

# **PROGRAMMENTWURF**

## **- PROTOKOLL**

**Name: [Obreiter, Nick] Matrikelnummer: [8107127]**

**Abgabedatum: [04.05.2025]**

# KAPITEL 1: EINFÜHRUNG (4P)

# ÜBERSICHT ÜBER DIE APPLIKATION (1P)

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

**FitTrackerPro** ist eine prototyp zur Erfassung und generierung von Workouts. Es wird dem Nutzern ermöglicht Trainingspläne zu erstellen, Fortschritte zu verfolgen und in Zukunft Ernährungsdaten zu verwalten.

## Frage

## Was macht die Applikation?

## Antwort

- Stellt einen durchsuchbaren Übungskatalog bereit (z. B. Kniebeugen, Bankdrücken).- Ermöglicht das Anlegen individueller Trainingspläne (Split, Ganzkörper u. a.).- Führt den Nutzer während des Workouts durch die geplanten Sätze

**In der Zukunft:** Speichern & Wiederholungen und erfasst dabei Gewicht, RPE usw.- Speichert alle Einträge und visualisiert Fortschritte (PR-Historie, Volumen-Trends).- (Roadmap) Integriert Ernährungs-Einträge, um Training & Kalorienaufnahme gemeinsam auszuwerten.

Wie  
funktioniert  
sie?

- Clean-Architecture-Ansatz mit klar getrennten Schichten: 1. *Domain* (Entitäten wie `Exercise`, `Workout`) 2. *Use-Cases* (z. B. `CreateWorkoutPlanUseCase`, `LogSetUseCase`) 3. *Interface Adapters* (CLI / GUI, Persistenz-Gateway).- Persistenz über eine eingebettete Datenbank → keine externe Installation nötig.- Java 21 + Maven liefern plattformunabhängige Ausführung (`java -jar FitTrackerPro.jar`).

## Welches Problem löst sie / Zweck?

- Viele Sportler nutzen Zettel, Excel oder mehrere Apps – Daten sind verstreut und schwer auswertbar.- FitTrackerPro konsolidiert Training + (in Zukunft) Ernährung in einem Tool, automatisiert die Fortschrittserfassung und macht Entwicklung transparent.- Durch die modulare Architektur kann die Anwendung leicht um neue Algorithmen zur Trainingsplan-Generierung oder zusätzliche Analytics- bzw. Export-Features erweitert werden.

---

# STARTEN DER APPLIKATION (1P)

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

Die Applikation kann mit dem normalen Java-Befehl in der Konsole ausgeführt werden:

```
java -jar FitTrackerPro.jar
```

Voraussetzungen:

- Installierte Java-Laufzeitumgebung version 21 (JRE) oder Java Development Kit (JDK)
- Das FitTrackerPro.jar-File muss sich im aktuellen Verzeichnis befinden.

# TECHNISCHER ÜBERBLICK (2P)

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

- **Java 21:** Verwendet für die Kernentwicklung aufgrund der verbesserten Performance und Features.
- **Maven:** Verwaltung von Abhängigkeiten und Build-Prozess.
- **JUnit:** Testframework für Unit-Tests.

- **H2-Datenbank:** Vorbereitung für Einsatz von Datenpersistenz sofern man keine InMemory verwendet.
- **PlantUML-Generator:** Automatische Erstellung von UML-Diagrammen zur Code-Dokumentation.
- **Commons-IO:** Bibliothek für erweiterte Dateioperationen (benötigt für PlantUML).
- SonarCube

# KAPITEL 2:

Softwarearchitektur (8P)

