Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

(Teil 06)

Objekte in einer Liste und Listen als Teil eines Objekts

- Wiederholung der Arbeit mit Listen und wie sie als Instanzvariablen verwendet werden können
- Beispiele: **Playlists** und **Freizeitparkfahrten**
- Einführung in die Nutzung von Listen zur Verwaltung von Objekten

Verwendung von Listen als Instanzvariablen

```java
public class Playlist {
    private ArrayList<String> songs;

    public Playlist() {
        this.songs = new ArrayList<>();
    }

    public void addSong(String song) {
        this.songs.add(song);
    }

    public void printSongs() {
        for (String song : this.songs) {
            System.out.println(song);
        }
    }
}
```

- Instanziierung einer **Playlist** mit einer Liste von Songs
- **ArrayList** als zentrale Struktur

Prinzip: Komposition in der Objektorientierung

- **Komposition**: Ein Objekt enthält andere Objekte als Instanzvariablen
- Prinzip: „Hat-eine“-Beziehung zwischen Objekten
- Beispiel: Ein **AmusementParkRide** enthält eine Liste von **Person**-Objekten

Komposition am Beispiel

```
public class AmusementParkRide {
    private ArrayList<Person> riding;

    public AmusementParkRide(String name, int minimumHeight) {
        this.riding = new ArrayList<>();
    }

    public void addPerson(Person person) {
        this.riding.add(person);
    }
}
```

- **AmusementParkRide** speichert Personen, die eine Fahrt machen
- Demonstration der Zusammenarbeit zwischen Klassen und Objekten

Dynamische Beziehungen zwischen Objekten

- Listen ermöglichen dynamische Datenstrukturen, die sich während der Laufzeit ändern
- Beispiel: Hinzufügen oder Entfernen von Personen in einer Fahrt (oder Songs in einer Playlist)
- Erhöhung der Flexibilität und Wiederverwendbarkeit von Code

Komposition und Prinzipien der Programmierung

- Förderung von **Modularität**: Jedes Objekt hat eine klare Verantwortung (z.B. eine Playlist verwaltet Songs)
- **Wiederverwendbarkeit**: Klassen wie **ArrayList** oder **Person** können in verschiedenen Kontexten wiederverwendet werden
- **Erweiterbarkeit**: Neue Funktionalitäten (wie zusätzliche Methoden) können einfach hinzugefügt werden

Objekte in Listen / Stack

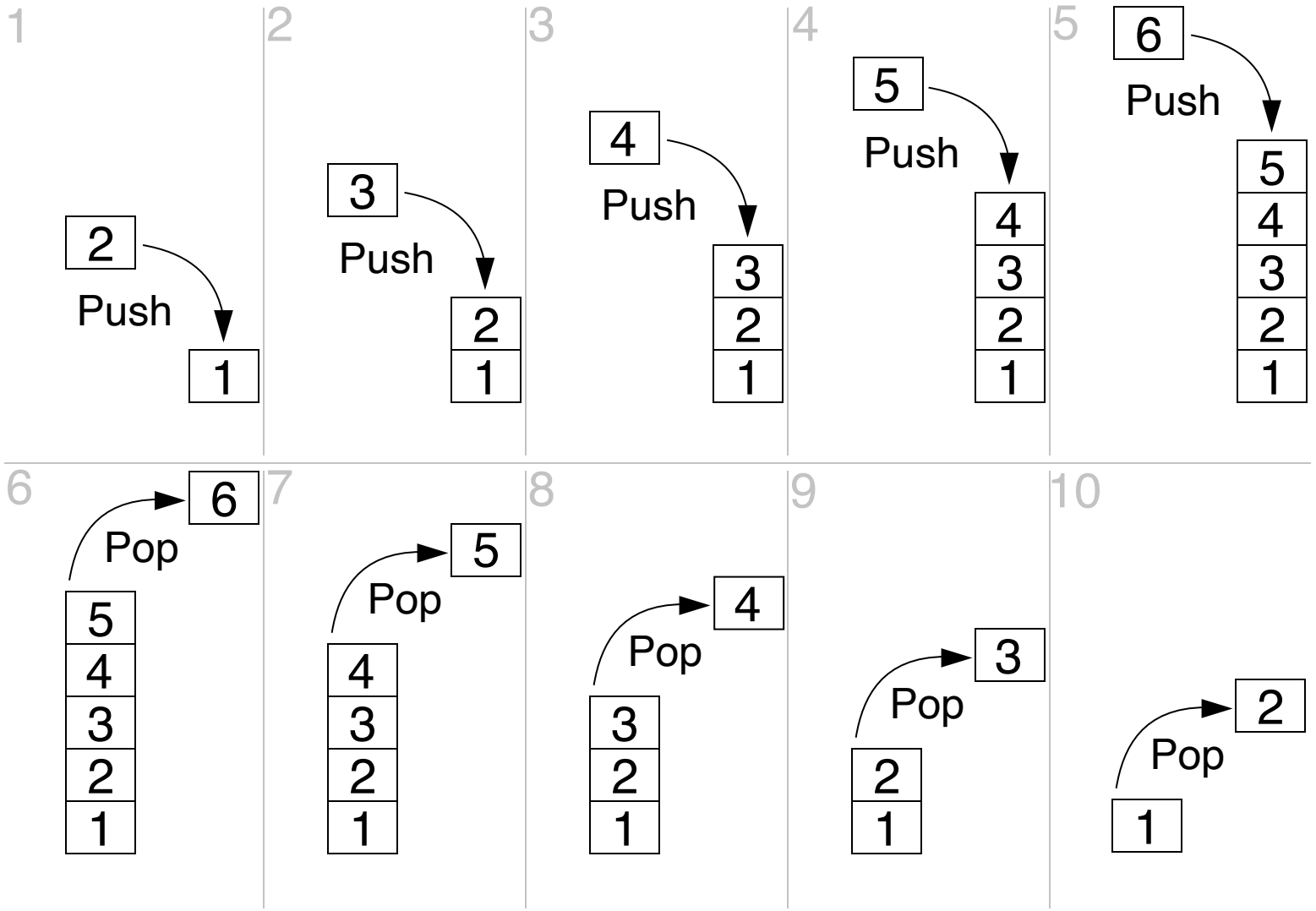
- Stacks als dynamische Datenstruktur, bei der Elemente hinzugefügt und entnommen werden können
- Beispiel einer Klasse für einen Stack:

```
public class Stack {
    private ArrayList<String> elements;

    public void push(String element) { elements.add(element); }
    public String pop() { return elements.remove(elements.size()-1); }
}
```

