DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Das Simpsons-Quiz: Finde deinen Zwilling in Springfield!

Programmentwurf

von

Dominik Veith

Abgabedatum:	28. Mai 2023
Bearbeitungszeitraum:	04.10.2022 - 28.05.2023
Matrikelnummer, Kurs:	3352220, TINF20B2

Inhaltsverzeichnis

Abkürzungsverzeichnis

JDK	Java Development Kit
IDE	Integrated Development Environment
ASCII	American Standard Code for Information Interchange
POM	Project Object Model

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einführung

1.1 Übersicht über die Applikation

Die Fernsehserie 'Die Simpsons' ist eine US-amerikanische Zeichentrickserie, die seit 1989 ausgestrahlt wird und mittlerweile über 700 Episoden hat. Die Serie handelt von der Familie Simpson, bestehend aus Homer, Marge, Bart, Lisa und Maggie, sowie zahlreichen Nebenfiguren, die in der fiktiven Stadt Springfield leben. Die Simpsons zeichnet sich durch ihren satirischen und humorvollen Stil aus, der sowohl politische als auch soziale Themen behandelt. Die Serie parodiert oft bekannte Filme, Serien, Persönlichkeiten und Institutionen und enthält viele popkulturelle Referenzen. Die Simpsons wurde mit zahlreichen Preisen ausgezeichnet, darunter 34 Emmy Awards, und gilt als eine der erfolgreichsten Fernsehserien aller Zeiten. Die Serie wurde in mehr als 100 Ländern ausgestrahlt und hat eine große Fangemeinde auf der ganzen Welt.[1]
Die Applikation Simpsons-Quiz ist ein Terminal-basierendes Minispiel. Durch gezielte Fragen, welche User:innen mittels Tastatureingaben beantworten, soll bestimmt werden, durch welchen Charakter des Simpsons Universum er oder sie am ehesten repräsentiert wird. Zusätzlich werden Informationen über den Zielcharakter ausgegeben. Dies umfasst den Wohnort, den Arbeitsplatz, die Art der Fortbewegung und das Lieblingsessen der Simpsons-Figur. Zusätzlich werden individuelle, charakterspezifische Fakten präsentiert. Die visuelle Ausgabe im Terminal wird durch eine American Standard Code for Information Interchange (ASCII)- Repräsentation des verifizierten Charakterbildes unterstützt. Nachdem alle Fragen beantwortet wurden und der oder die User:in seinen Simpsons-Charakter mit Erläuterungen erhalten hat, werden alle Informationen zusätzlich in einer Textdatei abgelegt um sie später noch einmal nachlesen zu können.

1.2 Start der Applikation

Zum Start der Applikation sind folgende Voraussetzungen notwendig:

- Das Java Development Kit (JDK) um den Code kompilieren und auszuführen zu können.
- Ein Integrated Development Environment (IDE) um die Ausführung des Codes komfortabler zu gestalten.

Um die Applikation zu starten kann der Code innerhalb einer IDE der Wahl geöffnet werden. Danach muss die Java Klasse 'SimpsonsTerminal' im Ordner 'SimpsonsQuiz/plugin-0/src/main/java/com/dhbw/ase/simpsons/plugin/SimpsonsTerminal.java' ausgeführt werden. Alle Interaktionen der Applikation mit dem User erfolgen anhand einer textbasierten Ausgabe über das Terminal der IDE. Im Anschluss an die Ausführung der Applikation wird eine Textdatei mit dem Namen 'YourCharacter.txt' erstellt. Diese enthält alle Informationen über den Simpsons-Charakter, welcher durch die Antworten des Users bestimmt wurde und im Terminal zu sehen waren.

