

Das Simpsons-Quiz: Finde deinen Zwilling in Springfield!

Programmentwurf

von

Dominik Veith

Abgabedatum: 15. April 2023

Bearbeitungszeitraum: 04.10.2022 - 15.04.2023

Matrikelnummer, Kurs: 3352220, TINF20B2

Inhaltsverzeichnis

Abkürzungsverzeichnis	III
Abbildungsverzeichnis	IV
Quellcodeverzeichnis	V
1 Einführung	1
1.1 Übersicht über die Applikation	1
1.2 Start der Applikation	2
1.3 Testen der Applikation	2
2 Clean Architecture	3
2.1 Was ist Clean Architecture?	3
2.2 Analyse der Dependency Rule	5
2.3 Analyse der Schichten	7
3 SOLID	9
3.1 Analyse Single-Responsibility-Principle (SRP)	9
3.2 Analyse Open-Closed-Principle (OCP)	11
3.3 Analyse Interface-Segregation-Principle (ISP)	12
4 Weitere Prinzipien	13
4.1 Analyse GRASP: Geringe Kopplung	13
4.2 Analyse GRASP: Hohe Kohäsion	15
4.3 Don't Repeat Yourself (DRY)	16
5 Unit Tests	17
5.1 10 Unit Tests	17
5.2 ATRIP: Automatic	18
5.3 ATRIP: Thorough	19
5.4 ATRIP: Professional	20
5.5 Code Coverage	22
5.6 Fakes und Mocks	23
6 Domain Driven Design	26
6.1 Ubiquitous Language	26
6.2 Entities	27
6.3 Value Objects	28
6.4 Repositories	28

6.5	Aggregates	29
7	Refactoring	30
7.1	Code Smells	30
7.2	2 Refactorings	32
8	Entwurfsmuster	35
8.1	Command Pattern	35
8.2	Strategy Pattern	37
	Literaturverzeichnis	VI

Abkürzungsverzeichnis

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

Abbildungsverzeichnis

2.1	Clean Architecture [2]	3
2.2	UML Klassendiagramm (UserBuild) Positiv Beispiel Dependency Rule	5
2.3	UML Klassendiagramm (ConsoleOutput) Positiv Beispiel Dependency Rule .	6
2.4	UML Klassendiagramm (Apu)	7
2.5	UML Klassendiagramm (QuestionManager)	8
3.1	Positiv Beispiel SRP der Klasse (UserBuild)	10
3.2	Negativ Beispiel SRP der Klasse (GameTerminal)	10
3.3	Positiv Beispiel OCP der Klasse (SimpsonsCharacter)	11
3.4	Positiv Beispiel ISP der Klasse (CharacterAction)	12
3.5	Negativ Beispiel ISP der Klasse (WorkplaceFeatures)	12
4.1	UML Klassendiagramm (FlandersHome) im Zusammenspiel mit Interface . .	14
4.2	UML Klassendiagramm (UserBuild) in Abhängigkeit der Charaktere	14
4.3	UML Klassendiagramm (Charaktere im Zusammenspiel mit Arbeitsstätte und Zuhause)	15
5.1	Maven Surfire Plugin	18
5.2	Maven JaCoCo Plugin	22
6.1	UML Diagramm für Value Objects	28
6.2	Repository UML Diagramm	29
7.1	UserBuild UML Diagramm	33
7.2	SimpsonsCharacter UML Diagramm	34
8.1	UserBuild UML Diagramm	35
8.2	QuestionManager UML Diagramm	37

Quellcodeverzeichnis

2.1	Dependency Rule in der Adapter Schicht	6
4.1	DRY Prinzip bei Ausgabe der Charakter Bilder	16
5.1	Unit Test der Klasse Apu	19
5.2	Unit Test der Klasse UserBuild	20
5.3	Unit Test der Klasse Workplaces	21
5.4	Mock Klasse für den Question Manager	23
7.1	Auslagern der Methode printPicture() in die Superklasse SimpsonsCharacter	30
7.2	Lösung der large class	31
7.3	Refactoring des Switch Statements	32
8.1	Command Pattern der UserBuild Klasse	36
8.2	Strategy Pattern der QuestionManager Klasse	38

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 ermittelten 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) 18 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, abhängig der Schichten der Architektur unter „test“ gefunden und einzeln ausgeführt werden.

Alternativ und komfortabler kann im Wurzelverzeichnis „SimpsonsQuiz“ auch die Kommandozeile geöffnet werden und mit dem Befehl `mvn test` alle Tests auf einmal 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:

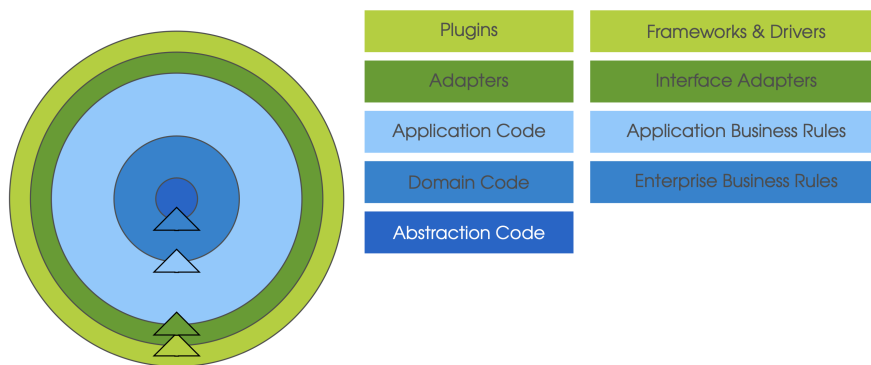


Abbildung 2.1: Clean Architecture [2]

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

Die Dependency Rule der Clean Architecture besagt, dass Abhängigkeiten immer von äußeren Schichten (näher am Framework und der Infrastruktur) zu inneren Schichten (näher am Domain-Modell und der Geschäftslogik) gerichtet sein müssen, um eine klare Trennung von Anliegen und eine unabhängige Testbarkeit der verschiedenen Schichten zu gewährleisten. 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. Dazu verwendet sie die Klasse QuestionManager 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.

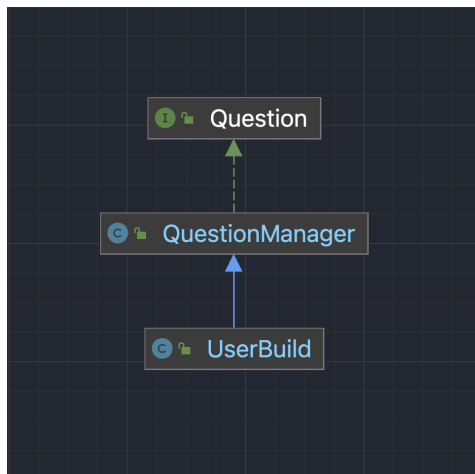


Abbildung 2.2: UML Klassendiagramm (UserBuild) Positiv Beispiel Dependency Rule

2.2.2 Negativ-Beispiel

Ein Negativ Beispiel oder Verletzung der Dependency Rule lässt sich leider nicht zeigen, da die Dependency Rule im 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

Gerne möchte ich aber ein zweites Positiv Beispiel hervorheben: In der Plugin Schicht wird durch Dependency Injection, wie in Abbildung 2.3 zu sehen, sichergestellt, dass wirklich nur die Plugin Schicht im Terminal eine Ausgabe erzeugt und nicht eine der unteren Schichten, was eine Verletzung der Dependency Rule darstellen würde.

