

Programmentwurf - Protokoll

MentalHealthApp

Name: Sorokovaia, Ekaterina

Martrikelnummer: 9266835

Abgabedatum: 04.05.2025

Inhaltsverzeichnis

Inhaltsverzeichnis	2
1. Kapitel 1: Einführung (4P)	1
1.1 Übersicht über die Applikation (1P)	1
1.2 Starten der Applikation (1P).....	1
1.3 Technischer Überblick (2P)	1
2. Kapitel 2: Softwarearchitektur (8P).....	2
2.1 Gewählte Architektur (4P).....	2
2.1.1 Schichtenmodell der Applikation:.....	2
2.1.2 UML-Diagramm.....	4
2.2 Domain Code (1P)	5
2.3 Analyse der Dependency Rule (3P).....	6
2.3.1 <i>Positiv-Beispiel: Dependency Rule</i>	6
2.3.2 <i>Negativ-Beispiel: Dependency Rule</i>	6
3. Kapitel 3: SOLID (8P).....	7
3.1 Analyse SRP (3P).....	7
3.1.1 Positiv-Beispiel.....	7
3.1.2 Negativ-Beispiel	8
3.2 Analyse OCP (3P)	9
3.2.1 Positiv-Beispiel.....	9
3.2.2 Negativ-Beispiel	9
3.3 Analyse DIP (2P)	10
3.3.1 Positiv-Beispiel (DIP)	10
3.3.2 Negativ-Beispiel (DIP).....	11
4. Kapitel 4: Weitere Prinzipien (8P)	12
4.1 Analyse GRASP: Geringe Kopplung (3P).....	12
4.2 Analyse GRASP: Pure Fabrication (3P)	13
4.3 DRY (2P)	14
5. Kapitel 5: Unit Tests (8P)	15
5.1 10 Unit Tests (2P).....	15
5.2 ATRIP: Automatic, Thorough und Professional (2P).....	20
5.3 Fakes und Mocks (4P).....	22
6. Kapitel 6: Domain Driven Design (8P).....	25
6.1 Ubiquitous Language (2P)	25
6.2 Repositories (1,5P).....	26
6.3 Aggregates (1,5P).....	27

6.4	Entities (1,5P)	28
6.5	Value Objects (1,5P).....	29
7.	Kapitel 7: Refactoring (8P)	30
7.1	Code Smells (2P).....	30
7.2	2 Refactorings (6P).....	31
8.	Kapitel 8: Entwurfsmuster (8P).....	33

1. Kapitel 1: Einführung (4P)

1.1 Übersicht über die Applikation (1P)

[Was macht die Applikation? Wie funktioniert sie? Welches Problem löst sie/welchen Zweck hat sie?]

Die Applikation ist eine textbasierte MentalHealthApp, die Benutzer bei der Förderung ihrer psychischen Gesundheit unterstützt. Sie bietet verschiedene Module wie einen Stimmungskalender, Achtsamkeits- und Atemübungen, Routinenmanagement, ein Zielsetzungsmodul sowie Funktionen zur Selbstreflexion und zur Bewältigung von Gedankenkreisen.

Die Applikation verfolgt das Ziel, tägliche Selbstfürsorge zur Gewohnheit werden zu lassen, emotionale Muster besser zu verstehen und langfristig den mentalen Zustand zu stabilisieren bzw. zu verbessern. Dabei wird ein strukturierter Rahmen geboten, um Routinen aufzubauen, Emotionen zu reflektieren, Fortschritte festzuhalten und persönliche Ziele zu verfolgen.

1.2 Starten der Applikation (1P)

[Wie startet man die Applikation? Was für Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]

- **Projekt klonen:**

```
git clone https://github.com/kattyterra/MentalHealthApp.git
```

```
cd MentalHealthApp
```

- **Kompilieren:**

```
javac App.java
```

- **Ausführen:**

```
java App
```

1.3 Technischer Überblick (2P)

[Nennung und Erläuterung der Technologien (z.B. Java, MySQL, ...), jeweils Begründung für den Einsatz der Technologien]

Die Applikation wurde in Java realisiert, einer objektorientierten Programmiersprache, die sich durch Plattformunabhängigkeit, gute Wartbarkeit und breite Unterstützung im Bereich der Softwareentwicklung auszeichnet. Java bietet

darüber hinaus geeignete Strukturen zur Anwendung etablierter Architektur- und Entwurfsprinzipien wie SOLID, GRASP und Domain-Driven Design (DDD).

Gemäß den Projektvorgaben erfolgt die Benutzerinteraktion über eine textbasierte Konsole. Diese Form der Ein- und Ausgabe ermöglicht eine schlanke Umsetzung ohne grafische Oberfläche und unterstützt die Plattformunabhängigkeit der Anwendung.

Auch die Datenspeicherung in UTF-8-kodierten Textdateien basiert auf den vorgegebenen Rahmenbedingungen. Die Speicherung erfolgt dabei strukturiert und nachvollziehbar, sodass eine einfache Weiterverarbeitung sowie Nachvollziehbarkeit der Nutzerdaten (z. B. Stimmungen, Routinen, Ziele) gewährleistet ist.

Die Architektur der Anwendung orientiert sich an den Prinzipien der Clean Architecture, wodurch eine klare Trennung zwischen fachlicher Logik, Anwendungslogik und Persistenzschicht realisiert wird. Diese Trennung unterstützt eine hohe Flexibilität und erleichtert zukünftige Erweiterungen, beispielsweise durch den Austausch der Speichermethode.

2. Kapitel 2: Softwarearchitektur (8P)

2.1 Gewählte Architektur (4P)

[*In der Vorlesung wurden Softwarearchitekturen vorgestellt. Welche Architektur wurde davon umgesetzt? Analyse und Begründung inkl. UML der wichtigsten Klassen, sowie Einordnung dieser Klassen in die gewählte Architektur*]

Die Applikation wurde auf Grundlage einer Schichtenarchitektur umgesetzt. Dieses Modell folgt dem Prinzip der Separation of Concerns, bei dem jede Schicht eine klar abgegrenzte Aufgabe übernimmt. Die Schichten kommunizieren dabei stets nur mit der jeweils direkt darunter liegenden Schicht.

2.1.1 Schichtenmodell der Applikation:

Präsentationsschicht (Presentation Layer):

Diese Schicht bildet die textbasierte Benutzerschnittstelle ab. Sie ist für die Interaktion mit dem Nutzer zuständig und verarbeitet die Ein- und Ausgaben über die Konsole. Die Menüklassen koordinieren dabei den Benutzerfluss, enthalten jedoch keine Anwendungslogik.

Zugehörige Klassen: MainMenu, ZielMenu, RoutinenMenu, ÜbungMenu, StimmungskalenderMenu, TagebuchMenu, GedankenkarussellMenu, InspirationssatzMenu

Anwendungsschicht (Application Layer / Services):

Diese Schicht enthält die zentrale Steuerlogik der Applikation. Sie ist in zustandslose Service-Klassen unterteilt, die für die Umsetzung konkreter Anwendungsfälle verantwortlich sind. Sie verwenden die Geschäftsobjekte aus der Domänenschicht und delegieren Speicher- bzw. Ladevorgänge an die darunterliegende Schicht.

Zugehörige Klassen: ZielVerwaltung, RoutinenVerwaltung, ÜbungenVerwaltung, StimmungskalenderVerwaltung, TagebuchVerwaltung, FortschrittsberichtService, InspirationssatzVerwaltung

Domänenschicht (Domain Layer)

In dieser Schicht sind die zentralen Geschäftsobjekte und Entitäten definiert. Diese Klassen modellieren die fachliche Logik der Anwendung, sind weitgehend zustandsbehaftet und kapseln Regeln oder Strukturen, die unabhängig von technischen Details sind.

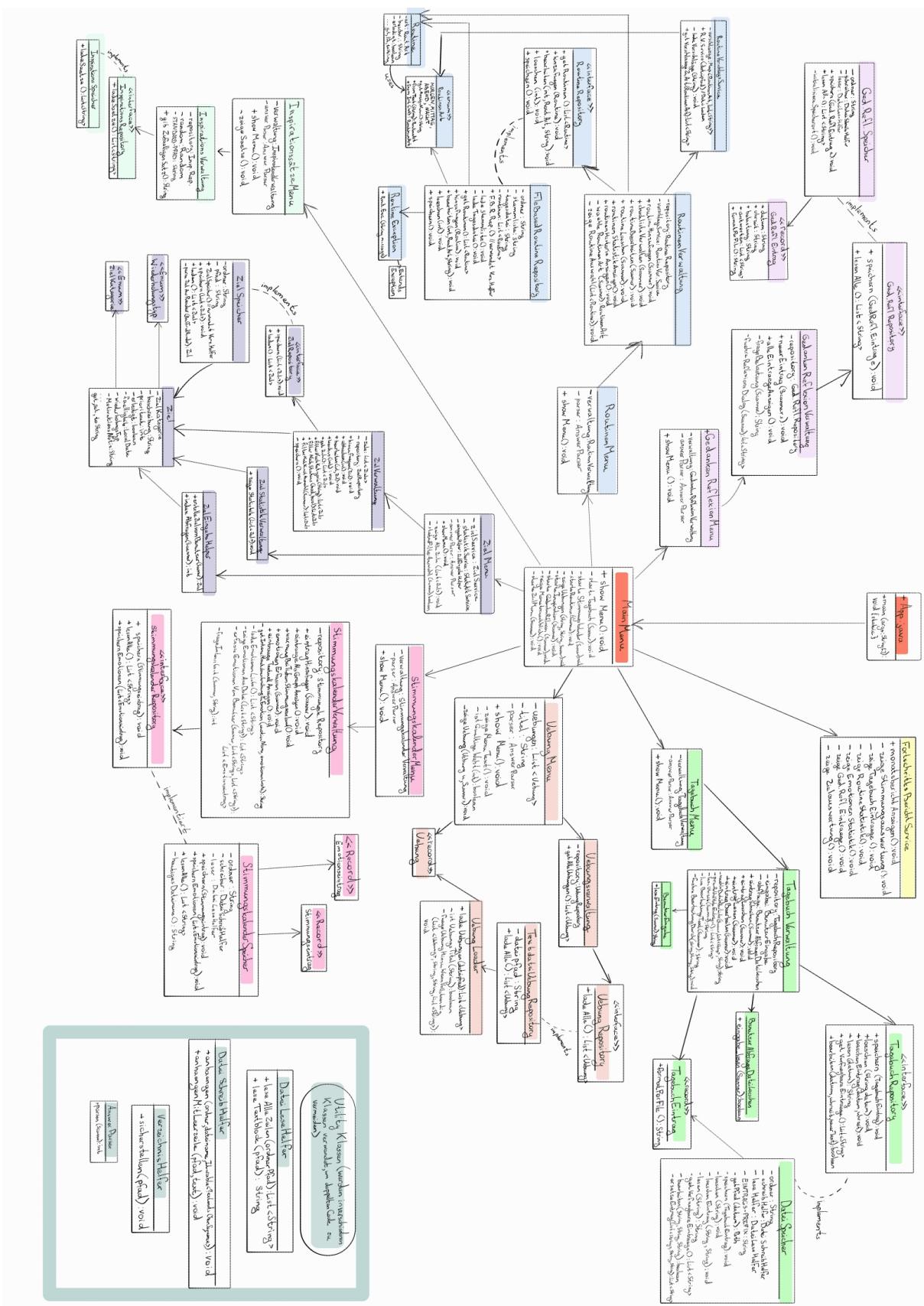
Zugehörige Klassen: Ziel, ZielStatus, Zielkategorie, Routine, RoutinenArt, Übung, ÜbungKategorie, StimmungsEintrag, Emotion, TagebuchEintrag, GedankenkarussellEintrag

Datenzugriffsschicht (Persistence Layer)

Diese Schicht ist zuständig für das Laden und Speichern von Daten in Dateien. Sie enthält Klassen, die Schnittstellen oder implementieren. Auf diese Weise wird die Speicherlogik vollständig von der Anwendungslogik getrennt.