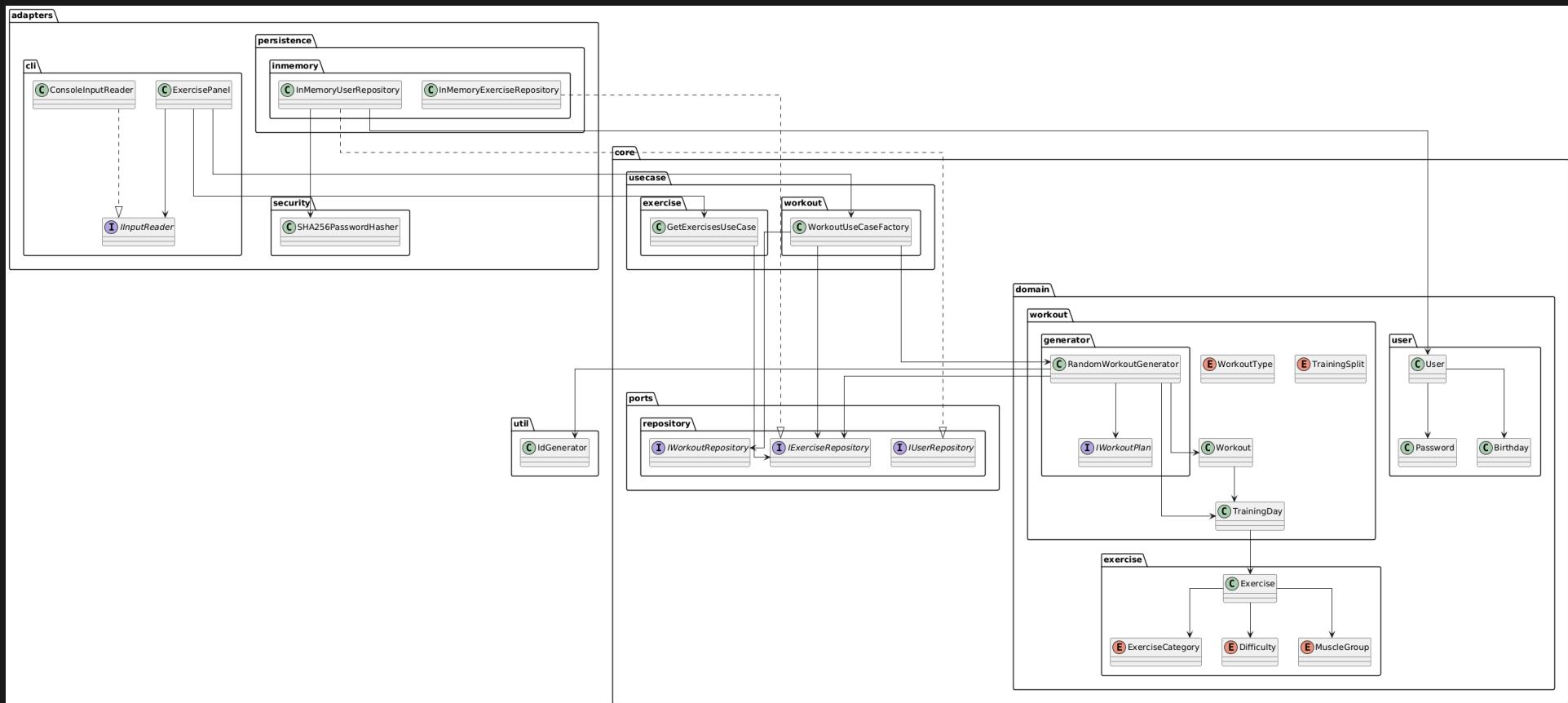
# GEWÄHLTE ARCHITEKTUR (4P)

[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]

# CLEAN ARCHITECTURE

- **Schichten:**

1. **Domain (Entities):** Enthält zentrale Domänenobjekte wie `Exercise`, `User` und `Workout`.
  2. **Use Cases:** Realisieren Geschäftsprozesse, z. B. `CreateWorkoutUseCase` oder `AuthenticationUserUseCase`.
  3. **Interface Adapters:** Übersetzen Daten zwischen interner Logik und externen Schnittstellen.
- **Dependency Rule:** Alle Abhängigkeiten verlaufen von außen nach innen, sodass die inneren Schichten (Domain und Use Cases) keine Kenntnis von technischen Implementierungen haben.