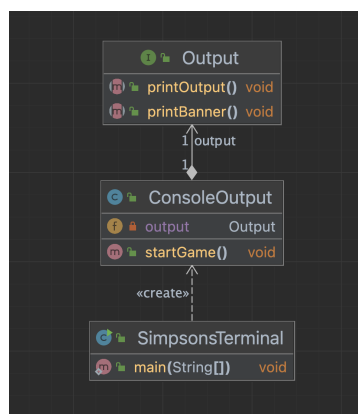


Abbildung 2.3: UML Klassendiagramm (ConsoleOutput) Positiv Beispiel Dependency Rule

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 2.4 zu sehen, eingeführt. Die Klasse Apu

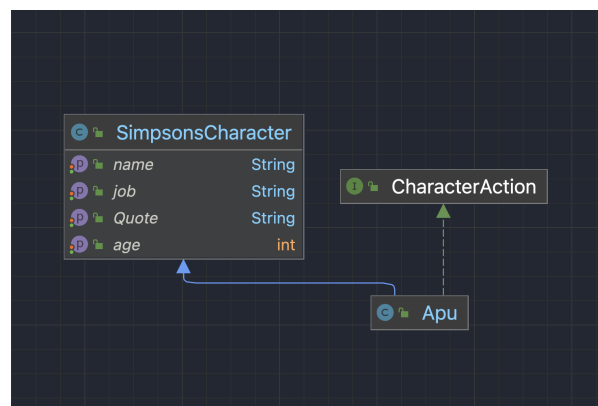


Abbildung 2.4: UML Klassendiagramm (Apu)

ist ein Value Object 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 2.5 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.

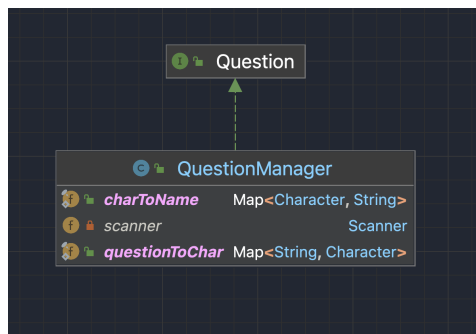


Abbildung 2.5: UML Klassendiagramm (`QuestionManager`)

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 auf

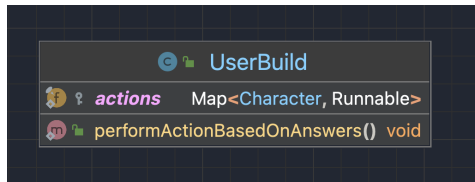


Abbildung 3.1: Positiv Beispiel SRP der Klasse (`UserBuild`)

der Summe der `Character`, welche durch die Beantwortung der Fragen generiert werden, die passende Simpsons Figur zu erstellen. Die `Character` repräsentieren dabei jeweils eine Figur aus dem Simpsons Universum.

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.

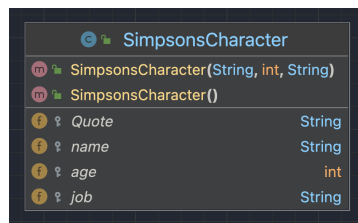


Abbildung 3.3: Positiv Beispiel OCP der Klasse (`SimpsonsCharacter`)

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-Segregation-Principle (ISP)

Das Prinzip besagt, dass eine Klasse keine Methoden implementieren muss, 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.

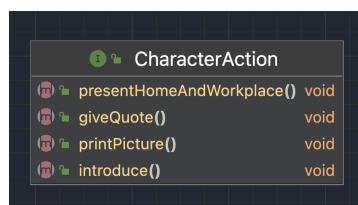


Abbildung 3.4: Positiv Beispiel ISP der Klasse (`CharacterAction`)

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.

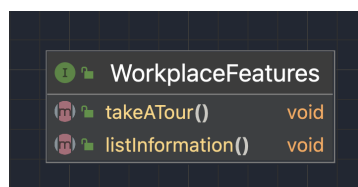


Abbildung 3.5: Negativ Beispiel ISP der Klasse (`WorkplaceFeatures`)

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. Besser ist die Nutzung von Abstraktionen (Interfaces). 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.

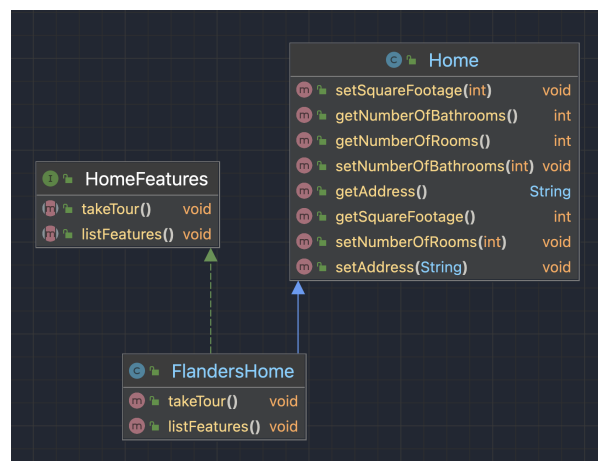


Abbildung 4.1: UML Klassendiagramm (`FlandersHome`) im Zusammenspiel mit Interface