- Illustration einer praktischen Anwendung von Listen als Stack

Stack


1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1

Verbindungen zu höheren Prinzipien

- **Abstraktion**: Container-Klassen kapseln die Verwaltung von Daten; Benutzer müssen sich nicht um die Details kümmern
- **Verantwortungsteilung**: Jede Klasse übernimmt klar definierte Aufgaben (z.B. Playlist speichert Songs, Stack verwaltet Daten)
- **Flexibilität**: Listen und Stacks bieten anpassbare Datenstrukturen für unterschiedliche Anforderungen in der Programmierung

Leeren der Liste eines Objekts

- **Methode zum Leeren einer Liste**: Mit der Methode `clear()` können alle Elemente aus einer Liste entfernt werden.
- Beispiel: **Leeren** der Liste von Personen, die eine Freizeitparkfahrt machen:

```
public void removeEveryoneOnRide() {
    this.riding.clear();
}
```

Anwendungsfall: Liste von Personen auf einer Freizeitparkfahrt Hurjakuru

- Die Methode `removeEveryoneOnRide()` entfernt alle Personen von der Liste, die auf der Fahrt sind.
- Dynamische Datenverwaltung in Objekten, die Listen als Instanzvariablen enthalten.

```
AmusementParkRide hurjakuru = new AmusementParkRide("Hurjakuru", 140);
hurjakuru.removeEveryoneOnRide();
```

Berechnung einer Summe aus Objekten auf einer Liste

- Eine Methode, die die **Durchschnittsgröße** der Personen auf einer Freizeitparkfahrt berechnet:

```
public double averageHeightOfPeopleOnRide() {
    int sumOfHeights = 0;
    for (Person per : riding) {
        sumOfHeights += per.getHeight();
    }
    return 1.0 * sumOfHeights / riding.size();
}
```

Berechnung aus Listen

- **Durchschnittswerte** aus einer Liste von Objekten berechnen:
  - Summierung der relevanten Daten (z.B. Größe)
  - Division der Summe durch die Anzahl der Objekte
- Rückgabe eines speziellen Werts wie `-1`, wenn die Liste leer ist.

Praxisbeispiel: Berechnung der Durchschnittsgröße

```
Person matti = new Person("Matti");
matti.setHeight(180);
hurjakuru.isAllowedOn(matti);
System.out.println(hurjakuru.averageHeightOfPeopleOnRide());
```

- Der Durchschnitt wird durch Addition der Größen der Personen auf der Fahrt und anschließende Division durch deren Anzahl berechnet (hier nicht gezeigt).

Abrufen eines spezifischen Objekts aus einer Liste

- Finden und Zurückgeben des größten Objekts aus einer Liste:

```
public Person getTallest() {
    Person tallest = riding.get(0);
    for (Person per : riding) {
        if (per.getHeight() > tallest.getHeight()) {
            tallest = per;
        }
    }
    return tallest;
}
```

- Nutzung von **Vergleichen** innerhalb der Schleife, um die Person mit der größten Größe zu finden.

Lernziele (Trennung der Benutzungsschnittstelle von der Programmlogik)

- **Verständnis entwickeln**: Erkennen, wie die Trennung zwischen Benutzungsschnittstelle und Programmlogik eine saubere und modularisierte Softwarearchitektur ermöglicht.
- **Trennung umsetzen**: Eine textbasierte Benutzungsschnittstelle erstellen, die eine klare Trennung zur Anwendungslogik hat.
- **Objektorientierung anwenden**: Verstehen, wie Klassen wie `WordSet` zur Kapselung von Logik beitragen und die Wiederverwendbarkeit des Codes erhöhen.
- **Praxisnahes Programmieren**: Methoden zur Benutzereingabe und -überprüfung entwickeln und in einem schrittweisen Prozess anwenden.

Trennung der Benutzungsschnittstelle von der Programmlogik

- **Ziel**: Eine klare Trennung zwischen der Benutzungsschnittstelle (UI) und der Anwendungslogik schaffen.
- **Vorteil**: Verbesserte Wartbarkeit, Verständlichkeit und Wiederverwendbarkeit des Codes.
- **Beispiel**: Implementierung einer Klasse `UserInterface`, die Benutzereingaben verarbeitet, ohne die Logik direkt zu implementieren.

```
public class UserInterface {
    private Scanner scanner;

    public UserInterface(Scanner scanner) {
        this.scanner = scanner;
    }

    public void start() {
        // UI-Logik ohne Anwendungslogik
    }
}
```