1.3 Testen der Applikation

Um die Applikation zu testen gelten folgende Voraussetzungen:

- Maven ist installiert
- Optional: Die IDE IntelliJ ist installiert

Zum Start des Test ist sicherzustellen, dass der Code in der IDE der Wahl geöffnet ist. Danach können alle Tests-Klassen im Verzeichnis unter:
SimpsonsGame/src/test/java/de/dhbw/ase/simpsons gefunden und einzeln ausgeführt werden.
Alternativ kann im Wurzelverzeichnis 'SimpsonsGame' auch die Kommandozeile geöffnet werden und mit dem Befehl 'mvn test' alle Tests ausgeführt werden.

2 Clean Architecture

2.1 Was ist Clean Architecture?

Die Clean Architecture ist ein Architekturmuster, das sich auf die Trennung von Verantwortlichkeiten und die Abhängigkeiten zwischen den Schichten konzentriert. Die Architektur besteht aus mehreren Schichten, die von innen nach außen, wie in Abbildung 2.1 zu sehen, angeordnet sind:

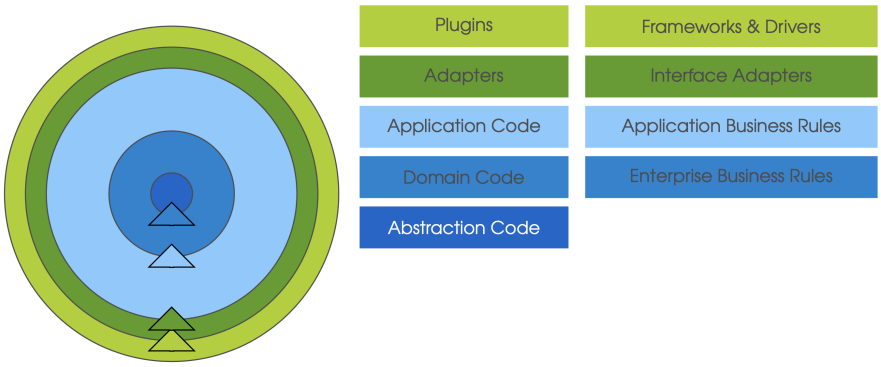

Abbildung 2.1: Clean Architecture [2]

1. **Abstraction Code (Schicht 4):** Code, welcher Konzepte, grundlegende Algorithmen und Datenstrukturen implementiert. Dieser Code enthält Domänenübergreifendes Wissen und wird nur selten berührt.
2. **Domain Code (Schicht 3):** Diese Schicht enthält die Kernlogik der Anwendung und wird am seltensten geändert. Hier werden die Geschäftsregeln und -modelle definiert. Die Entitäten der Domäne werden von der Application Schicht verwendet.
3. **Application Code (Schicht 2):** Diese Schicht enthält die Geschäftslogik der Anwendung(Use Cases welche direkt aus Anforderungen resultieren) und koordiniert den Datenfluss mit Hilfe der Entitäten der Domäne. Dabei sind die Regeln der anwendungsspezifischen Logik nicht projektweit gültig.

4. **Adapters (Schicht 1):** In der Adapter Schicht erfolgt die Kommunikation nach außen.Bei der Interaktion mit Plugins und anderen Anwendungen werden die von dort eingehenden Daten mit bereitgestellten Schnittstellen in interne Formate umgewandelt. Das primäre Ziel der Schicht ist die Entkopplung von innen und außen.

5. **Plugins(Schicht 0):** Die Plugin Schicht interagiert grundsätzlich nur mit der Adapterschicht und enthält unter keinen Umständen Anwendungslogik. Als Beispiel sind an der Stelle Frameworks oder Benutzeroberflächen zu nennen.

In der Betrachtung der Schichten von innen nach außen sei an der Stelle die Langlebigkeit des Codes erwähnt, welche nach außen hin abnimmt. Je weiter außen die Schicht liegt, desto häufiger wird sie geändert. Dabei weiß eine innere Schicht nichts von der äußeren, nur die äußere Schicht verwendet die innere. Das hat das Ziel, Code langlebiger zu machen und bei Änderungen der Technologien eine unveränderte Anwendung zu erhalten.

