



# Programmentwurf: Mental Health App

---

Ekaterina Sorokovaia, TINF22B5, 9266835



# Agenda

---

- Kapitel 1: Einführung
  - Kapitel 2: Softwarearchitektur
  - Kapitel 3: SOLID
  - Kapitel 4: Weitere Prinzipien
  - Kapitel 5: Unit Tests
  - Kapitel 6: Domain Driven Design
  - Kapitel 7: Refactoring
  - Kapitel 8: Entwurfsmuster
- 

# Kapitel 1 - Einführung

# Übersicht über die Appikation

- Tagebuch
- Stimmungskalender
- Routinenverwaltung
- Zielsetzung & Zielverfolgung
- Atemübungen
- Achtsamkeitsübungen
- Inspirationsecke
- Gedankenreflexion
- Monatlicher Fortschrittsbericht



**MentalHealthApp**

# Starten der Appikation



## Projekt starten

1. Projekt klonen:

```
git clone https://github.com/dein-benutzername/MentalHealthApp.git  
cd MentalHealthApp
```



2. Kompilieren:

```
javac App.java
```



3. Ausführen:

```
java App
```



Hinweis: Beim ersten Start werden automatisch Verzeichnisse und leere Dateien erstellt.

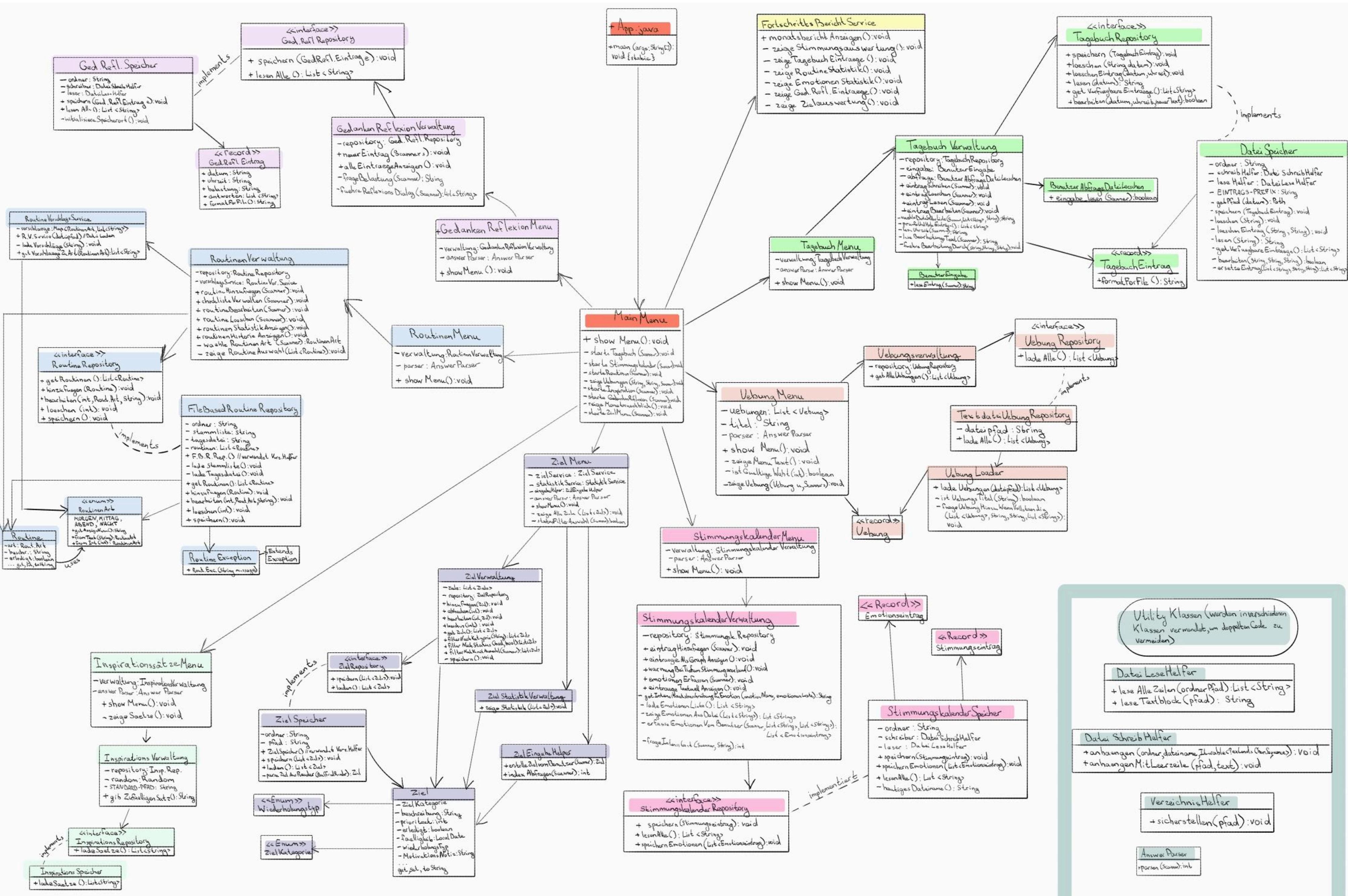
# Technischer Überblick

- **Sprache: Java**
  - Objektorientiert, plattformunabhängig, wartungsfreundlich
  - Gute Unterstützung für SOLID, GRASP & DDD
- **Benutzeroberfläche: Textbasierte Konsole**
  - Einfache Ein-/Ausgabe ohne grafische Oberfläche
  - Plattformunabhängig & schlank umgesetzt
- **Datenspeicherung: UTF-8 Textdateien**
  - Strukturiert, lesbar & nachvollziehbar
  - Speicherung aller Nutzerdaten wie Stimmungen, Ziele, Routinen etc.

# Kapitel 2 - Softwarearchitektur



Gewählte Architektur



# Domain Code

- Was ist Domain Code?

- Abbild der fachlichen Regeln & Abläufe der realen Welt
- Unabhängig von Technik (Datenbank, UI, Framework)
- Fokus auf „Was“ und „Warum“ – nicht auf „Wie“

- Beispiel: Klasse Routine

- Fachliche Repräsentation:

- Modelliert eine tägliche Gewohnheit zur mentalen Gesundheit

- Enthält Geschäftslogik:

- Kennt ihren Status („erledigt“ oder nicht)
  - Gibt sich selbst domänenspezifisch im `toString()` aus

- Technikunabhängig:

- Keine Datenbank, keine UI, kein Framework
  - Reiner Java-Code zur fachlichen Beschreibung der Routine

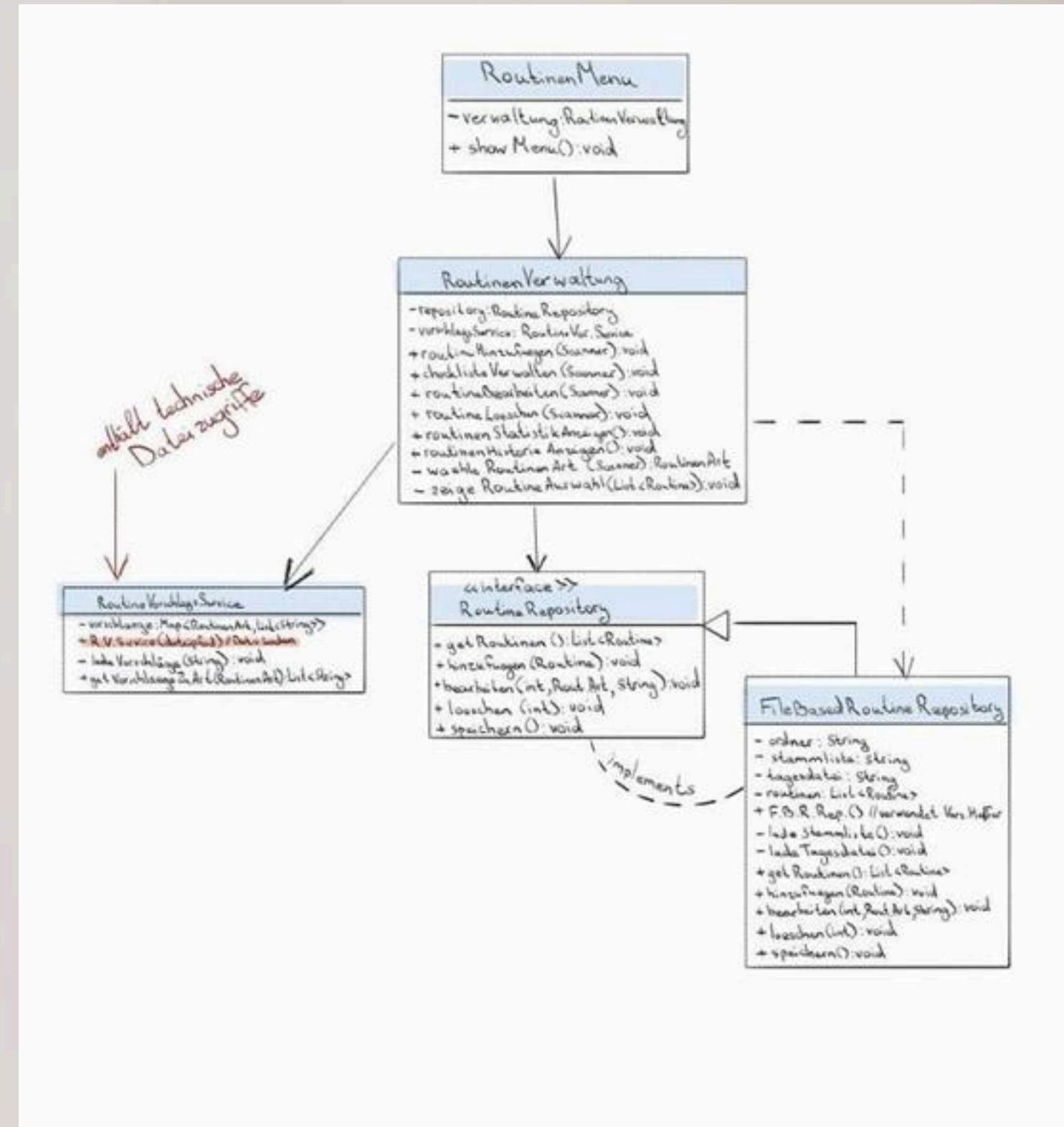
```
1 package routinen_logik;
2 /**
3  * Repräsentiert eine einzelne Routine in der MentalHealthApp.
4  */
5 public class Routine {
6
7     private RoutinenArt art;
8     private String beschreibung;
9     private boolean erledigt;
10
11    public Routine(RoutinenArt art, String beschreibung) {
12        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}
```

