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. 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.

2.2 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 2.1 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 Abbildung 2.1 sind die Value Objects in rosa dargestellt. 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 Member-Felder 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 Member-Felder 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.

2.3 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. Abbildung 2.1 zeigt die Aufteilung der Aggregate in die beiden Ordner.

2.4 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 (siehe Abbildung 2.1). Analog hierzu ermöglicht das KategorieRepository Zugriff auf die Kategorie als Root Entity des zugehörigen Aggregates (siehe Abbildung 2.1). 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.

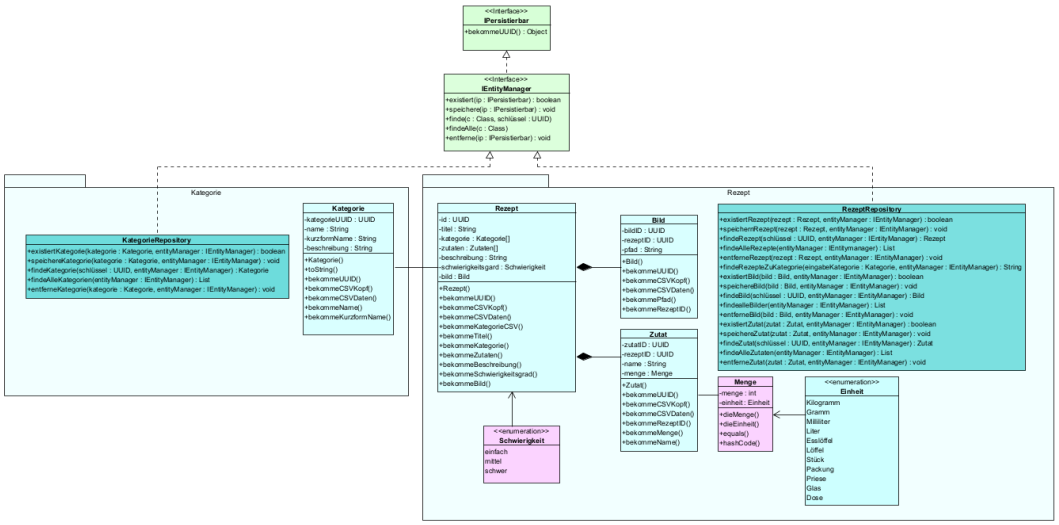

Abbildung 2.1: UML-Diagramm Repositories

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 haben wir die Adapters Code Schicht und die Application Code Schicht zusammengelegt. Der Code dieser Schichten befinden sich in der Schicht 1: Adapters Code.

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. In der Domain Code Schicht sind die beiden Aggregate und die enthaltenen Entitäten und Value Objects der Softwaredomäne. Für beide Aggregate wurde jeweils ein Repository erstellt, die sich auch in der Domain Code Schicht befinden. Die Repositories enthalten spezifische Methoden des EntityManagers zur Speicherung. Außerdem befindet sich im Domain Code das Interface IEntityManager für die Dependecy Injection. Eine genauere Erläuterung befindet sich in Abschnitt 3.4 Dependency Inversion. Die Interfaces IEntityManager und IPersistierbar wurden beide neu erstellt. Alle Entitäten der Domäne, welche gespeichert werden sollen, implementieren das Interface IPersistierbar. Dieses

wird dazu genutzt, um sicherzustellen, dass die Entitäten eine UUID zum Speichern im EntityManager haben. Der Code in dieser Schicht verwendet nur Java-Standards, sodass er als Kern und langlebigste Softwareschicht keine Abhängigkeiten aufweist. Abbildung 3.1 zeigt die Domain Code Schicht als UML Diagramm.