4.1.2 Negativ-Beispiel

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

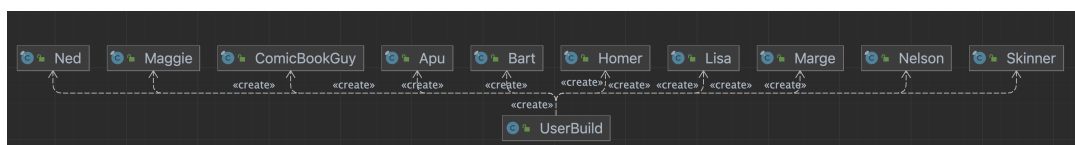


Abbildung 4.2: UML Klassendiagramm (`UserBuild`) in Abhängigkeit der Charaktere

4.2 Analyse GRASP: Hohe Kohäsion

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

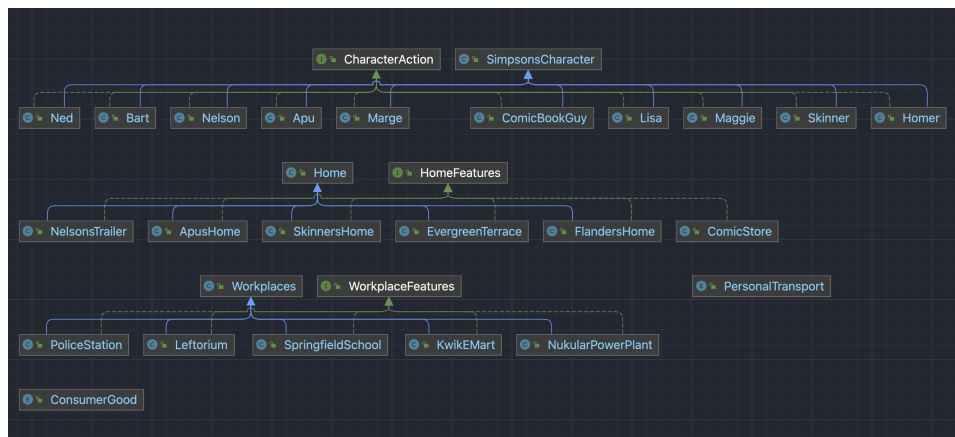


Abbildung 4.3: UML Klassendiagramm (Charaktere im Zusammenspiel mit Arbeitsstätte und Zuhause)

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

Unit Tests haben die Aufgabe einzelne Einheiten von Code auf Funktionalität zu überprüfen. Dabei kann eine Einheit eine einzelne Methode, eine Klasse oder ein Modul sein. Zweck ist die Sicherstellung dass jede Einheit der Software wie erwartet funktioniert und dass Änderungen an einer Einheit keine unerwarteten Auswirkungen auf andere Teile der Software haben. Darüber hinaus helfen Unit-Tests auch dabei, die Qualität und Zuverlässigkeit der Software zu verbessern, da sie sicherstellen, dass jeder Teil des Codes wie erwartet funktioniert und dass Fehler vermieden werden. Unit-Tests tragen somit dazu bei, dass die Software insgesamt stabiler und robuster wird.

5.1 10 Unit Tests

1. `UserBuildTest.testPerformActionBasedOnAnswers()` Testet ob für jeden Simpsons Charakter eine Aktion bereitgestellt wird.
2. `ApuTest.testIntroduce()` Testet ob die Methode `introduce()` den richtigen String zurückgibt. Die selben Tests wurden für die Simpsons Charaktere Bart, Homer, Marge, Lisa, Maggie, ComicBookGuy, Ned, Skinner und Nelson durchgeführt.
3. `SimpsonsCharacterTest.testFavoriteFood()` Testet ob abhängig vom Charakter der Simpsons das spezifische Lieblingsessen zurückgegeben wird.
4. `SimpsonsCharacterTest.testPersonalTransport()` Testet ob abhängig vom Charakter der Simpsons das spezifische Transportmittel zurückgegeben wird.
5. `ConsumerGoodsTest.testToString()` Testet ob abhängig vom jeweiligen Enum der richtige String zum Lieblingsessen zurückgegeben wird.
6. `PersonalTransportTest.testDisplayName()` Testet ob abhängig vom jeweiligen Enum der richtige String zum Transportmittel zurückgegeben wird.
7. `WorkplacesTest.testGettersAndSetters()` Testet ob die Getter und Setter der Klasse `Workplaces` funktionieren. (Negativ Beispiel)

5.2 ATRIP: Automatic

Test sollten nach Möglichkeit automatisiert ablaufen. Des Weiteren sollten auch die Ergebnisse eines Tests auf ihren positiven oder negativen Ausgang geprüft werden. In diesem Projekt wird dies mit dem Surefire Plugin von Maven realisiert, welches, wie in Abbildung 5.1 zu sehen, alle Tests auf einmal abrufen und angibt, ob ein Test fehlgeschlagen ist oder nicht. Abgerufen wird dieser Test mit dem Befehl 'mvn test' im Verzeichnis des Projekts über das Terminal.

```
[INFO] SimpsonsQuiz ..... SUCCESS [ 0.001 s]
[INFO] domain-3 ..... SUCCESS [ 1.511 s]
[INFO] application-2 ..... SUCCESS [ 0.357 s]
[INFO] adapters-1 ..... SUCCESS [ 0.408 s]
[INFO] plugin-0 ..... SUCCESS [ 0.052 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.389 s
[INFO] Finished at: 2023-04-08T14:46:54+02:00
```

Abbildung 5.1: Maven Surefire Plugin

5.3 ATRIP: Thorough

Thorough bezieht sich auf die Gründlichkeit eines Tests, insbesondere auf eine diverse Prüfung des gesamten Codes um sicherzustellen, dass alle Aspekte der Funktionalität geprüft und verschiedene Szenarien abgedeckt wurden. In diesem Projekt wurde dies durch die Erstellung von Unit Tests für jeden einzelnen Simpsons Charakter sichergestellt, wie in listing 5.1 zu sehen ist. Die Tests prüfen, ob die Methode `introduce()` den richtigen String zurückgibt. Die selben Tests wurden für die Simpsons Charaktere Bart, Homer, Marge, Lisa, Maggie, ComicBookGuy, Ned, Skinner und Nelson durchgeführt.

```
1  public class ApuTest {
2
3      @Test
4      public void testIntroduce() {
5          Apu apu = new Apu();
6          ByteArrayOutputStream outContent = new ↵
              ↳ ByteArrayOutputStream();
7          System.setOut(new PrintStream(outContent));
8          apu.giveQuote();
9          String expectedOutput = "Famous Quote: \"Never have I ↵
              ↳ seen you look so unhappy while purchasing such a ↵
              ↳ large quantity of ice cream!\"";
10         assertEquals(expectedOutput, ↵
              ↳ outContent.toString().trim());
11     }
12 }
```

Listing 5.1: Unit Test der Klasse Apu

Negativ gilt an dieser Stelle hervorzuheben, dass nicht alle Methoden der jeweiligen Klassen getestet werden.