# Analyse der Dependency Rule

## Positiv-Beispiel:

RoutinenVerwaltung → RoutineRepository → FileBasedRoutineRepository

- Nur Abstraktion bekannt:  
RoutinenVerwaltung kennt nur das Interface RoutineRepository
- Keine Kopplung an Technik: Fachliche Logik ist unabhängig von konkreter Dateiimplementierung
- Austauschbar: Technische Details (z.B. Speicherung) lassen sich ändern, ohne Logik zu verändern



## Negativ-Beispiel:

RoutinenVerwaltung → RoutineVorschlagsService

- Direkte technische Kopplung:  
RoutinenVerwaltung nutzt konkrete Klasse mit Dateizugriff
- Verletzung der Trennung: Fachliche Logik hängt von Dateiformat und Pfadstruktur ab
- Weniger wartbar: Änderungen an der Vorschlagsquelle erfordern Änderungen an der Logik
- Lösung: Einführung eines Interfaces wie RoutineVorschlagsProvider zur Trennung von Logik und Technik



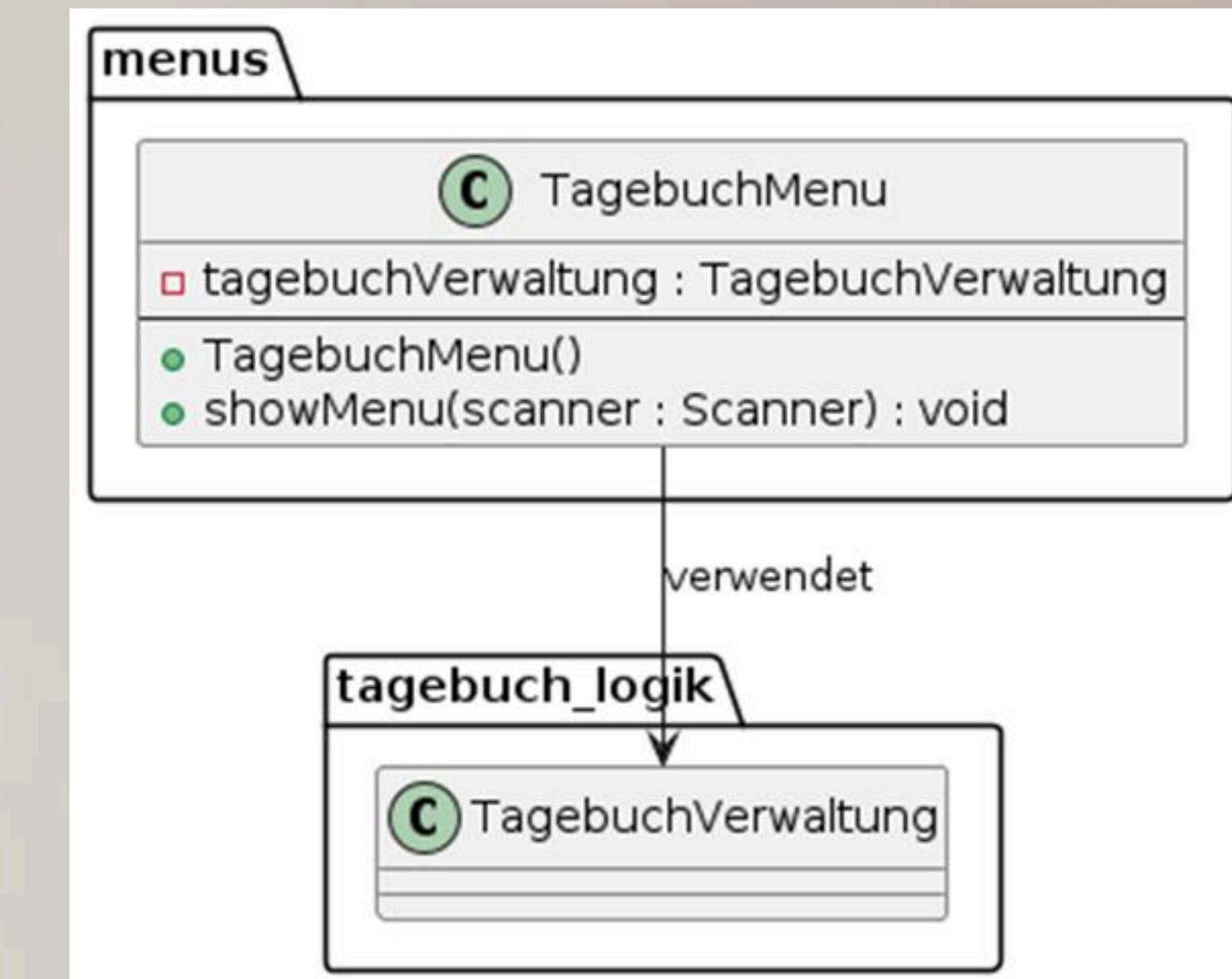
# Kapitel 3: - SΟΥΦΡΩΝ



# Analyse S<sup>T</sup>RIP

# Positiv-Beispiel

- **Klare Verantwortung:** Steuert ausschließlich die Benutzerführung im Tagebuchmodul
- **Trennung von Logik & Darstellung:**
  - Zeigt Menü
  - Liest Benutzereingaben
  - Leitet Aktionen an TagebuchVerwaltung weiter
- **Keine Fachlogik enthalten:** Kein Speichern, Bearbeiten oder Analysieren von Einträgen
- **Nur ein Änderungsgrund:** Anpassung der Menüführung



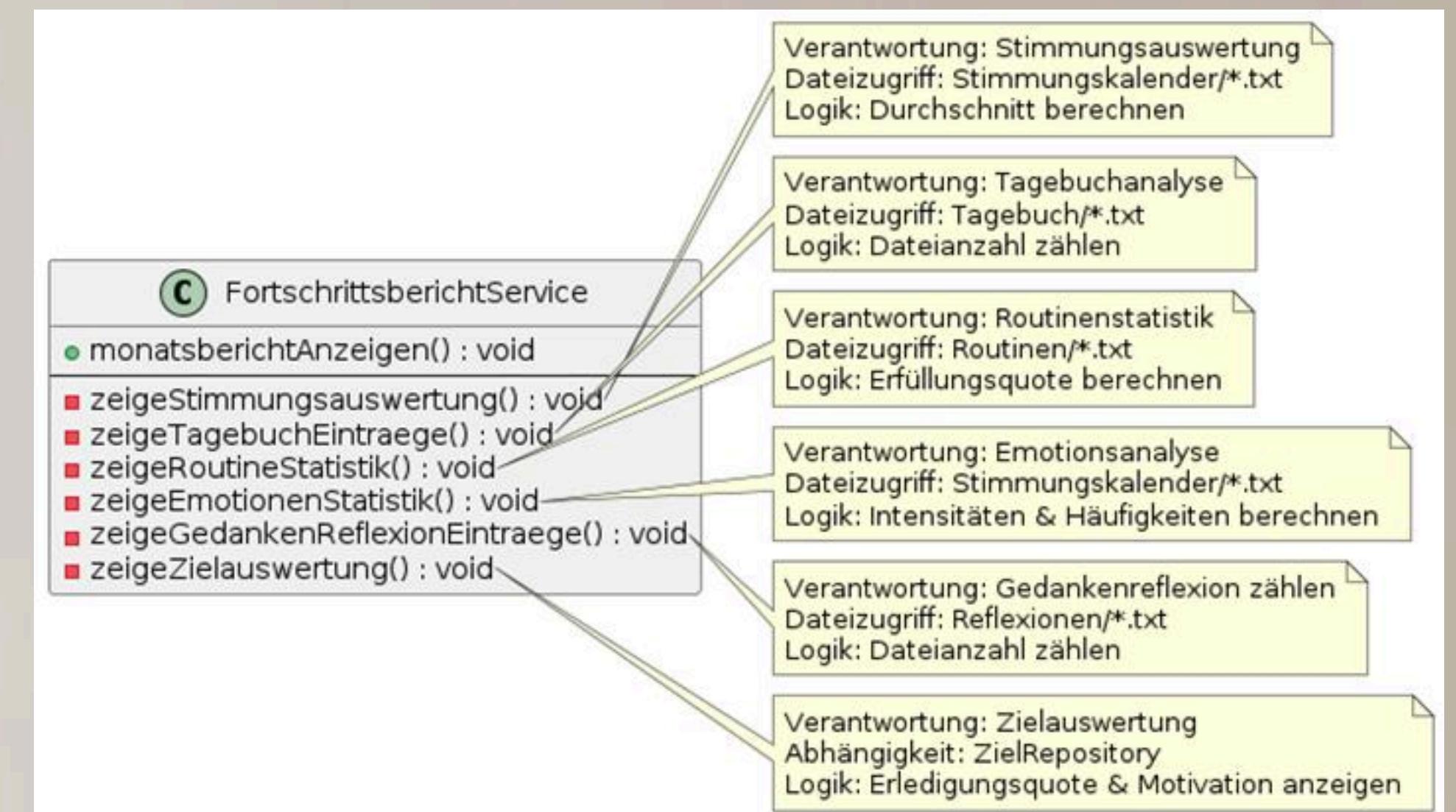
# Negativ-Beispie[

- **Mehrere Verantwortungen in einer Klasse:**

- Auswertung von Stimmung, Tagebuch, Routinen, Emotionen, Gedankenreflexion & Zielen
- Gleichzeitige Berichtserstellung und -ausgabe