Zugehörige Klassen: GedankenReflexionSpeicher, FileBasedRoutineRepository, ZielSpeicher, StimmungskalenderSpeicher, TextdateiUebungRepository, DateiSpeicher, DateiLeseHelper, DateiSchreibHelper

2.1.2 UML-Diagramm



2.2 Domain Code (1P)

[kurze Erläuterung in eigenen Worten, was Domain Code ist – 1 Beispiel im Code zeigen, das bisher noch nicht gezeigt wurde]

Domain Code ist der Teil einer Software, der die fachliche Logik abbildet – also die Regeln und Abläufe der realen Welt, für die die Anwendung entwickelt wurde. Er ist unabhängig von Technik wie Datenbanken oder Benutzeroberflächen und konzentriert sich auf das „Was“ und „Warum“ einer Funktion. Typisch für Domain Code sind Klassen wie Entities, Value Objects und Domain Services, die die Geschäftslogik klar und verständlich strukturieren.

Beispiel:

```
1 package routinen_logik;
2 /*
3  * Repräsentiert eine einzelne Routine in der MentalHealthApp.
4  */
5 public class Routine {
6
7     private RoutinenArt art;
8     private String beschreibung;
9     private boolean erledigt;
10
11    public Routine(RoutinenArt art, String beschreibung) {
12        this.art = art;
13        this.beschreibung = beschreibung;
14        this.erledigt = false;
15    }
16
17    public RoutinenArt getArt() { return art; }
18
19    public String getBeschreibung() { return beschreibung; }
20
21    public boolean isErledigt() { return erledigt; }
22
23    public void setArt(RoutinenArt art) { this.art = art; }
24
25    public void setBeschreibung(String beschreibung) { this.beschreibung = beschreibung; }
26
27    public void setErledigt(boolean erledigt) { this.erledigt = erledigt; }
28
29    @Override
30    public String toString() {
31        String status = erledigt ? "[√]" : "[ ]";
32        return status + " [" + art + "] - " + beschreibung;
33    }
34}
```

- **Fachliche Repräsentation:**

- o Die Klasse Routine bildet ein zentrales Konzept der Domäne ab – nämlich eine tägliche Gewohnheit oder Aktivität zur mentalen Gesundheit.

- **Geschäftslogik enthalten:**

- o Die Klasse weiß, ob eine Routine erledigt ist oder nicht, und stellt das

im `toString()` dar. Das ist keine technische Logik, sondern domänenspezifische Darstellung.

- **Unabhängig von Technik:**

- o Es gibt keinerlei technische Abhängigkeiten – keine Datenbank, keine Benutzeroberfläche, kein Framework. Nur Java-Logik, die ausdrückt, was eine Routine inhaltlich ist.

2.3 Analyse der Dependency Rule (3P)

[In der Vorlesung wurde im Rahmen der ‘Clean Architecture’ die s.g. Dependency Rule vorgestellt. Je 1 Klasse zeigen, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

2.3.1 Positiv-Beispiel: Dependency Rule

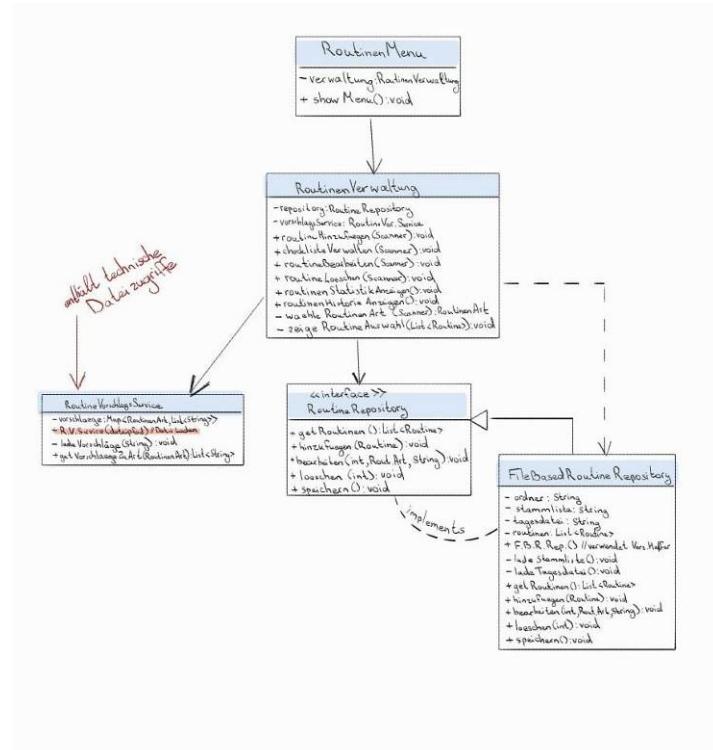
RoutinenVerwaltung → RoutineRepository → FileBasedRoutineRepository

- Die Klasse RoutinenVerwaltung kennt nur das Interface RoutineRepository
- Die konkrete Implementierung FileBasedRoutineRepository wird nicht direkt verwendet, sondern per Abstraktion (Interface)
- Die fachliche Logik (RoutinenVerwaltung) hängt nicht von technischer Infrastruktur ab, sondern nur von einer Abstraktion

2.3.2 Negativ-Beispiel: Dependency Rule

RoutinenVerwaltung → RoutineVorschlagsService

Die Klasse RoutineVorschlagsService verletzt die Dependency Rule, da sie direkt in die Geschäftslogik der RoutinenVerwaltung eingebunden ist und beim Erstellen Vorschläge aus einer konkreten Textdatei lädt. Dadurch hängt die zentrale Logik von technischen Details wie dem Dateiformat und dem Dateisystem ab. Änderungen an der Vorschlagsquelle würden somit Änderungen an der Geschäftslogik erzwingen. Um diese Kopplung zu vermeiden, sollte stattdessen ein Interface wie RoutineVorschlagsProvider eingeführt werden, das die Vorschlagslogik abstrahiert. So bleibt die RoutinenVerwaltung unabhängig von der konkreten Implementierung und besser erweiterbar.



UML-Diagramm für Positiv- und Negativ-Beispiel

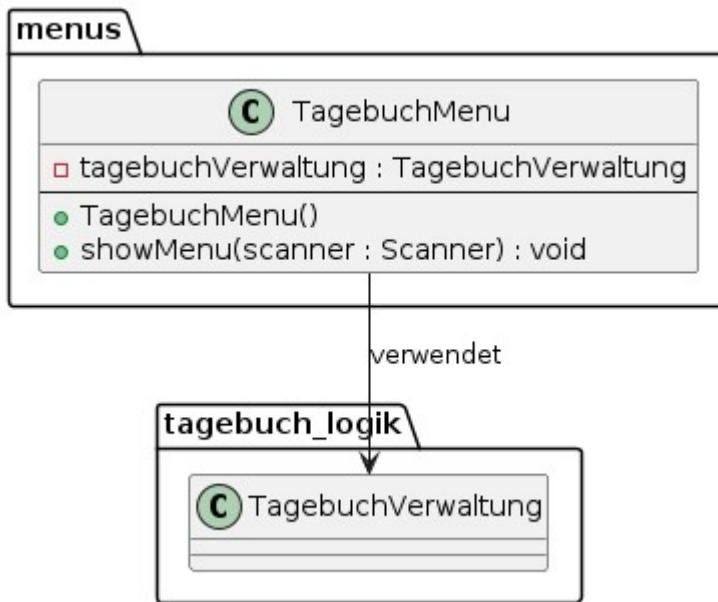
3. Kapitel 3: SOLID (8P)

3.1 Analyse SRP (3P)

[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

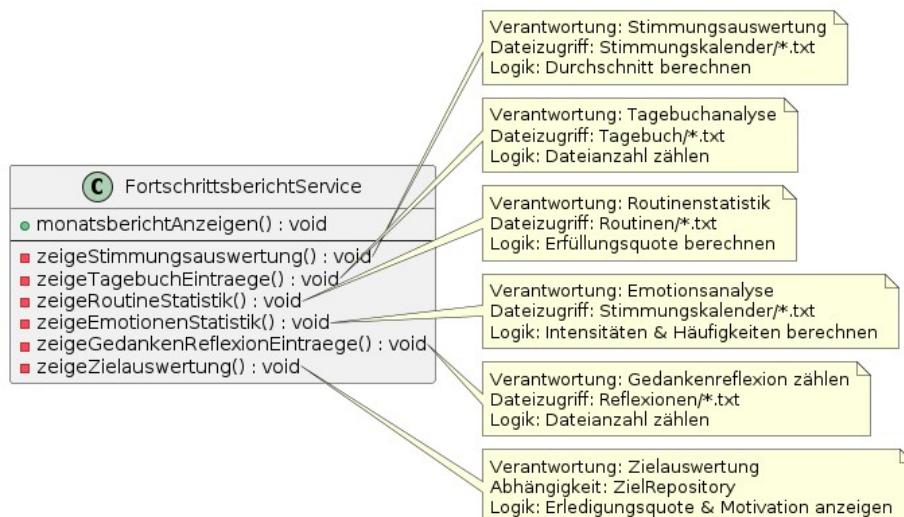
3.1.1 Positiv-Beispiel

Die Klasse TagebuchMenu ist SRP-konform, da sie eine klar abgegrenzte und eindeutig definierte Verantwortung erfüllt: Sie steuert die Benutzerführung innerhalb des Tagebuchmoduls. Ihre Aufgaben beschränken sich auf die Anzeige des Menüs, das Einlesen und Verarbeiten der Benutzereingaben sowie die Weiterleitung der gewählten Aktionen an die zuständige Fachklasse TagebuchVerwaltung. Sie enthält keine eigene Logik zur Bearbeitung, Speicherung oder Analyse von Tagebucheinträgen, sondern delegiert diese Aufgaben konsequent. Dadurch besteht für die Klasse nur ein Grund zur Änderung – nämlich wenn sich die Menüstruktur oder die Benutzerführung ändert. Fachliche Änderungen in der Tagebuchverwaltung erfordern keine Anpassungen an dieser Menüklasse. Somit entspricht TagebuchMenu dem Single-Responsibility-Prinzip.



3.1.2 Negativ-Beispiel

Die Klasse `FortschrittsberichtService` verstößt gegen das Single-Responsibility-Prinzip (SRP), da sie mehrere voneinander unabhängige Aufgaben übernimmt. Sie wertet Daten aus verschiedenen Modulen wie Stimmung, Tagebuch, Routinen, Emotionen, Gedankenreflexion und Zielverwaltung aus und gibt diese gleichzeitig aus. Jede dieser Aufgaben stellt eine eigene Verantwortung dar und würde bei Änderungen unterschiedliche Anpassungen erfordern. Dadurch hat die Klasse mehrere Gründe, sich zu ändern – was SRP widerspricht. Eine SRP-konforme Lösung wäre, die Auswertungslogiken in spezialisierte Klassen auszulagern und die Hauptklasse nur für die Koordination und Ausgabe des Berichts zu verwenden.