5.4 ATRIP: Professional

Da Tests den selben Qualitätsstandards wie Produktivcode unterliegen, sollte auch hierbei darauf geachtet werden, dass die Tests mit der notwendigen Professionalität erstellt werden. Dies bedeutet, dass die Tests gut lesbar und wartbar sind. Außerdem sollten sie so geschrieben werden, dass sie leicht zu verstehen sind und dass sie sich leicht erweitern lassen. Ein positives Beispiel ist die der Test des User Builds in listing 5.2. Dieser Test prüft, ob für jeden Simpsons Charakter eine Aktion bereitgestellt wird. Dies wird durch die Methode `testPerformActionBasedOnAnswers()` realisiert. Dabei spielt es keine Rolle wie viele Charakter momentan vorhanden sind, oder ob noch welche hinzugefügt werden.

```
1  class UserBuildTest {  
2  
3      @Test  
4      void testPerformActionBasedOnAnswers() {  
5          // Ensure an action is provided for each character  
6          UserBuild.actions.forEach((character, action) -> ↵  
              ↵ assertNotNull(action));  
7      }  
8  
9  }
```

Listing 5.2: Unit Test der Klasse UserBuild

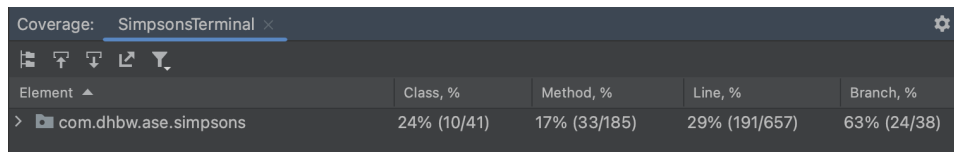
Unnötige Tests wie beispielsweise der Test von Getter und Setter Methoden sind hingegen nicht professionell. Dies ist in listing 5.3 zu sehen. Dieser Test prüft, ob die Getter und Setter der Klasse Workplaces funktionieren. Dies ist unnötig, da man keine Tests nur des Testens wegen schreiben sollte. Des Weiteren ist der Code sehr repetitiv ist und nicht DRY, weil zwei mal sehr ähnliche assertEquals-Blöcke aufgerufen werden.

```
1  public class WorkplacesTest {
2
3      @Test
4      void testGettersAndSetters() {
5          Workplaces wp = new Workplaces("Nuclear Power Plant", ↵
6              ↵ "Springfield", 100, "Mr. Burns");
7
8          assertEquals("Nuclear Power Plant", wp.getName());
9          assertEquals("Springfield", wp.getLocation());
10         assertEquals(100, wp.getNumberOfEmployees());
11         assertEquals("Mr. Burns", wp.getOwner());
12
13         wp.setName("Krusty Burger");
14         wp.setLocation("Downtown");
15         wp.setNumberOfEmployees(20);
16         wp.setOwner("Krusty the Clown");
17
18         assertEquals("Krusty Burger", wp.getName());
19         assertEquals("Downtown", wp.getLocation());
20         assertEquals(20, wp.getNumberOfEmployees());
21         assertEquals("Krusty the Clown", wp.getOwner());
22     }
23 }
```

Listing 5.3: Unit Test der Klasse Workplaces

5.5 Code Coverage

Code Coverage ist ein Maß dafür, wie viel Prozent des Quellcodes einer Software durch Tests abgedeckt werden. Eine Code Coverage von 100% würde bedeuten, dass jeder Teil des Codes durch Tests abgedeckt wurde, während eine niedrigere Code Coverage darauf hinweist, dass einige Teile des Codes nicht durch Tests überprüft wurden. In diesem Projekt wurde die Code Coverage mit dem JaCoCo Plugin von Maven ermittelt. Dieses Plugin ist in Abbildung 5.2 zu sehen. Die Code Coverage beträgt 24% der Klassen und 17% der Methoden, womit also noch Bedarf an Optimierung im Hinblick auf die Methoden besteht. Für die Klassen ist die Code Coverage jedoch ausreichend, da viele Klassen ähnlich aufgebaut sind.



Element	Class, %	Method, %	Line, %	Branch, %
> com.dhbw.ase.simpsons	24% (10/41)	17% (33/185)	29% (191/657)	63% (24/38)

Abbildung 5.2: Maven JaCoCo Plugin

5.6 Fakes und Mocks

Mocks sind im Kontext von Unit-Tests Platzhalter-Objekte, die das Verhalten von Abhängigkeiten simulieren, die für den Test nicht verfügbar sind oder unerwünschte Nebenwirkungen haben könnten. Sie helfen, Tests schneller auszuführen, indem sie die Interaktionen der zu testenden Komponente mit ihren Abhängigkeiten imitieren. Mocks können auch zur Überprüfung von Interaktionen verwendet werden, indem sie aufgezeichnete Methodenaufrufe und Parameter speichern und anschließend überprüfen, ob sie den erwarteten Werten entsprechen. In diesem Projekt wird, wie in listing 5.4 zu sehen, der Unit Test `QuestionManagerTest` mit einem Mock Input verwendet um die Eingabe durch einen Nutzer zu simulieren.

```
1 public class MockInputGenerator {
2     private final int size;
3     private final long seed;
4
5     public MockInputGenerator(int size, long seed) {
6         this.size = size;
7         this.seed = seed;
8     }
9
10    public String generateMockInput() {
11        StringBuilder mockInputBuilder = new
12            ↳ StringBuilder(size);
13        Random random = new Random(seed);
14        for (int i = 0; i < size; i++) {
15            char c = random.nextBoolean() ? 'y' : 'n';
16            mockInputBuilder.append(c).append('\n');
17        }
18        return mockInputBuilder.toString();
19    }
20 }
21
22
23
```

```
24 public class QuestionManagerTest {
25     @Test
26     public void testAskQuestions() {
27         // Create a new instance of QuestionManager
28         QuestionManager questionManager = new QuestionManager();
29         // Prepare input stream with mock user input
30         int size = questionManager.charToName.size() * 2;
31         MockInputGenerator mockInputGenerator = new ↵
32             ↵ MockInputGenerator(size, 42L);
33         String mockInput = ↵
34             ↵ mockInputGenerator.generateMockInput();
35         System.setIn(new ↵
36             ↵ ByteArrayInputStream(mockInput.getBytes()));
37
38         InputStream inputStream = new ↵
39             ↵ ByteArrayInputStream(mockInput.getBytes());
40         System.setIn(inputStream);
41
42         QuestionManager questionManager2 = new ↵
43             ↵ QuestionManager();
44         // Call the method to be tested
45         Character result = questionManager2.askQuestions();
46
47         // Assert that the result is one of the expected ↵
48             ↵ characters
49         List<Character> validChars = Arrays.asList('H', 'M', ↵
50             ↵ 'L', 'B', 'A', 'N', 'C', 'X', 'Y', 'S');
51         assertTrue(validChars.contains(result));
52     }
53 }
```

Listing 5.4: Mock Klasse für den Question Manager

Die Klasse enthält eine Test-Methode `testAskQuestions()`, die eine Instanz von `QuestionManager` erstellt und die `askQuestions()` - Methode dieser Instanz testet. Zunächst wird eine zufällige Benutzereingabe erstellt, indem eine `StringBuilder`-Instanz mit der Größe der `charToName`-Map der `QuestionManager`-Klasse initialisiert wird. Die Schleife durchläuft dann jede Zeichenposition und fügt zufällig 'y' für Ja oder 'n' für Nein hinzu, um die Benutzereingabe zu simulieren. Durch den festen Seed des Random Generators wird eine gleiche Sequenz bei erneuter Ausführung sichergestellt. Anschließend wird eine `InputStream`-Instanz erstellt, um die generierte Benutzereingabe zu setzen. Dann wird die `askQuestions()` - Methode aufgerufen, um die Antwort des Frage-Managers zu erhalten. Schließlich wird geprüft, ob das Ergebnis ein gültiges Zeichen enthält, das in der Liste der erwarteten Zeichen 'validChars' enthalten ist.