Iteration und Abbruchbedingungen

- **Schrittweise Entwicklung**: Zunächst eine Schleife, die kontinuierlich Eingaben verarbeitet.
- **Abbruchbedingung**: Das Programm soll enden, wenn ein bestimmtes Kriterium erfüllt ist.

```
while (true) {
    String word = scanner.nextLine();
    if (alreadyEntered(word)) {
        break;
    }
}
System.out.println("You gave the same word twice!");
```

Speicherung von Daten

- **Ziel**: Eingegebene Wörter speichern, um die Abbruchbedingung zu überprüfen.
- **Datenstruktur**: Eine `ArrayList` wird verwendet, um die Eingaben zu speichern.

```
private ArrayList<String> words;

public UserInterface(Scanner scanner) {
    this.scanner = scanner;
    this.words = new ArrayList<String>();
}
```

Überprüfung auf Duplikate

- **Methode zur Prüfung**: Die Methode `alreadyEntered` überprüft, ob ein Wort bereits eingegeben wurde.
- **Verbesserung der Logik**: Die Methode kann leicht angepasst oder erweitert werden.

```
public boolean alreadyEntered(String word) {
    return this.words.contains(word);
}
```

Kapselung der Logik in einer eigenen Klasse

- **Wichtiges Prinzip**: Die Logik wird aus der UI ausgelagert, um saubereren und wiederverwendbaren Code zu erzeugen.
- **Klasse `WordSet`**: Diese Klasse kapselt die Verwaltung der eingegebenen Wörter.

```
public class WordSet {
    private ArrayList<String> words = new ArrayList<>();

    public boolean contains(String word) {
        return words.contains(word);
    }

    public void add(String word) {
        words.add(word);
    }
}
```

Erweiterung der Funktionalität

- **Palindrome erkennen**: Neue Funktionalitäten können leicht zur Klasse `WordSet` hinzugefügt werden, ohne die UI zu verändern.

```
public int palindromes() {
    int count = 0;
    for (String word : words) {
        if (isPalindrome(word)) {
            count++;
        }
    }
    return count;
}
```

Konsequente Trennung von Logik und UI

- **Prinzip**: Die UI sollte nur für die Interaktion mit dem Nutzer verantwortlich sein.
- **Erweiterbarkeit**: Änderungen in der Logik haben keine Auswirkungen auf die UI, da die Logik in einer separaten Klasse gekapselt ist.

```
public class UserInterface {
    private WordSet wordSet;
    private Scanner scanner;

    public UserInterface(WordSet wordSet, Scanner scanner) {
        this.wordSet = wordSet;
        this.scanner = scanner;
    }
}
```

```java
public class UserInterface {
    private WordSet wordSet;
    private Scanner scanner;

    public UserInterface(WordSet wordSet, Scanner scanner) {
        this.wordSet = wordSet;
        this.scanner = scanner;
    }

    public void start() {

        while (true) {
            System.out.print("Enter a word: ");
            String word = scanner.nextLine();

            if (this.wordSet.contains(word)) {
                break;
            }

            this.wordSet.add(word);
        }
```

Saubere Abtrennung der UI

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    WordSet set = new WordSet();

    UserInterface userInterface = new UserInterface(set, scanner);
    userInterface.start();
}
```

Von einer Einheit zu vielen Teilen

- Einführung in die Aufteilung eines Programms in kleinere, modulare Teile.
- Beispiel: Punkte in Noten umwandeln und die Verteilung als Sterne anzeigen.
- Ziel: Das Programm effizient und wartbar gestalten.

Beispielprogramm - Als Einheit

```
public class Program {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        ArrayList<Integer> grades = new ArrayList<>();

        while (true) {
            System.out.print("Points: ");
            String input = scanner.nextLine();
            if (input.equals("")) {
                break;
            }

            int score = Integer.valueOf(input);

            if (score < 0 || score > 100) {
                System.out.println("Impossible number.");
                continue;
            }

            int grade = 0;
            if (score < 50) {
                grade = 0; }
            else if // usw

            grades.add(grade);
        }

        System.out.println("");
```

Trennung in kleinere Teile

- **Programmlogik**: Verantwortlich für das Speichern und Verarbeiten der Noten.
- **Benutzungsschnittstelle**: Verantwortlich für die Eingabe der Punkte und die Ausgabe der Ergebnisse.

Ziel: Verbesserung der Lesbarkeit, Wartbarkeit und Testbarkeit des Programms.

Klasse `GradeRegister`

- Kapselt die Logik für die Verarbeitung der Noten.
- Bietet Methoden zum Hinzufügen von Noten basierend auf Punkten und zur Ausgabe der Anzahl der erhaltenen Noten.

```java
public class GradeRegister {
    private ArrayList<Integer> grades;

    public void addGradeBasedOnPoints(int points) {
        this.grades.add(pointsToGrades(points));
    }

    public int numberOfGrades(int grade) {
        // Zählt die Anzahl der erhaltenen Noten
    }

    public static int pointsToGrades(int points) {

        int grade = 0;
        if (points < 50) {
            grade = 0;
        } else if
        // usw.

        return grade;
    }
}
```

Programm nach der Trennung

- Die Logik zur Notenverarbeitung wird in der Klasse `GradeRegister` ausgelagert.
- `main` ist jetzt fokussierter auf die Programminitialisierung.

Beispielprogramm - `GradeRegister` ausgelagert

```
public class Program {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        GradeRegister register = new GradeRegister();

        while (true) {
            System.out.print("Points: ");
            String input = scanner.nextLine();
            if (input.equals("")) {
                break;
            }

            int score = Integer.valueOf(input);

            if (score < 0 || score > 100) {
                System.out.println("Impossible number.");
                continue;
            }

            register.addGradeBasedOnPoints(score);
        }

        System.out.println("");
        int grade = 5;
        while (grade >= 0) {
            int stars = register.numberOfGrades(grade);
            System.out.print(grade + ": ");
            while (stars > 0) {
```

Textbasierte Benutzungsschnittstelle - Hauptprogramm

```java
import java.util.Scanner;

public class Program {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        GradeRegister register = new GradeRegister();

        UserInterface userInterface = new UserInterface(register, scanner);
        userInterface.start();
    }
}
```

Klasse `UserInterface`

- Verwaltet die Benutzereingaben und gibt die Notenverteilung aus.
- Interagiert mit der Programmlogik (Klasse `GradeRegister`).