2.2 Analyse der Dependency Rule

Im Projekt wurde dies konkret durch die Verwendung von Maven umgesetzt. Maven ist ein Build-Management-Tool, welches die Abhängigkeiten zwischen den einzelnen Schichten verwaltet. Es folgen zwei Beispiele, welche die Dependency Rule einhalten und verletzen.

2.2.1 Positiv-Beispiel

Als Positiv Beispiel sei an der Stelle die Klasse UserBuild eingeführt, wie in Abbildung 2.2 zu sehen. Die Klasse liegt in der Adapter Schicht, da sie die Aufgabe hat die jeweilige Simpsons-Figur der Wahl aus den inneren Schichten zu vermitteln. Verwendet wird die Klasse in der QuestionManager Klasse der Application Schicht. Dadurch bleiben die Abhängigkeiten von Adapters zu Application Code bis zu Domain Code, in dem die separaten Klassen der Simpsons Figuren liegen, erhalten.

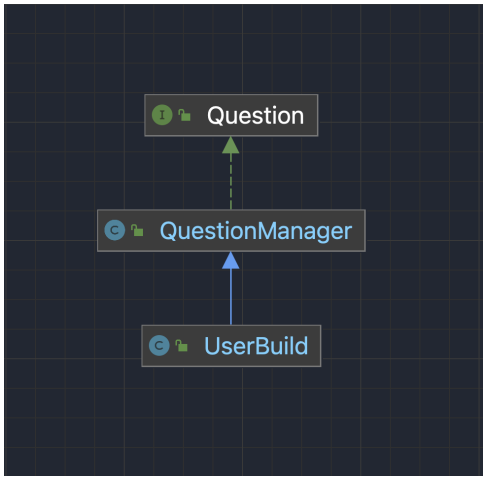

Abbildung 2.2: UML Klassendiagramm (UserBuild) Positiv Beispiel Dependency Rule

2.2.2 Negativ-Beispiel

Ein Negativ Beispiel oder Verletzung der Dependency Rule lässt sich leider nicht zeigen, da die Dependency Rule in unserem Projekt nicht verletzt wurde. Dies liegt daran, dass die einzelnen Schichten durch Maven verwaltet werden und somit die Abhängigkeiten zwischen den einzelnen Schichten klar definiert sind. Listing 2.1 zeigt die Abhängigkeiten welche im Project Object Model (POM) der Adapter Schicht festgelegt wurden.
Ein mögliches Negativ Beispiel wäre, wenn zum Beispiel die Klasse Apu aus der Domain Schicht mit der introduce() Methode das Banner der GameTerminal Klasse verwenden würde. Dadurch würde ein Verstoß gegen die Dependency Rule vorliegen, da die Domain Schicht nicht auf die Adapter Schicht zugreifen darf.

```
1        <dependency>
2            <groupId>com.dhbw.ase</groupId>
3            <artifactId>application-2</artifactId>
4            <version>1.0-SNAPSHOT</version>
5        </dependency>
```

Listing 2.1: Dependency Rule in der Adapter Schicht

2.3 Analyse der Schichten

In diesem Abschnitt werden zwei Klassen aus verschiedenen Schichten der Clean Architecture vorgestellt.

2.3.1 Domain Code: Apu

Zunächst sei die Klasse Apu, wie in Abbildung 2.3 zu sehen, eingeführt. Die Klasse Apu

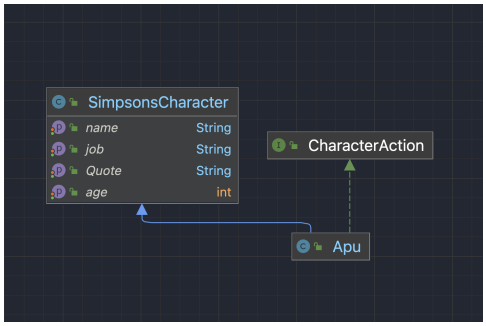

Abbildung 2.3: UML Klassendiagramm (Apu)

ist eine Entität in der Domain Schicht und eine von 10 Figuren der Simpsons, welche in diesem Projekt implementiert wurden. Sie muss nicht angepasst werden, da sich die Eigenschaften einer Figur selten bis nie ändern. Sie wird dazu benutzt um im Verlauf des Quiz Ähnlichkeiten zum User aufzuzeigen und wird von der UserBuild Klasse im weiteren Spielverlauf verwendet.