6 Domain Driven Design

Domain Driven Design ist ein Ansatz der Software Entwicklung, bei dem die Domäne und deren Logik im Mittelpunkt steht. Dabei hilft DDD der Komplexität von Software vorzubeugen, indem es den Fokus auf das Verständnis der Domäne legt. [4]

6.1 Ubiquitous Language

Ubiquitous Language ist ein zentrales Konzept im Domain-Driven Design und bezieht sich auf eine gemeinsame, konsistente Sprache, die von allen Beteiligten im Projekt verwendet wird, um die Domäne und ihre Anforderungen zu beschreiben. Diese einheitliche Sprache soll Kommunikationsprobleme zwischen Entwicklern, Fachexperten, Stakeholdern und anderen Teammitgliedern vermeiden und ein gemeinsames Verständnis der Domäne fördern. In der Praxis bedeutet dies, dass die Begriffe, die in der Geschäftsdomäne verwendet werden, konsistent in den Diskussionen, Dokumentationen und im Code selbst verwendet werden sollten. Ubiquitous Language wird im gesamten Entwicklungsprozess eingesetzt, von der Anforderungsanalyse über das Design bis hin zur Implementierung. Im Folgenden wird die Ubiquitous Language des Simpsons Quiz Projekts vorgestellt.

- **Charakter:** Ein Charakter ist eine Person der Simpsons Serie. Dabei ist es egal ob es sich um einen Hauptcharakter oder einen Nebencharakter handelt.
- **Portrait:** Ein Portrait ist ein Bild, welches einem Charakter der Serie zugeordnet ist.
- **Workplace:** Ein Workplace ist ein Arbeitsplatz, welcher einem Charakter der Serie zugeordnet ist.
- **Home:** Ein Home ist ein Wohnort, welcher einem Charakter der Serie zugeordnet ist.
- **LuxuryFood:** Ein LuxuryFood ist das Lieblingsessen, welches einem Charakter der Serie zugeordnet ist.

- Transport: Ein Transport ist das präferierte Transportmittel, welches einem Charakter der Serie zugeordnet ist.

Die Begriffe zählen zur Ubiquitous Language, da sie essentiell für das Verständnis des Simpsons Quiz sind und die Hauptmerkmale des Aufbaus der jeweiligen Figur im Rahmen des Quiz darstellen. Dabei werden sie bei der Implementierung als Gruppierung der einzelnen Attribute verwendet.

6.2 Entities

Entitäten repräsentieren Objekte, die innerhalb einer Geschäftsdomäne eine eindeutige Identität besitzen. Im Gegensatz zu Value Objects, die nur durch ihre Attribute definiert sind, haben Entitäten eine Identität, die unabhängig von ihren Eigenschaften ist. Das bedeutet, dass selbst wenn sich der Zustand einer Entität im Laufe der Zeit ändert, ihre Identität konstant bleibt. Beispiele für Entitäten können Kunden, Produkte, Bestellungen oder Mitarbeiter sein. In all diesen Fällen ist die Identität des Objekts entscheidend, um es von anderen Objekten desselben Typs unterscheiden zu können. Im Simpsons Quiz stellen Workplace, Home, und Transport Entitäten dar.

6.3 Value Objects

Value Objects sind ein wichtiger Bestandteil von Domain-Driven Design (DDD) und repräsentieren Objekte, die keine eigene Identität haben und ausschließlich durch ihre Eigenschaften oder Attribute definiert sind. Im Gegensatz zu Entitäten, die eine eindeutige Identität besitzen und deren Zustand sich im Laufe der Zeit ändern kann, sind Value Objects unveränderlich und können bei Bedarf einfach ersetzt werden. Im Simpsons Quiz stellen die einzelne Charaktere des Spiels Value Objects dar. Sie sind anhand ihrer Attribute wie Arbeitsplätze, Vorlieben oder Zitaten einzigartig und können nur durch einen anderen Charakter ersetzt, aber nicht verändert werden.

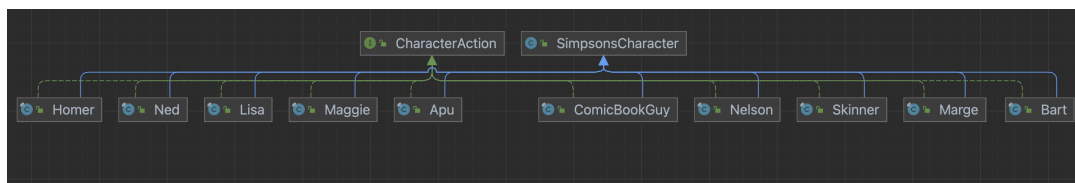


Abbildung 6.1: UML Diagramm für Value Objects

6.4 Repositories

Repositories vermitteln zwischen der Domäne und dem Modell und stellen Methoden bereit um Aggregates aus dem Persistenzspeicher zu lesen oder zu speichern. Im Simpsons Quiz wird, wie in Abbildung 6.2 zu sehen, ein Repository verwendet, das generierte Textdokumente auf der Festplatte speichert. Dazu wird eine Document-Entität und ein DocumentRepository-Interface verwendet, das von einer konkreten Implementierung `FileDocumentRepository` realisiert wird. Das `FileDocumentRepository` speichert jedes Document-Objekt (die erstellte Charakter Textdatei) als eine separate Textdatei auf der Festplatte. Die Dateien werden im Projekt Verzeichnis abgelegt.