```
import java.util.Scanner;

public class UserInterface {

    private GradeRegister register;
    private Scanner scanner;

    public UserInterface(GradeRegister register, Scanner scanner) {
        this.register = register;
        this.scanner = scanner;
    }

    public void start() {
        readPoints();
        System.out.println("");
        printGradeDistribution();
    }

    public void readPoints() { // naechste Folie
    }

    public void printGradeDistribution() { // naechste Folie
    }
}
```

```
    public void readPoints() {
        while (true) {
            System.out.print("Points: ");
            String input = scanner.nextLine();
            if (input.equals("")) {
                break;
            }

            int points = Integer.valueOf(input);

            if (points < 0 || points > 100) {
                System.out.println("Impossible number.");
                continue;
            }

            this.register.addGradeBasedOnPoints(points);
        }
    }

    public void printGradeDistribution() {
        int grade = 5;
        while (grade >= 0) {
```

Vorteile der Trennung

- **Modularität**: Bessere Strukturierung des Codes.
- **Wartbarkeit**: Änderungen in der Benutzungsschnittstelle betreffen die Logik nicht und umgekehrt.
- **Wiederverwendbarkeit**: Die Klasse `GradeRegister` kann in anderen Projekten wiederverwendet werden.

Prinzipien der Programmierung: Abstraktion und Modularität

- **Abstraktion**: Die Reduktion auf wesentliche Details, während unwichtige Aspekte verborgen bleiben.
  - Abstrakte Konzepte helfen, **komplexe Probleme zu vereinfachen**.
  - Beispielsweise abstrahieren wir die Interaktion mit einer Datenstruktur (wie einer Liste) von der konkreten Implementierung.
- **Modularität**: Unterteilung eines Programms in kleinere, unabhängig funktionierende Einheiten.
  - **Jede Einheit** (Modul, Klasse) erfüllt eine klar definierte Aufgabe.

Prinzipien der Programmierung: Trennung von Bedenken

- **Separation of Concerns**: Ein fundamentales Prinzip, bei dem **unabhängige Aspekte** eines Programms voneinander getrennt werden.
    - **Beispiel**: Die Benutzungsschnittstelle (UI) sollte keine Logik enthalten, die das Programm steuert – diese gehört in ein eigenes Modul.
- **Wiederverwendbarkeit**: Durch die Trennung von Bedenken wird es möglich, **Teile des Programms** in anderen Kontexten wiederzuverwenden.
    - **Beispiel**: Eine Klasse zur Verwaltung von Noten kann in verschiedenen Anwendungen verwendet werden – sowohl in Konsolenanwendungen als auch in grafischen Benutzungsschnittstellen.

Prinzipien der Programmierung: Kapselung und Informationsversteckung

- **Kapselung**: Ein Prinzip, das den Zugriff auf interne Daten oder Implementierungsdetails eines Objekts beschränkt.
    - **Nur Methoden** eines Objekts sollten auf dessen Daten zugreifen dürfen.
- **Informationsversteckung**: Reduziert die Komplexität, indem nur notwendige Informationen freigegeben werden.
    - **Vorteil**: Andere Teile des Programms müssen nicht wissen, wie intern gearbeitet wird, sondern nur, **was** eine Komponente tut.

Beispiel: Kapselung und Modularität

Im Beispiel der Notenverwaltung:

- **Kapselung**: Die Klasse `GradeRegister` kapselt die Logik zur Verwaltung der Noten.
- **Modularität**: Die `UserInterface`-Klasse übernimmt nur die Verantwortung für die Interaktion mit dem Benutzer.

```
public class GradeRegister {
    private ArrayList<Integer> grades;