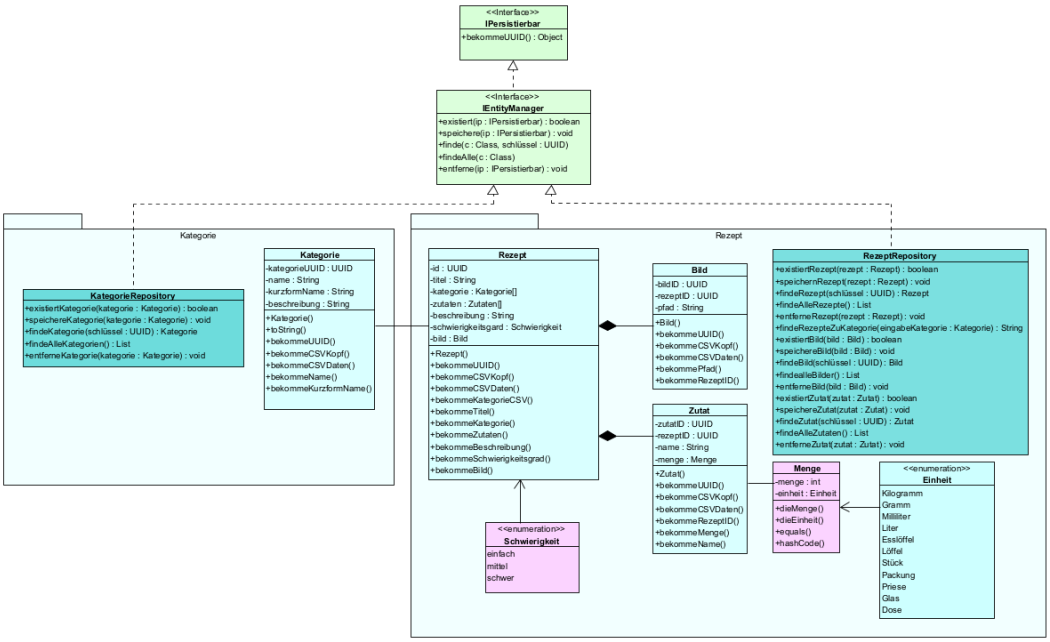

Abbildung 3.1: UML-Diagramm Domain Code Schicht

3.2 Schicht 1: Adapters

Die Adapters-Schicht in Clean Architecture stellt die Schnittstelle zwischen der Anwendung und externen Systemen dar. Sie ermöglicht die Kommunikation der Anwendung mit der Umgebung und besteht aus Adaptern, die dafür verantwortlich sind, die Daten von der Anwendung in eine Form zu konvertieren, die von den externen Systemen verarbeitet werden kann und umgekehrt. In unserer Anwendung befinden sich in der Adapters-Schicht die Funktionalität für die Speicherung und für die GUI.

Der EntityManager, welcher für die Datenhaltung zuständig ist, implementiert nun das Interface IEntityManager aus der Domain Code-Schicht (siehe Abschnitt 3.4 Dependency Inversion). Darüber hinaus befindet sich in dieser Schicht die Konvertierung der Daten, die für die Speicherung notwendig ist. In dieser Anwendung wurde sich für eine Speicherung

der Daten in CSV-Dateien entschieden. Die CSV Funktionalitäten werden möglichst in die äußeren Schichten implementiert, um die Art der Speicherung austauschbar zu gestalten. Für die Konvertierung wurde der Ordner Datenpersistenz erstellt und das Interface ICSVPersistierbar erstellt. ICSVPersistierbar wird dazu genutzt, um sicherzustellen, dass die zugrunde liegenden persistierbaren Entitäten eine UUID zum Speichern sowie einen Kopf und Daten für die CSV Dateien bereitstellen. Außerdem befinden sich im Ordner zu jeder persistierbaren Entität des Domain Codes eine eigene Klasse, die dieses Interface implementiert und somit explizite Funktionalität zur Speicherung in CSV Dateien beinhaltet.

Zusätzlich wurde der DataReader implementiert, der die Funktionalitäten zum Laden der Daten aus den CSV Dateien in den EntityManager und das Speichern in CSV Dateien beinhaltet. Hierfür erstellt der DataReader die Instanz des EntityManagers und ruf dann die Funktionalität des CSVReaders oder des CSVWriters auf, welche die Daten laden oder speichern.

