Programmentwurf

Advanced Software Engineering (T3INF3001)

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Katharina Braun (7032223)

Michaela Haag (3098671)

Abgabedatum:	30. April 2023
Bearbeitungszeitraum:	03.10.2022 - 30.04.2023
Kurs:	TINF20B1
Gutachter der Dualen Hochschule:	Daniel Lindner

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Wir haben uns entschieden, für das Advanced Software-Engineering Projekt, eine von uns entwickelte Rezeptverwaltung zu verwenden. Diese Anwendung soll später auch als mobile App umgesetzt werden, weshalb dafür eine Codeverbesserung sehr sinnvoll ist.

1.1 Funktionalität

Die Anwendung wurde entwickelt, um Personen die Planung Ihrer Mahlzeiten zu vereinfachen. In der Anwendung ist es möglich, alle Rezepte zentral an einer Stelle zu verwalten, anstelle von mehreren verteilen Rezeptbüchern. Beim Anlegen neuer Rezepte ist es neben der Angabe des Namen, der Zutaten, der Beschreibung, des Schwierigkeitsgrads und eines Bildes auch möglich, das Rezept vordefinierten Kategorien (z. B. Grundzutaten) zuzuordnen. Auf der Startseite der App gibt es dann die Möglichkeit, sich eine Liste aller verfügbaren Rezepte anzuzeigen oder nur die Rezepte einzelner Kategorien. Wird in der Rezeptliste ein Rezept ausgewählt, so bekommt der Anwender eine Detailansicht des Rezepts. Des Weiteren hat die Anwendung noch die Funktion, dass der Anwender sich ein zufälliges Rezept generieren kann. Möchte der Anwender das Rezept kochen, dann kann er sich die Details des Rezepts anzeigen lassen. Wenn der Anwender mit der Auswahl unzufrieden ist, so gibt es die Möglichkeit, ein neues zufälliges Rezept generieren zu lassen.

1.2 Kundennutzen

Der Kundennutzen liegt darin, dass Anwender alle Ihre Lieblingsrezepte an einem Ort sammeln und nach Ihren Wünschen organisieren können. Dadurch verhindern wir langes recherchieren nach einem online Rezept oder das Suchen nach Omas Rezepten auf einem Zettel. Nutzer können einfach eigene Rezepte anlegen und ein Foto eines Rezepts hochladen, z. B. aus einem Buch oder einer Zeitschrift. Außerdem möchten wir die

Vielfalt beim Essen vergrößern, indem der Nutzer ein Zufall Rezept aus einem großen Pool von Rezepten vorgeschlagen bekommt oder indem er direkt nach Kategorien filtert. Mit unserer Anwendung erleichtern wir somit den Anwendern die Entscheidung, welches Gericht jeden Tag gekocht werden soll.

1.3 Technologie

Die Anwendung ist aktuell in Java entwickelt und als Architektur wurde das MVC-Konzept verwendet (Trennung von Daten, Benutzeroberfläche und Logik). Für die Datenhaltung haben wir uns entschieden, alle Daten in CSV-Dateien zu speichern.

2 Domain Driven Design

In diesem Kapitel wird die Ubiquitous Language unseres Projektes analysiert. Außerdem werden die Entities & Value Objects basierend auf der entwickelten Software modelliert und abschließend die verwendeten Repositories und Aggregates analysiert und erklärt.

2.1 Ubiquitous Language

2.1.1 Sprache

Ubiquitous Language bedeutet, eine Sprache und die Begriffe so zu wählen, dass die Domänenexperten und die Entwickler minimalen Übersetzungsaufwand haben. Aufgrund einer deutschen Domäne und Anwendung haben wir deutsche Domänen Begriffe verwendet. Die Dateinamen und Ordner der Javaklassen wurden so gewählt, dass sie den Domänenexperten und den Nicht-Entwicklern mit kurzen und aussagekräftigen Worten Aufschluss über die Funktionalität geben. textbf!!! Folgendes wurde von Merlin inspiriert (umgeschrieben) übernommen und sollte hier erwähnt werden, wenn wir es gemacht haben: Handelt es sich um eine Entität oder ein Value Objekt, werden diese entsprechend benannt. Da sich Ubiquitous Language auf das gesamte Projekt bezieht, wurden auch die Tests in der Domain Sprache geschrieben und sollen für alle Leser verständlich sein. Der Test XXX, repräsentiert sehr gut die verwendete Ubiquitous Language. !!!