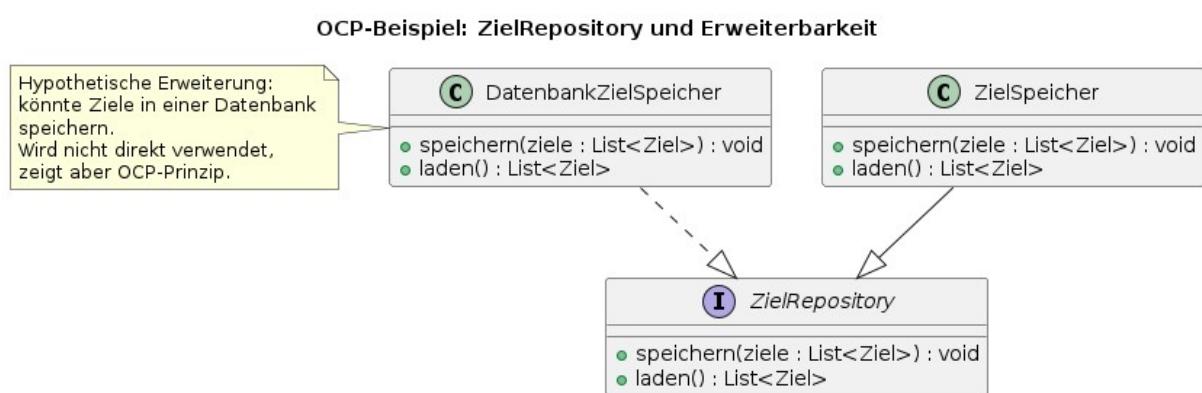
3.2 Analyse OCP (3P)

[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]

3.2.1 Positiv-Beispiel

Die Klassen ZielRepository und ZielSpeicher sind ein Beispiel für die Anwendung des Open/Closed Principle (OCP). Das Interface ZielRepository definiert eine klare, abstrahierte Schnittstelle zum Speichern und Laden von Zielen, ohne sich auf eine bestimmte Speicherform festzulegen. Dadurch ist der Code für Erweiterungen offen: Wenn in Zukunft z.B. eine Datenbank- oder Cloud-Speicherung statt der dateibasierten Variante genutzt werden soll, kann einfach eine neue Klasse wie DatenbankZielSpeicher implementiert werden, ohne den bestehenden Anwendungscode oder die aufrufenden Klassen verändern zu müssen.

Gleichzeitig ist der bestehende Code geschlossen für Änderungen, denn der vorhandene Code in ZielSpeicher oder in den Klassen, die ZielRepository nutzen, muss bei einer solchen Erweiterung nicht angepasst werden – nur die neue Implementierung wird zusätzlich geschrieben und eingebunden. Damit wird das OCP vollständig erfüllt.

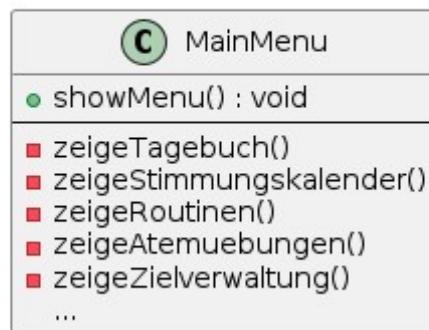


3.2.2 Negativ-Beispiel

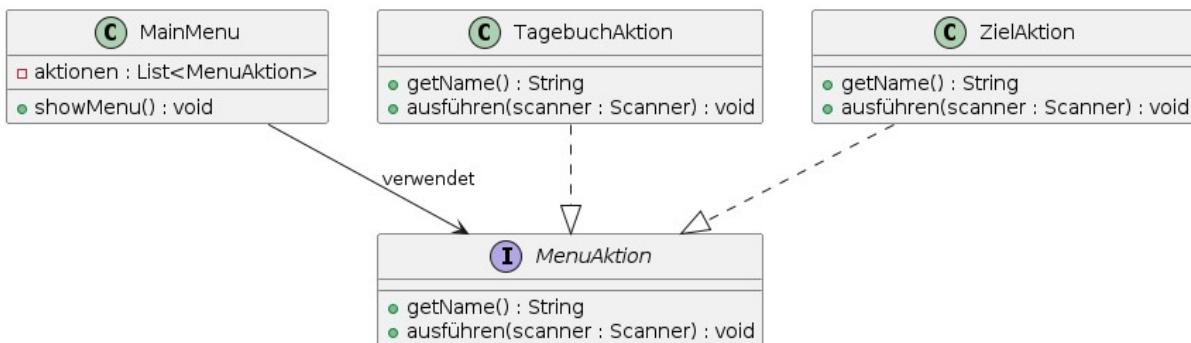
Die Klasse MainMenu verstößt gegen das Open/Closed Principle, da sie bei jeder Erweiterung des Hauptmenüs (z. B. durch neue Module oder Funktionen) angepasst werden muss. Die Menülogik ist aktuell fest im switch-Block implementiert, wodurch

jede neue Funktion eine direkte Änderung am Quellcode der MainMenu-Klasse erfordert. Damit ist die Klasse nicht für Erweiterungen offen, sondern nur durch Modifikation anpassbar – was dem OCP widerspricht. Eine bessere Lösung wäre es, das Menü modular über eine gemeinsame Schnittstelle für Menüaktionen aufzubauen, sodass neue Funktionen einfach hinzugefügt werden können, ohne die bestehende Klasse zu verändern.

MainMenu -> nicht OCP-konform



OCP-konformes Menü



3.3 Analyse DIP (2P)

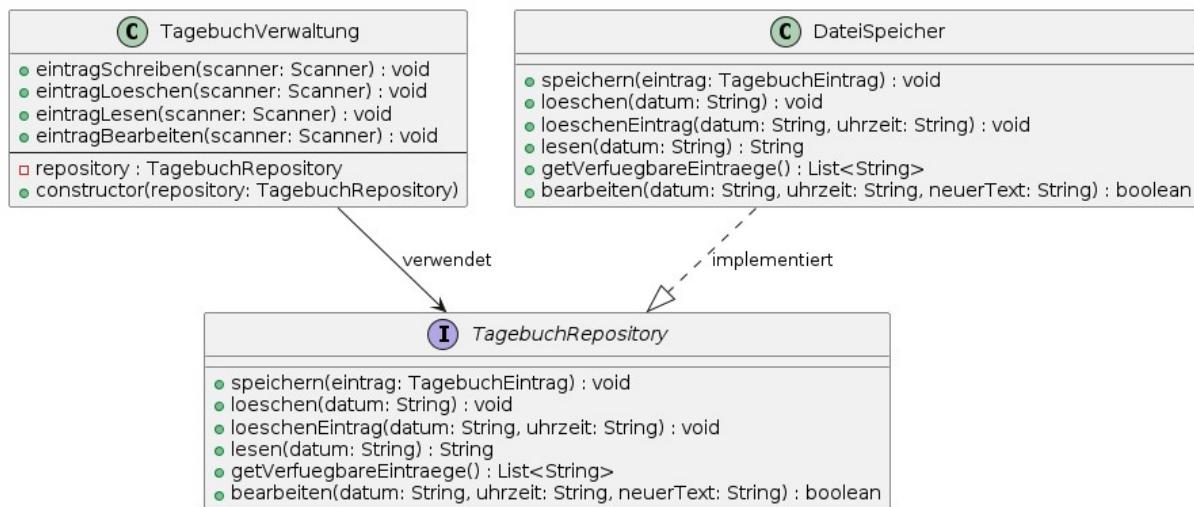
[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP; jeweils UML und Begründung, warum hier das Prinzip erfüllt/nicht erfüllt wird; beim Negativ-Beispiel UML einer möglichen Lösung hinzufügen]

[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

3.3.1 Positiv-Beispiel (DIP)

Die Klasse **TagebuchVerwaltung** ist ein gutes Beispiel für die Anwendung des **Dependency Inversion Principle**. Sie hängt nicht direkt von einer konkreten Speicherstrategie ab, sondern von der Abstraktion **TagebuchRepository**. Dadurch

kann die Speicherstrategie jederzeit ausgetauscht werden (z. B. durch eine Datenbanklösung oder ein anderes Speichersystem), ohne dass Änderungen an der Verwaltungsklasse notwendig sind. Die konkrete Implementierung des Repositories wird über den Konstruktor injiziert, was sicherstellt, dass TagebuchVerwaltung nur mit einer Abstraktion arbeitet und nicht mit einer festen, konkreten Implementierung. Dies entspricht dem DIP und sorgt für hohe **Flexibilität, Erweiterbarkeit** und **Testbarkeit**, da neue Implementierungen von TagebuchRepository problemlos hinzugefügt werden können, ohne die Logik der TagebuchVerwaltung zu ändern.



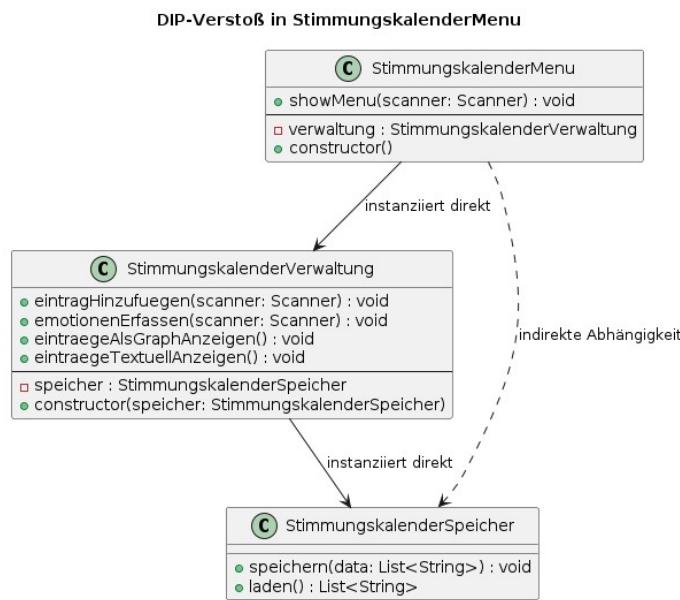
3.3.2 Negativ-Beispiel (DIP)

(Gefixt im Commit am 18.04)

Die Klasse StimmungskalenderMenu verstößt gegen das Dependency Inversion Principle, da sie selbst festlegt, welche konkrete Speicherstrategie innerhalb der Verwaltung verwendet wird. Die Klasse erzeugt intern ein konkretes Objekt vom Typ StimmungskalenderVerwaltung mit fest eingebautem Stimmungskalender-Speicher:

```
this.verwaltung = new StimmungskalenderVerwaltung(new StimmungskalenderSpeicher());
```

Dadurch ist sie indirekt an die Implementierung StimmungskalenderSpeicher gekoppelt. Eine DIP-konforme Lösung besteht darin, die Verwaltung über den Konstruktor zu übergeben. So hängt StimmungskalenderMenu nur noch von Abstraktionen ab, bleibt flexibel erweiterbar und muss bei Änderungen der Speicherlogik nicht angepasst werden.