Um die Code-Qualität und Wartbarkeit zu verbessern, wurden die Funktionen, die zuvor in den GUI-Klassen enthalten waren, in separate Funktionen-Klassen ausgelagert. Diese Funktionen-Klassen sind nun Teil der Adapters-Schicht, die als Vermittler zwischen der Benutzeroberfläche und den Daten fungiert.

3.3 Schicht 0: Plugins

In der Clean Architecture sind Plugins optional einsetzbare Komponenten, die von der Kernanwendung getrennt sind und über Schnittstellen integriert werden können. Sie ermöglichen es, die Anwendung um zusätzliche Funktionen oder Integrationsmöglichkeiten zu erweitern, ohne die Kernanwendung selbst zu verändern. In unserer Anwendung haben wir die Plugins in Plugins und Plugins-Main aufgeteilt. In den Plugins sind alle GUI-Klassen implementiert. Die Plugins-Main Schicht beinhaltet die Main Methode zum Starten der App. In der Main-Methode wird die DataReader Instanz instanziiert und die Daten aus den CSV Dateien geladen, bevor die Startseite gestartet wird. Da die darunter liegenden Schichten aufgrund der Dependency Rule dann nicht mehr auf den DataReader und somit auch nicht auf den EntityManager zugreifen könnten, wird die Instanz des DataReaders an die darunter liegenden Schichten beim Aufruf weitergegeben.

3.4 Dependency Inversion

In der ursprünglichen Architektur haben Repositories innerhalb der Domain-Schicht den EntityManager aus der Adapters-Schicht aufgerufen. Dies verstieß gegen das Prinzip der Dependency Rule, das besagt, dass Abhängigkeiten immer von innen nach außen gerichtet sein sollten.

Um dieses Problem zu beheben, werden die Prinzipien der Dependency Inversion und Injection eingesetzt, um sicherzustellen, dass der EntityManager in der Domain-Schicht aufgerufen werden kann, ohne die Abhängigkeiten zwischen den Schichten zu verletzen. Zu diesem Zweck wird das Interface IEntityManager, in der Domain-Schicht erstellt, welches die Methoden des EntityManager beinhaltet. Der EntityManager selbst wird in die Adapters-Schicht verschoben und implementiert dieses Interface. Um sicherzustellen, dass die Repositories auf den EntityManager zugreifen können, wird beim Aufruf des Repositories ein EntityManager Objekt übergeben. Da der EntityManager nun als IEntityManager-Typ im Repository-Code deklariert ist, kann er auch in der Domain-Schicht aufgerufen werden, ohne die Dependency Rule zu verletzen. Außerdem war hierbei wichtig, dass Entities in Domain Schicht IPersistierbar implementieren, da dies in EntityManager vorausgesetzt wird.

Durch die Verwendung von Dependency Inversion und Injection wird die Abhängigkeit zwischen der Domain-Schicht und der Adapters-Schicht umgekehrt, sodass die Domain-Schicht unabhängig von der Adapters-Schicht bleibt. Dies verbessert die Flexibilität und Wartbarkeit der Anwendung und ermöglicht es, Komponenten einfach auszutauschen oder zu erweitern, ohne dass dies Auswirkungen auf andere Schichten hat.

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. Der ActionListener in Java hat einige Limitationen. Eine Limitation ist, dass der ActionListener nur für eine Aktion auf