2.1.2 Begriffsdefinition

Die Anwendungsdomäne befasst sich mit der Verwaltung von Rezepten. Daher ist der erste zentrale Begriff der Domäne das **Rezept**. Ein Rezept setzt sich aus einer ID, einem Titel, einer Beschreibung, einer Menge von **Zutaten**, einer Menge von Rezept **Kategorien**, einer **Schwierigkeit** und optional einem **Bild** zusammen. Eine **Zutat** enthält Informationen über den Namen der Zutat, in welcher **Menge** die Zutat in das Rezept gehört, sowie die dazugehörende **Einheit**. Die **Einheit** setzt sich aus einem Namen

und einer Beschreibung zusammen. Ein weiterer wichtiger Begriff in der Domäne sind die Rezept **Kategorien**. Sie enthalten einen Namen, eine Beschreibung und zusätzlich noch eine Kurzform des Namens für eine schönere Visualisierung in der Benutzeroberfläche. Die **Schwierigkeit** eines Rezeptes kann in der Domäne **einfach**, **mittel** oder **schwer** sein. Das **Bild** enthält neben dem zugehörigen Rezept noch den **Pfad**, an dem ein Bild gespeichert ist.

2.1.3 Entities & Value Objects

In diesem Unterkapitel werden wir die Entities und Value Objects in unserer Rezepte-Anwendung genauer analysieren. Entities sind Objekte, die eine Identität haben. Sie haben eine eindeutige ID und können von anderen Objekten referenziert werden. In unserer Anwendung haben wir verschiedene Entities. In Abbildung XX sind die Entities in Blau dargestellt.

Rezept ist eine Entity, da es eine eindeutige Identität hat und veränderliche Eigenschaften hat. Ein Nutzer kann in der Anwendung die Rezepte nach dem Erstellen immer wieder bearbeiten. Die Zutaten eines Rezepts setzen sich aus einer Menge, dem Namen der Zutat, einer ID und der ID des zugehörigen Rezepts zusammen. Außer den beiden IDs können auch die Eigenschaften von Zutaten verändert werden. Daher ist auch die Zutat ein Entity. Das Bild, das zu einem Rezept gehört, wird in unserer Anwendung auch als Entity betrachtet, da es eindeutig identifizierbar ist, durch eine eindeutige ID mit dem Rezept verknüpft werden kann und über die Eigenschaft Pfad verfügt. Der Pfad des Bildes kann verändert werden. Kategorie ist die vierte Entity der Domäne. Rezepte werden verschiedenen Kategorien zugeordnet und die Nutzer können sich eine Liste mit allen Rezepten für eine Kategorie anschauen, dafür ist es notwendig, dass Kategorie eine eindeutige ID hat und referenzierbar ist.

Value Objects hingegen haben keine Identität. Sie werden lediglich durch ihre Eigenschaften definiert und können nicht von anderen Objekten referenziert werden. In unserer Anwendung gibt es das Value Object: Menge. Eine Zutaten-Menge hat keine eindeutige Identität, sondern beschreibt einfach den Umfang oder die Menge einer bestimmten Zutat, die in einem Rezept verwendet wird. Eine Zutaten-Menge hat keine eigenen Eigenschaften oder Verhaltensweisen und kann nicht auf andere Objekte referenzieren. Eine Menge setzt sich aus der Menge und einer Einheit zusammen.

