann gesteuert  
werden

# 2. Komposition > Vererbung

## Was ist Komposition?

“Wiederverwendung durch Zusammensetzung bestehender Objekte zu einem neuen mit erweiterter Funktionalität”

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○○

Java API  
○○○○○○

OOP Designprinzipien  
○○○○●○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○

# 2. Komposition > Vererbung

## Komposition



```
class Object3D {
    Vector3D position, rotation, scale;
}

class Physics3D {
    Object3D influenceOn;
    Vector3D gravity, velocity;
    public Physics3D(Object3D object) { ... }
    ...
}

class Player {
    Object3D position;
    Physics3D physics;
}
```

Besser

## Vererbung

```
abstract class Object3D {
    Vector3D position, rotation, scale;
    ...
}

abstract class PhysicsObject extends Object3D {
    Vector3D gravity, velocity;
    ...
}

class Player extends PhysicsObject {
    ...
}
```

Schlecht

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○○

Java API  
○○○○○○

OOP Designprinzipien  
○○○○●○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○

# 2. Komposition > Vererbung

Sehr ausführliches Video dazu:  
[The Flaws of Inheritance - CodeAesthetic](#)

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○○

Java API  
○○○○○○

OOP Designprinzipien  
○○○○●○○○○○○○○

Commands  
○○○○

Aufgabe: Commands  
○○

# 3. Schnittstellen statt Implementierungen

## ■ Beispiel: Erweiterung eines 3D Games durch Fahrzeuge

zu spezifisch

```
class Car {...}
class Player {
    ...
    public void enterCar(Car car) {...}
}
```

Schlecht

möglichst unspezifisch

```
interface Vehicle {...}
class Car implements Vehicle {...}
class Truck implements Vehicle {...}
...
class Player {
    ...
    public void enterVehicle(Vehicle vehicle) {...}
}
```

Besser

# 4. Open-Closed-Principle (OCP)

- Offen für Erweiterung, Geschlossen für Änderung
- ⇒ bei Erweiterung neuen Code hinzufügen, nicht alten Code ändern!

```
Employee[] employees;  
  
for (Employee employee : employees) {  
    if (employee instanceof Intern) {  
        employee.paySalary(0);  
    } else if (employee instanceof Manager) {  
        employee.paySalary(69420);  
    } ...  
}
```

Schlecht

Bei Erweiterung muss alter  
Code verändert werden!

```
Employee[] employees;  
  
for (Employee employee : employees) {  
    salary = employee.getRole().getSalary();  
    employee.paySalary(salary);  
}
```

Besser



# SOLID-Prinzipien

## 1. Single Responsibility Principle

- Jede Klasse sollte nur eine Verantwortung („Grund zur Änderung“) haben.

## 2. Open/Closed Principle

- Klassen sollten Erweiterungen erlauben, ohne dabei ihr Verhalten zu ändern

## 3. Liskov Substitution Principle

- Eine Instanz einer abgeleiteten Klasse sollte sich so verhalten, dass jemand, der meint, ein Objekt der Basisklasse vor sich zu haben, nicht durch unerwartetes Verhalten überrascht wird.

## 4. Interface Segregation Principle

- Klassen sollten durch Interface nicht gezwungen werden, nicht notwendige Methoden zu implementieren; stattdessen Interface auftrennen

## 5. Dependency Inversion Principle

- Abstraktes soll nicht von Details abhängen

# Weitere Prinzipien (für Schnittstellen)

1. Verstecken von interner Repräsentation → möglichst generelle Superklasse/Schnittstelle verwenden
2. Interfaces gut abstrahieren → nicht ein “Alles-Interface” sondern sinnvoll trennen
3. Konsistentes Abstraktionsniveau
4. Methodennamen sinnvoll benennen

# Aufgaben (1/2)

## ■ Welche Designprinzipien bricht der folgende Code?

```
public class OnlineShop {
    ...
    public void sell(Customer c,
                     Product p, int amount) {
        if (i instanceof DiscountedProduct) {
            c.bill(i.price * amount * i.discount);
        } else {
            c.bill(i.price * amount);
        }
        this.ship(c, i, amount);
    }

    public void add(ArrayList<Product> toAdd) {
        this.items.addAll(toAdd);
    }
    ...
}
```

```
public class Product {
    public String name;
    public double price;
    ...
}

public class DiscountedProduct extends Product
{
    public double discount;
    ...
}
```