4. Kapitel 4: Weitere Prinzipien (8P)

4.1 Analyse GRASP: Geringe Kopplung (3P)

[eine bis jetzt noch nicht behandelte Klasse als positives Beispiel geringer Kopplung; UML mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt; es müssen auch die Aufrufer/Nutzer der Klasse berücksichtigt werden]

Klasse UebungsVerwaltung:

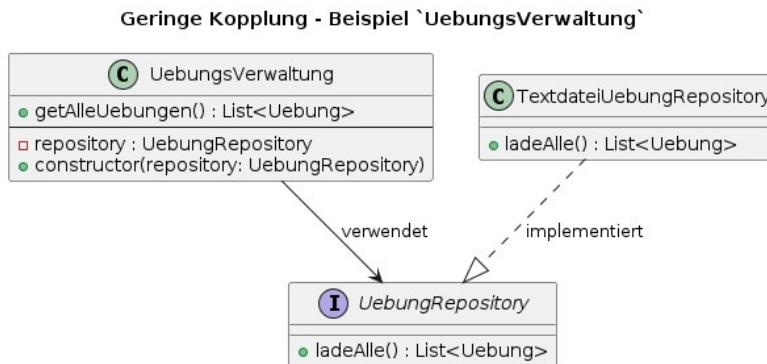
Die UebungsVerwaltung verwaltet Übungen in der Anwendung und greift dabei auf ein Repository (z. B. UebungRepository) zu, das die Speicherung und den Abruf der Übungen ermöglicht. Sie arbeitet mit der Abstraktion UebungRepository und kennt keine konkrete Implementierung wie TextdateiUebungRepository.

Warum geringe Kopplung?

Abstraktion statt konkreter Implementierung: Die UebungsVerwaltung ist nur von der Abstraktion UebungRepository abhängig und nicht von einer spezifischen Implementierung wie TextdateiUebungRepository. Das bedeutet, dass die Implementierung des Repositories einfach ausgetauscht werden kann, ohne die UebungsVerwaltung zu ändern.

Flexibilität: Änderungen an der Speicherstrategie (z. B. von Textdateien zu einer Datenbank) erfordern keine Änderungen an der UebungsVerwaltung.

Durch die Abstraktion kann die UebungsVerwaltung einfach mit Mock-Implementierungen des Repositories getestet werden, ohne auf echte Daten zugreifen zu müssen.



4.2 Analyse GRASP: Pure Fabrication (3P)

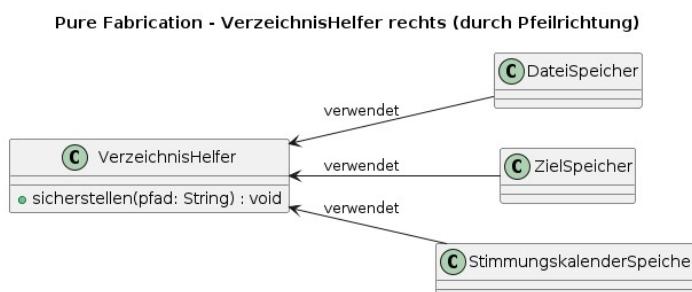
[*eine Klasse als positives Beispiel entweder von Polymorphismus oder von Pure Fabrication; UML Diagramm und Begründung, warum es hier zum Einsatz kommt*]

Die Klasse **VerzeichnisHelper** ist ein gutes Beispiel für **Pure Fabrication**. Diese Klasse wurde eingeführt, um eine **technische Aufgabe** vom fachlichen Code zu trennen: das **Sicherstellen eines Verzeichnisses im Dateisystem**.

Wäre diese Funktion direkt in jeder Speicherklasse (z. B. DateiSpeicher, ZielSpeicher, StimmungskalenderSpeicher) eingebaut, gäbe es:

- **Duplikation** desselben Codes in mehreren Klassen.
- eine **Vermischung von Fach- und Techniklogik**.
- eine **höhere Kopplung** an die konkrete Dateisystem-API

Mit der Auslagerung in die Klasse **VerzeichnisHelper** wurde diese technische Aufgabe **zentralisiert, wiederverwendbar** und vom Fachcode **entkoppelt**. Dadurch bleibt die Fachlogik in den Repository-Klassen klar fokussiert auf das **Speichern von Daten**, und die technische Infrastruktur (Ordnerstruktur) ist ausgelagert.



4.3 DRY (2P)

[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher) einfügen; begründen und Auswirkung beschreiben – ggf. UML zum Verständnis ergänzen]

Commit e579e5c

```

MentalHealthApp/src/main/java/utility/VerzeichnisHelper.java □

@@ -0,0 +1,22 @@
+ package utility;
+
+ import java.io.File;
+
+ public class VerzeichnisHelper {
+
+     /**
+      * Erstellt ein Verzeichnis, wenn es nicht existiert.
+      * Gibt true zurück, wenn es erfolgreich erstellt oder schon vorhanden ist.
+      *
+      * @param pfad Pfad zum Verzeichnis
+      */
+     public void sicherstellen(String pfad) {
+         File dir = new File(pfad);
+         if (!dir.exists()) {
+             boolean success = dir.mkdirs();
+             if (!success) {
+                 System.err.println("Ordner '" + pfad + "' konnte nicht erstellt werden!");
+             }
+         }
+     }
+ }

public DateiSpeicher() {
-     File dir = new File(ordner);
-     if (!dir.exists()) {
-         boolean success = dir.mkdirs();
-         if (!success) {
-             System.err.println("Ordner '" + ordner + "' konnte nicht erstellt werden!");
-         }
-     }
- }
+     VerzeichnisHelper verzeichnisHelper = new VerzeichnisHelper();
+     verzeichnisHelper.sicherstellen(ordner);
+     this.schreibHelper = new DateiSchreibHelper();
+     this.leseHelper = new DateiLeseHelper();
}

```

Im Commit e579e5c wurde duplizierter Code zur Verzeichnis-Erstellung ausgelagert und somit das DRY-Prinzip umgesetzt.

Vorher (Verstoß gegen DRY):

In der Klasse DateiSpeicher war die Logik zum Anlegen eines Verzeichnisses direkt eingebettet. Diese Logik wurde identisch auch in anderen Klassen verwendet (z. B. ZielSpeicher, StimmungskalenderSpeicher). Änderungen oder Fehlerbehebungen hätten dort an mehreren Stellen erfolgen müssen – das widerspricht dem DRY-Prinzip.

Nachher (DRY-konform):

Die Verzeichnis-Erstellung wurde in die neue Hilfsklasse VerzeichnisHelper ausgelagert. Damit befindet sich der gesamte Logikblock in nur einer einzigen Methode (sicherstellen(String pfad)). Die betroffenen Speicherklassen rufen nun zentral diese Methode auf.

Begründung & Wirkung:

- **Zentrale Verwaltung:** Änderungen (z. B. Logging, Fehlerbehandlung) können an einer Stelle durchgeführt werden.
- **Wartbarkeit** steigt, weil Wiederholungen vermieden wurden.
- **Kopplung wird reduziert**, da die Speicherklassen nun nicht mehr direkt mit dem Dateisystem arbeiten.
- **Wiederverwendbarkeit:** VerzeichnisHelper kann nun auch in anderen Modulen verwendet werden.

5. Kapitel 5: Unit Tests (8P)

5.1 10 Unit Tests (2P)

[Zeigen und Beschreiben von 10 Unit-Tests und Beschreibung, was getestet wird]

1. Klasse: DateiSchreibHelperTest – Test der Dateischreiblogik

Die Klasse DateiSchreibHelper ist für das Anhängen von Textzeilen an Dateien zuständig, sowohl mit als auch ohne Leerzeile. Um sicherzustellen, dass diese Funktionalitäten korrekt umgesetzt sind, wurden folgende fünf Tests entwickelt:

1. anhaengen_schreibtMehrereZeilenAnDatei

Dieser Test prüft, ob mehrere Zeilen korrekt an eine neue Datei geschrieben

werden. Dabei wird ein temporäres Verzeichnis verwendet, um die reale Dateiumgebung zu simulieren, ohne auf das produktive Dateisystem zuzugreifen.

2. **anhaengen_haengtAnBestehendeDateiAn**

Hier wird sichergestellt, dass bestehende Dateien korrekt erweitert werden und die neuen Inhalte tatsächlich an die vorhandenen Zeilen angehängt werden.

3. **anhaengenMitLeerzeile_fuegtLeerzeileWennDateiNichtLeer**

Dieser Test überprüft das Verhalten bei bereits existierenden Inhalten. Wenn die Datei nicht leer ist, muss vor dem neuen Eintrag eine Leerzeile stehen. Diese Anforderung wird durch diesen Test überprüft.

4. **anhaengenMitLeerzeile_ohneLeerzeileWennDateiLeer**

Um eine korrekte Formatierung sicherzustellen, wird hier getestet, ob bei einer leeren Datei keine unnötige Leerzeile vorangestellt wird, sondern direkt der Inhalt geschrieben wird.

5. **anhaengenMitLeerzeile_erstelltDateiWennNichtVorhanden**

Dieser Test stellt sicher, dass beim ersten Aufruf der Methode auch eine neue Datei korrekt erstellt wird, falls sie noch nicht vorhanden ist.

Durch diese fünf Tests wird sichergestellt, dass alle Varianten des Dateizugriffs robust und korrekt funktionieren, auch im Zusammenspiel mit vorhandenen Inhalten oder fehlenden Dateien.

2. Klasse: **TagebuchVerwaltungTest – Test der Benutzerinteraktion im Tagebuchmodul**

Die Klasse TagebuchVerwaltung steuert sämtliche Benutzerinteraktionen im Tagebuch-Modul – vom Schreiben über das Bearbeiten bis zum Löschen von Einträgen. Um die Funktionalität dieser Klasse zu prüfen, wurden folgende fünf Unit-Tests erstellt:

1. **eintragSchreiben_speichertEintragMitHeutigemDatumUndText**

Es wird geprüft, ob ein Tagebucheintrag mit dem aktuellen Datum und dem vorgegebenen Text korrekt gespeichert wird. Hierbei kommen Mocks für die Benutzereingabe und das Repository zum Einsatz.

2. eintragLesen_gibtEintragFuerGewaehltesDatumAus

Dieser Test überprüft, ob das gewählte Datum korrekt an das Repository übergeben wird, wenn der Benutzer einen Eintrag lesen möchte. Dabei wird ein Scanner mit simulierter Eingabe verwendet.

3. eintragBearbeiten_ruftBearbeitenMitEingabenAuf

Der Test stellt sicher, dass beim Bearbeiten eines Eintrags die Eingaben für Uhrzeit und neuer Text korrekt verarbeitet und an das Repository übergeben werden.

4. eintragLoeschen_ganzeDateiWirdGeloeschtWennAbfrageTrue

Es wird simuliert, dass der Benutzer die gesamte Datei löschen möchte. Der Test prüft, ob in diesem Fall nur das Datum (und keine Uhrzeit) an das Repository übergeben wird – ein Indikator für die Löschung der gesamten Datei.