- **Mehrere Gründe für Änderungen:**

- Änderungen an einem Modul (z.B. Emotionen) beeinflussen die gesamte Klasse
- Erhöhtes Risiko für Seiteneffekte & schwierige Wartung



# Analyse OCP

# Positiv-Beispiel

- **Abstraktion durch Interface:**

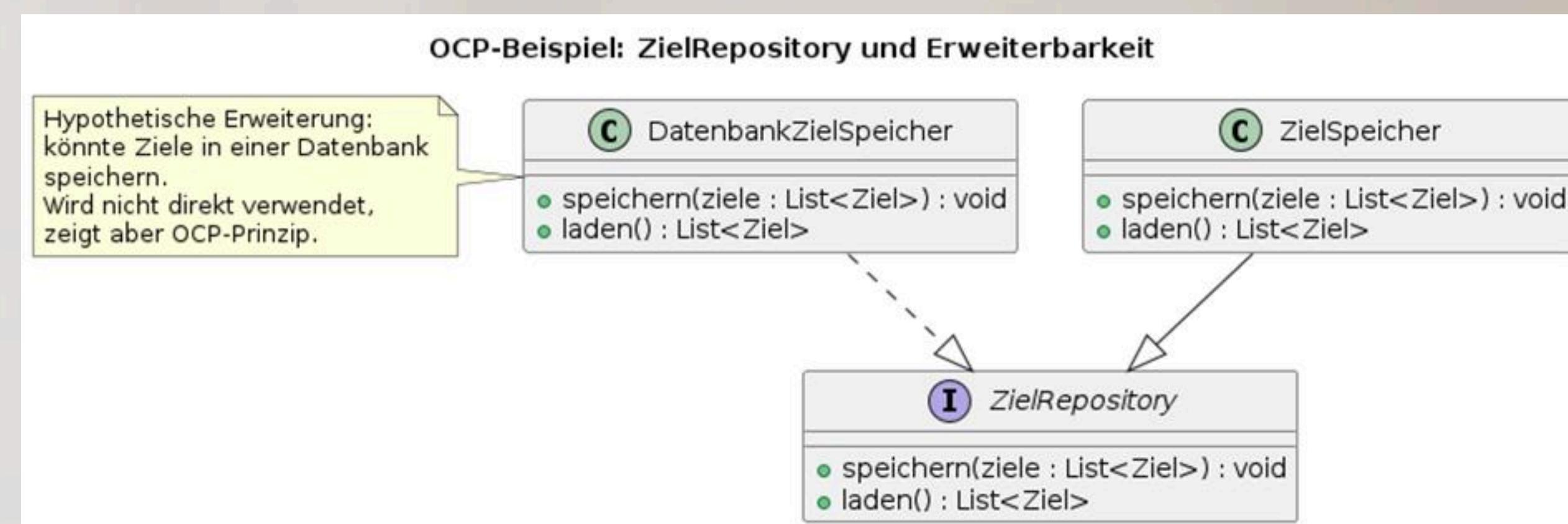
- ZielRepository definiert die Schnittstelle zum Speichern/Laden von Zielen
- Keine Festlegung auf konkrete Implementierung

- **Offen für Erweiterung:**

- Neue Varianten wie DatenbankZielSpeicher können hinzugefügt werden
- Ohne bestehende Klassen (z.B. ZielVerwaltung) zu ändern

- **Geschlossen für Modifikation:**

- ZielSpeicher und nutzende Klassen bleiben unangetastet
- Nur neue Klassen kommen hinzu – bestehender Code bleibt stabil



# Negativ-Beispiele

- Direkte Verletzung des Open/Closed Principle:

- Neue Menüoptionen erfordern Änderung des switch-Blocks
- Jede Funktionserweiterung → direkter Eingriff in MainMenu

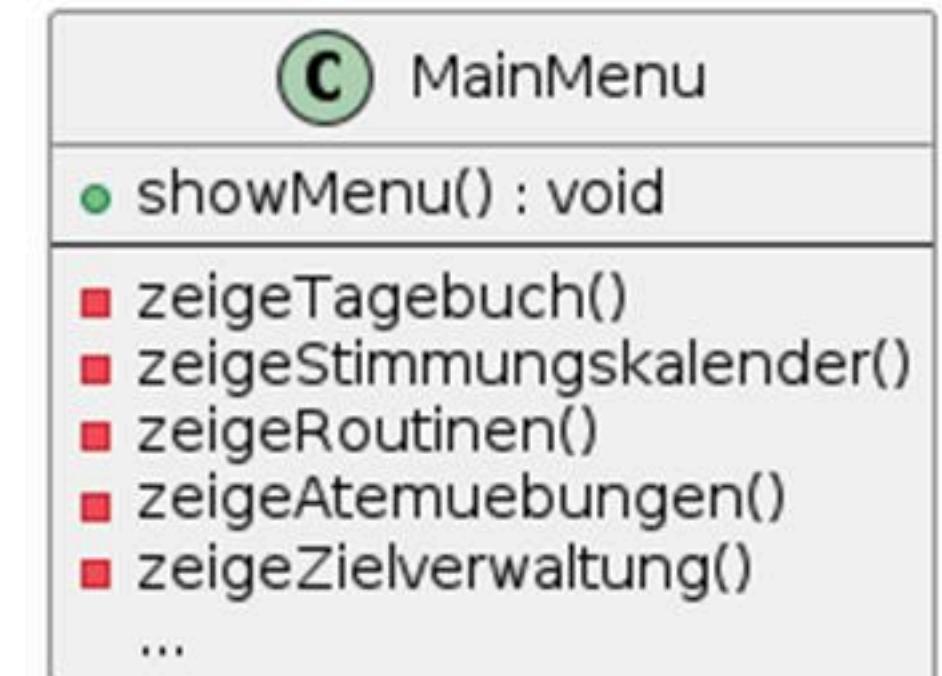
- Nicht offen für Erweiterung:

- Erweiterung = Änderung am Quellcode
- Risiko für Fehler & schlechtere Wartbarkeit steigt

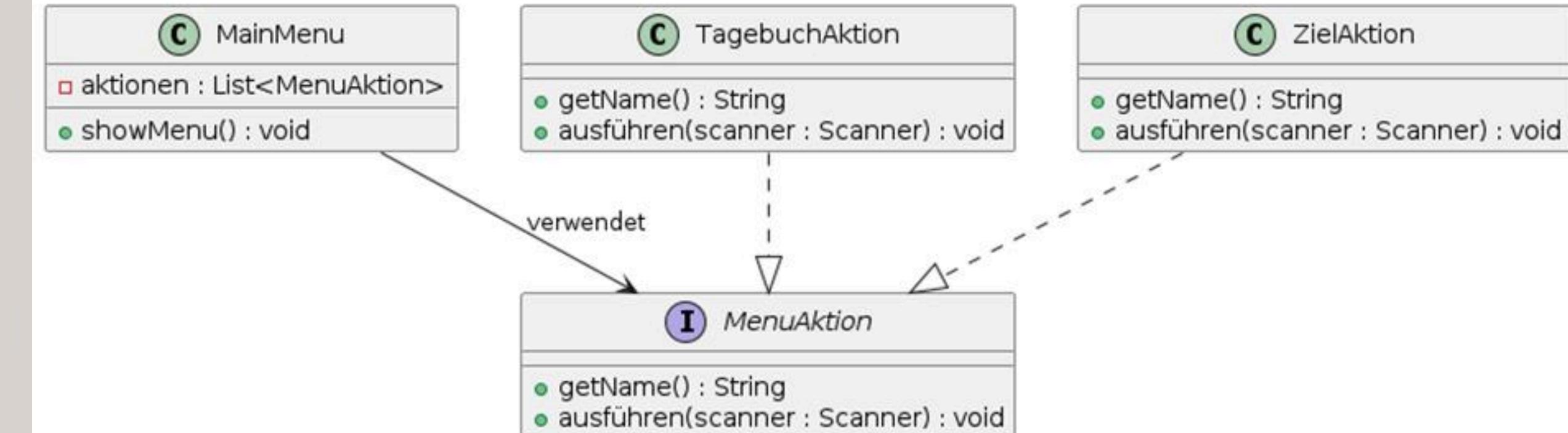
- Lösungsvorschlag (OCP-konform):

- Einführung eines Interface wie MenuAktion
- Menüeinträge als modulare Klassen (z. B. ZielMenuAktion, TagebuchMenuAktion)
- MainMenu iteriert nur über eine Liste von Aktionen – keine Änderungen nötig bei neuen Einträgen