In der Domäne gibt es zusätzlich die Enumerationen Schwierigkeit und Einheit. Enumerationen können eine Sammlung von Value Objects sein, wenn in den Enum-Instanzen kein veränderbarer Zustand hinterlegt ist. Die Enumeration Schwierigkeit kann nur die Werte: Einfach, Normal und Schwer annehmen und diese Instanzen haben keine Eigenschaften und sind nicht veränderbar. Darum handelt es sich bei dem Enum Schwierigkeit um ein Value Object. Bei der Enumeration Einheit ist das ähnlich. Einheit hat keine eindeutige Identität und beschreibt lediglich die Art der Messung, die verwendet wird, um die Menge einer Zutat zu beschreiben, z.B. Gramm oder Teelöffel. Es hat allerdings die Eigenschaften Name und Beschreibung. Daher wird die Einheit, die bei der Angabe der Menge einer Zutat verwendet wird, als Entity betrachtet.

Wir haben uns dazu entschieden, die ID bei allen Entities mit Surrogatschlüsseln umzusetzen. Dafür haben wir allen Entity Elementen in unserer Domäne eine UUID gegeben. Universally Unique Identifier, kurz UUID, ist ein Standard für Identifikationsnummern. Vorteile von UUIDs sind, dass sie jederzeit generierbar sind und dass sie anwendungsübergreifend eindeutig sind. Außerdem ist hier die Verteilung der IDs einfacher, da eine UUID generiert werden kann, ohne dass sie mit den bereits vorhandenen UUIDs verglichen werden muss, da es sehr unwahrscheinlich ist, dass versehentlich doppelte UUIDs generiert werden. Die Nachteile, wie: nicht sprechend, keine Bedeutung in der Domäne und eventuelle Engpässe bei Generierung in Hochlastsystemen sind in unserer Anwendung nicht von Bedeutung, da der Schlüssel einfach aus der CSV ausgelesen werden kann.

2.1.4 Aggregates

Aggregate gruppieren die Entities und Value Objects zu gemeinsam verwalteten Einheiten. Die Verwendung von Aggregaten ermöglicht das Entkoppeln der Objektbeziehungen, das Bilden natürlicher Transaktionsgrenzen und die kontinuierliche Übereinstimmung mit den Domänenregeln. Alle Klassen im Paket Rezept bilden Eigenschaften eines Rezeptes ab, daher werden diese zu einem Aggregate zusammengefasst. Dazu gehören Rezept, Kategorie, Bild und Schwierigkeit. Die Root Entität ist dabei das Rezept selbst. Im zweiten Aggregat befindet sich nur die Kategorie, da die Kategorie eigen verwaltet wird.

2.1.5 Repositories

Für den Zugriff auf den persistenten Speicher werden zwei Repositories gemäß dem Grundsatz „Ein Repository pro Aggregate“ definiert: Das RezeptRepository erlaubt den Zugriff auf das Rezept, also die Root Entity des entsprechenden Aggregates. Analog hierzu ermöglicht das KategorieRepository Zugriff auf die Kategorie als Root Entity des zugehörigen Aggregates. Im RezeptReposetory werden die Daten in 3 verschiedenen CSV Dateien gespeichert. Dafür gibt es verschiedene Gründe. Die Zutaten mit den jeweiligen Eigenschaften werden separat voneinander gespeichert, da hier eine 1:n Beziehung herrscht und wir hier Redundanz der Daten vermeiden wollen. In der CSV für die Zutaten wird daher die ID des jeweiligen Rezeptes mitgespeichert. Bei den Bildern eines Rezeptes ist das ähnlich. Auch hier herrscht eine 1:n Beziehung. Zusätzlich haben wir uns für eine getrennte Speicherung zur besseren Organisation entschieden.