5. eintragLoeschen_einzelnerEintragWirdGeloeschtWennAbfrageFalse

Im Gegensatz zum vorherigen Test wird hier geprüft, ob beim Löschen eines einzelnen Eintrags zusätzlich die Uhrzeit korrekt berücksichtigt wird. Dies wird ebenfalls über einen Mock realisiert.

Durch diese Tests wird sichergestellt, dass die gesamte Eingabesteuerung und Weitergabe der Informationen an das Repository vollständig und korrekt abläuft – unabhängig davon, ob der Benutzer Einträge erstellt, liest, bearbeitet oder löscht.

```
1 package utility_tests;
2
3 import org.junit.jupiter.api.Test;
4 import org.junit.jupiter.api.io.TempDir;
5 import utility.DateiSchreibHelper;
6
7 import java.io.IOException;
8 import java.nio.file.*;
9 import java.util.List;
10
11 import static org.junit.jupiter.api.Assertions.*;
12
13 class DateiSchreibHelperTest {
14
15     @Test
16     void anhaengen_schreibtMehrereZeilenAnDatei(@TempDir Path tempDir) throws IOException {
17         Path datei = tempDir.resolve("test.txt");
18         DateiSchreibHelper helper = new DateiSchreibHelper();
19
20         helper.anhaengen(tempDir.toString() + "/", "test.txt", List.of("Zeile 1", "Zeile 2"));
21
22         List<String> zeilen = Files.readAllLines(datei);
23         assertEquals(List.of("Zeile 1", "Zeile 2"), zeilen);
24     }
25
26     @Test
27     void anhaengen_haengtAnBestehendeDateiAn(@TempDir Path tempDir) throws IOException {
28         Path datei = tempDir.resolve("anhang.txt");
29         Files.writeString(datei, List.of("Vorhanden"));
30
31         DateiSchreibHelper helper = new DateiSchreibHelper();
32         helper.anhaengen(tempDir.toString() + "/", "anhang.txt", List.of("Neu 1", "Neu 2"));
33
34         List<String> zeilen = Files.readAllLines(datei);
35         assertEquals(List.of("Vorhanden", "Neu 1", "Neu 2"), zeilen);
36     }
37
38     @Test
39     void anhaengenMitLeerzeile_fuegtLeerzeileWennDateiNichtLeer(@TempDir Path tempDir) throws IOException {
40         Path datei = tempDir.resolve("mitLeerzeile.txt");
41         Files.writeString(datei, "Vorher");
42
43         DateiSchreibHelper helper = new DateiSchreibHelper();
44         helper.anhaengenMitLeerzeile(datei.toString(), "Nachher");
45
46         List<String> zeilen = Files.readAllLines(datei);
47         assertEquals(List.of("Vorher", "Nachher"), zeilen);
48     }
49
50     @Test
51     void anhaengenMitLeerzeile_ohneLeerzeileWennDateiLeer(@TempDir Path tempDir) throws IOException {
52         Path datei = tempDir.resolve("leer.txt");
53
54         DateiSchreibHelper helper = new DateiSchreibHelper();
55         helper.anhaengenMitLeerzeile(datei.toString(), "Erste Zeile");
56
57         List<String> zeilen = Files.readAllLines(datei);
58         assertEquals(List.of("Erste Zeile"), zeilen);
59     }
60
61     @Test
62     void anhaengenMitLeerzeile_ergibtDateiWennNichtVorhanden(@TempDir Path tempDir) throws IOException {
63         Path datei = tempDir.resolve("neu.txt");
64
65         assertFalse(Files.exists(datei)); // Datei darf noch nicht da sein
66
67         DateiSchreibHelper helper = new DateiSchreibHelper();
68         helper.anhaengenMitLeerzeile(datei.toString(), "Startzeile");
69
70         assertTrue(Files.exists(datei));
71         assertEquals(List.of("Startzeile"), Files.readAllLines(datei));
72     }
73 }
```

Programmentwurf – Protokoll: MentalHealthApp

```
1 package tagebuch_logik_tests;
2
3 import org.junit.jupiter.api.BeforeEach;
4 import org.junit.jupiter.api.Test;
5 import tagebuch_logik.*;
6
7 import java.time.LocalDate;
8 import java.util.List;
9 import java.util.Scanner;
10
11 import static org.junit.jupiter.api.Assertions.*;
12
13 class TagebuchVerwaltungTest {
14
15     private MockTagebuchRepository repository;
16     private MockBenutzerEingabe eingabe;
17     private MockAbfrage abfrage;
18     private TagebuchVerwaltung verwaltung;
19
20     static class MockTagebuchRepository implements TagebuchRepository {
21         public TagebuchEintrag gespeicherterEintrag;
22         public String zuletztGelesenesDatum;
23         public String geloeschtesDatum;
24         public String geloeschteUhrzeit;
25         public String bearbeitetesDatum;
26         public String bearbeiteteUhrzeit;
27         public String bearbeiteterText;
28         public boolean bearbeitenErgebnis = true;
29
30         @Override
31         public void speichern(TagebuchEintrag eintrag) {
32             this.gespeicherterEintrag = eintrag;
33         }
34
35         @Override
36         public void loeschen(String datum) {
37             this.geloeschtesDatum = datum;
38         }
39
40         @Override
41         public void loeschenEintrag(String datum, String uhrzeit) {
42             this.geloeschtesDatum = datum;
43             this.geloeschteUhrzeit = uhrzeit;
44         }
45
46         @Override
47         public String lesen(String datum) {
48             this.zuletztGelesenesDatum = datum;
49             return "Test-Inhalt";
50         }
51
52         @Override
53         public List<String> getVerfuegbareEintraege() {
54             return List.of("2025-04-18");
55         }
56
57         @Override
58         public boolean bearbeiten(String datum, String uhrzeit, String neuerText) {
59             this.bearbeitetesDatum = datum;
60             this.bearbeiteteUhrzeit = uhrzeit;
61             this.bearbeiteterText = neuerText;
62             return bearbeitenErgebnis;
63         }
64     }
65
66     static class MockAbfrage extends BenutzerAbfrageDateiLoeschen {
67         private final boolean ganzeDatei;
68
69         public MockAbfrage(boolean ganzeDatei) {
70             this.ganzeDatei = ganzeDatei;
71         }
72
73         @Override
74         public boolean eingabe_lesen(Scanner scanner) {
75             return ganzeDatei;
76         }
77     }
78
79     static class MockBenutzerEingabe extends BenutzerEingabe {
80         private final String text;
81
82         public MockBenutzerEingabe(String text) {
83             this.text = text;
84         }
85
86         @Override
87         public String leseEintrag(Scanner scanner) {
88             return text;
89         }
90     }
91
92     @BeforeEach
93     void setup() {
94         repository = new MockTagebuchRepository();
95         eingabe = new MockBenutzerEingabe("Testtext");
96         abfrage = new MockAbfrage(false); // default: nur Eintrag löschen, nicht ganze Datei
97         verwaltung = new TagebuchVerwaltung(repository, eingabe, abfrage);
98     }
99
100    @Test
101    void eintragSchreiben_speichertEintragMitHeutigemDatumUndText() {
102        verwaltung.eintragSchreiben(new Scanner(""));
103
104        assertEquals("Testtext", repository.gespeicherterEintrag.text());
105        assertEquals(LocalDate.now().toString(), repository.gespeicherterEintrag.datum());
106    }
107
108    @Test
109    void eintraglesen_gibtEintragFuerGewähltesDatumAus() {
110        Scanner scanner = new Scanner("\n");
111        verwaltung.eintraglesen(scanner);
112
113        assertEquals("2025-04-18", repository.zuletztGelesenesDatum());
114    }
115
116    @Test
117    void eintragBearbeiten_ruftBearbeitenMitEingabenAuf() {
118        Scanner scanner = new Scanner("\n08:00\nNeuer Text\n");
119        verwaltung.eintragbearbeiten(scanner);
120
121        assertEquals("2025-04-18", repository.bearbeitetesDatum());
122        assertEquals("08:00:00", repository.bearbeiteteUhrzeit());
123        assertEquals("Neuer Text", repository.bearbeiteterText());
124    }
125
126    @Test
127    void eintragLoeschen_ganzeDateiWirdGelöschtWennAbfrageTrue() {
128        abfrage = new MockAbfrage(true);
129        verwaltung = new TagebuchVerwaltung(repository, eingabe, abfrage);
130        Scanner scanner = new Scanner("\n");
131
132        verwaltung.eintragLoeschen(scanner);
133
134        assertEquals("2025-04-18", repository.geloeschtesDatum());
135        assertNull(repository.geloeschteUhrzeit()); // weil ganze Datei gelöscht
136    }
137
138    @Test
139    void eintragLoeschen_einzelterEintragWirdGelöschtWennAbfrageFalse() {
140        abfrage = new MockAbfrage(false);
141        verwaltung = new TagebuchVerwaltung(repository, eingabe, abfrage);
142        Scanner scanner = new Scanner("\n08:00:00\n");
143
144        verwaltung.eintragLoeschen(scanner);
145
146        assertEquals("2025-04-18", repository.geloeschtesDatum());
147        assertEquals("08:00:00", repository.geloeschteUhrzeit());
148    }
149}
```

5.2 ATRIP: Automatic, Thorough und Professional (2P)

[je Begründung/Erläuterung, wie ‘Automatic’, ‘Thorough’ und ‘Professional’ realisiert wurde – bei ‘Thorough’ zusätzlich Analyse und Bewertung zur Testabdeckung]

Automatic

Die Tests sind vollständig automatisiert und erfordern keinerlei manuelles Eingreifen während der Ausführung. Sie werden über das Build-Tool **Maven** gesteuert, wodurch ein standardisierter, reproduzierbarer Testlauf jederzeit möglich ist. Durch den modularen Aufbau der Testpakete (z.B. `tagebuch_logik_tests`, `zielverwaltung_logik_tests`) kann Maven alle Tests mit dem Befehl **mvn test** automatisiert ausführen.

Jeder Test überprüft das Verhalten der getesteten Komponente eigenständig durch den Einsatz von Assertions wie `assertEquals`, `assertTrue` oder `assertThrows`. Das Ergebnis eines Tests ist damit eindeutig binär – entweder *bestanden* oder *nicht bestanden (failed)*.

Ein Beispiel hierfür ist `DateiSpeicherTest`, wo automatisiert geprüft wird, ob ein Tagebucheintrag nach dem Speichern korrekt wieder aus der Datei gelesen werden kann. Auch in **GedankenReflexionVerwaltungTest** oder **Stimmungskalender-VerwaltungTest** wird das Verhalten bei typischen und fehleranfälligen Nutzungsszenarien automatisiert validiert – etwa beim Hinzufügen und Entfernen von Einträgen oder bei der Verarbeitung fehlerhafter Eingaben.

Benutzereingaben über die Konsole werden gezielt simuliert, etwa durch das Mocking von Scanner-Objekten in den Tests wie `ZielEingabeHelperTest`, was auch interaktive Komponenten automatisiert testbar macht. Dadurch lässt sich die

```
Auswahl: [INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.021 s -- in zielverwaltung_logik_tests.ZielVerwaltungTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 81, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  13.929 s
[INFO] Finished at: 2025-05-01T11:36:02+02:00
[INFO] -----
PS C:\Users\admin\Desktop\SWE_Projekt\MentalHealthApp> 
```

gesamte Applikationslogik deterministisch und ohne Nutzereingriff überprüfen.