**MainMenu -> nicht OCP-konform**



**OCP-konformes Menü**



# Analyse DIP

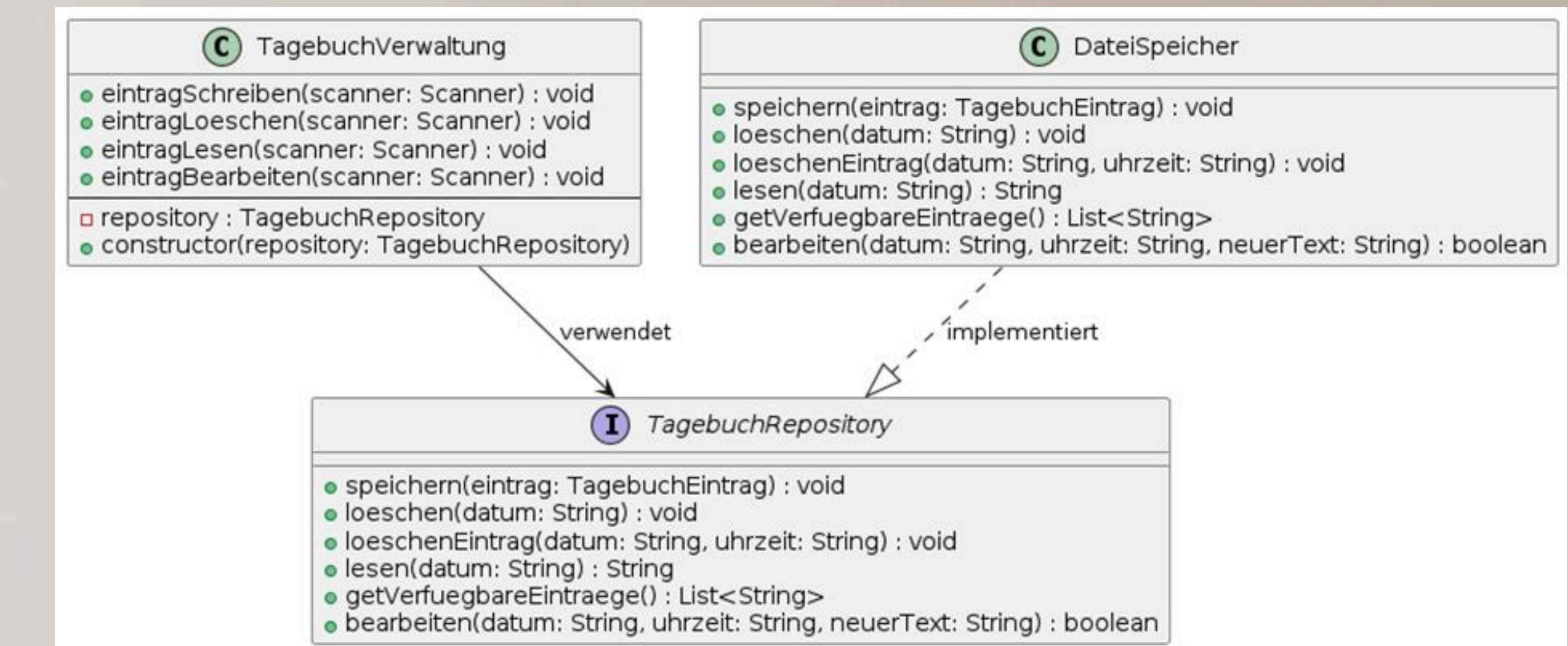
# Positiv-Beispiel

- **Abhangigkeit zur Abstraktion:**

- TagebuchVerwaltung nutzt das Interface TagebuchRepository
- Keine direkte Kopplung an DateiSpeicher oder andere konkrete Klassen

- **Injektion uber Konstruktor:**

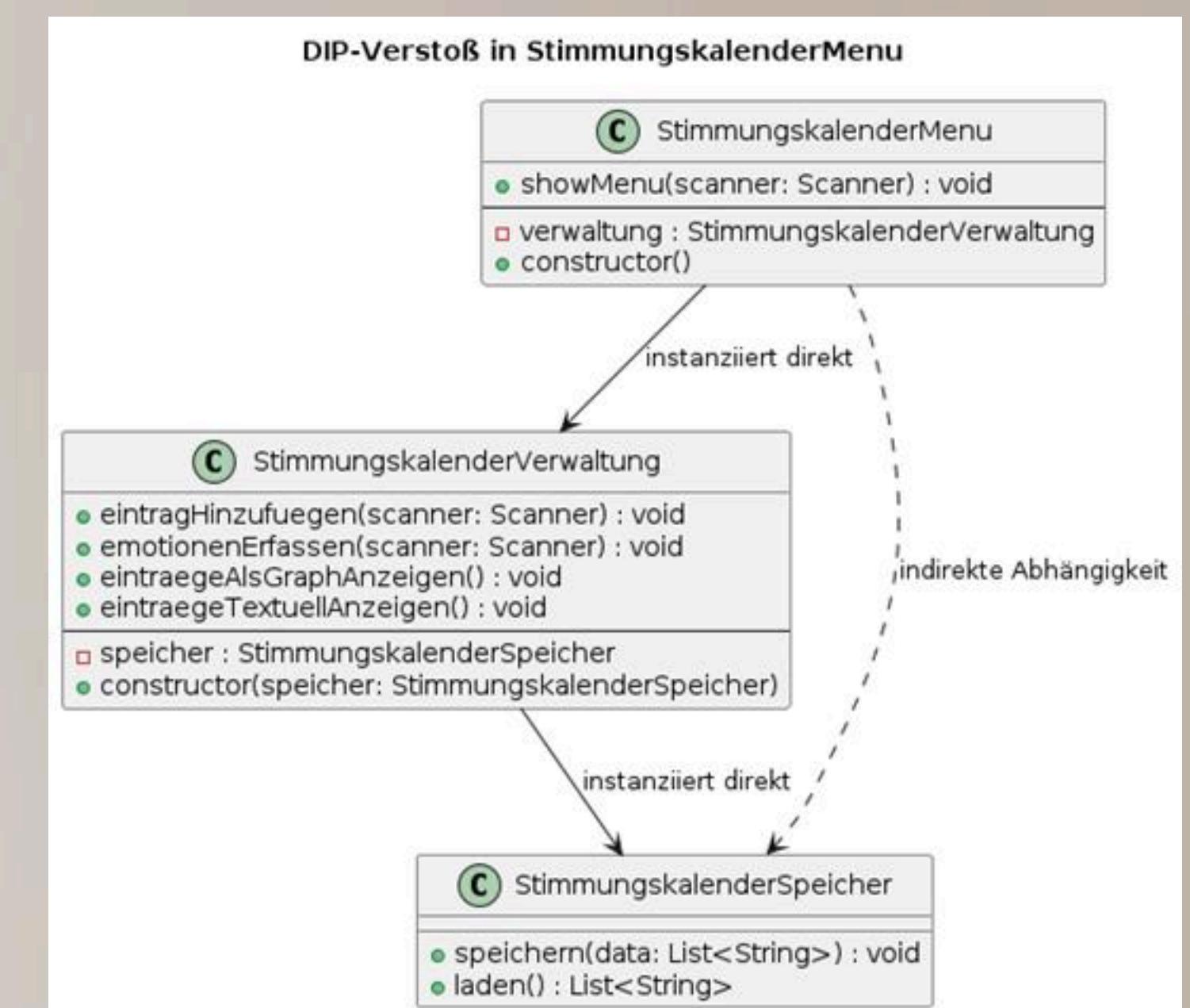
- Speicherstrategie wird zur Laufzeit injiziert
- Austausch von Implementierungen ohne nderung an der Logikklasse



# Negativ-Beispiele

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

- **StimmungskalenderMenu** erstellt selbst **StimmungskalenderVerwaltung** mit festem **StimmungskalenderSpeicher** → direkte Abhängigkeit von konkreter Implementierung (statt Abstraktion)
- **Folge:** schwer erweiterbar, schlecht testbar
- **Lösung:** Verwaltung per Konstruktor übergeben (Dependency Injection) → Menü hängt nur von Abstraktion ab, bleibt flexibel und DIP-konform