# DOMAIN CODE (1P)

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

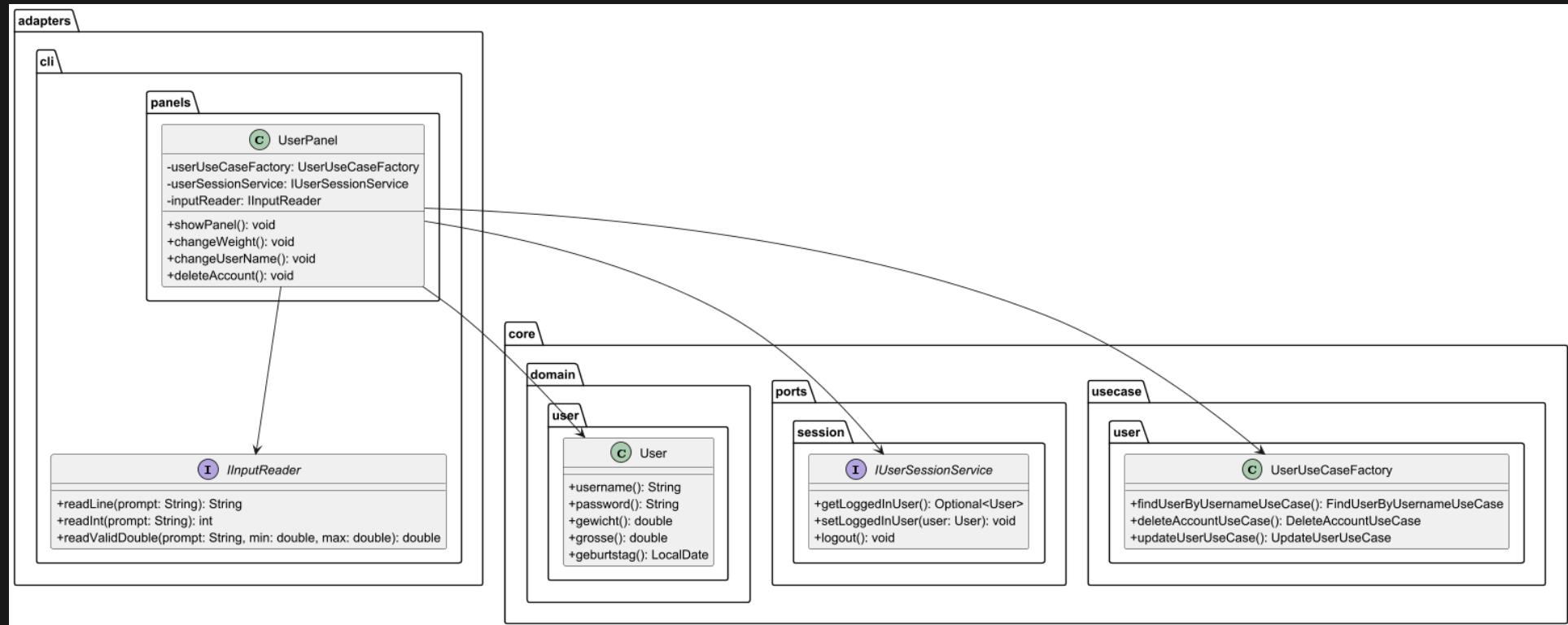
Im Kontext von Domain-Driven Design (DDD) und Clean Architecture bildet der **\*\*Domain Code\*\*** den fachlichen Kern von **\*\*FitTrackerPro\*\***. Er umfasst alle zentralen Konzepte rund um das Training – wie z. B. Benutzer, Workouts, Übungen oder Trainingspläne – und bildet deren Eigenschaften und Regeln unabhängig von technischen Details ab. Diese Schicht stellt sicher, dass die Geschäftslogik (z. B. was ein `Workout` oder ein `User` ist) **\*\*klar getrennt\*\*** von der Benutzeroberfläche, Datenspeicherung oder sonstigen Implementierungen bleibt.

```
1 package core.domain.exercise;
2
3 import java.util.List;
4
5 /**
6  * Record: Exercise
7  *
8  * @author NeeroxZ
9  * @date 21.10.2024
10 */
11
12 public record Exercise(
13     int id,
14     String name,
15     ExerciseCategory category,
16     Difficulty difficulty,
17     String description,          // Detaillierte Beschreibung der Übung
18     List<MuscleGroup> targetMuscles // Primär trainierte Muskelgruppen
19 ) implements Comparable<Exercise>
20 {
21
22     @Override
23     public int compareTo(Exercise other)
24     {
25         return this.name.compareTo(other.name);
26     }
27 }
28
29
30
```