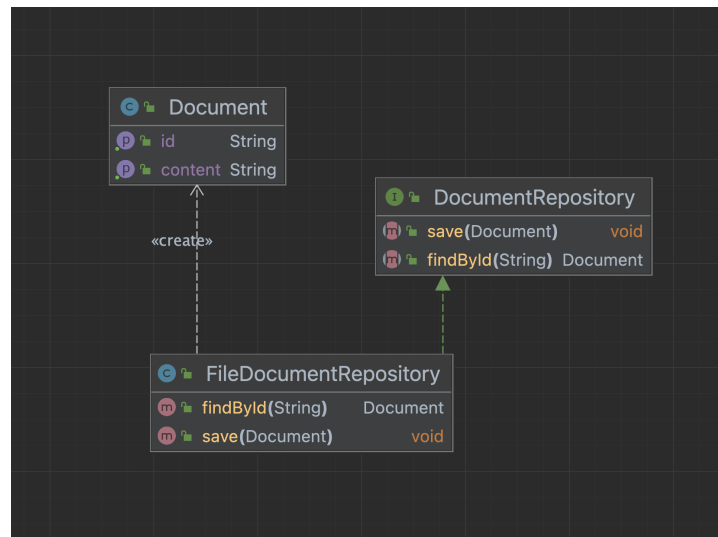


Abbildung 6.2: Repository UML Diagramm

6.5 Aggregates

Ein Aggregate dient dazu, eine Gruppe von assoziierten Objekten, die zusammenarbeiten, zu kapseln und die Konsistenz der Geschäftsregeln innerhalb dieser Gruppe sicherzustellen. Aggregate bestehen aus einer oder mehreren Entitäten und/oder Value Objects und haben eine klar definierte Grenze, innerhalb derer sie als eine Einheit behandelt werden. Jedes Aggregate hat eine Root-Entität, die als Einstiegspunkt für die Interaktion mit dem Aggregate dient. Andere Objekte innerhalb des Aggregats sind von außerhalb des Aggregats nur über die Root-Entität zugänglich. Die Root-Entität ist verantwortlich für die Einhaltung von Geschäftsregeln und die Konsistenz der assoziierten Objekte. Die Identität eines Aggregats wird durch die Identität seiner Root-Entität bestimmt. Diese Identität bleibt unverändert, selbst wenn sich die Zustände der anderen Objekte innerhalb des Aggregats ändern. Im Simpsons Quiz sorgt das Aggregate SimpsonsCharacter als Root für die verschiedenen Figuren des Quiz. Die spezifischen Figuren fungieren als Value Objects innerhalb des Aggregats.

7 Refactoring

7.1 Code Smells

Code Smells sind Anzeichen oder Muster in der Software, die auf mögliche Probleme oder schlechte Praktiken im Code hindeuten. Sie sind nicht unbedingt Fehler oder Bugs, können jedoch die Wartbarkeit, Lesbarkeit und Qualität des Codes beeinträchtigen. In diesem Kapitel werden zwei Code Smells im Projekt vorgestellt und behoben.

7.1.1 Code Smell: Duplicated Code

Das Vorhandensein von ähnlichem oder identischem Code an mehreren Stellen im Projekt kann auf schlechte Modularisierung oder mangelnde Wiederverwendbarkeit hindeuten. Jeder Simpsons Charakter hat neben seinen Attributen und Methoden, welche sein Zuhause oder Transportmittel beschreiben, auch eine Methode welche nachdem der Charakter ausgewählt wurde, ein entsprechendes Bild über das Terminal ausgibt. Dieser Code ist in allen Charakterklassen vorhanden und wird in jeder Klasse aufgerufen. Dieser Code ist also mehrfach vorhanden und kann durch Auslagern in eine Superklasse gelöst werden. Wie in listing 7.1 zu sehen wurde die Methode in die Superklasse SimpsonsCharacter ausgelagert und in den jeweiligen Subklassen nur noch aufgerufen.

```
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 7.1: Auslagern der Methode printPicture() in die Superklasse SimpsonsCharacter

7.1.2 Code Smell: Large Class

Eine Large Class ist eine Klasse, die zu viele Methoden oder eine große Anzahl von Codezeilen hat. Dies kann ein Anzeichen dafür sein, dass die Klasse zu viele Verantwortlichkeiten trägt. Eine große Klasse kann es schwierig machen, den Code zu verstehen, zu warten und zu erweitern. Um dieses Problem zu lösen, sollte die Klasse in kleinere, fokussierte Klassen aufgeteilt werden, die jeweils eine einzige Verantwortung haben. Im Simpsons Quiz gibt es für jeden Charakter bereits eine separate Klasse. Allerdings war bis vor dem Refactoring die Bilder der Charakter ebenfalls mit in der Klasse gespeichert. Das Auslagern in extra Klassen sorgt für eine deutliche Reduktion der Code Zeilen der Charakter Klassen. Im Folgenden Commit sind die Änderungen zu sehen: https://github.com/Crixos86/AdvancedSE_DHBW/commit/2f4d1b

```
1 @Override
2 public void printPicture() {
3     printPicture(HomerPicture.Picture);
4 }
```

Listing 7.2: Lösung der large class

7.2 2 Refactorings

Refactorings sind strukturierte Änderungen am Code, die darauf abzielen, die interne Struktur und Qualität des Codes zu verbessern, ohne das äußere Verhalten oder die Funktionalität der Software zu ändern. Das Hauptziel von Refactoring ist es, den Code lesbarer, wartbarer und besser verständlich zu machen, was die Produktivität der Entwickler erhöht und die Wahrscheinlichkeit von Fehlern oder Bugs reduziert.

7.2.1 Switch Statements

Im Simpsons Quiz wird je nach Antworten des Spielers ein anderer Charakter der Serie ausgewählt. Dies wurde initial durch ein Switch Statement realisiert was aber nicht sehr gut wartbar ist und auch in Teilen gegen das Open Closed Principle verstößt. Der Code wurde, wie in listing 7.3 zu sehen, entsprechend angepasst.