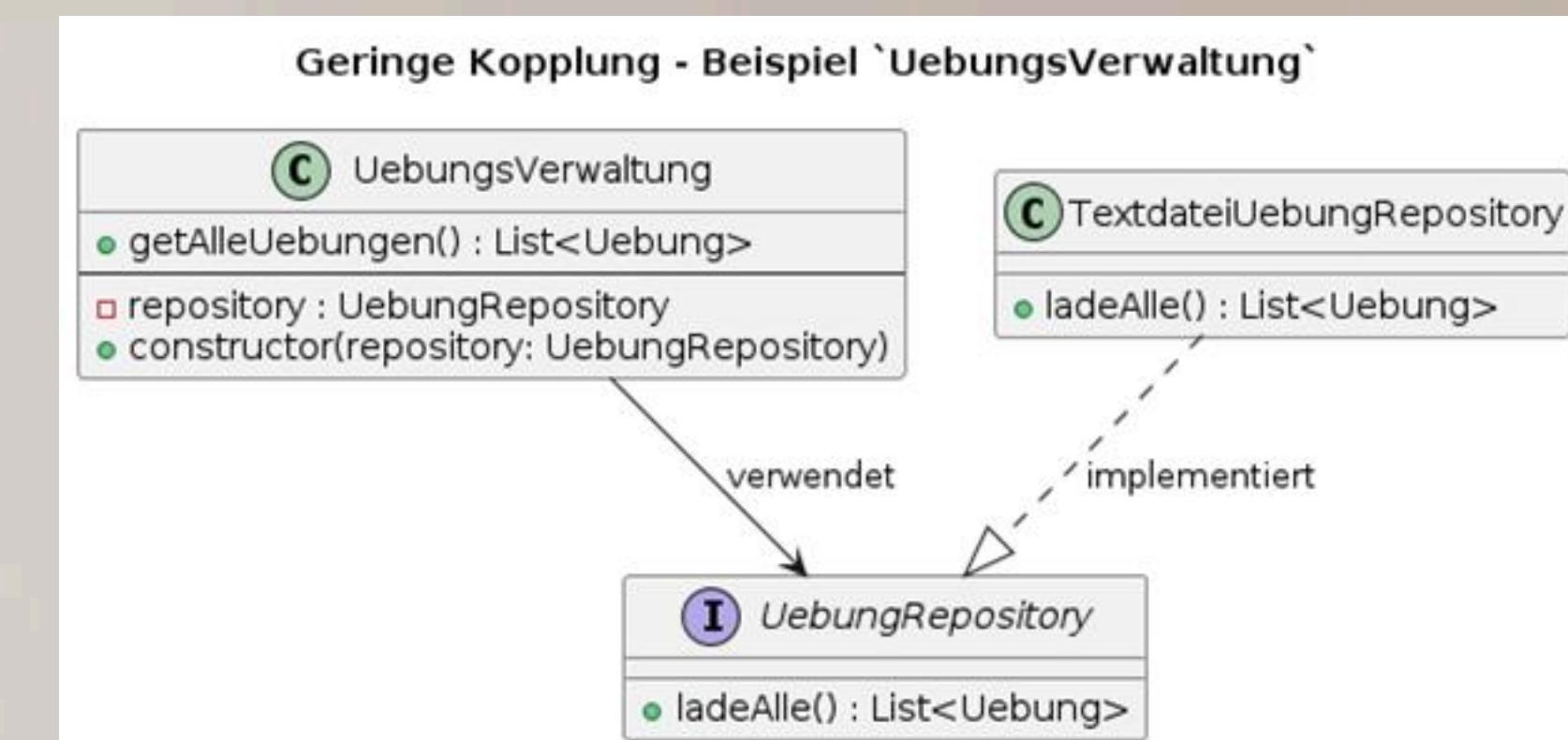




# Kapitel 4 - Weitere Prinzipien

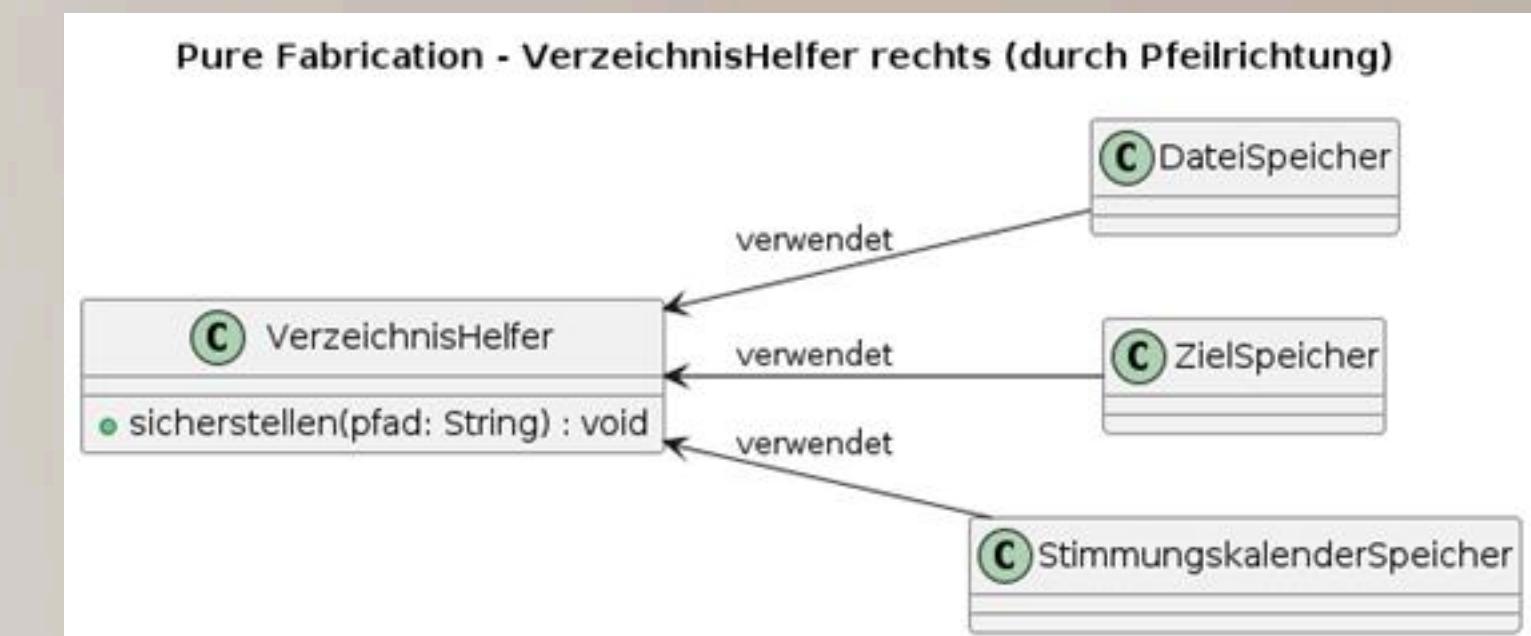
# Analyse GRASP

- **Abhängig von Abstraktion statt konkreter Klasse:** Die Klasse verwendet das Interface UebungRepository anstelle einer konkreten Implementierung wie TextdateiUebungRepository.
- **Flexibel erweiterbar:** Änderungen an der Speicherstrategie (z.B. Umstieg auf Datenbank oder Webservice) erfordern keine Änderungen an der Klasse selbst.
- **Hohe Testbarkeit:** Durch die Abhängigkeit zur Schnittstelle kann UebungsVerwaltung problemlos mit einem Mock-Repository getestet werden – ganz ohne Dateizugriff.
- **Saubere Trennung der Verantwortlichkeiten:** Die Klasse konzentriert sich auf die fachliche Logik (Verwaltung der Übungen), nicht auf technische Details der Datenspeicherung.



# Pure Fabrication

- **Zweck:** Technische Aufgabe (Verzeichnis anlegen) wird ausgelagert, um Fachlogik sauber zu halten.
- **Problem ohne Auslagerung:**
  - Duplikation in mehreren Speicherklassen (DateiSpeicher, ZielSpeicher, StimmungskalenderSpeicher),
  - Vermischung von Fach- und Infrastrukturcode,
  - stärkere Kopplung ans Dateisystem.
- **Lösung:** VerzeichnisHelper übernimmt einmalig die technische Verantwortung.
- **Vorteile:** Wiederverwendbar, leichter testbar, Fachklassen bleiben schlank und fokussiert.



# DRY

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

... @@ -0,0 +1,22 @@
1 + package utility;
2 +
3 + import java.io.File;
4 +
5 + public class VerzeichnisHelper {
6 +
7 +     /**
8 +      * Erstellt ein Verzeichnis, wenn es nicht existiert.
9 +      * Gibt true zurück, wenn es erfolgreich erstellt oder schon vorhanden ist.
10 +     *
11 +     * @param pfad Pfad zum Verzeichnis
12 +     */
13 +    public void sicherstellen(String pfad) {
14 +        File dir = new File(pfad);
15 +        if (!dir.exists()) {
16 +            boolean success = dir.mkdirs();
17 +            if (!success) {
18 +                System.err.println("Ordner '" + pfad + "' konnte nicht erstellt werden!");
19 +            }
20 +        }
21 +    }
22 +}
```

```
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();
}
```

# Kapitel 5 - Unit Tests

# Unit Tests

```
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.write(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_erstelltDateiWennNichtVorhanden(@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 }
```

- **Neue Datei schreiben:** Inhalt wird korrekt gespeichert
- **An bestehende Datei anhängen:** Vorheriger Inhalt bleibt erhalten
- **Leerzeile nur bei Bedarf:** Korrekte Formatierung bei vorhandenen Inhalten
- **Datei wird bei Bedarf erstellt:** Kein Fehler bei fehlender Datei
- **Tests nutzen Temp-Verzeichnis:** Kein Einfluss auf echte Dateien

# Unit Tests