3 Clean Architecture

Dieses Kapitel beschreibt die Architektur der entwickelten Software. Es wurde nach den in der Vorlesung erläuterten Clean-Architecture-Prinzipien gebaut. Clean Architecture ist ein Softwarearchitektur-Muster, welches zum Ziel hat, die Abhängigkeiten innerhalb einer Anwendung zu minimieren und die Wiederverwendbarkeit und Testbarkeit zu maximieren. Es besteht aus einer inneren Schicht von unabhängigen Entitäten, die von einer äußeren Schicht von Abhängigkeiten umgeben sind. Dies ermöglicht es Entwicklern, Anwendungen zu erstellen, die leicht zu testen, zu verstehen und zu warten sind und die flexibel sind, um schnell auf Änderungen reagieren zu können.

Im Folgenden werden die verwendeten Schichten genauer erläutert. Die Schicht „Abstraction Code“ wurde in unserer Anwendung nicht verwendet, da für die in der Domäne behandelten Themengebiete kein Domänen übergreifendes Wissen notwendig war, welches Teil dieser Schicht hätte sein müssen.

- Außerdem wird auf die Application Code Schicht verzichtet (eigentlich weil wir nicht mehr genug Code für die Schicht haben, das sollten wir aber anders begründen) evtl. Begründen, dass wir Schicht 2 und 1 zusammengelegt haben

3.1 Schicht 3: Domain Code

Die Domain Code Schicht umfasst die unabhängigen und wiederverwendbaren Geschäftslogik-Komponenten der Anwendung. Sie enthalten keine Abhängigkeiten von anderen Schichten und sind in der Regel unabhängig von der Benutzeroberfläche oder der Datenpersistenz.

- Enthält alle Objekte aus DDD (Entitys, VOs) - Unterteilung in beide Aggregate - Pro Aggregat ein Repository erstellt (Enthält spezifische Methoden des Entity Managers, als Zwischenschicht) - Alle Entitäten implementieren Interface IPersistierbar, welches sicherstellt, dass diese eine UUID zum Speichern im EntityManager haben - IEntityManager für Dependecy Injection (genauer erklärt in Kapitel Dependency Injection) - beide interfaces wurden neu erstellt

3.2 Schicht 1: Adapters

- Entity Manager für interne Datenhaltung in diese Schicht geschoben, implementiert IEntityManager Interface - Ordner Datenpersistenz erstellt, beinhaltet alles für CSV Speicherung (CSV Funktionaliät wird möglichst nach außen in Schichten gelegt, dass schnell und einfach austauschbar ist) - Erstellung von Data Transfer Objekte, basierend auf den zugrundelegenden persistierbaren Entitäten, welche explizite Funktionalitäten zur Speicherung in CSV Dateien beinhalten. Deshalb implementieren alle Objekte ICSVPersistierbar - CSVReader und Writer dort rein verschoben, welche Objekte aus CSV Dateien lesen oder dort rein schreiben können - DataReader erstellt, dieser beinhaltet Funktionalität zum Laden der Daten aus den CSV Dateien in den EntityManager und zurück speichern (Funktionen aus Controller verschoben) - DataReader beinhaltet Instanz des Entity Managers (hier noch ergänzen, was Herr Lindner gesagt hat, dass wir es mit im Text erwähnen sollen) - Funktionalität wurde aus den GUI Klassen in sogenannte Funktionen Klassen ausgelagert und in Adapters Schicht gezogen als Zwischenstück zwischen GUI und Daten (Kathi vlt. noch Ergänzungen von dir)

3.3 Schicht 0: Plugins

- aufgeteilt in plugins und plugins-main - beinhaltet eigentlich nur GUI und die Main Methode der APP - in RezeptApp Main wird DataReader Instanz erstellt und überall nach unten in Schichten weiter gegeben (durch UI), sodass man von überall auf EntityManager zugreifen kann - da man von inneren schichten nicht auf datareader Instanz in plugins mehr zugreifen kann wird er in Schichten nach unten durchgereicht Dependecy Rule

3.4 Dependency Inversion

Ausgangssituation: Hatten Repositories in Domain-Schicht, welche EntityManager von Adapters Schicht aufrufen (Dependecy Rule verletzt (Aufrufe von innen nach außen))