# 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

# POSITIV-BEISPIEL: DEPENDENCY RULE



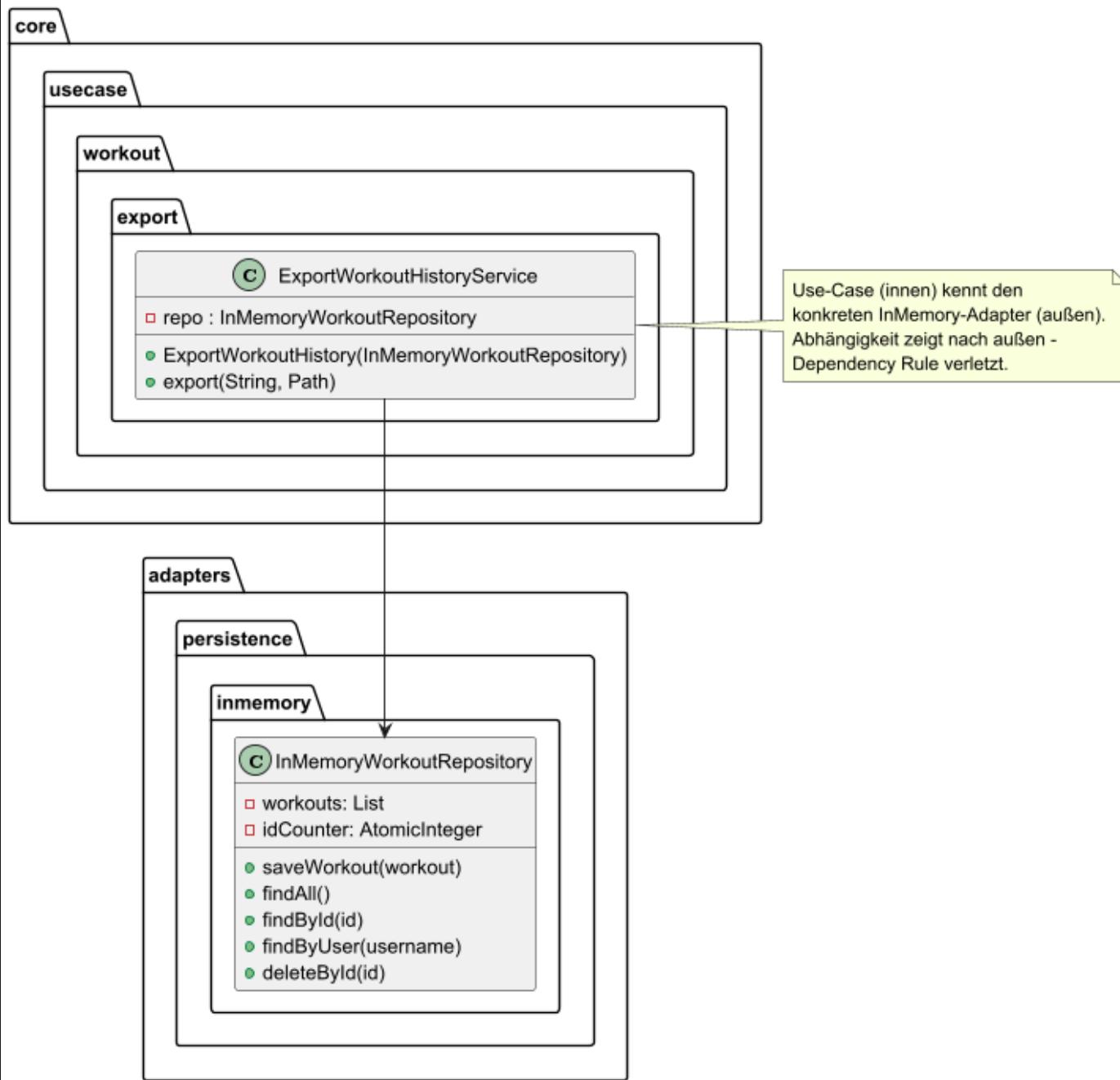
- **UserPanel** kennt nur Interfaces und **UseCaseFactory**.

# NEGATIV-BEISPIEL

: Dependency Rule

Alle äußeren Schichten kennen die gegenüberliegende innere Schicht \*\*nur über abstrakte Schnittstellen\*\* (Ports) was hier nicht der Fall ist.