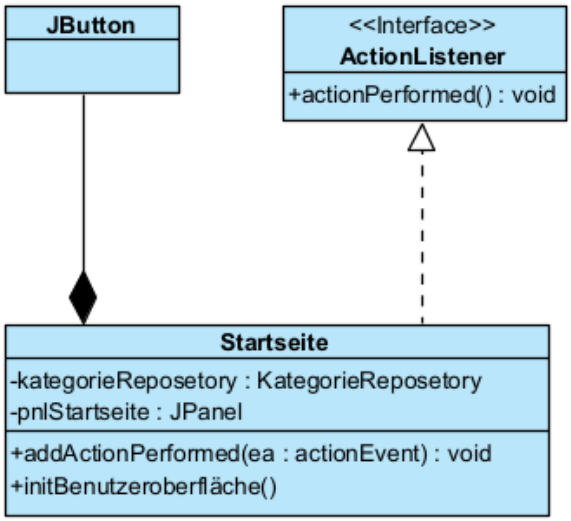

Abbildung 4.1: UML-Diagramm Observer-Pattern Entwurfmuster

einem bestimmten Komponenten-Objekt registriert werden kann, wie z.B. das Klicken auf eine Schaltfläche. Außerdem kann der ActionListener keine Werte zurückgeben, was bedeutet, dass er keine Möglichkeit hat, Feedback oder Informationen an den Aufrufer zurückzugeben, da die Methode actionPerformed vom Typ void ist. Daher kann der ActionListener auch keine Exceptions werfen. Eine weitere Limitation ist, dass der ActionListener asynchron ausgeführt wird. Er blockiert, bis die Aktion abgeschlossen ist. Des Weiteren ist ActionListener nur für Mausereignisse geeignet und kann nicht für Tastatureingaben verwendet werden. Dazu kommt, dass der ActionListener nur auf ein Mausereignis reagieren kann, auf das Klicken auf eine Schaltfläche oder ein Menüelement. Andere Mausereignisse wie das Bewegen der Maus oder das Drücken der rechten Maustaste werden vom ActionListener nicht akzeptiert. Unser Code arbeitet innerhalb der Limitationen, daher ist der ActionListener für unsere Anwendung gut geeignet.

5 Programming Principles

Programming principles (Programmierprinzipien) sind Grundsätze, die beim Entwurf, der Entwicklung und dem Testen von Software angewendet werden können. Sie sind eine Verallgemeinerung wiederkehrender Erkenntnisse in der Softwareentwicklung und liefern Entwicklern Richtlinien für einen bestimmten Programmierstil. Im Allgemeinen zielen sie darauf ab, die Qualität und Robustheit von Software zu verbessern. Im folgenden Abschnitt werden drei Programming principles vorgestellt und deren Anwendung in unserem Projekt analysiert.

5.1 SOLID

Die SOLID-Prinzipien sind ein Konzept für objektorientierte Programmierung, das fünf Grundsätze für die Entwicklung von hochwertigem und wartbarem Code definiert. Indem man diese Prinzipien befolgt, kann man sicherstellen, dass der Code besser strukturiert und leichter zu erweitern ist, und dass er weniger fehleranfällig ist und besser gewartet werden kann. Jeder Buchstabe in SOLID steht für einen dieser Grundsätze:

5.1.1 Single Responsibility Principle (SRP)

Das Single Responsibility Principle (SRP) besagt, dass ein einzelnes Objekt oder eine Klasse nur für eine einzige Aufgabe oder Verantwortlichkeit zuständig sein sollte. In unserer Anwendung ist beispielsweise eine Klasse, die dieses Prinzip strikt einhält: FunktionenZufallsGenerator. Diese Klasse hat lediglich die Aufgabe, aus einer Liste von Rezepten ein „zufälliges“ Rezept auszuwählen. Ein Negativbeispiel für eine Klasse, die das Single Responsibility Principle nicht einhält, ist unser EntityManager. Unser EntityManager wird verwendet, um eine Verbindung zwischen den Objekten der Anwendung und der zugrunde liegenden Datenbank herzustellen. Mit dem EntityManager wollten wir die Verwaltung von Objekten und deren Zuständen vereinfachen. Das bedeutet allerdings, dass der EntityManager mehrere Aufgaben (Verantwortungen) hat und somit das Prinzip

verletzt. Andere Positivbeispiele für die Einhaltung des Single Responsibility Principle sind beispielsweise die Klassen FunktionenRezeptBearbeiten, FunktionenNeuesRezept, ButtonRenderer und FunktionenListenÜbersicht. Die Klasse FunktionenListenÜbersicht ist beispielsweise nur dafür verantwortlich alle Rezepte zu einer angeklickten Kategorie zurückzugeben.