    public void addGrade(int grade) { grades.add(grade); }
    public ArrayList<Integer> getGrades() { return grades; }
}
```

Prinzipien der Programmierung: Single Responsibility Principle (SRP)

- **Ein Modul, eine Verantwortung**: Jede Klasse oder Methode sollte genau **eine** Verantwortung haben.
    - **Beispiel**: Eine Klasse sollte entweder die Benutzereingaben verwalten oder die Notenlogik. Niemals beides.
- **Vorteile**:
    - **Einfacheres Testen**: Jede Komponente kann isoliert getestet werden.
    - **Wartbarkeit**: Änderungen an einer Funktionalität wirken sich nicht auf andere Module aus.

Verwendete Prinzipien der Programmierung

- **Verifikation und Validierung**: Durch die **Trennung von Logik und Benutzungsschnittstelle** wird es einfacher, einzelne Module zu testen und zu verifizieren.
- **Skalierbarkeit**: Ähnlich wie wissenschaftliche Modelle erweiterbar sind, ist auch der Code durch klare Trennung leicht **skalierbar**.
    - **Beispiel**: Die `GradeRegister`-Klasse könnte um komplexe Berechnungen erweitert werden, ohne die Benutzungsschnittstelle zu beeinflussen.

Zusammenfassung: Prinzipien der Programmierung im Kontext

- **Kohäsion**: Jede Komponente des Programms hat einen klaren Zweck.
- **Modularität und Abstraktion**: Teile des Programms sind klar getrennt, um Komplexität zu reduzieren.
- **Wiederverwendbarkeit**: Durch Trennung der Logik und Benutzungsschnittstelle entsteht flexibler, wiederverwendbarer Code.
- Diese Prinzipien entsprechen **wissenschaftlichen Ansätzen**, bei denen komplexe Systeme durch klar definierte Module und Abstraktionen modelliert werden.

Einführung in das Model-View-Controller (MVC) Muster

- **Model**: Enthält die Daten und die Logik der Anwendung.
- **View**: Präsentiert Daten dem Benutzer und nimmt Benutzereingaben entgegen.
- **Controller**: Vermittelt zwischen Model und View, verarbeitet Benutzereingaben und aktualisiert Model und View entsprechend.

MVC und die Trennung von Verantwortlichkeiten

- **Ziel**: Noch klarere Trennung der Komponenten, um Wartbarkeit und Erweiterbarkeit zu erhöhen.
- **Vorteil**: Jede Komponente kann unabhängig entwickelt, getestet und modifiziert werden.
- In den vorherigen Beispielen wurde diese Trennung **nicht konsequent** umgesetzt.

Anwendung von MVC auf unser Beispiel

Wir erstellen eine Anwendung, die Benutzereingaben verarbeitet und eine Liste von Aufgaben verwaltet.

- **Model**: Klasse `TaskList`, die Aufgaben speichert und verwaltet.
- **View**: Klasse `TaskView`, die das UI darstellt.
- **Controller**: Klasse `TaskController`, die die Eingaben verarbeitet und Model und View koordiniert.

Das Model: Klasse `TaskList`

```java
public class TaskList {
    private ArrayList<String> tasks;

    public TaskList() {
        this.tasks = new ArrayList<>();
    }

    public void addTask(String task) {
        this.tasks.add(task);
    }

    public void removeTask(int index) {
        if(index >= 0 && index < tasks.size()) {
            this.tasks.remove(index);
        }
    }

    public ArrayList<String> getTasks() {
        return new ArrayList<>(this.tasks);
    }
}
```

Die View: Klasse `TaskView`

```java
import java.util.Scanner;

public class TaskView {
    private Scanner scanner;

    public TaskView(Scanner scanner) {
        this.scanner = scanner;
    }

    public void displayTasks(ArrayList<String> tasks) {
        System.out.println("Aktuelle Aufgaben:");
        for(int i = 0; i < tasks.size(); i++) {
            System.out.println((i+1) + ": " + tasks.get(i));
        }
    }

    public String getUserInput() {
        System.out.print("Eingabe: ");
        return scanner.nextLine();
    }

    public void showMessage(String message) {
        System.out.println(message);
    }
}
```

Der Controller: Klasse `TaskController`

```
public class TaskController {
    private TaskList model;
    private TaskView view;

    public TaskController(TaskList model, TaskView view) {
        this.model = model;
        this.view = view;
    }

    public void start() {
        while(true) {
            view.displayTasks(model.getTasks());
            view.showMessage("Optionen: [1] Aufgabe hinzufügen, [2] Aufgabe entfernen, [0] Beenden");
            String input = view.getUserInput();
            if(input.equals("0")) {
                break;
            } else if(input.equals("1")) {
                view.showMessage("Neue Aufgabe eingeben:");
                String task = view.getUserInput();
                model.addTask(task);
            } else if(input.equals("2")) {
                view.showMessage("Nummer der zu entfernenden Aufgabe:");
                int index = Integer.parseInt(view.getUserInput()) - 1;
                model.removeTask(index);
            } else {
                view.showMessage("Ungültige Option.");
            }
        }
    }
}
```

Hauptprogramm mit MVC-Struktur

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        TaskList model = new TaskList();
        Scanner scanner = new Scanner(System.in);
        TaskView view = new TaskView(scanner);
        TaskController controller = new TaskController(model, view);
        controller.start();
    }
}
```

Vorteile der Verwendung von MVC

- **Klare Trennung**: Daten, Darstellung und Steuerung sind voneinander getrennt.
- **Bessere Wartbarkeit**: Änderungen in einer Komponente beeinflussen die anderen nicht direkt.
- **Erweiterbarkeit**: Einfaches Hinzufügen neuer Features oder Austausch einer Komponente (z.B. GUI statt Konsole).

Zusammenfassung

- Das MVC-Muster hilft dabei, Anwendungen sauber zu strukturieren.
- In unseren vorherigen Beispielen haben wir die Logik und die UI getrennt, aber **nicht konsequent** das MVC-Muster angewendet.
- Durch die Einführung eines Controllers können wir die Trennung weiter verbessern und die Prinzipien der Programmierung effektiver umsetzen.

Lernziele (Testen, Fehler, Verifikation)

- Sie können einige Probleme beschreiben, die durch Softwarefehler verursacht werden.
- Sie wissen, was ein Stack Trace ist, kennen die Schritte zur Fehlerbehebung und können einem Scanner textuelle Testeingaben geben.
- Sie wissen, was Unit-Tests sind und können Unit-Tests schreiben.
- Sie sind mit testgetriebener Softwareentwicklung vertraut.