Thorough

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cqty	Missed	Lines	Missed	Methods	Missed	Classes
Total	2,690 of 5,465	50%	309 of 473	34%	269	444	733	1,42	63	188	12	45
menus		0%		0%	75	75	282	282	18	18	8	8
routinen_logik		53%		34%	68	108	162	358	10	39	1	6
fortschrittsbericht_logik		0%		0%	37	37	85	85	15	15	1	1
stimmungskalender_logik		42%		22%	36	51	95	155	4	17	0	4
tagebuch_logik		78%		68%	22	53	53	177	4	22	0	5
zielverwaltung_logik		86%		54%	21	66	26	195	5	42	0	7
gedanken_reflexion_logik		87%		100%	2	12	9	57	2	10	0	3
utility		84%		88%	2	20	9	53	0	11	0	4
uebungen		89%		92%	3	14	5	36	2	7	1	4
default		0%	n/a		2	2	4	4	2	2	1	1
inspirations_logik		84%		100%	1	6	3	18	1	5	0	2

Die Teststrategie des Projekts erfüllt das Kriterium der Gründlichkeit in zentralen Punkten. Im Fokus stehen die logiktragenden Module wie **gedanken_reflexion_logik**, **zielverwaltung_logik**, **utility**, **inspirations_logik** sowie **tagebuch_logik**, die jeweils eine hohe Instruktionsabdeckung von über 78 % bis zu 87 % aufweisen. Auch die Module **routinen_logik**, **uebungen** und **stimmungskalender_logik** sind durch gezielte Tests abgesichert. Obwohl deren Abdeckung im Vergleich etwas geringer ist – etwa 34 % bis 42 % – wurden alle geschäftskritischen Funktionen wie das Speichern, Bearbeiten und Auswerten von Einträgen, Routinen oder Übungen durch aussagekräftige Unit-Tests abgedeckt.

Besonders hervorzuheben ist das Modul **stimmungskalender_logik**, das trotz seiner relativ komplexen Datenstrukturen – etwa Emotionseinträge mit Intensitäten, Ursachen und Zeitstempeln – sorgfältig getestet wurde. Die zugehörigen Tests decken nicht nur typische Nutzungsszenarien ab, sondern auch Randfälle wie unvollständige Eingaben oder fehlerhafte Dateien, wodurch potenzielle Fehlverhalten frühzeitig erkannt und abgefangen werden können.

Nicht alle Module sind gleich stark abgedeckt – beispielsweise **menus**, **fortschrittsbericht_logik** und **default** weisen derzeit noch keine getesteten Pfade auf. Diese Komponenten sind entweder UI-nah, relativ neu oder enthalten konfigurationsnahe Logik. Die Teststrategie folgt damit dem Prinzip der **gezielten Priorisierung**, indem besonders fehleranfällige und fachlich relevante Bereiche priorisiert abgesichert wurden, während unterstützende oder weniger kritische Teile in einer späteren Testausbaustufe folgen können.

Professional

Der Testcode des Projekts erfüllt in weiten Teilen professionelle Standards und

orientiert sich klar an den Prinzipien guter Softwareentwicklung. Die Struktur der Tests folgt der Aufbauweise des Produktivcodes: Jede fachliche Domäne besitzt ein eigenes dediziertes Testpaket (z. B. gedanken_reflexion_logik_tests, zielverwaltung_logik_tests, stimmungskalender_logik_tests), wodurch sich eine klare Trennung und Zuordnung zwischen Anwendung und Testlogik ergibt.

Die Benennung der Testklassen entspricht konsistent dem Muster KlassennameTest, was die Nachvollziehbarkeit und Wartbarkeit deutlich erhöht. Auch innerhalb der Testmethoden werden verständliche Namen verwendet, die den geprüften Sachverhalt beschreiben (z. B. testZielHinzufuegenMitKategorie, testEintragLoeschenMitFehlendemPfad). Durch gezielten Einsatz von Mocks – etwa für Benutzerinteraktion über Scanner – wird sichergestellt, dass die Tests reproduzierbar, isoliert und unabhängig von tatsächlichen I/O-Operationen bleiben.

Auf künstliche Testfälle oder das bloße Erhöhen der Testanzahl wurde bewusst verzichtet. Stattdessen liegt der Fokus auf den tatsächlich relevanten und logiktragenden Klassen. Unkritische Module wie die menus oder reine Datenklassen werden nachvollziehbar nicht mit Unit-Tests abgedeckt, was dem Grundsatz folgt, dass Testcode kein Selbstzweck sein darf.

Die Verständlichkeit des Testcodes ist durchweg gegeben: Komplexe Testlogik wurde vermieden, stattdessen liegt der Schwerpunkt auf klaren, lesbaren Assertions und strukturierter Vorbereitung der Testdaten. So bleibt der Testcode auch für Dritte gut nachvollziehbar – ein zentrales Qualitätsmerkmal professioneller Softwaretests.

5.3 Fakes und Mocks (4P)

[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten (die Fakes und Mocks sind ohne Dritthersteller-Bibliothek/Framework zu implementieren); zusätzlich jeweils UML Diagramm mit Beziehungen zwischen Mock, zu mockender Klasse und Aufrufer des Mocks]

Im Projekt wurden an mehreren Stellen gezielt selbst implementierte Fakes und Mocks eingesetzt, um Abhängigkeiten zu entkoppeln und einzelne Komponenten isoliert testen zu können. Zwei exemplarische Lösungen werden im Folgenden dargestellt und jeweils durch eine einfache UML-Beziehung ergänzt.

Beispiel 1: Fake für ZielRepository

Verwendung: In der Testklasse ZielVerwaltungTest wird ein eigens erstellter Fake

verwendet, der das Interface ZielRepository implementiert. Anstelle von Dateizugriffen nutzt dieser Fake eine List<Ziel>, um Ziele im Speicher zu halten.

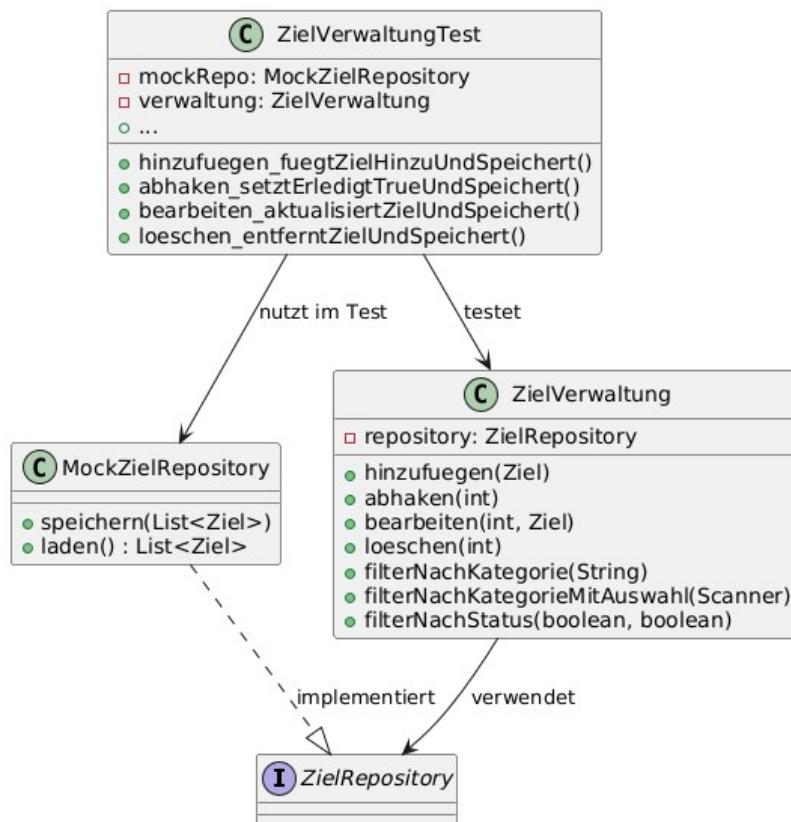
Zweck: Er ersetzt die Persistenzschicht, sodass die Logik von ZielVerwaltung getestet werden kann, ohne auf Dateisystem oder externe Abhängigkeiten angewiesen zu sein.

Nutzen: Ermöglicht schnelle, deterministische und wiederholbare Tests auf Methoden wie Zielerstellung, Kategorienfilterung und Fortschrittsstatistiken.

```
static class MockZielRepository implements ZielRepository { 2
    public boolean speichernAufgerufen = false; 5 usages
    List<Ziel> gespeicherteZiele = new ArrayList<>(); 1 usage

    @Override 1 katty.terra
    public void speichern(List<Ziel> ziele) {
        speichernAufgerufen = true;
        gespeicherteZiele = new ArrayList<>(ziele);
    }

    @Override 1 katty.terra
    public List<Ziel> laden() {
        return new ArrayList<>();
    }
}
```



Beispiel 2: Mock für TagebuchRepository

Verwendung: In TagebuchVerwaltungTest wird ein Mock-Objekt des Interfaces TagebuchRepository verwendet, z.B. als MockTagebuchRepository. Diese Klasse speichert und liefert Tagebucheinträge intern aus einer Liste zurück, ohne Dateien zu lesen oder zu schreiben.

Zweck: Der Mock kontrolliert gezielt das Verhalten der Speicherkomponente, etwa um zu testen, ob neue Einträge korrekt übergeben und gespeichert werden oder wie sich die Verwaltung bei doppelten bzw. fehlenden Einträgen verhält.

Nutzen: Tests sind dadurch vollständig unabhängig vom Dateisystem und fokussieren ausschließlich auf die Logik innerhalb von TagebuchVerwaltung.

```
static class MockTagebuchRepository implements TagebuchRepository { 2 usages ▲ katty.terra
    public TagebuchEintrag gespeicherterEintrag; 3 usages
    public String zuletztGelesenesDatum; 2 usages
    public String geloeschtesDatum; 4 usages
    public String geloeschteUhrzeit; 3 usages
    public String bearbeitetesDatum; 2 usages
    public String bearbeiteteUhrzeit; 2 usages
    public String bearbeiteterText; 2 usages
    public boolean bearbeitenErgebnis = true; 1 usage

    @Override ▲ katty.terra
    public void speichern(TagebuchEintrag eintrag) { this.gespeicherterEintrag = eintrag; }

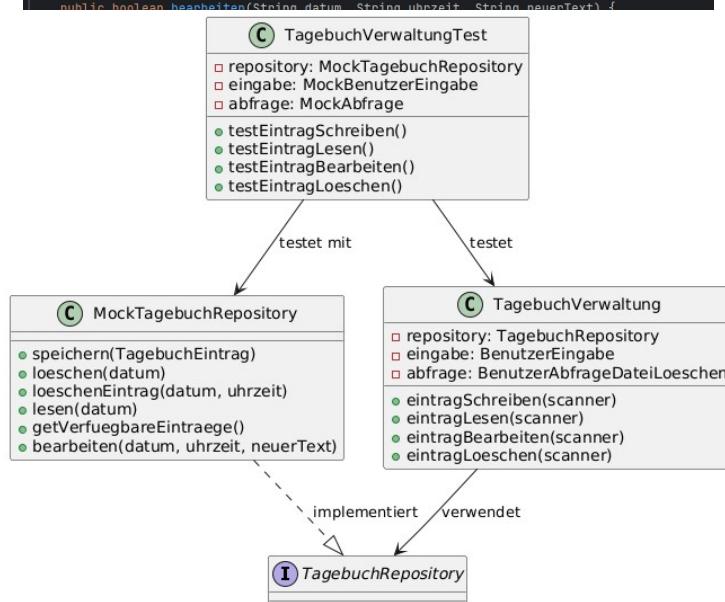
    @Override ▲ katty.terra
    public void loeschen(String datum) { this.geloeschtesDatum = datum; }

    @Override 3 usages ▲ katty.terra
    public void loeschenEintrag(String datum, String uhrzeit) {
        this.geloeschtesDatum = datum;
        this.geloeschteUhrzeit = uhrzeit;
    }

    @Override ▲ katty.terra
    public String lesen(String datum) {
        this.zuletztGelesenesDatum = datum;
        return "Test-Inhalt";
    }

    @Override 4 usages ▲ katty.terra
    public List<String> getVerfuegbareEintraege() { return List.of( e1: "2025-04-18"); }

    @Override ▲ katty.terra
    public boolean bearbeiten(String datum, String uhrzeit, String neuerText) {
```