5.1.2 Open-Closed-Prinzip

Das Open-Closed-Prinzip besagt, dass Software-Entitäten offen für Erweiterungen sein sollte, aber geschlossen bezüglich Veränderungen. Das bedeutet, dass bestehender Code nicht mehr geändert werden sollte, sondern neue Funktionalitäten hinzugefügt werden. Dadurch soll sichergestellt werden, dass die bestehende Funktionalität nicht beeinträchtigt wird und dass die Erweiterung der Software einfacher und sicherer ist. Die Klasse Entity-Manager, die auch das Single Responsibility Principle verletzt, ist auch ein Negativbeispiel für das Open-Closed-Principle. Da diese Klasse jeweils alle Use-Cases implementiert, die die Entitytäten betreffen, muss bei geänderten oder neuen Anforderungen diese bestehende Klassen verändert werden. Ein Beispiele für die Einhaltung des Open-Closed-Principle sind die GUI Funktionen in der Adapterschicht. Hier existiert für jeden einzelnen Use-Case eine separate Klasse, sodass bei neuen Anforderungen lediglich eine neue Klasse implementiert werden müsste und damit der bestehende Code nur erweitert. Die einzige Änderung an bestehendem Code würde in den GUI Klassen stattfinden, da dort neue Events bzw. die zugehörigen Callbacks registriert werden.

5.1.3 Liskov substitution principle (LSP)

Das Liskov substitution principle besagt, dass es möglich sein muss, Instanzen von Objekten durch ihre Subtypen zu ersetzen, ohne die Korrektheit des Programms zu beeinträchtigen. Kurz gesagt, soll die Ableitungsklasse alle Eigenschaften und Methoden der Basisklasse beibehalten und diese nicht modifizieren oder verletzen. Problematisch kann es sein, wenn Subtypen eine Spezialisierung des Supertypen sind. Ein Beispiel einer Verletzung der Regel: Quadrat wird als Subtyp eines Rechteckes implementiert. Subtypen sind eine Spezialisierung des Supertyps. In dem vorliegenden Projekt ist keine Vererbungsbeziehung vorhanden, die auf das Liskov Substitution Principle untersucht

werden kann, da in keinem Fall von einer eigenen konkreten Klasse geerbt wird, sondern nur von abstrakten Klassen. Es könnte beispielsweise die Klassen Startseite auf das Liskov Substitution Principle untersucht werden, jedoch ist es hier trivial, dass das Prinzip eingehalten wird, da die Subtypen keine Funktionalität des Supertypen überschreiben.

5.1.4 Interface-Segregation-Principle (ISP)

Das Interface-Segregation-Prinzip besagt, dass mehrere spezifische Interfaces besser sind als ein Allround-Interface. Die Schnittstellen sollten also schlank und spezifisch sein, damit Clients nur das implementieren müssen, was sie tatsächlich brauchen, anstatt gezwungen zu sein, unnötige Methoden zu implementieren, um Abhängigkeiten und Kopplung zwischen Modulen oder Klassen zu reduzieren und die Wartbarkeit und Flexibilität des Codes zu verbessern.

Gute Beispiele für die Einhaltung des Interface-Segregation-Principles finden sich in den Klassen ICSVPersistierbar und IPersistierbar wieder. Beide Interfaces haben jeweils nur eine oder wenige Methode, die nur einen einzigen Nutzen definieren. Das Interface ICSVPersistierbar enthält Methoden die zur Speicherung der Objekte notwendig sind und in den Objekten implementiert werden müssen. IPersistierbar wird verwendet, um die UUID zu lesen, die im EntityManager benötigt wird. Für dieses Vorhaben definiert dieses Interface die Methode bekommeUUID.