```
1 public class UserBuild extends QuestionManager{
2
3     protected static final Map<Character, Runnable> actions = ↵
4         ↵ Map.of(
5             'H', () -> new Homer().introduce(),
6             'M', () -> new Marge().introduce(),
7             'L', () -> new Lisa().introduce(),
8             'B', () -> new Bart().introduce(),
9             'A', () -> new Apu().introduce(),
10            'N', () -> new Ned().introduce(),
11            'C', () -> new ComicBookGuy().introduce(),
12            'X', () -> new Nelson().introduce(),
13            'Y', () -> new Maggie().introduce(),
14            'S', () -> new Skinner().introduce()
15        );
16 }
```

Listing 7.3: Refactoring des Switch Statements

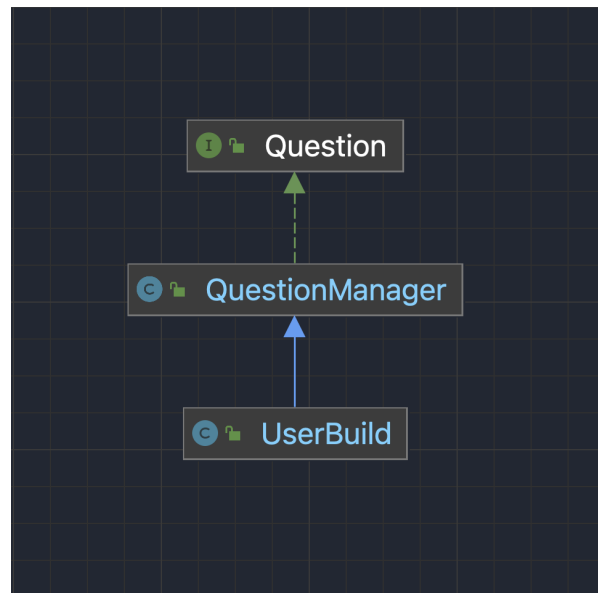


Abbildung 7.1: UserBuild UML Diagramm

Die Verwendung der Runnable Interfaces und einer Map mit Lambdas bieten gegenüber dem Switch Statement einige Vorteile:

- Lesbarkeit: Die Verwendung einer Map und Lambdas bietet eine klarere und leichter verständliche Struktur, da sie die Logik zur Ausführung der Aktionen auf einer höheren Abstraktionsebene kapselt.
- Erweiterungen: Es ist einfacher, der Map neue Aktionen hinzuzufügen oder vorhandene Aktionen zu ändern, ohne den gesamten Code umzuschreiben. Im Gegensatz dazu erfordert ein Switch-Statement oft eine umfangreichere Änderung des Codes, um neue Fälle hinzuzufügen oder bestehende zu ändern.
- Open/Closed Principle: Durch die Verwendung einer Map und Lambdas folgt der Code eher dem Open/Closed Principle, das besagt, dass Software-Einheiten (Klassen, Module, Funktionen usw.) für Erweiterungen offen, aber für Modifikationen geschlossen sein sollten. Hier kann die Implementierung leicht erweitert werden, ohne dass der bestehende Code geändert werden muss.
- Effizienz: Da die Map auf Hash-basierten Schlüsseln arbeitet, ist der Zugriff auf die zugehörige Aktion in der Regel schneller als ein Switch-Statement, insbesondere wenn es eine große Anzahl von Fällen gibt.

7.2.2 Polymorphismus

Polymorphismus ist ein grundlegendes Konzept der objektorientierten Programmierung, das es ermöglicht, verschiedene Objekte durch eine gemeinsame Schnittstelle oder Basisklasse zu behandeln. Polymorphismus kann als Refactoring-Technik verwendet werden, um den Code sauberer, modularer und leichter wartbar zu gestalten. Mit Einführung der `SimpsonsCharacter` Klasse wurden gemeinsame Eigenschaften und Methoden der Charaktere ausgelagert. Im folgenden Commit ist das Refactoring zu sehen: https://github.com/Crixos86/AdvancedSE_DHBW/commit/7a1e03

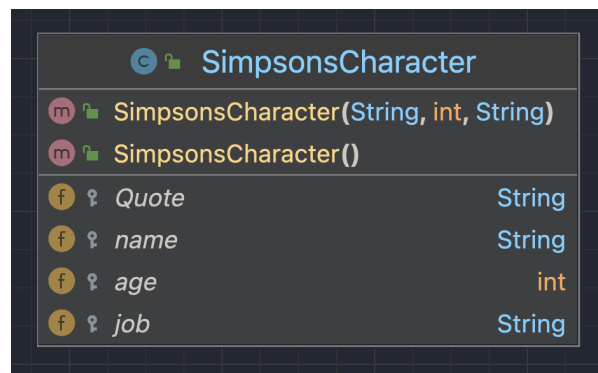


Abbildung 7.2: SimpsonsCharacter UML Diagramm

Folgende Vorteile wurden durch die Einführung der Superklasse erreicht:

- **Bessere Abstraktion:** Polymorphismus ermöglicht es, eine gemeinsame Schnittstelle oder Basisklasse für unterschiedliche Verhaltensweisen oder Implementierungen zu definieren. Dies führt zu einer klareren Abstraktion und kapselt die unterschiedlichen Implementierungen besser.
- **Erhöhte Wiederverwendbarkeit:** Durch die Verwendung von Polymorphismus kann Code, der auf der gemeinsamen Schnittstelle oder Basisklasse basiert, wiederverwendet werden. Dies reduziert die Code-Redundanz und verbessert die Modularität.
- **Einfachere Erweiterbarkeit:** Mit Polymorphismus können neue Verhaltensweisen oder Implementierungen hinzugefügt werden, indem neue Klassen erstellt werden, die die gemeinsame Schnittstelle oder Basisklasse erweitern. Dies verbessert die Erweiterbarkeit des Codes und erleichtert das Hinzufügen neuer Funktionen.

8 Entwurfsmuster

8.1 Command Pattern

Das Command-Entwurfsmuster ist ein Verhaltensmuster, welches darauf abzielt, die Anfrage zur Ausführung einer Aktion von der eigentlichen Ausführung der Aktion zu entkoppeln. Dies wird erreicht, indem die Aktionen in Objekten gekapselt werden, die als Befehle (Commands) bezeichnet werden. Die Klasse `UserBuild`, wie in listing 8.1 verwendet das Command-Entwurfsmuster in Kombination mit einer Map, um Aktionen basierend auf den Antworten des Benutzers auszuführen. Das Command-Pattern kapselt eine Anfrage als Objekt, in diesem Fall als `Runnable` Objekt, sodass es leicht übergeben, gespeichert und ausgeführt werden kann. Im 'UserBuild'-Code werden verschiedene Aktionen in einer Map als `Runnable`-Objekte gespeichert. Jede Aktion wird durch einen Charakter gekennzeichnet. Die Methode `performActionBasedOnAnswers()` verwendet diese Map, um die entsprechende Aktion basierend auf der Eingabe (dem meistgewählten Charakter) auszuführen. Das Command-Pattern ermöglicht es, die Aktionen und deren Ausführung zu entkoppeln, sodass sie leicht erweitert oder geändert werden können, ohne die Hauptlogik der Anwendung zu beeinflussen. In diesem Fall ermöglicht das Command-Pattern eine saubere und leicht erweiterbare Implementierung für die Verwaltung von Aktionen, die auf den Benutzerantworten basieren.

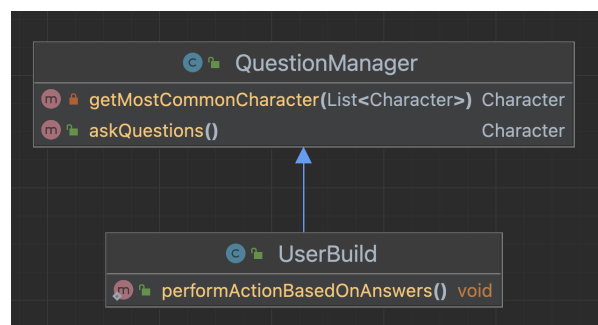


Abbildung 8.1: UserBuild UML Diagramm