6. Kapitel 6: Domain Driven Design (8P)

6.1 Ubiquitous Language (2P)

[4 Beispiele für die Ubiquitous Language; jeweils Bezeichnung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]

Bezeichnung	Bedeutung	Begründung
Stimmungseintrag	Repräsentiert die Tagesstimmung eines Benutzers, inklusive Bewertung und Beschreibung.	Der Begriff beschreibt klar ein zentrales Konzept der Domäne – nämlich das Erfassen und Reflektieren der eigenen Tagesstimmung. Er ist sprachlich eindeutig und für Fachanwender wie auch Nutzer unmittelbar verständlich.
Routine	Beschreibt eine wiederkehrende Aufgabe (z. B. „Wasser trinken“), die der Nutzer abhaken kann.	„Routine“ ist ein etabliertes Wort aus dem Alltag und drückt exakt aus, worum es geht: wiederkehrende, gesundheitsförderliche Handlungen. Der Begriff passt intuitiv zur Domäne der Selbstfürsorge und Gesundheitsförderung.
Ziel	Ein persönliches Ziel des Nutzers, das kategorisiert, priorisiert und mit Fälligkeitsdatum versehen werden kann.	Das Konzept „Ziel“ ist grundlegend für persönliche Weiterentwicklung. Es ist fachlich klar definiert und stellt einen festen Bestandteil

		psychologischer und therapeutischer Arbeit dar – und wird deshalb als Begriff direkt übernommen.
Gedankenreflexion	Ein Modul zur schriftlichen Auseinandersetzung mit belastenden Gedanken (Gedankenkarussell stoppen).	Dieser Ausdruck beschreibt einen psychologischen Prozess, bei dem belastende Gedanken beobachtet, hinterfragt und verarbeitet werden. Er ist präzise, fachlich korrekt und unterstützt eine klare Kommunikation zwischen allen Beteiligten.

6.2 Repositories (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde]

Das Interface TagebuchRepository definiert eine abstrakte Schnittstelle für alle Speicher- und Verwaltungsoperationen von Tagebucheinträgen – wie speichern, lesen, bearbeiten oder löschen. Es abstrahiert den Zugriff auf die Speichertechnologie (z. B. Dateisystem), sodass die Logik der Anwendung nicht direkt von der konkreten Speicherung abhängig ist.

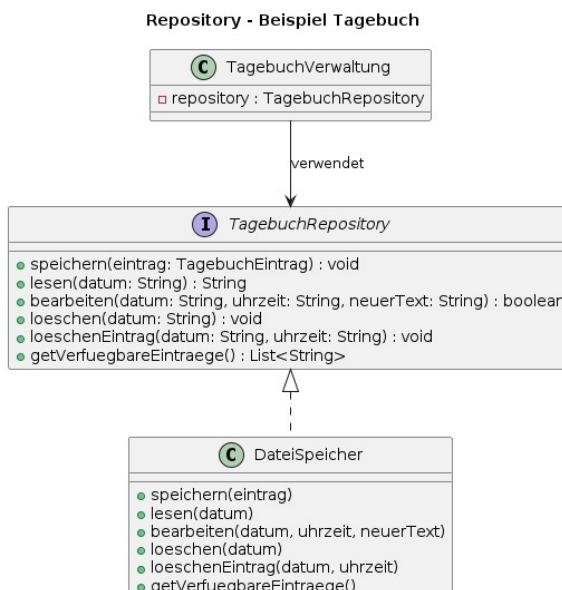
Die konkrete Implementierung DateiSpeicher übernimmt die technische Aufgabe der Dateioperationen und erfüllt somit die Anforderungen der Persistenz – ohne dass die Domänenlogik direkt mit dem Dateisystem arbeiten muss.

Das TagebuchRepository erfüllt auch die Kriterien eines klassischen Domain-Driven-Design-Repositories:

- Es arbeitet mit einem klar abgegrenzten **Aggregate (TagebuchEintrag)**

zusammen.

- Es gibt für dieses Aggregate genau **ein zugehöriges Repository**.
- Die Rückgabe erfolgt über die **Aggregate Root**, wodurch die Nutzung des gesamten Aggregats ermöglicht wird.
- Die **Definition** des Repositories liegt im **Domänenbereich**, während die **Implementierung** (z. B. DateiSpeicher) außerhalb, im technischen Teil, liegt.



6.3 Aggregates (1,5P)

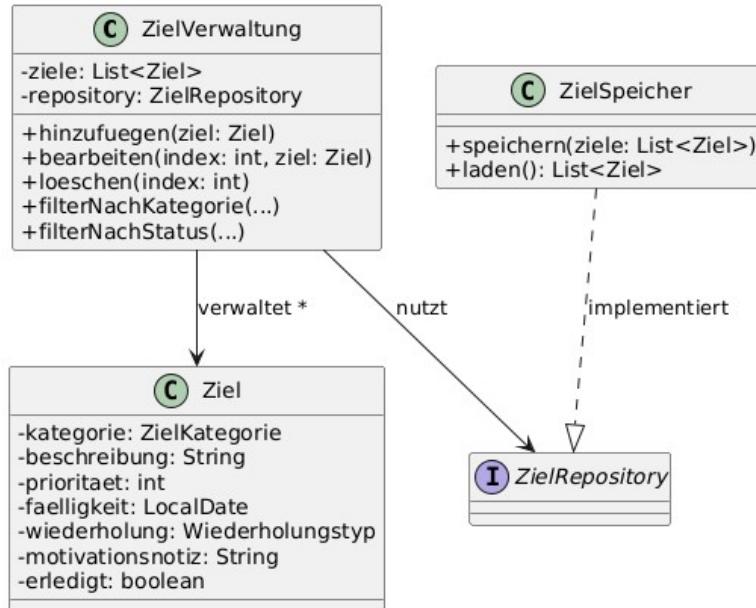
[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]

Im Modul zielverwaltung_logik ist mit der Klasse ZielVerwaltung ein typisches Aggregat im Sinne von Domain-Driven Design umgesetzt. Sie fungiert als **Aggregate Root** und verwaltet eine Liste von Ziel-Entitäten, die jeweils eine eigene Identität und Lebensdauer besitzen. Alle CRUD-Operationen – wie Hinzufügen, Bearbeiten, Löschen und Filtern – erfolgen ausschließlich über die Methoden der ZielVerwaltung, wodurch eine klare **Transaktionsgrenze** geschaffen wird.

Die Ziele werden **nie direkt von außen verändert**, sondern über die Root-Klasse kontrolliert. So wird die **fachliche Konsistenz** gewahrt, etwa bei Statusänderungen

oder Kategoriefiltern. Zudem wird bei jeder Änderung die gesamte Zielstruktur über das Repository gespeichert, was der Regel entspricht, dass ein Aggregat stets als Einheit geladen und persistiert wird.

Durch diese Struktur kapselt ZielVerwaltung nicht nur die Daten, sondern auch alle Domänenregeln der Zielverwaltung – und erfüllt damit die Rolle eines Aggregats im Sinne von DDD.



6.4 Entities (1,5P)

[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keine geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde]

Im Modul zielverwaltung_logik ist die Klasse Ziel ein typisches Beispiel für eine **Entity** im Sinne des Domain-Driven Design. Eine Entity zeichnet sich dadurch aus, dass sie eine **dauerhafte Identität** besitzt, die über ihre Lebenszeit hinweg erhalten bleibt – auch wenn sich ihre Eigenschaften ändern.

Das trifft auf Ziel klar zu: Ein Ziel hat Eigenschaften wie Beschreibung, Priorität, Fälligkeitsdatum, Kategorie, Wiederholungstyp und Erledigungsstatus, die sich im Lauf der Nutzung verändern können. Trotzdem bleibt es als **individuelles Ziel** in der Anwendung stets eindeutig identifizierbar. Diese dauerhafte fachliche Identität unterscheidet es z. B. von einem bloßen Wertobjekt.

Die Klasse Ziel wird in einer Liste innerhalb des Aggregats ZielVerwaltung verwaltet, bearbeitet und gespeichert. Ihre Lebensdauer reicht von der Erstellung bis zur möglichen Löschung durch den Benutzer – damit erfüllt sie alle Kriterien für eine eigenständige Entität in der Domäne.



6.5 Value Objects (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde]

Ein Value Object ist ein objektorientiertes Konzept, das keine eigene Identität besitzt und ausschließlich über seine Attributwerte definiert ist. Es dient der Modellierung von Eigenschaften, die inhaltlich bedeutsam sind, aber für sich genommen keine Lebensdauer oder Nachverfolgbarkeit erfordern.

Im vorliegenden Projekt erfüllt die Klasse ZielKategorie diese Eigenschaften. Sie wird als Enumeration (Enum) verwendet, um Ziele thematisch zu klassifizieren (z. B. GESUNDHEIT, BERUF, FREIZEIT). ZielKategorie besitzt keine individuelle Identität, ist unveränderlich und wird ausschließlich als wertbasierte Eigenschaft innerhalb der Entität Ziel verwendet.



7. Kapitel 7: Refactoring (8P)

7.1 Code Smells (2P)

[jeweils 1 Code-Beispiel zu 2 unterschiedlichen Code Smells (die benannt werden müssen) aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommenen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]

(Beide wurden im Commit vom 01.05.2025 entfernt)

Code Smell 1: Long Method

Beispiel:

In der Methode eintragBearbeiten(Scanner scanner) aus TagebuchVerwaltung befinden sich mehrere Aufgaben: Benutzereingaben einlesen, vorhandene Einträge laden, Fehler prüfen, bearbeiten und Rückmeldung geben. Die Methode ist dadurch relativ lang und übernimmt zu viele Verantwortlichkeiten auf einmal.

```
public void eintragBearbeiten(Scanner scanner) {  
    // 1. Datum wählen  
    String datum = waehleDatum(scanner);  
    // 2. Uhrzeit eingeben  
    System.out.println("Uhrzeit:");  
    String uhrzeit = scanner.nextLine();  
    // 3. Text eingeben  
    System.out.println("Neuer Text:");  
    String neuerText = scanner.nextLine();  
    // 4. Bearbeiten  
    boolean erfolg = repository.bearbeiten(datum, uhrzeit, neuerText);  
    // 5. Rückmeldung  
    if (erfolg) {  
        System.out.println("Eintrag bearbeitet.");  
    } else {  
        System.out.println("Bearbeiten fehlgeschlagen.");  
    }  
}
```

Lösungsweg:

Zerlegung in kleinere Methoden mit je einer klaren Aufgabe. Dadurch wird die Lesbarkeit und Testbarkeit verbessert.