2.3.2 Application Code: QuestionManager

Die zweite Klasse ist die Klasse QuestionManager, wie in Abbildung 2.4 zu sehen. Ihre Aufgabe ist die Verwaltung der Fragen und Antworten, welche im Quiz verwendet werden. Dabei enthält sie die notwendige Logik um basierend auf dem jeweiligen Use Case den entsprechenden Charakter zu ermitteln. Es besteht die Möglichkeit, dass die Klasse erweitert wird, da sich die Anforderungen an das Quiz im Laufe der Zeit ändern können.

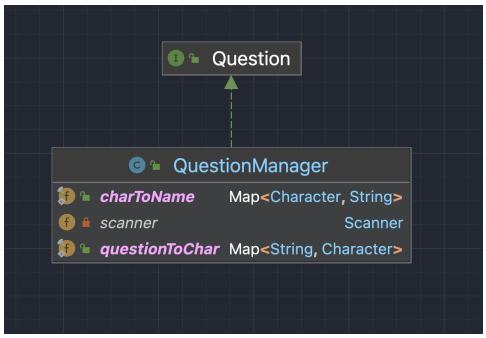

Abbildung 2.4: UML Klassendiagramm (QuestionManager)

3 SOLID

SOLID ist ein Akronym für fünf fundamentale Prinzipien der objektorientierten Programmierung:

1. Single Responsibility Principle (SRP)
2. Open Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

Sie sollen dazu beitragen, dass Software-Systeme leichter zu verstehen, zu entwickeln, zu testen und zu warten sind. [3]

3.1 Analyse Single-Responsibility-Principle (SRP)

Das Prinzip besagt, dass eine Klasse nur eine einzige Verantwortung und Aufgabe haben sollte. Durch die Aufteilung von Aufgaben an mehrere Klassen wird die Klasse lesbarer, wartbarer und hat nur einen Grund geändert zu werden. [3]

3.1.1 Positiv-Beispiel

Ein Beispiel einer Klasse, welche dieses Prinzip umsetzt, ist die Klasse UserBuild aus der Adapter Schicht, wie in Abbildung 3.1 zu sehen. Ihre primäre Aufgabe ist es, basierend

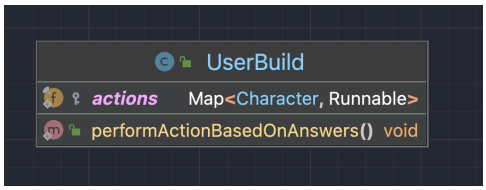

Abbildung 3.1: Positiv Beispiel SRP der Klasse (UserBuild)

auf der Summe der Character aus den Fragen, die passende Simpsons Figur zu erstellen.

3.1.2 Negativ-Beispiel

Die Klasse GameTerminal aus der Adapter Schicht hat, wie in Abbildung 3.2 zu sehen, mehrere Aufgaben. Sie ist neben der Erstellung eines Banners und Textes zur Begrüßung des Spieler zusätzlich für die Ausgabe der Fragen und Resultate zuständig. Diese Aufgaben

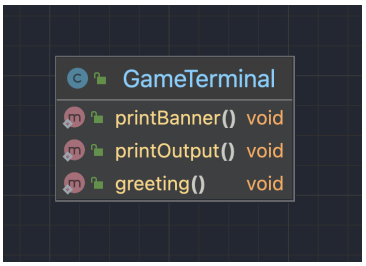

Abbildung 3.2: Negativ Beispiel SRP der Klasse (GameTerminal)

sollten in zwei verschiedene Klassen aufgeteilt werden, um die Lesbarkeit und Wartbarkeit zu verbessern.