```
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     static class MockAbfrage extends BenutzerAbfrageDateiLoeschen {
66         private final boolean ganzeDatei;
67
68         public MockAbfrage(boolean ganzeDatei) {
69             this.ganzeDatei = ganzeDatei;
70         }
71
72         @Override
73         public boolean eingabe_lesen(Scanner scanner) {
74             return ganzeDatei;
75         }
76     }
77
78     static class MockBenutzerEingabe extends BenutzerEingabe {
79         private final String text;
80
81         public MockBenutzerEingabe(String text) {
82             this.text = text;
83         }
84
85         @Override
86         public String leseEintrag(Scanner scanner) {
87             return text;
88         }
89     }
90
91     @BeforeEach
92     void setup() {
93         repository = new MockTagebuchRepository();
94         eingabe = new MockBenutzerEingabe("Testtext");
95         abfrage = new MockAbfrage(false); // default: nur Eintrag löschen, nicht ganze Datei
96         verwaltung = new TagebuchVerwaltung(repository, eingabe, abfrage);
97     }
98
99     @Test
100    void eintragSchreiben_speichertEintragMitHeutigemDatumUndText() {
101        verwaltung.eintragSchreiben(new Scanner(""));
102
103        assertEquals("Testtext", repository.gespeicherterEintrag.text());
104        assertEquals(LocalDate.now().toString(), repository.gespeicherterEintrag.datum());
105    }
106
107    @Test
108    void eintragLesen_gibtEintragFuerGewaehltesDatumAus() {
109        Scanner scanner = new Scanner("1\n");
110        verwaltung.eintragLesen(scanner);
111
112        assertEquals("2025-04-18", repository.zuletztGelesenesDatum);
113    }
114}
```

- **Eintrag schreiben:** Speichert korrekt Datum + Text (mit Mock)
- **Eintrag lesen:** Übergibt korrekt das gewählte Datum
- **Eintrag bearbeiten:** Prüft Übergabe von Datum, Uhrzeit & neuem Text
- **Gesamteintrag löschen:** Prüft Löschung ganzer Datei bei Bestätigung
- **Einzelnen Eintrag löschen:** Prüft gezielte Löschung per Uhrzeit

→ Alle Tests validieren korrekte Benutzersteuerung und saubere Weiterleitung an das Repository.

# AMRIP

- **Automatic – Vollautomatisiert & integriert:**

- Tests werden automatisch mit mvn test über Maven ausgeführt
- Kein manuelles Eingreifen nötig
- Tests sind deterministisch (klar bestanden/nicht bestanden)
- Konsoleninteraktionen werden durch Scanner-Mocking simuliert

- **Thorough – Gründlich & priorisiert:**

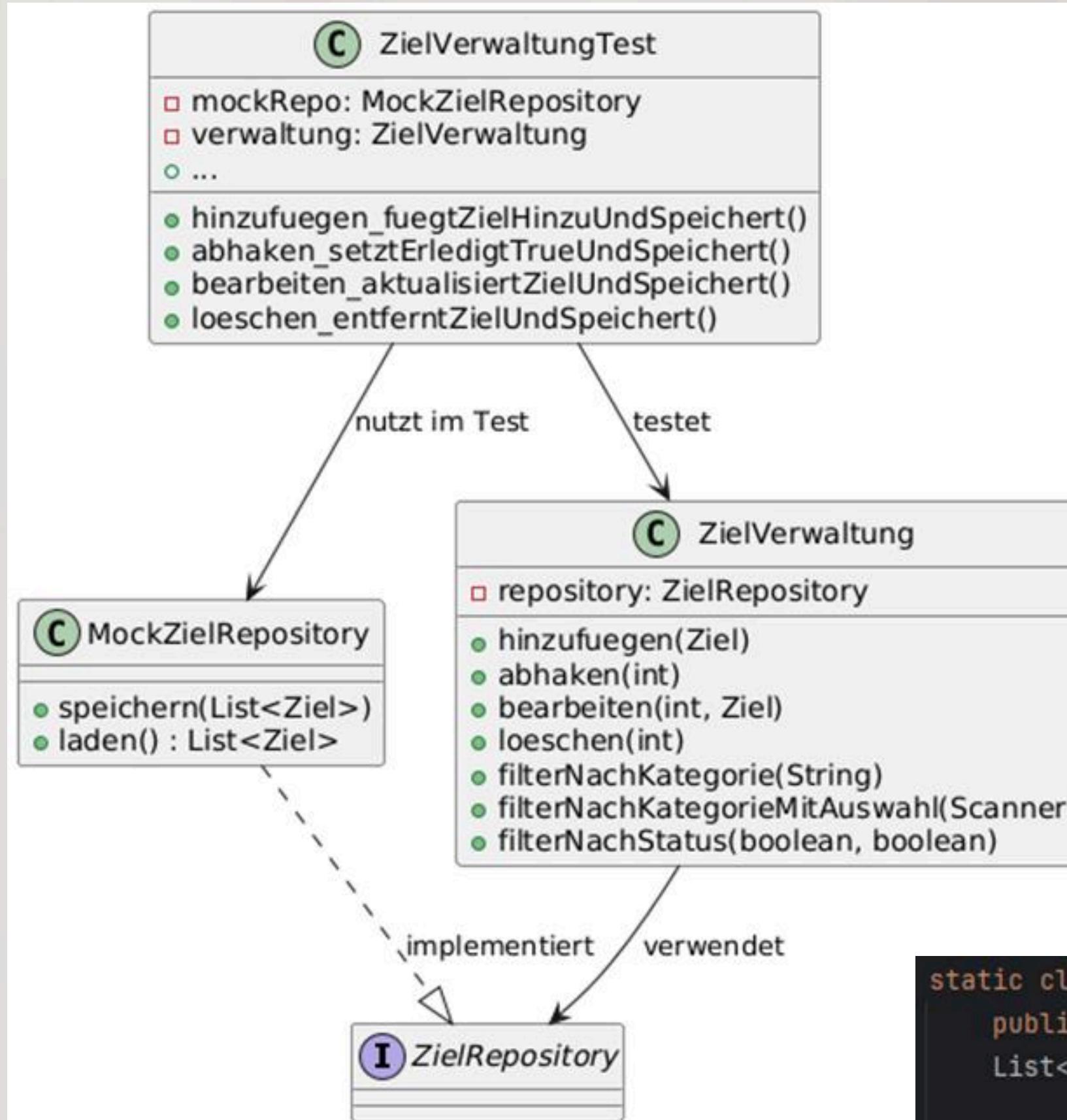
- Hohe Testabdeckung bei zentralen Modulen (78 %-87 %)
- Kritische Funktionen wie Speichern, Bearbeiten, Filtern sind getestet
- Auch Randfälle und Fehlersituationen abgedeckt
- Geringere Testabdeckung bei UI-nahen oder neuen Komponenten, aber begründet priorisiert

- **Professional – Klar, wartbar, gut strukturiert:**

- **Namenskonvention:** KlasseTest, sprechende Methodennamen
- Tests logisch nach Domänenmodulen organisiert
- Fokus auf reale, relevante Szenarien statt künstlicher Testfälle
- Einsatz von Mocks für isolierte, wiederholbare Tests
- Verständlicher Code ohne unnötige Komplexität

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

# Fakes und Mocks



- Ort der Verwendung:** In `ZielVerwaltungTest`
- Technik:** Eigene Fake-Klasse implementiert das Interface `ZielRepository`
- Funktion:** Verwaltet Ziele im Arbeitsspeicher(mittels `List<Ziel>`) statt in Dateien
- Ziel:** Isolierter Test der Geschäftslogik von `ZielVerwaltung` ohne Dateisystem
- Vorteile:**
  - Schnellere Testdurchläufe
  - Keine Seiteneffekte oder externe Abhängigkeiten
  - Wiederholbarkeit garantiert

```

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

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

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

# Fakes und Mocks

```
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; }

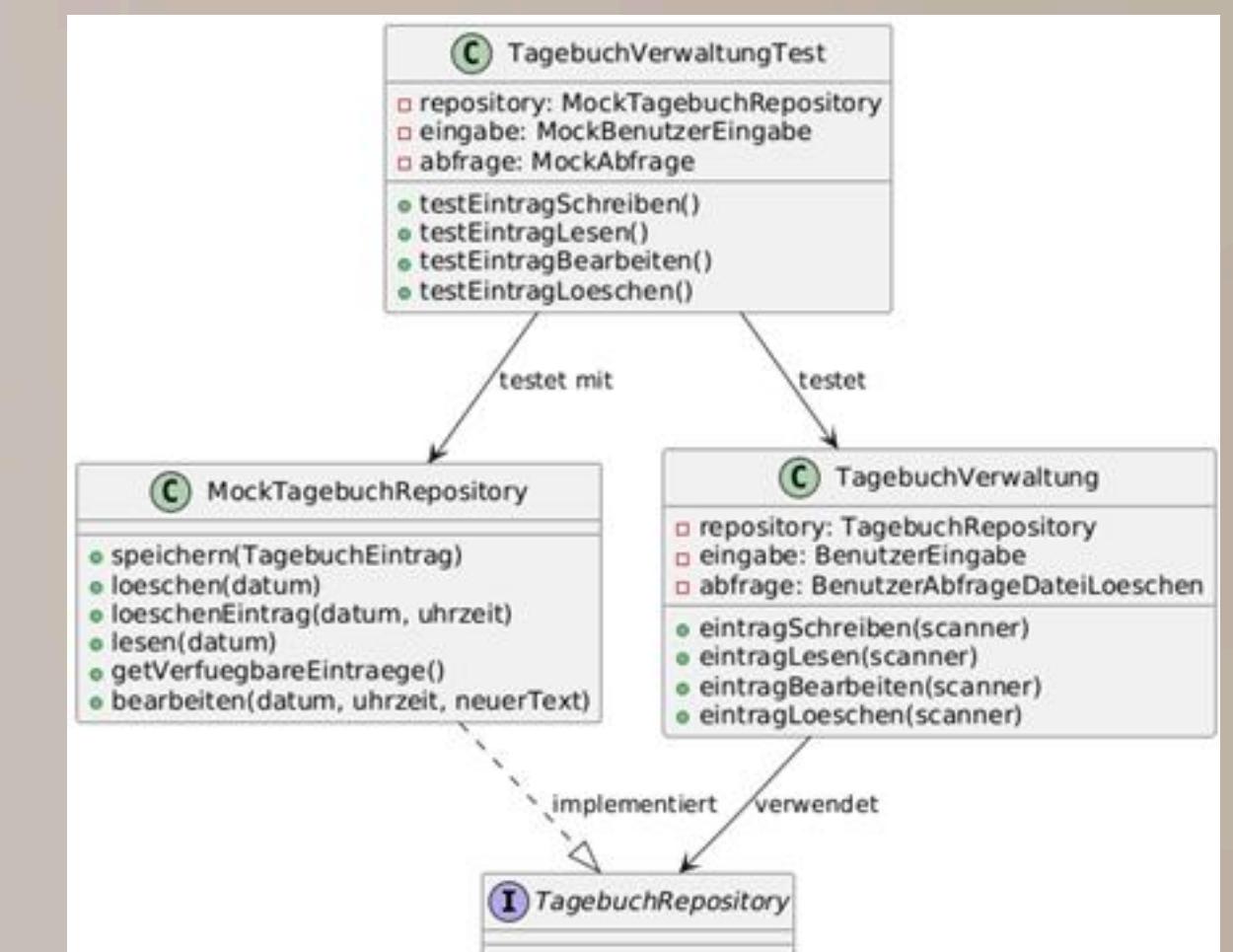
    @Override 3 usages ▾ katty.terra
    public void loeschenEintrag(String datum, String uhrzeit) {
        this.geloeschtesDatum = datum;
        this.geloeschteUhrzeit = uhrzeit;
    }

    @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("2025-04-18"); }

    @Override ▾ katty.terra
    public boolean bearbeiten(String datum, String uhrzeit, String neuerText) {
        this.bearbeitetesDatum = datum;
        this.bearbeiteteUhrzeit = uhrzeit;
        this.bearbeiteterText = neuerText;
        return bearbeitenErgebnis;
    }
}
```

- **Ort der Verwendung:** In TagebuchVerwaltungTest
- **Technik:** Eigener Mock(MockTagebuchRepository) implementiert das Interface TagebuchRepository
- **Funktion:** Verarbeitet Einträge in einer internen List<TagebuchEintrag>, ganz ohne Dateizugriff
- **Ziel:** Testet das Verhalten von TagebuchVerwaltung bei Speicherung, Löschung und Bearbeitung von Einträgen
- **Vorteile:**
  - Tests laufen unabhängig vom Dateisystem
  - Logik kann isoliert und gezielt geprüft werden
  - Edge Cases (z.B. fehlende Uhrzeit) lassen sich gezielt simulieren



# Kapitel 6 - Domain Driven Design

# Ubiquitous Language

- **Stimmungseintrag**

- Bezeichnet eine dokumentierte Tagesstimmung eines Nutzers einschließlich Bewertung und Beschreibung.
- Der Begriff ist fachlich präzise, domänenspezifisch und für Anwender eindeutig verständlich.