# Aufgaben (1/2) - Lösung

Single Responsibility

```
public class OnlineShop {  
    ...  
    public void sell(Customer c,  
                     Product p, int amount) {  
        if (p instanceof DiscountedProduct) {  
            c.bill(p.price * amount * p.discount);  
        } else {  
            c.bill(p.price * amount);  
        }  
        this.ship(c, p, amount);  
    }  
  
    public void add(ArrayList<Product> toAdd) {  
        this.items.addAll(toAdd);  
    }  
    ...  
}
```

Methodenname ungenau

Open/Closed Principle!

Schnittstellen anstatt Klassen bevorzugen  
+  
Internen Aufbau verstecken

Datenkapselung!

```
public class Product {  
    public String name;  
    public double price;  
    ...  
}  
  
public class DiscountedProduct extends Product {  
    public double discount;  
    ...  
}
```

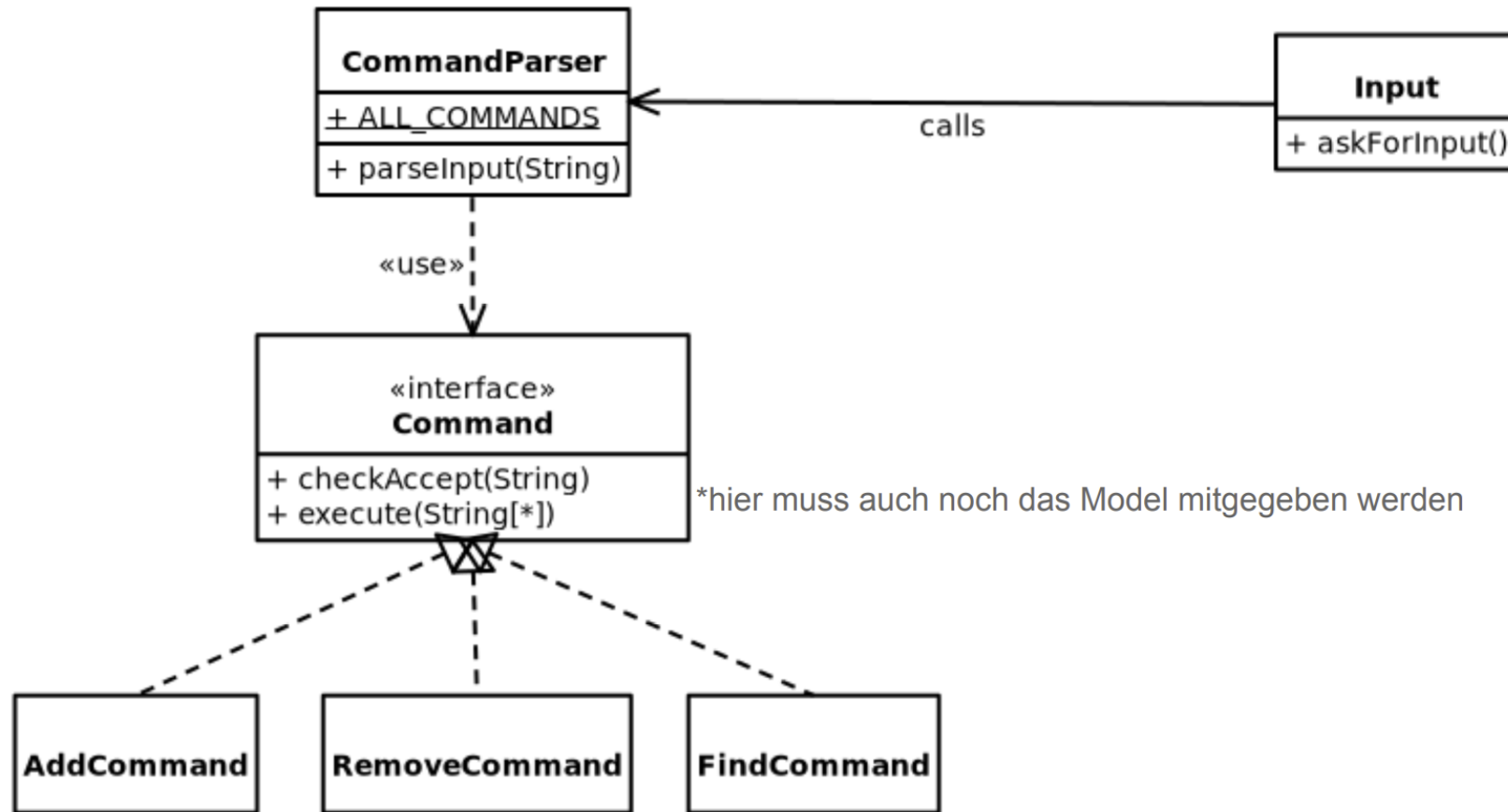
Komposition statt Vererbung bevorzugen

# Aufgaben (2/2)

## ■ Aufgabe: Verbessere den folgenden Command-Parser

```
public boolean parseCommand() {  
    String input = this.scanner.nextLine();  
    String[] args = input.split( );  
    // Parse input  
    if (args[0].equals(add)) {  
        this.model.add(args[1]);  
        return true;  
    } else if (args[0].equals(find)) {  
        this.model.find(args[1]);  
        return true;  
    } else if (args[0].equals(remove)) {  