3.2 Analyse Open-Closed-Principle (OCP)

Ein Prinzip, welches besagt, dass Klassen offen für Erweiterungen, aber geschlossen für Änderungen sein sollten. Das bedeutet, dass Änderungen an einer Klasse vermieden werden, sobald sich die Bedingungen ändern. [3]

3.2.1 Positiv-Beispiel

Im Quiz finden sich 10 Figuren aus dem Simpsons Universum, welche alle aus der Klasse SimpsonsCharacter, wie in Abbildung 3.3 zu sehen, hervorgehen.

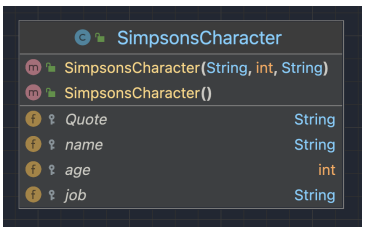

Abbildung 3.3: Positiv Beispiel OCP der Klasse (SimpsonsCharacter)

Egal wie viele Figuren durch beispielsweise eine Erweiterung noch hinzugefügt werden, die Klasse SimpsonsCharacter muss nicht verändert werden, da sie nur die Grundfunktionen der Figuren enthält.

3.2.2 Negativ-Beispiel

Sollte durch eine Erweiterung eine neuer Figur implementiert werden, wäre die Klasse UserBuild aus Abbildung 3.1 ein Verstoss gegen das OCP, da sie bei jeder neuen Figur angepasst werden müsste. Dies ließe sich durch die Auslagerung der Figuren in ein Interface lösen, welches dann als Erweiterung in die Klasse implementiert werden kann.

3.3 Analyse Interface-Segreggation- (ISP)

Das Prinzip besagt, dass eine Klasse nicht von Methoden abhängig sein sollte, die sie nicht benötigt. Durch die Aufteilung von Schnittstellen in kleinere, spezifischere Schnittstellen können unnötige Abhängigkeiten vermieden werden.

3.3.1 Positiv-Beispiel

Ein Beispiel, welche das Prinzip erfüllt, ist das Interface CharacterAction, wie in Abbildung 3.4 zu sehen. Es enthält nur die Methoden, welche für die Figuren benötigt werden. Information über beispielsweise die Features eines Arbeitsplatzes der Figur sind in ein separates Interface ausgelagert worden.

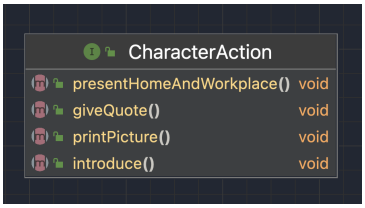

Abbildung 3.4: Positiv Beispiel ISP der Klasse (CharacterAction)

3.3.2 Negativ-Beispiel

Um das genannte Beispiel ins Negative zu kehren, wäre lediglich ein Erweitern des Interfaces um die Methoden aus dem Interface WorkplaceFeatures, wie in Abbildung 3.5 zu sehen, möglich. Dies würde als Konsequenz bedeuten, dass die Klasse SimpsonsCharacter zusätzlich die Methoden für die Arbeitsplätze verarbeiten müsste und somit unnötige Abhängigkeiten entstehen würden.

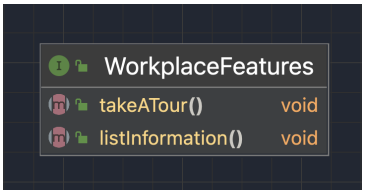

Abbildung 3.5: Negativ Beispiel ISP der Klasse (WorkplaceFeatures)

4 Weitere Prinzipien

General Responsibility Assignment Software Patterns (GRASP) sind eine Sammlung von Prinzipien, welche in der Softwareentwicklung verwendet werden. In diesem Kapitel werden die Prinzipien Geringe Kopplung und Hohe Kohäsion vorgestellt. Außerdem wird das Prinzip Don't Repeat Yourself (DRY) vorgestellt.