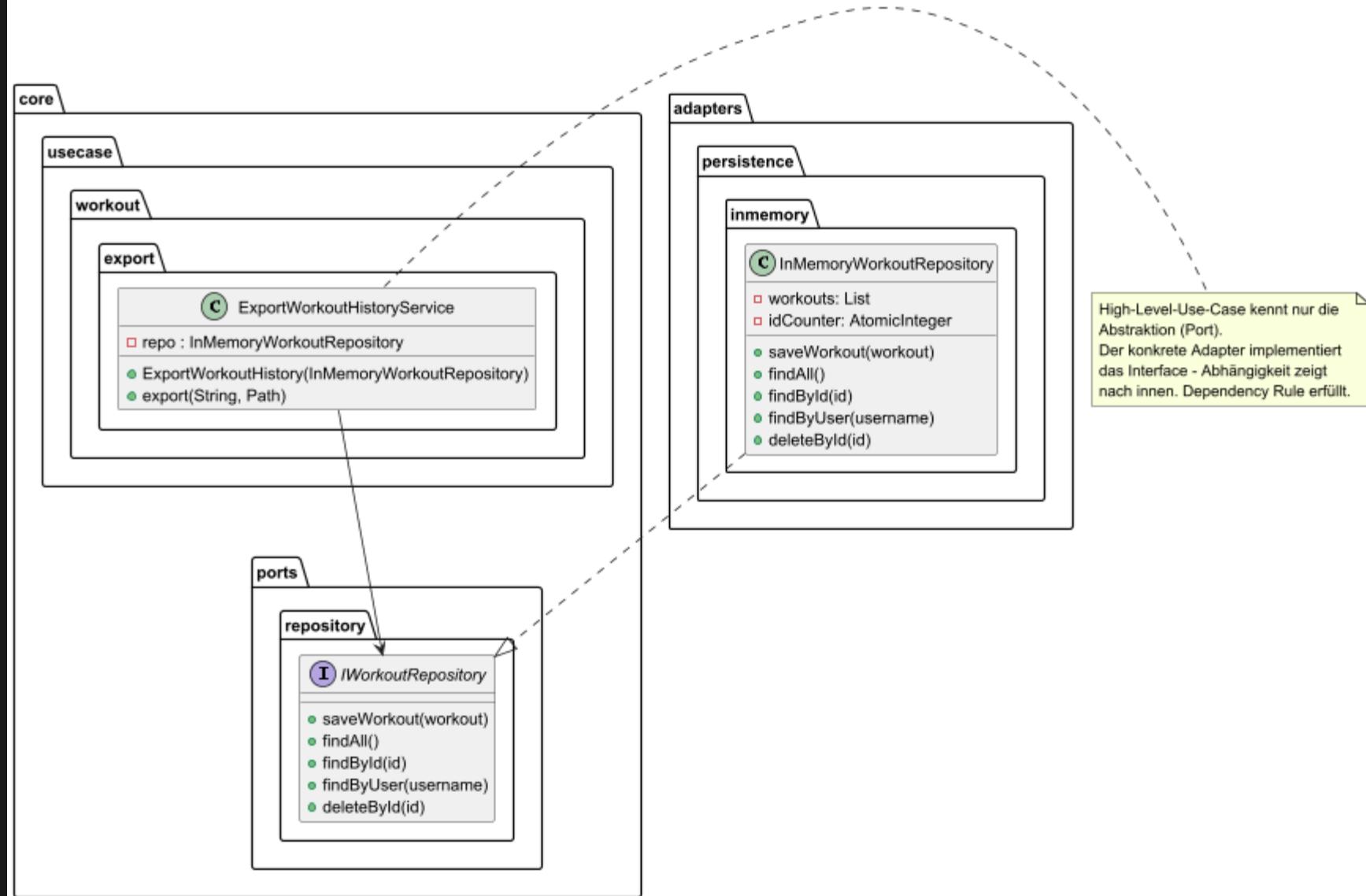
## Verletzung der Dependency Rule (ExportWorkoutHistoryServiceBad)



Use-Case (innen) kennt den konkreten InMemory-Adapter (außen). Abhängigkeit zeigt nach außen - Dependency Rule verletzt.