        this.model.remove(args[1]);  
        return true;  
    }  
    return false;  
}
```

# Aufgaben (2/2) - Lösung(-svorschlag)



Mögliche Erweiterungen:

- **CommandResult** Klasse für Rückgabewert
- **ALL\_COMMANDS** als **HashMap**

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○○

Java API  
○○○○○○

OOP Designprinzipien  
○○○○○○○○○○○○●

Commands  
○○○○

Aufgabe: Commands  
○○

# Ein kleines Musterlösungssystem

## Siehe

`SimpleCommandSystem.java`

→ Einfache Lösung

→ Wird sehr schnell unübersichtlich bei vielen Commands

Organisatorisches  
○

Übungsblatt 4  
○○○

Rekursion: Wiederholung  
○○

Java API  
○○○○○○

OOP Designprinzipien  
○○○○○○○○○○○○○○○○

Commands  
●○○○

Aufgabe: Commands  
○○

# Das "offizielle" Command System

## Was ist das?

- Eines der *twenty-three well-known gang of four design patterns*
- Mehr zu diesen Entwurfsmustern in SWT I



# Das "offizielle" Command System

## Was ist das?

- Eines der *twenty-three well-known gang of four design patterns*
- Mehr zu diesen Entwurfsmustern in SWT I

## Vorteile

- Sorgt für gute Struktur und Schafft Übersichtlichkeit
- Man kann Befehlsausführungen verwalten und so sehr leicht eine undo Funktion implementieren
- Separiert Behfelsstruktur vom Rest der Programmlogik Neue commands können leicht ergänzt werden Lose Kopplung von Befehlen zum Rest des Programmes