4.1 Analyse GRASP: Geringe Kopplung

Low Coupling bezeichnet eine geringe Kopplung zwischen Klassen. Eine geringe Kopplung zwischen Klassen bedeutet, dass eine Klasse nur auf wenige andere Klassen angewiesen ist. Eine Klasse, die auf viele andere Klassen angewiesen ist, ist schwieriger zu warten und zu erweitern. Außerdem ist es schwieriger, die Funktionalität einer Klasse zu testen, wenn diese auf viele andere Klassen angewiesen ist. Zusätzlich lässt sie sich einfacher tauschen oder ersetzen.

4.1.1 Positiv-Beispiel

Ein Beispiel, bei dem das Prinzip der Geringen Kopplung umgesetzt wurde ist die Klasse FlandersHome als Beispiel für die verschiedenen Klassen aus dem Package Homes, welche die Wohnungen der Simpsons Figuren repräsentiert. Wie in Abbildung 4.1 zu sehen ist, ist die Klasse FlandersHome nur von der Klasse Home abhängig. Zusätzlich implementiert sie notwendige Methoden des HomeFeature Interfaces. Gibt es beispielsweise eine Änderung am Heim von Rektor Skinner, betrifft es diese Klasse nicht.

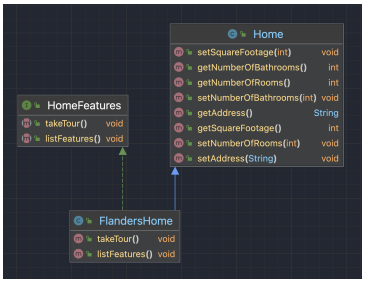

Abbildung 4.1: UML Klassendiagramm (FlandersHome) im Zusammenspiel mit Interface

4.1.2 Negativ-Beispiel

Führt man das zuvor genannte Beispiel eine Ebene nach oben, wird allerdings deutlich, dass die Klasse QuestionManager von jeder Klasse, welche eine der Simpsons Figuren repräsentiert, abhängig ist und somit eine hohe Kopplung aufweist. Abbildung 4.2 zeigt die Klasse QuestionManager im Zusammenspiel mit den Klassen aus dem Package der Charaktere.

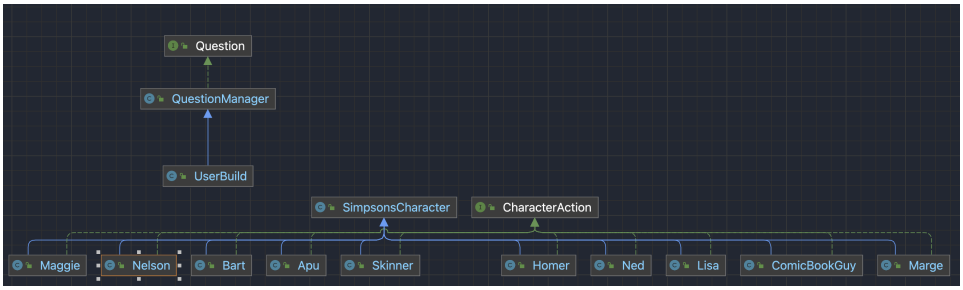

Abbildung 4.2: UML Klassendiagramm (QuestionManager) in Abhängigkeit der Charaktere

4.2 Analyse GRASP: Hohe Kohäsion

Hohe Kohäsion ist wichtig, da Objekte so organisiert werden sollten, dass die Methoden und Attribute zusammengehören und eine klar definierte Aufgabe erfüllen. Zusätzlich erhöht es wesentlich die Überschaubarkeit und des Codes. Abbildung 4.3 zeigt die Klassen der einzelnen Figuren im Zusammenhang mit ihren Arbeitsstätten und Wohnorte.