# LÖSUNG

## Lösung mit Port-Interface (ExportWorkoutHistoryService)

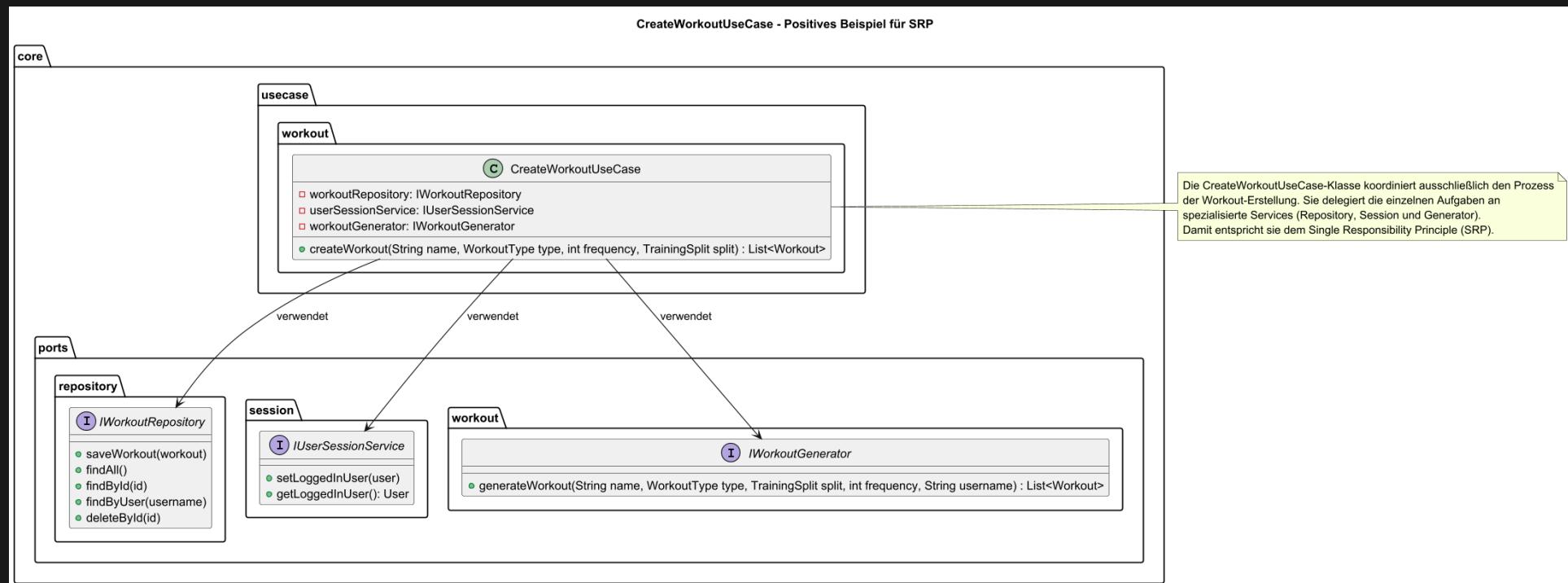


# KAPITEL 3: SOLID (8P)

# 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) ]

# POSITIV-BEISPIEL



# NEGATIV-BEISPIEL

**adapters**

cli

panels

**C** ExerciseCreatorCLI

- scanner : Scanner
- exerciseService : GetExercisesUseCaseFactory
- ExerciseCreatorCLI(exerciseService: GetExercisesUseCase)
- createOwnExercise() : void
- getCategoryInput() : ExerciseCategory
- getDifficultyInput() : Difficulty
- getTargetMuscles() : List<MuscleGroup>
- generateId() : int

core

usecase

user

**C** GetExercisesUseCaseFactory

- getExerciseByIdUseCase : GetExerciseByIdUseCase
- getAllExercisesUseCase : GetAllExercisesUseCase
- createExerciseUseCase : CreateExerciseUseCase
- getExercisesByTypeUseCase : GetExercisesByTypeUseCase
- filterExercisesBySplitUseCase : FilterExercisesBySplitUseCase
- getExerciseById(int id) : Optional<Exercise>
- getAllExercises() : List<Exercise>
- createExercise(Exercise exercise)
- getExercisesByType(WorkoutType type) : List<Exercise>
- filterExercisesBySplit(TrainingSplit split) : List<Exercise>