Ein Negativbeispiel für die Einhaltung des Interface-Segregation-Principles findet sich in dem Interfaces IEntityManager. Diese kombiniert jeweils den schreibenden Zugriff, das Löschen und das Suchen von Objekten in einem einzigen Interface. Dieses Interface könnte aufgeteilt werden auf jeweils ein Interface für den schreibenden Zugriff, eins für das Löschen von Objekten und ein weiteres Interface für das Suchen nach Objekten. Dadurch bräuchte ein Klient, welcher nur schreibende Zugriffe benötigt, keine Abhängigkeiten auf ein Interface, welches auch löschenden Zugriff erlaubt. Durch eine solche Aufteilung könnte eine Zugriffsverwaltung wesentlich leichter implementiert werden.

5.1.5 Dependency inversion principle (DIP)

Das Dependency-Inversion-Principle besagt, dass Klassen höherer Ebenen nicht von Klassen niederer Ebenen abhängig sein sollen, sondern beide von Interfaces. Durch die Verwendung von abstrakten Schnittstellen oder Interfaces können Änderungen an einer konkreten Implementierung vorgenommen werden, ohne dass dies Auswirkungen auf die anderen Module oder Klassen hat, die von dieser Implementierung abhängen. Das Dependency-Inversion-Principle trägt somit zur Flexibilität, Erweiterbarkeit und Wartbarkeit von Software bei. Ein Beispiel für das Dependency-Inversion-Principle ist zum beispiel unser Entity Manager. Hier wurde die allgemeine Schnittstelle IEntityManager definiert, welches die Methoden des EntityManager beinhaltet. Der EntityManager selbst wird in die Adapters-Schicht implementiert. Dadurch haben die Entitäten im Domain Code keine Abhängigkeit auf die konkreten Implementierungen, sondern erhält diese lediglich durch Aufrufe der Methoden.

5.2 GRASP

GRASP steht für General Responsibility Assignment Software Patterns/Prinziples und ist ein Muster oder Prinzipg, die sich mit der Zuweisung von Verantwortlichkeiten an Objekte befasst. GRASP stellt 9 Lösungsprinzipien für die Softwareentwicklung vor. Zum Grundkonzept gehören zwei der Lösungsprinzipien: Low Coupling und High Cohesion. Beide gehören mit zuden wichtigsten Prinzipen für das GRASP-Programmierprinzip. Daher werden beide im folgenden anhand unserer Anwendung erläutert und analysiert.

5.2.1 Low Coupling

Low Coupling ist eines der Muster, das sich auf die Reduzierung der Abhängigkeiten einer Klasse von ihrer Umgebung konzentriert. Das Ziel ist es, die Verbindungen zwischen den verschiedenen Komponenten im System zu minimieren, um eine höhere Flexibilität, leichtere Anpassbarkeit, gute Testbarkeit, erhöhte Wiederverwendbarkeit und Erweiterbarkeit zu erreichen. Außerdem je loser die Kopplung, desto leichter ist die Austauschbarkeit der Funktionalität. Auch die in Kapitel 3 gemachten Änderungen haben darauf abgezielt die Kopplung zu reduzieren. Denn das Ziel von Clean Architecture ist, die Abhängigkeiten