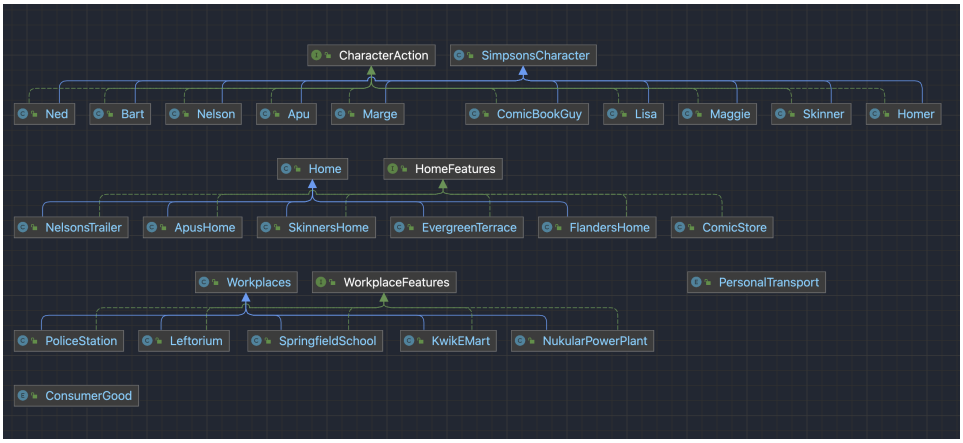

Abbildung 4.3: UML Klassendiagramm (Charaktere im Zusammenspiel mit Arbeitsstätte und Zuhause)

4.3 Don't Repeat Yourself (DRY)

DRY ist ein Akronym für Don't Repeat Yourself und ein weiteres Prinzip der Softwareentwicklung. Es bedeutet, dass jede Funktionalität oder Information in einem Programm nur einmal definiert werden sollte um Redundanzen zu vermeiden. So kann beispielsweise Code ausgelagert werden um ihn wiederverwenden zu können ohne ihn zu duplizieren. Außerdem wird die Wartbarkeit des Codes erhöht, da Änderungen nur an einer Stelle vorgenommen werden müssen. So wurde beispielsweise, wie in Listing 4.1 zu sehen ist, der Code für die Ausgabe der Charakter Bilder in die SimpsonsCharacter Klasse ausgelagert und kann so direkt von den jeweiligen Klassen aufgerufen werden.

```
1  public void printPicture(String[] picture) {
2      for (String line : picture) {
3          System.out.println(line);
4          try {
5              Thread.sleep(300); // Pause between printing lines
6          } catch (InterruptedException e) {
7              e.printStackTrace();
8          }
9      }
10 }
```

Listing 4.1: DRY Prinzip bei Ausgabe der Charakter Bilder

5 Unit Tests

5.1 10 Unit Tests

5.2 ATRIP: Automatic

5.3 ATRIP: Thorough

5.4 ATRIP: Professional

5.5 Code Coverage

5.6 Fakes und Mocks

6 Domain Driven Design

6.1 Entities

6.2 Value Objects

6.3 Aggregates

7 Refactoring

7.1 Code Smells

7.2 2 Refactorings

Charakter klassen entzerrt mit einführen einer Superklasse und einem Interface

8 Entwurfsmuster

8.1 Muster 1

8.2 Muster 2

Literaturverzeichnis

[1] Kelly, S. *Die Simpsons*. Hrsg. von IMDB. https://www.imdb.com/title/tt0096697/. o.O., 2023. (Einsichtnahme: 19. 03. 2023).

[2] Briem, L. *Clean Architecture*. Hrsg. von DHBW. https://github.com/briemla/ase-tinf20/blob/main/briem/clean-architecture.pdf. o.O., 2023. (Einsichtnahme: 19. 03. 2023).

[3] Martin, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. o.O.: Prentice Hall, 2017.