Softwarefehler und ihre Folgen

- **Softwarefehler** können erhebliche Auswirkungen haben.
  - Ein Beispiel ist der Fehler bei der Verwendung von unterschiedlichen Maßeinheiten, der zur Zerstörung eines Satelliten führte (Mars Climate Orbiter der NASA, der Satellit ging 1999 verloren, weil eine Softwareeinheit des Bodenteams Maße in Pound-Force (lb-f) lieferte, während das Raumfahrzeug Newton (N) erwartete).
  - Fehler können sowohl kleine Unannehmlichkeiten als auch katastrophale Folgen verursachen.
- **Fehler zu machen ist unvermeidlich**, aber sie sind der beste Weg zu lernen. Man sollte keine Angst davor haben, Fehler zu machen und sie bewusst zu analysieren.

Stack Trace: Ein nützliches Werkzeug

- Ein **Stack Trace** zeigt die Liste der Methodenaufrufe, die zu einem Fehler geführt haben.
- Beispiel:

```
Exception in thread "main" ...
    at Program.main(Program.java:15)
```

- Dies hilft, den genauen Ort des Fehlers zu identifizieren, um gezielt debuggen zu können.

Beispiel: Stack Trace in Java

Ein **Stack Trace** zeigt die Methode und die Zeile, in der ein Fehler aufgetreten ist, sowie die Kette von Methoden, die zum Fehler geführt haben.

Beispiel-Code

```
public class StackTraceExample {
    public static void main(String[] args) {
        try {
            causeError();
        } catch (Exception e) {
            e.printStackTrace(); // Gibt den Stack Trace aus
        }
    }

    public static void causeError() {
        String str = null;
        str.length(); // Fehler: NullPointerException
    }
}
```

Erklärung des Stack Trace

Beispiel-Stack-Trace:

```
java.lang.NullPointerException: Cannot invoke "String.length()" because "str" is null
    at StackTraceExample.causeError(StackTraceExample.java:11)
    at StackTraceExample.main(StackTraceExample.java:5)
```

Wichtige Punkte:

1. **Art des Fehlers**:
   - `java.lang.NullPointerException`: `str` ist `null`.
   - Nachricht: `Cannot invoke "String.length()" because "str" is null`.
2. **Methode und Zeile des Fehlers**:
   - Fehler trat in der Methode `causeError` auf (Zeile 11).
   - `main` hat `causeError` aufgerufen (Zeile 5).
3. **Rückverfolgung (Call Stack)**:
   - Der Stack Trace zeigt die Abfolge der Methoden, die zum Fehler geführt haben. Dies hilft beim Debugging.

Fehlerbehebung: Eine Checkliste

1. Überprüfen Sie die Syntax (fehlende Klammern, falsche Variablennamen).
2. Testen Sie verschiedene Eingaben, um herauszufinden, wann der Fehler auftritt.
3. Verwenden Sie Ausgabebefehle, um den Wert von Variablen zu verfolgen.
4. Stellen Sie sicher, dass alle Variablen initialisiert sind.
5. Verwenden Sie den **Debugger**, um schrittweise durch den Code zu gehen.

Automatisierte Testeingaben

- Es ist möglich, dem **Scanner**-Objekt eine Testeingabe als String zu übergeben.
  - Dies vereinfacht das Testen, ohne dass manuell Eingaben gemacht werden müssen.
  - **Beispiel**:

```
String input = "one\n" + "two\n" + "three\n";
Scanner reader = new Scanner(input);
```

Unit-Tests: Automatisiertes Testen kleiner Einheiten

- **Unit-Tests** testen einzelne Methoden und Klassen isoliert.
- **Vorteile**:
  - Sie gewährleisten, dass jede Komponente korrekt funktioniert.
  - Sie machen das Testen großer Programme einfacher, da Tests systematisch geschrieben werden.
- Beispiel eines Unit-Tests:

```
@Test
public void calculatorInitialValueZero() {
    Calculator calculator = new Calculator();
    assertEquals(0, calculator.getValue());
}
```

Java-Annotationen: Ein (völlig unvollständiger) Überblick

Annotation	Obligatorisch?	Wichtig für die Funktionalität?	Beschreibung
`@Override`	Nein	Nein, aber schützt vor Fehlern bei Überschreibungen.	Kennzeichnet Methoden, die eine Methode überschreiben.
`@Test`	Ja (für Tests)	Ja, ohne wird die Methode nicht als Test erkannt.	Markiert eine Methode als Testmethode (z. B. in JUnit).
`@Deprecated`	Nein	Nein, aber zeigt an, dass eine Methode/Klasse veraltet ist.	Kennzeichnet Elemente, die in Zukunft entfernt werden könnten.
`@SuppressWarnings`	Nein	Nein, dient nur zur Unterdrückung von Warnungen.	Unterdrückt Compiler-Warnungen, z. B. für nicht verwendeten Code.

Wichtige Punkte zu Java-Annotationen

Überblick

- `@Override` : Nicht erforderlich, aber schützt vor Fehlern (z. B. Tippfehler bei Methodenüberschreibungen).
- `@Test` : Zwingend notwendig in Test-Frameworks wie JUnit, damit Methoden als Testmethoden erkannt werden.
- `@Deprecated` : Nicht obligatorisch, aber nützlich, um veraltete APIs zu kennzeichnen und Warnungen zu erzeugen.
- `@SuppressWarnings` : Nützlich, um unnötige Compiler-Warnungen zu unterdrücken (z. B. bei generischen Typen).

Prinzipien der Unit-Tests

- **Single Responsibility Principle (SRP)**: Jede Klasse sollte nur eine Verantwortung haben, was Unit-Tests vereinfacht.
- **Wiederholbarkeit**: Tests sollten reproduzierbar sein und unabhängig von der Umgebung dieselben Ergebnisse liefern.
- **Schnelligkeit**: Unit-Tests sollten schnell ausgeführt werden, um die Entwicklung effizient zu unterstützen.