- **Routine**

- Bezeichnet eine regelmäßig wiederkehrende Tätigkeit (z.B. „Wasser trinken“), die vom Nutzer ausgeführt und markiert werden kann.
- Der Begriff ist im Alltag etabliert, spiegelt gesundheitsförderliche Handlungsmuster wider und ist dem Anwendungsbereich klar zuordenbar.

- **Ziel**

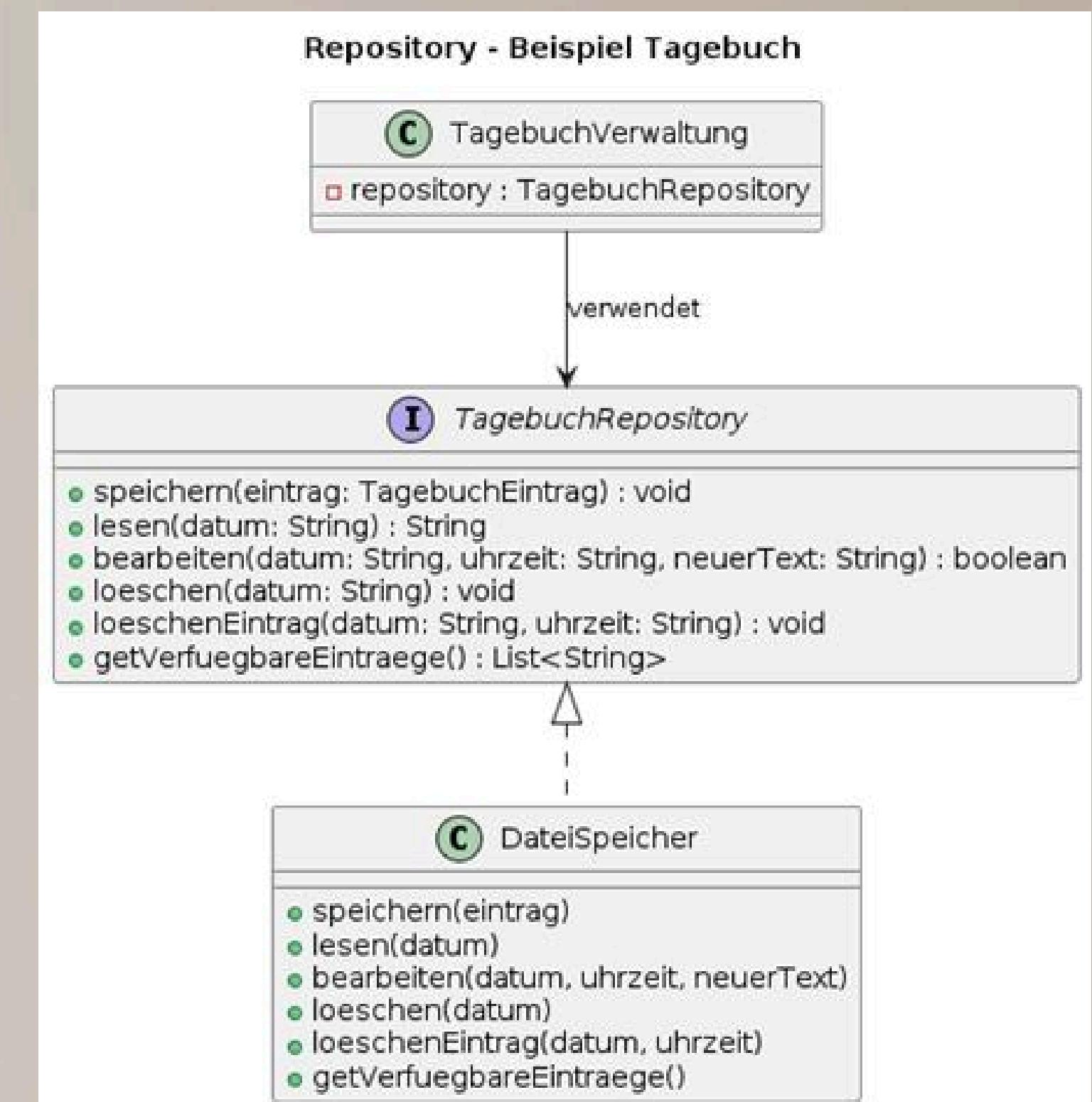
- Ein individuell gesetztes Vorhaben des Nutzers, das strukturiert (z.B. mit Priorität und Fälligkeitsdatum) verwaltet wird.
- Der Begriff ist fachlich etabliert, unterstützt die Selbstentwicklung und findet in therapeutischen Kontexten breite Anwendung.

- **Gedankenreflexion**

- Modul zur strukturierten schriftlichen Auseinandersetzung mit belastenden Gedanken.
- Der Begriff beschreibt einen psychologischen Verarbeitungsprozess und ist fachlich korrekt sowie kommunikativ eindeutig.

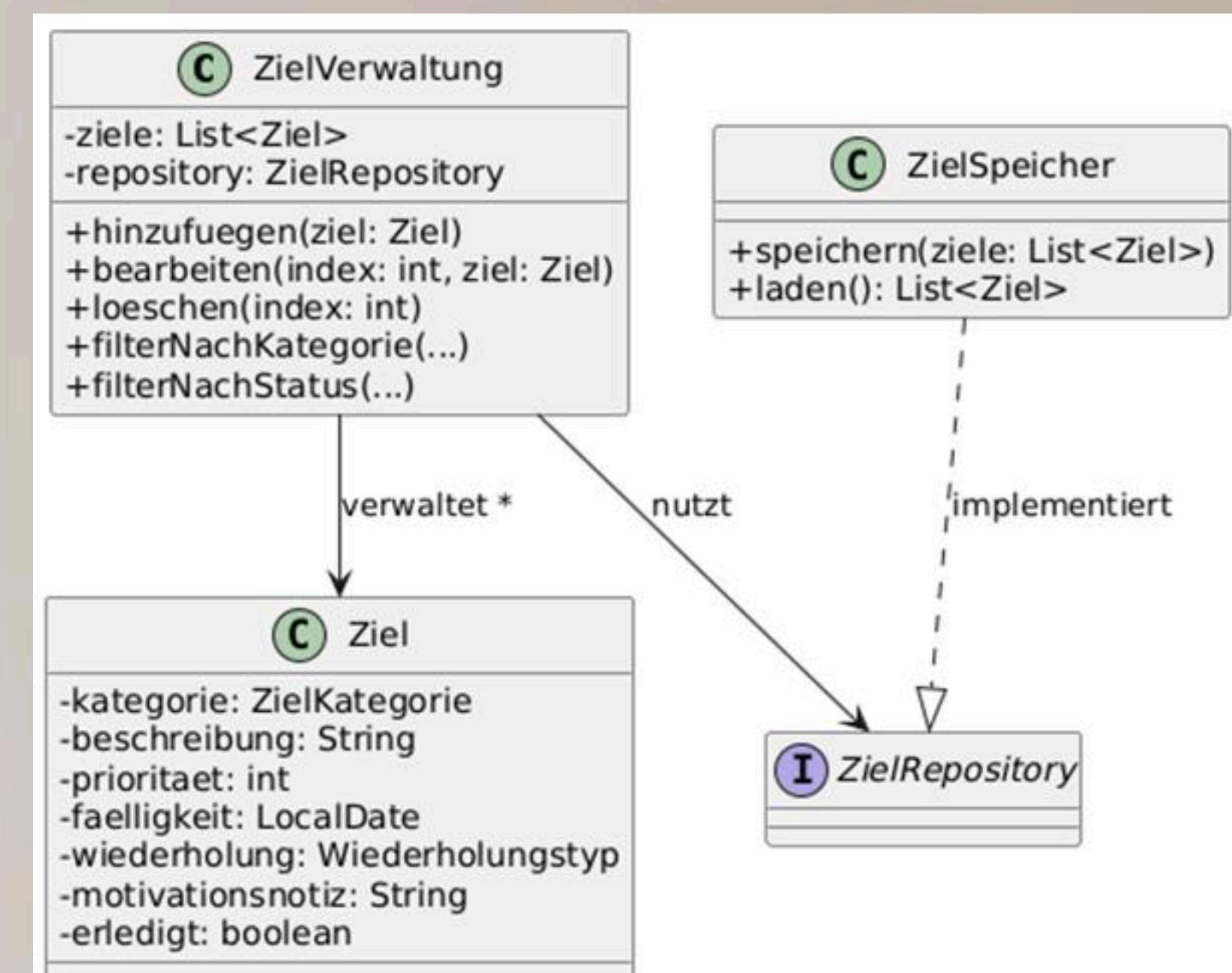
# Repositories

- **Abstraktion der Speicherung:** Das Interface TagebuchRepository definiert eine abstrakte Schnittstelle für grundlegende Operationen wie Speichern, Lesen, Bearbeiten und Löschen von Tagebucheinträgen, ohne sich auf eine konkrete Speicherform festzulegen.
- **Technische Umsetzung getrennt:** Die Implementierung DateiSpeicher übernimmt die konkrete Ausführung der Dateioperationen, wodurch die Domänenlogik von technischen Details entkoppelt bleibt.
- **DDD-konforme Repository-Struktur**
  - Das Interface arbeitet mit einem klar definierten Aggregat (TagebuchEintrag).
  - Es existiert genau ein zugehöriges Repository für dieses Aggregat.
  - Die Methoden liefern konsistent die Aggregate Root zurück.
  - Die Schnittstelle ist im Domänenmodell verortet, während die Implementierung außerhalb liegt (Trennung von Domäne und Infrastruktur).