Refactored (Pseudo-)Code:

```
public void eintragBearbeiten(Scanner scanner) {  
    String datum = waehleDatum(scanner);  
    String uhrzeit = leseUhrzeit(scanner);  
    String text = leseText(scanner);  
    bearbeiteEintrag(datum, uhrzeit, text);  
}  
  
private String leseUhrzeit(Scanner scanner) { ... }  
private String leseText(Scanner scanner) { ... }  
private void bearbeiteEintrag(String datum, String uhrzeit, String text) { ... }
```

Code Smell 2: Shotgun Surgery

Beispiel:

In der Klasse ZielVerwaltung wird innerhalb mehrerer Methoden wie hinzufuegen(), bearbeiten(), loeschen() und abhaken() jeweils direkt repository.speichern(ziele) aufgerufen. Jede Änderung an der internen Liste der Ziele führt somit zu einem separaten Speichervorgang, der direkt an Ort und Stelle implementiert ist.

Problem:

Sollte sich das Speicherverhalten (z. B. durch zusätzliches Logging, Validierung oder Transaktionsmanagement) ändern, müssten alle Methoden der Klasse einzeln angepasst werden. Dies führt zu sogenannter **Shotgun Surgery**: Eine kleine Änderung hat Auswirkungen an vielen Stellen im Code, was Wartung und Weiterentwicklung erschwert.

Lösungsweg:

Zur Vermeidung mehrfacher identischer Speicheraufrufe wurde eine zentrale Methode aktualisiereUndSpeichere() eingeführt. Diese übernimmt die Ausführung der konkreten Änderung über ein Lambda-Argument und ruft anschließend zentral die Speicherfunktion auf.

```
private void aktualisiereUndSpeichere(Runnable änderung) {
    änderung.run();
    repository.speichern(ziele);
}

public void loeschen(int index) {
    aktualisiereUndSpeichere(() -> ziele.remove(index));
}
```

7.2 2 Refactorings (6P)

[2 unterschiedliche Refactorings aus der Vorlesung jeweils benennen, anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen – die Refactorings dürfen sich nicht mit den Beispielen der Code überschneiden]

Refactoring 1: Extract Method

Ort: ZielVerwaltung.filterNachKategorieMitAuswahl(...)

Problem: Die Methode enthält sowohl Benutzerdialog als auch Logik zur Filterung → vermischt Zuständigkeiten.

Lösung: Auslagerung der Benutzeroberfläche in eine eigene Methode `waehleKategorie(...)`.

Vorher:

```
public List<Ziel> filterNachKategorieMitAuswahl(Scanner scanner) {
    ZielKategorie[] kategorien = ZielKategorie.values();
    System.out.println("\nWähle eine Zielkategorie:");
    for (int i = 0; i < kategorien.length; i++) {
        System.out.println((i + 1) + " - " + kategorien[i]);
    }
    System.out.print("Auswahl: ");
    try {
        int katIndex = Integer.parseInt(scanner.nextLine()) - 1;
        ZielKategorie gewaehlteKategorie = kategorien[Math.max(0, Math.min(katIndex, kategorien.length - 1))];
        return filterNachKategorie(gewaehlteKategorie.name());
    } catch (Exception e) {
        System.out.println("⚠ Ungültige Kategorieauswahl.");
        return List.of();
    }
}
```

Nachher:

```
public List<Ziel> filterNachKategorieMitAuswahl(Scanner scanner) { 3 usages ± kattyterra *
    ZielKategorie gewaehlteKategorie = waehleKategorie(scanner);
    List<Ziel> gefiltert = filterNachKategorie(gewaehlteKategorie.name());
    if (gefiltert.isEmpty()) {
        System.out.println("⚠ Keine Ziele in Kategorie \"" + gewaehlteKategorie + "\" gefunden.");
    }
    return gefiltert;
}

private ZielKategorie waehleKategorie(Scanner scanner) { 1 usage new *
    ZielKategorie[] kategorien = ZielKategorie.values();
    System.out.println("\nWähle eine Zielkategorie:");
    for (int i = 0; i < kategorien.length; i++) {
        System.out.println((i + 1) + " - " + kategorien[i]);
    }
    System.out.print("Auswahl: ");
    try {
        int index = Integer.parseInt(scanner.nextLine()) - 1;
        return kategorien[Math.max(0, Math.min(index, kategorien.length - 1))];
    } catch (Exception e) {
        System.out.println("⚠ Ungültige Kategorieauswahl. Standard: GESUNDHEIT.");
        return ZielKategorie.GESUNDHEIT;
    }
}
```

Begründung: Durch die Auslagerung wird die Hauptmethode kürzer, leichter testbar und folgt dem SRP-Prinzip. Die neue Methode kann auch an anderen Stellen wiederverwendet werden (z. B. beim Erstellen neuer Ziele).

Commit vom 01.05.2025

Refactoring 2: Replace Temp with Query

Ort: filterNachKategorie(...)

Problem: Temporäre Variable filterKategorie speichert nur ein Zwischenresultat eines Methodenaufrufs.

Lösung: Direktes Einsetzen der valueOf(...)-Abfrage im Stream-Filter (sofern keine Mehrfachnutzung nötig ist).

Vorher:

```
public List<Ziel> filterNachKategorie(String kategorie) {  
    try {  
        ZielKategorie filterKategorie = ZielKategorie.valueOf(kategorie.toUpperCase());  
        return ziele.stream()  
            .filter(z -> z.getKategorie() == filterKategorie)  
            .toList();  
    } catch (IllegalArgumentException e) {  
        System.out.println("⚠️ Ungültige Kategorie: " + kategorie);  
        return List.of();  
    }  
}
```

Nachher:

```
public List<Ziel> filterNachKategorie(String kategorie) { 3 usages ↗ kattyterra *  
    try {  
        return ziele.stream()  
            .filter( Ziel z -> z.getKategorie() == ZielKategorie.valueOf(kategorie.toUpperCase()))  
            .toList();  
    } catch (IllegalArgumentException e) {  
        System.out.println("⚠️ Ungültige Kategorie: " + kategorie);  
        return List.of();  
    }  
}
```

Begründung: Die temporäre Variable war nur ein einmal verwendetes Zwischenresultat. Das direkte Einfügen erhöht die Lesbarkeit und reduziert unnötige lokale Zustände.

Commit vom 01.05.2025

8. Kapitel 8: Entwurfsmuster (8P)

[2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils benennen, sinnvoll einsetzen, begründen und UML-Diagramm]

Entwurfsmuster: Strategy (4P)

Mögliche Verwendung im Projekt:

Im Modul zielverwaltung_logik kann das Strategy-Muster verwendet werden, um die Filterlogik für Ziele flexibel zu gestalten. Statt mehrere filterNachKategorie, filterNachStatus, filterNachWiederholung einzeln zu implementieren, lassen sich unterschiedliche Filterkriterien als austauschbare Strategieobjekte kapseln.

Begründung:

Die Filterlogik ist zur Laufzeit variabel (z. B. Kategorieauswahl durch Benutzer), und neue Filterarten (z. B. nach Fälligkeit) lassen sich ohne Änderung der ZielVerwaltung hinzufügen. Das Strategy-Muster entkoppelt die Filterregeln von der Sammlung selbst.



Beispiel, wie es in diesem Projekt aussehen könnte:

- Interface: ZielFilterStrategy
 - ```
public interface ZielFilterStrategy {
 List<Ziel> filter(List<Ziel> ziele);
}
```
- Strategie 1: Kategorie-Filter
  - ```
public class KategorieFilter implements ZielFilterStrategy {
    private final ZielKategorie kategorie;
    public KategorieFilter(ZielKategorie kategorie) {
        this.kategorie = kategorie;
    }
    @Override
    public List<Ziel> filter(List<Ziel> ziele) {
        return ziele.stream()
            .filter(z -> z.getKategorie() == kategorie)
    }
}
```

```
        .toList();
    }
}

• Strategie 2: Status-Filter

○ public class StatusFilter implements ZielFilterStrategy {
    private final boolean erledigt;
    public StatusFilter(boolean erledigt) {
        this.erledigt = erledigt;
    }
    @Override
    public List<Ziel> filter(List<Ziel> ziele) {
        return ziele.stream()
            .filter(z -> z.isErledigt() == erledigt)
            .toList();
    }
}
```

- Verwendung in ZielVerwaltung

```
○ public class ZielVerwaltung {
    private final List<Ziel> ziele = new ArrayList<>();
    public List<Ziel> filter(ZielFilterStrategy strategie) {
        return strategie.filter(ziele);
    }
}
```

Entwurfsmuster: Builder (4P)

Mögliche Verwendung im Projekt:

Beim Erstellen von Ziel-Objekten mit vielen optionalen Attributen (z. B. Motivationsnotiz, Fälligkeit, Wiederholungstyp) bietet sich ein **Builder** an. Dieser ermöglicht ein flexibles und lesbares Erstellen komplexer Objekte – besonders hilfreich in Tests oder bei Benutzerinteraktion.

Begründung:

Der Konstruktor von Ziel wird bei vielen Parametern schnell unübersichtlich. Der Builder macht die Initialisierung nachvollziehbar und unterstützt method chaining. So entsteht ein klarer, wartbarer Code für die Erstellung von Entitäten mit optionalen Attributen.



Beispiel, wie es in diesem Projekt aussehen könnte:



```
public class ZielBuilder {
    private ZielKategorie kategorie;
    private String beschreibung;
    private int prioritaet;
    private LocalDate faelligkeit;
    private Wiederholungstyp wiederholung;
    private String motivationsnotiz;

    public ZielBuilder mitKategorie(ZielKategorie kategorie) {
        this.kategorie = kategorie;
        return this;
    }

    public ZielBuilder mitBeschreibung(String beschreibung) {
        this.beschreibung = beschreibung;
        return this;
    }

    public ZielBuilder mitPrioritaet(int prioritaet) {
        this.prioritaet = prioritaet;
        return this;
    }

    public ZielBuilder mitFaelligkeit(LocalDate faelligkeit) {
        this.faelligkeit = faelligkeit;
        return this;
    }

    public ZielBuilder mitWiederholung(Wiederholungstyp typ) {
        this.wiederholung = typ;
        return this;
    }

    public ZielBuilder mitMotivationsnotiz(String notiz) {
        this.motivationsnotiz = notiz;
        return this;
    }

    public Ziel build() {
        Ziel ziel = new Ziel(kategorie, beschreibung, prioritaet, faelligkeit, wiederholung);
        ziel.setMotivationsnotiz(motivationsnotiz);
        return ziel;
    }
}

Ziel ziel = new ZielBuilder()
    .mitKategorie(ZielKategorie.GESUNDHEIT)
    .mitBeschreibung("Mehr trinken")
    .mitPrioritaet(2)
    .mitFaelligkeit(LocalDate.now().plusDays(5))
    .mitWiederholung(Wiederholungstyp.TAEGLICH)
    .mitMotivationsnotiz("Hydration ist wichtig")
    .build();
```