Testgetriebene Entwicklung (TDD)

- **Testgetriebene Entwicklung (TDD)** ist ein Ansatz, bei dem zuerst Tests geschrieben werden, bevor die Funktionalität implementiert wird.
- **Schritte**:
    i. Schreiben Sie einen Test.
    ii. Führen Sie die Tests aus und überprüfen Sie, ob sie fehlschlagen.
    iii. Schreiben Sie den Code, der den Test besteht.
    iv. Führen Sie die Tests erneut aus.
    v. Refaktorisieren Sie den Code bei Bedarf.

Test Beispiel: `Calculator`-Klasse

```java
public class Calculator {

    private int value;

    public Calculator() {
        this.value = 0;
    }

    public void add(int number) {
        this.value = this.value + number;
    }

    public void subtract(int number) {
        this.value = this.value + number;
    }

    public int getValue() {
        return this.value;
    }
}
```

Erster Unit-Test: Initialer Wert des Calculators

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculatorTest {

    @Test
    public void calculatorInitialValueZero() {
        Calculator calculator = new Calculator();
        assertEquals(0, calculator.getValue());
    }
}
```

Testausgabe:

[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] BUILD SUCCESS

Test für Addition

```java
@Test
public void valueTenWhenTenAdded() {
    Calculator calculator = new Calculator();
    calculator.add(10);
    assertEquals(10, calculator.getValue());
}
```

Testausgabe:

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] BUILD SUCCESS

Test für Subtraktion (Fehler)

```java
@Test
public void valueMinusTwoWhenTwoSubtracted() {
    Calculator calculator = new Calculator();
    calculator.subtract(2);
    assertEquals(-2, calculator.getValue());
}
```

Testausgabe (Fehler):

[ERROR] CalculatorTest.valueMinusTwoWhenTwoSubtracted -- expected: <-2> but was: <2>

[INFO] BUILD FAILURE

Fehlerkorrektur in der `Calculator`-Klasse

```
public void subtract(int number) {
    this.value = this.value - number;
}
```

Fehlerarten (unvollständig)

Fehlerart	Beschreibung	Typ
Build Failure	Der Build-Prozess konnte nicht abgeschlossen werden, häufig aufgrund von Problemen beim Testen oder Zusammenstellen des Codes.	Build-Prozess
Compilation Failure	Der Code lässt sich aufgrund von Syntaxfehlern oder Typfehlern nicht kompilieren.	Build-Prozess
Test Failure	Ein Unit-Test schlägt fehl, weil das erwartete Ergebnis nicht mit dem tatsächlichen übereinstimmt.	Test-Prozess
Runtime Error	Der Code wird zwar erfolgreich kompiliert, führt aber zur Laufzeit zu einem Fehler (z.B. NullPointerException).	Ausführungsfehler
Assertion Error	Ein Test schlägt fehl, weil eine Assert-Anweisung nicht erfüllt wird.	Test-Prozess
Initialization Error	Ein Fehler tritt auf, bevor Tests überhaupt ausgeführt werden können, oft aufgrund von Problemen bei der Testinitialisierung.	Test-Prozess
(Linker Error)	Fehler, die durch fehlerhafte Referenzen auf externe Bibliotheken entstehen, insbesondere beim Kompilieren und Binden von Abhängigkeiten.	Build-Prozess
Timeout Error	Ein Test hat die vorgegebene Zeitüberschreitung erreicht, bevor er abgeschlossen wurde.	Test-Prozess

Zusammenfassung

- **Fehler und Tests**: Fehler sind unvermeidlich, aber durch systematisches Testen und Debugging kann ihre Zahl reduziert werden.
- **Unit-Tests** helfen dabei, Programme stabiler und wartbarer zu machen, indem sie kleine Einheiten testen.
- **TDD** fördert saubere und testbare Code-Entwicklung, indem Tests zuerst geschrieben werden.

Testen und Verifikation von Code

Das Testen und die Verifikation von Software sind entscheidend für die Qualitätssicherung in der Softwareentwicklung. Beide Ansätze zielen darauf ab, Fehler im Code zu entdecken und sicherzustellen, dass Programme wie erwartet funktionieren.

- **Testen**: Überprüft die Korrektheit des Programms basierend auf bestimmten Eingaben und erwarteten Ausgaben.
- **Verifikation**: Stellt (evtl. sogar mathematisch) sicher, dass der Code in allen möglichen Fällen korrekt funktioniert, oft durch formale Methoden.
- **Verifikation** fragt: "Haben wir die Software / das Programm / das Produkt richtig gebaut?", **Validierung** fragt: "Haben wir das richtige Produkt gebaut?"

Methoden zur Überprüfung von Software

Es gibt verschiedene Test- und Verifikationsmethoden:

1. **Unit-Tests**: Testen von isolierten Codekomponenten (z.B. Methoden oder Klassen) auf Korrektheit.
2. **Integrationstests**: Überprüfen das Zusammenspiel mehrerer Komponenten.
3. **Systemtests**: Überprüfen die gesamte Anwendung.
4. **Formale Verifikation**: Beweist mathematisch die Korrektheit eines Programms oder Algorithmus durch Invarianten und Beweise.

Unit-Tests als Verifikationsmethode

- **Unit-Tests**: Eine praxisorientierte Methode, die auf das Testen kleiner Teile des Programms (z.B. eine Methode) abzielt.
- Sie prüfen die Funktionalität durch gezielte Eingaben und überprüfen die erwartete Ausgabe.
- Vorteile: Schnell durchführbar, unterstützt kontinuierliche Integration.

Beispiel eines Unit-Tests in Java:

```java
@Test
public void testAddition() {
    Calculator calculator = new Calculator();
    calculator.add(5);
    assertEquals(5, calculator.getValue());
}
```

Formale Verifikation und Invarianten

- **Formale Verifikation**: Ein mathematischer Ansatz zur Verifizierung der Programmkorrektheit.
- **Schleifeninvariante**: Eine Eigenschaft, die zu jedem Zeitpunkt der Schleifenausführung wahr ist.

Beispiel: Formale Verifikation einer Schleife

Verifikation einer Schleifeninvariante:

```
void methode(int n)
{
    int k = 0;
    int x = 0;
    int m = 2*n;
    while (k < m) {
        x = x + k + 1;
        k = k + 2;
    }
    // Invarianten: (1) k ist gerade, (2) x=(k/2)^2
}
```

für die Invariante "k ist gerade" (verkürzt):

1. **Basisfall** k ist bei ersten Erreichen der Schleife gerade (k = 0)
2. **Induktionsschritt**: Angenommen, die Invariante gilt bei jedem Eintritt der Schleife. Da k um zwei erhöht wird, gilt die Invariante auch beim Austritt.

Sehr bekannte Beispiele für Softwarekatastrophen durch Fehler

1. **Ariane 5 Explosion (1996)**:
    - Fehler: Konvertierung eines 64-Bit-Gleitkommawerts in eine 16-Bit-Ganzzahl verursachte einen Überlauf.
    - Resultat: Raketenzerstörung 37 Sekunden nach dem Start.
2. **Therac-25 Strahlenunfälle (1985-87)**:
    - Fehler: Softwareproblem in einer Strahlentherapiemaschine führte zu Überdosierung von Strahlung.
    - Resultat: Mehrere Todesfälle.
3. **Mars Climate Orbiter (1999)**:
    - Fehler: Einheitenverwechslung (metrisch vs. imperial) führte zu fehlerhaften Berechnungen.
    - Resultat: Verlust der Raumsonde im Wert von 125 Millionen US-Dollar.

Einige wenige Softwarefehler der letzten 10 Jahre

Boeing 737 MAX MCAS Software (2018-2019)

- **Problem**: Fehlfunktion des Maneuvering Characteristics Augmentation System (MCAS) durch falsche Sensordaten.
- **Konsequenz**: Zwei Flugzeugabstürze (Lion Air Flug 610, Ethiopian Airlines Flug 302).
- **Todesopfer**: 346.
- **Ursache**: Falsche Sensordaten und unzureichende Software-Redundanz.
- **Ergebnis**: Weltweites Grounding der 737 MAX, Milliardenverluste.

British Airways IT-Ausfall (2017)

- **Problem**: IT-Ausfall legte das weltweite Buchungssystem und Gepäckabwicklung lahm.
- **Konsequenz**: Über 75.000 betroffene Passagiere, tausende Flugstreichungen.
- **Ursache**: Energieversorgungsproblem und mangelnde Redundanz.
- **Ergebnis**: Millionenverluste und Reputationsschaden.

Volkswagen Dieselgate (2015)

- **Problem**: Manipulative Software erkannte Emissionstests und reduzierte nur während dieser Tests die Emissionen.
- **Konsequenz**: Milliardenstrafen, Rückrufaktionen, Reputationsschaden.
- **Ursache**: Software zur Umgehung von Emissionsstandards.
- **Ergebnis**: Größter Automobilskandal des Jahrzehnts.

NHS Test and Trace (2020, UK)

- **Problem**: Verlust von Tausenden von COVID-19-Testfällen durch Verwendung eines ungeeigneten Excel-Formats.
- **Konsequenz**: Verzögerungen bei der Kontaktverfolgung, mögliche weitere Virusverbreitung.
- **Ursache**: Softwarearchitekturfehler, Verwendung veralteter Formate.
- **Ergebnis**: Große Ineffizienz im Krisenmanagement.

T-Mobile Netzwerk-Ausfall (2020, USA)

- **Problem**: Fehlerhafte Routerkonfiguration führte zu Netzwerkausfällen.
- **Konsequenz**: Millionen von Kunden hatten stundenlang keinen Zugang zu Mobilfunkdiensten, einschließlich Notrufdiensten.
- **Ursache**: Netzwerkfehlkonfiguration, problematische Softwareupdates.
- **Ergebnis**: Massiver Reputationsschaden für T-Mobile.

- Diese Beispiele zeigen die realen Auswirkungen von Softwarefehlern.
- Sie verdeutlichen die Notwendigkeit strikter Test- und Verifikationsmethoden.
- Fehler in Software können weitreichende Folgen haben, von finanziellen Verlusten bis hin zu menschlichen Opfern.

Zusammenfassung

- **Testen**: Praktische Methode, Fehler im Code zu finden. Unit-Tests sind verbreitet und effizient.
- **Verifikation**: Mathematische Sicherheit, dass Code korrekt ist. Formale Methoden wie Invarianten spielen hier eine wichtige Rolle.
- **Beispiele**: Fehlerhafte Software kann katastrophale Folgen haben, was die Bedeutung von Testen und Verifikation betont.

Hintergrund: Prinzipien der Programmierung in Bezug auf Tests

- **Abstraktion**: Die Tests isolieren und abstrahieren die zu testende Logik von der restlichen Anwendung.
- **Modularität**: Jede Komponente sollte testbar sein, was bedeutet, dass sie eine klar definierte Verantwortung haben muss.
- **Kohäsion**: Klassen und Methoden sollten fokussiert sein, um klare und präzise Tests zu ermöglichen.

Ende Teil 06