```
1 public class UserBuild extends QuestionManager{
2
3     protected static final Map<Character, Runnable> actions = ↵
        ↵ Map.of(
4         'H', () -> new Homer().introduce(),
5         'M', () -> new Marge().introduce(),
6         'L', () -> new Lisa().introduce(),
7         'B', () -> new Bart().introduce(),
8         'A', () -> new Apu().introduce(),
9         'N', () -> new Ned().introduce(),
10        'C', () -> new ComicBookGuy().introduce(),
11        'X', () -> new Nelson().introduce(),
12        'Y', () -> new Maggie().introduce(),
13        'S', () -> new Skinner().introduce()
14    );
15    public static void performActionBasedOnAnswers() {
16        Character mostCommonChar = new ↵
            ↵ QuestionManager().askQuestions();
17        String ChosenCharacter = charToName.get(mostCommonChar);
18        System.out.printf("The Simpsons character you are most ↵
            ↵ like is %s %n", ChosenCharacter );
19        if (mostCommonChar == null) {
20            System.out.println("No character found");
21            return;
22        }
23        Runnable action = actions.get(mostCommonChar);
24        if (action != null) {
25            action.run();
26        } else {
27            System.out.println("No action found for " + ↵
                ↵ mostCommonChar);        }
28    }
29 }
```

Listing 8.1: Command Pattern der UserBuild Klasse

8.2 Strategy Pattern

Das Strategy-Entwurfsmuster ist ein Verhaltensmuster, das darauf abzielt, eine Familie von austauschbaren Algorithmen zu definieren, die unabhängig von den Clients verwendet werden können, die sie verwenden. Das Strategy-Muster ermöglicht es, den Algorithmus, der von einem Objekt verwendet wird, dynamisch zur Laufzeit zu ändern, ohne das Objekt selbst zu ändern. Die Klasse `QuestionManager` verwendet, wie in listing 8.2 zu sehen, einige Aspekte des Strategy-Musters, indem sie die Fragen und zugehörigen Antworten in einer Map speichert. Dies ermöglicht es, die Fragen und Antworten leicht zu ändern oder zu erweitern, ohne die Hauptlogik der Klasse ändern zu müssen. In diesem Fall ist es jedoch nicht vollständig angewendet, da die Algorithmen selbst (die Fragen) direkt in der Klasse definiert sind und keine austauschbaren Strategieobjekte verwendet werden.

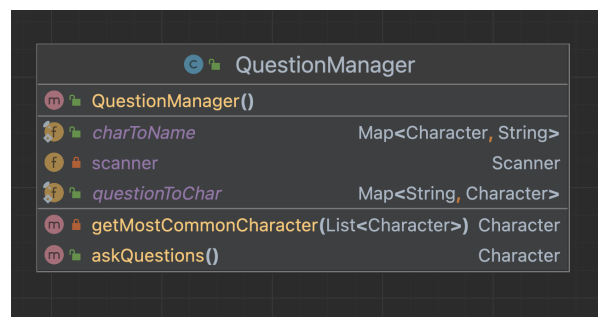


Abbildung 8.2: QuestionManager UML Diagramm

```
1 public class QuestionManager implements Question {
2
3     private Scanner scanner;
4
5
6     public final static Map<String, Character> questionToChar ↵
        ↳ = new HashMap<>();
7     public final static Map<Character, String> charToName = ↵
        ↳ new HashMap<>();
8
9     public QuestionManager() {
10         scanner = new Scanner(System.in);
11         questionToChar.put("Donuts are your favourite food?", ↵
            ↳ 'H');
12         questionToChar.put("Is beer your life?", 'H');
13         questionToChar.put("Do you like art?", 'M');
14         questionToChar.put("Is the ocean very inspiring for ↵
            ↳ you?", 'M');
15         questionToChar.put("Do you like going to school?", 'L');
16         questionToChar.put("Is it your dream to attend a Ivy ↵
            ↳ League University?", 'L');
17         questionToChar.put("Do you often suffer from brain ↵
            ↳ freeze due to too much Squishy?", 'B');
18         questionToChar.put("Do you like to ride your ↵
            ↳ Skateboard?", 'B');
19         questionToChar.put("You love to work 24 hours in your ↵
            ↳ own supermarket?", 'A');
20         questionToChar.put("Sorting groceries is a dream of ↵
            ↳ yours?", 'A');
21         questionToChar.put("Helping your lovely neighbours is ↵
            ↳ part of your daily routine?", 'N');
22         questionToChar.put("Attending church is very important ↵
            ↳ to you?", 'N');
23         questionToChar.put("Comics are your life?", 'C');
24         questionToChar.put("Eating Fast Food is the best?", ↵
            ↳ 'C');
```

```
25         questionToChar.put("Do you solve your conflicts with ↵
           ↵ violence?", 'X');
26         questionToChar.put("Do you like to laugh at people?", ↵
           ↵ 'X');
27         questionToChar.put("Do you like to sleep?", 'Y');
28         questionToChar.put("Playing with toys is awesome?", ↵
           ↵ 'Y');
29         questionToChar.put("Do you like to teach annoying ↵
           ↵ pupils a lesson?", 'S');
30         questionToChar.put("Do you still live with your ↵
           ↵ parents?", 'S');
31
32         charToName.put('H', "Homer Simpson");
33         charToName.put('M', "Marge Simpson");
34         charToName.put('L', "Lisa Simpson");
35         charToName.put('B', "Bart Simpson");
36         charToName.put('A', "Apu Nahasapeemapetilon");
37         charToName.put('N', "Ned Flanders");
38         charToName.put('C', "Comic Book Guy");
39         charToName.put('X', "Nelson");
40         charToName.put('Y', "Maggie");
41         charToName.put('S', "Seymour Skinner");
42
43
44     }
45
46     @Override
47     public Character askQuestions() {
48         List<Character> answers = new ArrayList<>();
49
50         for (String question : questionToChar.keySet()) {
51             System.out.println(question);
52
53             String answer;
54             while (true) {
55                 answer = scanner.nextLine().toLowerCase();
```

```
56         if (answer.equals("y") || answer.equals("n")) {
57             break;
58         }
59         System.out.println("Please answer the question ↵
        ↳ with [y] yes OR [n] no.");
60     }
61
62     if (answer.equals("y")) {
63         answers.add(questionToChar.get(question));
64     }
65 }
66
67 if (answers.isEmpty()) {
68     System.out.println("You didn't answer any ↵
        ↳ question.");
69     return null;
70 } else {
71     return getMostCommonCharacter(answers);
72 }
73
74 }
75
76 private Character getMostCommonCharacter(List<Character> ↵
    ↳ characters) {
77     Map<Character, Integer> charCounts = new HashMap<>();
78     for (Character nametag : characters) {
79         charCounts.put(nametag, ↵
            ↳ charCounts.getDefault(nametag, 0) + 1);
80     }
81
82     List<Character> mostCommonChars = new ArrayList<>();
83     int mostCommonCharCount = 0;
84     for (Character nametag : charCounts.keySet()) {
85         int count = charCounts.get(nametag);
86         if (count > mostCommonCharCount) {
87             mostCommonChars.clear();
```

```
88         mostCommonChars.add(nametag);
89         mostCommonCharCount = count;
90     } else if (count == mostCommonCharCount) {
91         mostCommonChars.add(nametag);
92     }
93 }
94
95 if (mostCommonChars.size() == 1) {
96     return mostCommonChars.get(0);
97 } else {
98     return mostCommonChars.get(new ↵
          ↵ Random().nextInt(mostCommonChars.size()));
99 }
100 }
101
102
103
104 }
```

Listing 8.2: Strategy Pattern der QuestionManager Klasse

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/asetinf20/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.
- [4] Evans, E. *Domain Driven Design*. Hrsg. von Addison-Wesley Professional. 2003.