# Das "offizielle" Command System

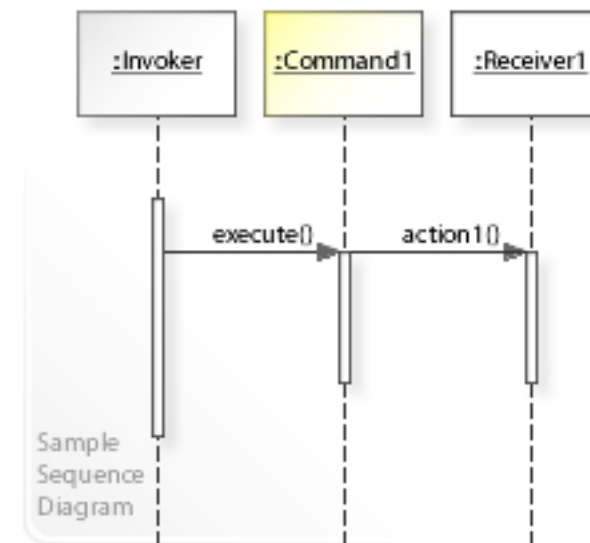
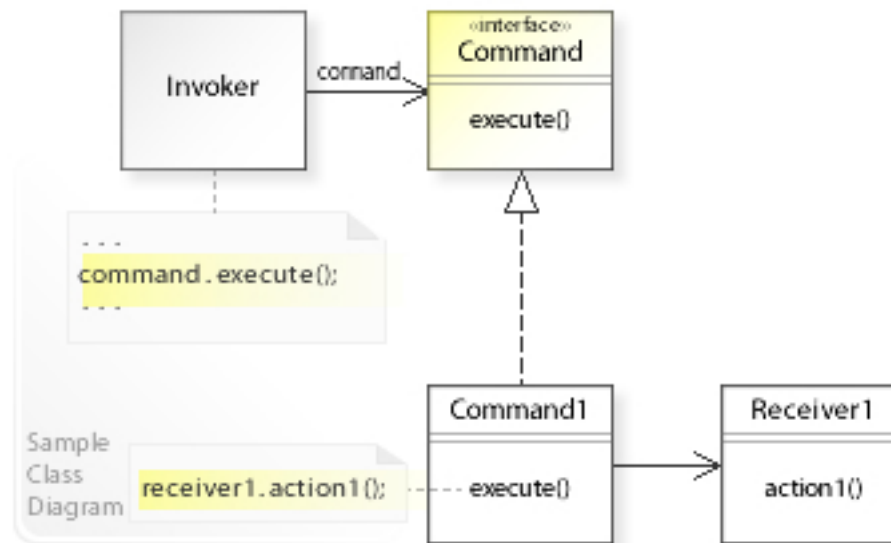
## Bestandteile

- Ein *Command* Interface
  - Hat eine `execute()` Methode
  - Hat idr. Eine Methode zum Namen erhalten
- Eine konkrete Klasse pro Command
  - Implementiert das Command Interface
  - Implementiert alle Methoden aus dem Interface auf passende Weise
  - Hält Referenz auf die nötige(n) Receiver Klasse(n) um Methoden auf diesen Aufrufen zu können
- Eine *Caller* Klasse
  - Ruft den passenden Command auf
  - Kennt die konkreten Command Klassen nicht, nur das Interface
  - Hält idr. eine Liste oder ähnliches aller möglichen Commands

# Das "offizielle" Command System

## Bestandteile

- Eine oder mehrere *Receiver* Klasse(n)
  - Die Klasse an der tatsächlich etwas geändert wird
  - Bei einem Spiel könnte das zum Beispiel die Spiel Klasse sein um die nächste Runde einzuleiten



By Vanderjoe - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=62530466>

# Aufgabe: Commands

## Aufgabe

Schreibe ein kleines Commandsystem mit kompletter Paketstruktur, das sich mit „quit“ beendet. Es soll einen Kleinen Taschenrechner unterstützen, also folgende Befehle implementieren:

- `add <num|ans> <num|ans>` Addiert die zwei Argumente
- `sub <num|ans> <num|ans>` Zieht das zweite vom ersten Argument ab
- `abs <num|ans>` Berechnet den Betrag des Argumentes
- `sqrt <num|ans>` Berechnet die Wurzel des Argumentes
- `ans` Gibt das letzte Ergebnis aus
- `undo` Macht den letzten Befehl rückgängig

# Aufgabe: Commands

## Info

- <num|ans> Steht hier jeweils für eine Zahl oder den String "ans", bei dem das letzte Ergebnis verwendet werden soll
- Arbeitet so weit wie ihr kommt, vereinfacht die Aufgabe falls es schon spät ist
- Es geht nicht um die eigentliche Logik sondern darum das Entwurfsmuster anzuwenden
- Geht davon aus, dass alle Eingaben korrekt sind. Mehr dazu wie man Eingaben validiert gibt's im Tutorium zum parsen