zwischen verschiedenen Schichten eines Systems zu minimieren. Das bedeutet, dass jede Schicht in der Architektur so gestaltet wurde, dass sie nur von den Schichten darunter abhängt und keine Kenntnis über die Schichten darüber hat. Da das Projekt der Clean Architecture entspricht, gilt allgemein, dass die inneren Schichten eine geringe Kopplung zu den äußeren Klassen haben sollten. Somit sollten die Klassen im Domain Code generell eine geringere Kopplung als die Klassen in den Plugins haben. Allerdings können die Klassen in den untern Schichten auch innerhalb einer Schicht viele Abhängigkeiten haben. So haben die Klasse Bild und Schwierigkeit eine geringe Kopplung, wobei Rezept eine hohe Kopplung hat. In der Klasse Rezept koppeln sich die Abhängigkeiten mit den Klassen Kategorie, Zutat, Bild und Schwierigkeit. Für eine zusätzliche stärkere Kopplung sorgt die Instanziierung der Arraylisten von Kategorie und Zutat im Konstruktor. Auf der anderen Seite sind neben einigen Domain-Code Klassen unsere GUI-Klassen Beispiele für Klassen mit schwacher Kopplung. Diese Klassen besitzen jeweils nur wenige Abhängigkeit (Kopplung) zueinander. Ein weiteres Beispiel für geringe Kopplung ist die Klasse FunktionenStartseite. Die Klasse weist eine Abhängigkeit mit der Klasse NeueKategorie auf.

5.2.2 High Cohesion

High Cohesion ist ein Maß für den inneren Zusammenhalt einer Klasse, zeigt also wie eng die Methoden und Attribute einer Klasse zusammenarbeiten. Mit anderen Worten, eine Klasse sollte eng miteinander verbundene und verwandte Funktionalitäten enthalten, um eine hohe Kohäsion zu erreichen. Die Klasse EntityManager der GUI Funktionen zeugt von geringerer Kohäsion, da hier mehrere Use-Cases in einer Klasse implementiert werden. Der EntityManager dient der Datenhaltung und hat die Aufgaben, Objekte zu speichern/löschen und zu finden. Eine verbesserte Implementierung bietet sich folgendermaßen an, indem die einzelnen Operationen separat in Klassen eines Moduls implementiert werden.

FunktionenZufallsGenerator zeigt von hoher Kohäsion, da diese lediglich einen Use-Case behandelt. Diese Klasse hat lediglich die Aufgabe, aus einer Liste von Rezepten ein „zufälliges“ Rezept auszuwählen. Dafür besitzt diese Klasse lediglich die Methode zufälligeRezeptUUID, die diese Logik implementiert.

5.3 DRY

Das DRY-Prinzip steht für "Don't Repeat Yourselfünd besagt, dass man eine bestimmte Information oder Funktionalität in einer Software nur an einer Stelle definieren sollte, um Redundanz zu vermeiden. Jeder Wissensaspekt darf nur eine einzige, unzweideutig verbindliche Repräsentation in einem System besitzen. Das bedeutet, dass man sich bemüht, Code-Duplikationen zu vermeiden und stattdessen wiederverwendbare Komponenten und Funktionen zu schaffen, die an verschiedenen Stellen in der Software verwendet werden können. Auf diese Weise wird der Code einfacher zu warten, zu testen und zu erweitern.

Ein Negativbeispiel für das Nichteinhalten des DRY-Prinzips ist in der Implementierung der GUI Klassen. Der generelle Aufbau der GUI wurde in jeder Klasse implementiert und lediglich der Hauptinhalt ausgetauscht. Das führt zu einer Dopplung der Informationen. Würden die Entwickler entscheiden beispielsweise den Footer der GUIs ändern zu wollen und beispielsweise den Button Text Startseite durch HomePage ersetzen zu wollen, müsste diese kleine Änderung in 3 Klassen geändert werden. Diese Verletzung des DRY-Prinzips wurde aus Unerfahrenheit der Entwickler begangen und soll in Zukunft vermieden werden.

Was sind weitere Dopplungen oder eher auslagerungen?

Ein Beispiel für die Einhaltung des DRY-Prinzips ist in der Verifizierung der Parameter eines Callbacks zu sehen. Hier wird zentral in jeder Callback-Klasse durch die Metho28 Programmentwurf Programming Principles de getRequiredParameters definiert, welche Parameter benötigt werden. Das CLI Plugin nutzt diese Informationen, um die übergebenen Parameter in der Methode ensureRequiredParameters auf Vollständigkeit für den entsprechenden Callback zu prüfen sowie um die Benutzerhilfe in constructOptions zu generieren.