Deshalb Dependency Inversion und Injection: IEntityManager mit Methoden aus EntityManger erstellt in Domain Schicht und EntityManager implementiert das Interface Repositories bekommen einen Entitymanager beim Aufruf übergeben, aber in Repository

code hat dieser nur den Typ IEntityManager Somit kann man auch den EnitiyManager in Domain Schicht äufrufen"

Hierbei war wichtig, dass Entitys in Domain Schicht IPersistierbar implementieren, da dies in EntityManager vorausgesetzt wird.

4 Entwurfsmuster

Entwurfsmuster sind bewährte Methoden, um wiederkehrende Probleme in der Softwareentwicklung zu lösen und stellen somit eine Art Blaupause dar, die zur Verbesserung der Struktur, Klarheit und Flexibilität von Software beitragen. Auch in diesem Projekt wurden Entwurfsmuster eingesetzt. Eines dieser Entwurfsmuster soll im nachfolgenden Abschnitt genauer erläutert werden.

4.1 Observer Pattern

In der Rezept-Anwendung wurde vermehrt mit dem Entwurfsmuster Observer-Pattern (Beobachter) gearbeitet. Das Observer-Entwurfsmuster ermöglicht es, Änderungen an einem Objekt den anderen Objekten mitzuteilen, die sich dafür registriert haben. Der Observer gehört zu der Kategorie der Verhaltensmuster. Das Entwurfsmuster besteht aus zwei Hauptkomponenten: dem Observable (Subjekt) und den Observern (Beobachtern). Das Subjekt hat einen Zustand, der sich im Laufe der Zeit ändern kann. Die Beobachter registrieren sich beim Subjekt und werden automatisch benachrichtigt, wenn sich der Zustand des Subjekts ändert. Das Observer-Entwurfsmuster ermöglicht es, Objekte lose zu koppeln, da die Observer keine Kenntnis über den Zustand der Objekte haben müssen, auf die sie reagieren. Dies führt zu einer flexibleren und wartungsfreundlicheren Softwarearchitektur. Im Folgenden wird das Entwurfsmuster anhand eines Beispiels genauer erläutert.

In der entwickelten Anwendungen wurde mit Java Swing gearbeitet, welche einen Observer (Beobachter) zur Verfügung stellt. Java Swing wurde genutzt, um die Benutzeroberfläche in der Java Anwendung zu erstellen. Diese Benutzeroberfläche ist interaktiv. Um auf die Benutzerinteraktionen zu reagieren, stellt Java Swing das Interface Action Listener bereit. Der Action Listener ist in unserer Anwendung der konkrete Observer (Beobachter) und der JButton das konkrete Subjekt.

Abbildung 4.1 ist das UML-Diagramm unseres Observers.

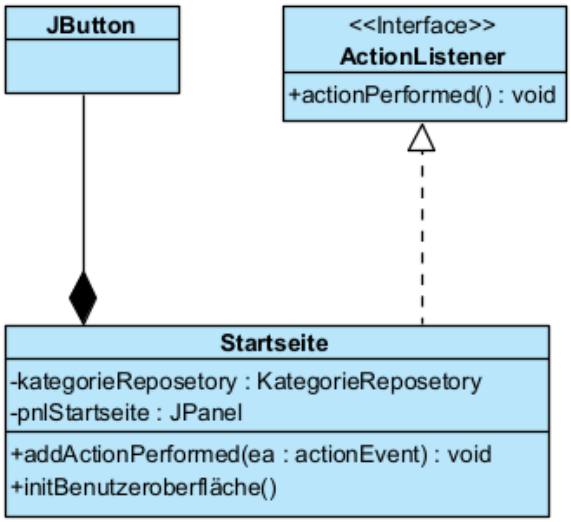

Abbildung 4.1: UML-Diagramm Observer-Pattern Entwurfmuster