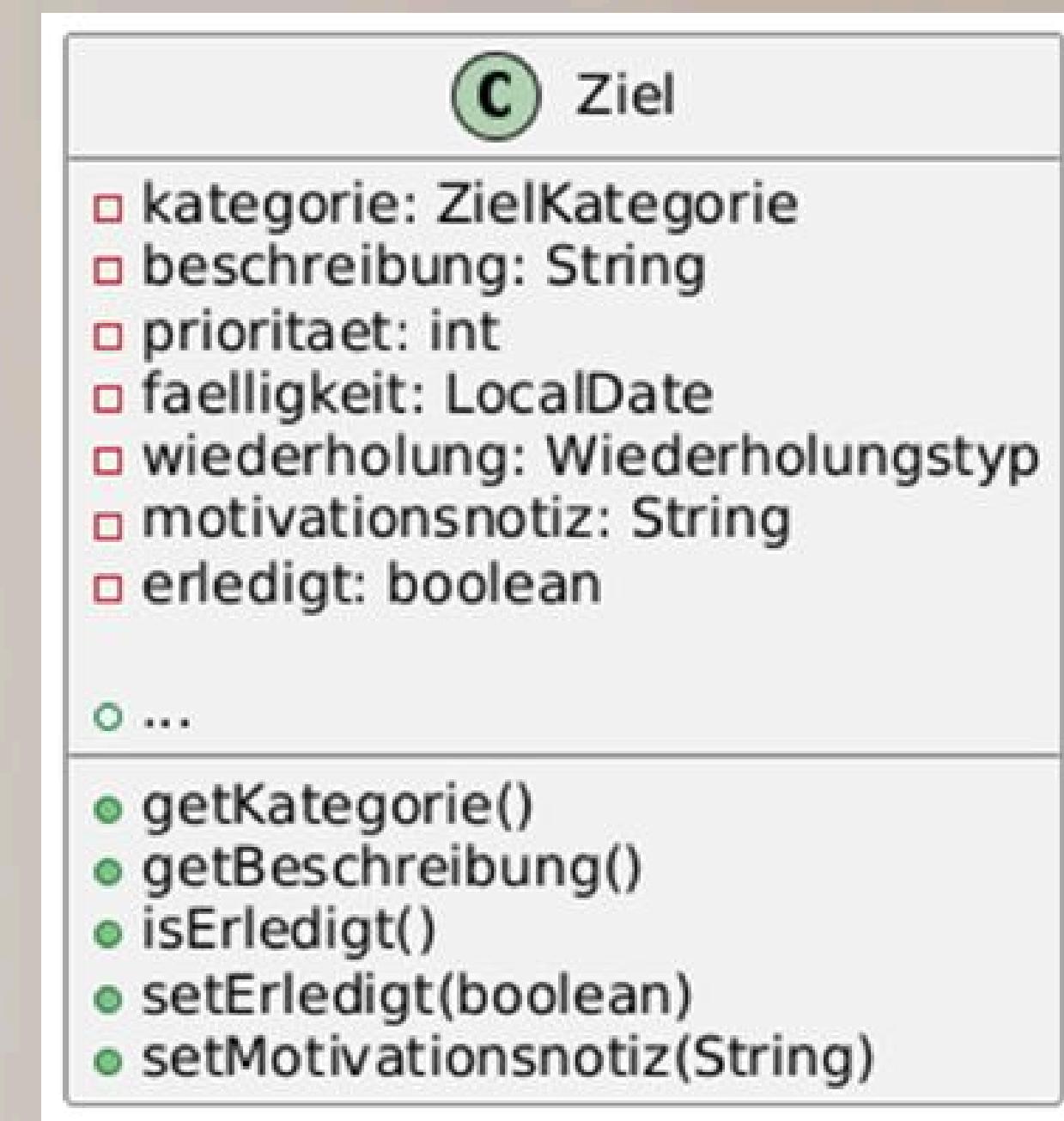
# Aggregates

- Die Klasse **ZielVerwaltung** bildet ein typisches Aggregat im Sinne von Domain-Driven Design.
- Sie ist die Aggregate Root und verwaltet alle zugehörigen Ziel-Entitäten, die eine eigene Identität besitzen.
- Alle Operationen wie Hinzufügen, Bearbeiten oder Löschen erfolgen ausschließlich über die Methoden dieser Root - direkte Änderungen an einzelnen Zielen von außen sind nicht möglich.
- Dadurch entsteht eine klare fachliche Einheit mit definierter Transaktionsgrenze.
- Änderungen werden stets gesammelt über das Repository gespeichert, was der Aggregat-Regel entspricht, dass ein Aggregat als Ganzes geladen und persistiert wird.
- So sichert die Klasse die fachliche Konsistenz und zentralisiert die Geschäftslogik der Zielverwaltung.



# Entities

- Die Klasse **Ziel** ist eine typische Entity im Sinne von Domain-Driven Design.
- Sie besitzt eine dauerhafte Identität, die unabhängig von Veränderungen ihrer Eigenschaften bestehen bleibt.
- Eigenschaften wie Beschreibung, Priorität, Fälligkeitsdatum oder Status können sich ändern – die Identität des Ziels bleibt jedoch gleich.
- Die Ziele werden durch das Aggregat ZielVerwaltung verwaltet und über ihre Lebensdauer hinweg eindeutig identifiziert.
- Damit erfüllt die Klasse alle Merkmale einer fachlichen Entität in der Domäne.



# Value Objects

- Value Objects sind Objekte ohne eigene Identität - sie werden rein über ihre Attributwerte definiert.
- Im Projekt ist ZielKategorie ein Beispiel dafür: eine unveränderliche Enum, die Ziele thematisch klassifiziert (z.B. GESUNDHEIT, BERUF).
- Sie wird wertbasiert innerhalb der Entität Ziel verwendet und besitzt keine eigene Lebensdauer.
- Damit erfüllt sie alle Merkmale eines klassischen Value Objects im Sinne von Domain-Driven Design.

E	ZielKategorie
	GESUNDHEIT
	BERUF
	FREIZEIT
	SOZIAL
	PERSOENLICHKEIT
	ORGANISATION

# Kapitel 7 - Refactoring

# Code Smell 1: Long Method

```
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.");  
    }  
}
```

```
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) { ... }
```

- **Beispiel:** Die Methode `eintragBearbeiten(Scanner scanner)` in der Klasse `TagebuchVerwaltung` ist zu lang und übernimmt mehrere Aufgaben gleichzeitig:
  - Anzeige und Auswahl eines vorhandenen Eintrags
  - Einlesen von Uhrzeit und neuem Text
  - Validierung der Eingaben
  - Durchführung der Bearbeitung
  - Rückmeldung an den Benutzer
- Diese Kombination führt zu geringer Übersichtlichkeit und erschwert Wiederverwendung sowie gezielte Tests einzelner Logikbestandteile.
- **Lösungsweg:** Aufteilung in klar benannte Hilfsmethoden mit jeweils nur einer Verantwortung gemäß dem Single-Responsibility-Prinzip.

# Code Smell 2: Shotgun Surgery

- **Problem:**

- repository.speichern(ziele) mehrfach in verschiedenen Methoden in der Klasse ZielVerwaltung
- Änderungen am Speicherverhalten erfordern viele Codeänderungen.
- Erhöhtes Fehlerpotenzial, erschwert Wartung.

- **Lösung:**

- Einführung zentraler Methode aktualisiereUndSpeichere(Runnable änderung).
- Änderung wird per Lambda übergeben, Speichern erfolgt zentral.

- **Vorteile:**

- Reduzierte Duplizierung.
- Einfachere Erweiterung (z.B. Logging).
- Klarere Struktur, bessere Wartbarkeit.

```
private void aktualisiereUndSpeichere(Runnable änderung) {  
    änderung.run();  
    repository.speichern(ziele);  
}  
  
public void loeschen(int index) {  
    aktualisiereUndSpeichere(() -> ziele.remove(index));  
}
```

# Refactoring 1: Extract Method

```
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();  
    }  
  
    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;  
        }  
    }
```

- **Ort:**  
ZielVerwaltung.filterNachKategorieMitAuswahl(...)
- **Problem:** Die Methode enthält sowohl Benutzerdialog als auch Logik zur Filterung → vermischt Zuständigkeiten.
- **Lösung:** Auslagerung der Benutzeroauswahl in eine eigene Methode waehleKategorie(...).
- **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).

# Refactoring 2: Replace Temp with Query

```
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();  
    }  
}
```

```
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();  
    }  
}
```

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

# Kapitel 8 - Entwurfsmuster (8P)

# Strategy

- **Problem:**

- Viele einzelne Filtermethoden (z.B. nach Kategorie, Status, Wiederholung).
- Neue Filter erfordern Codeänderung an der Klasse ZielVerwaltung.

- **Lösung:**

- Einsatz des Strategy-Musters zur Kapselung der Filterlogik. Jeder Filter ist eine eigene Klasse, die das Interface ZielFilterStrategy implementiert.

- **Vorteile:**

- Erweiterbarkeit ohne Änderung bestehender Logik.
- Flexibler Austausch von Filterstrategien zur Laufzeit.
- Saubere Trennung von Logik & Anwendungsklasse.

- Interface: ZielFilterStrategy
  - o 

```
public interface ZielFilterStrategy {  
    List<Ziel> filter(List<Ziel> ziele);  
}
```
- Strategie 1: Kategorie-Filter
  - o 

```
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
  - o 

```
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
  - o 

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

# Builder

- **Anwendungsszenario:** Erstellen von Ziel-Objekten mit vielen optionalen Attributen (z.B. Fälligkeit, Wiederholung, Motivationsnotiz).
- **Problem:** Konstruktor mit vielen Parametern ist unübersichtlich und fehleranfällig.
- **Lösung:** Einsatz des Builder-Musters für klare, lesbare Objektkonstruktion mit method chaining.
- **Vorteile:**
  - Verbesserte Lesbarkeit und Wartbarkeit.
  - Kein Bedarf an zahlreichen Konstruktorüberladungen.
  - Besonders hilfreich in Tests und bei interaktiver Benutzereingabe.

```
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();
```



Vielen Dank für Ihre  
Aufmerksamkeit