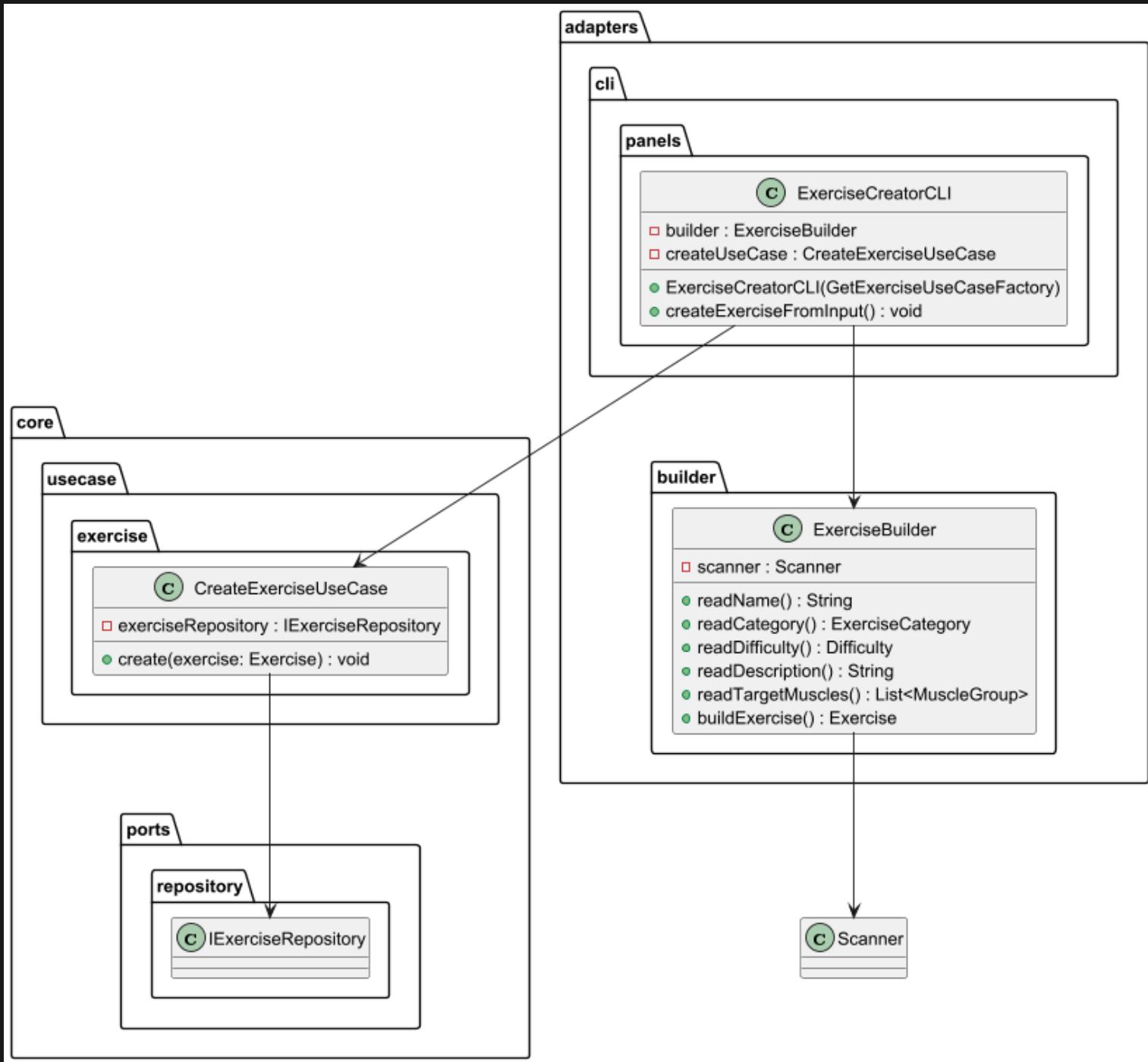
**C** Scanner

# ExerciseCreatorCLI` übernimmt mehrere Verantwortlichkeiten:

1. **Benutzereingabe** (Scanner, Inputfragen, Konsolentexte)
2. **Validierungslogik & Umwandlung** (z. B. Strings zu Enums)
3. **Geschäftslogik** (Erstellen & Speichern der Übung)
4. **ID-Erzeugung (Technische Verantwortung)**

Daher hat diese Klasse mehrere Gründe zur Änderung  
– was gegen das SRP verstößt.

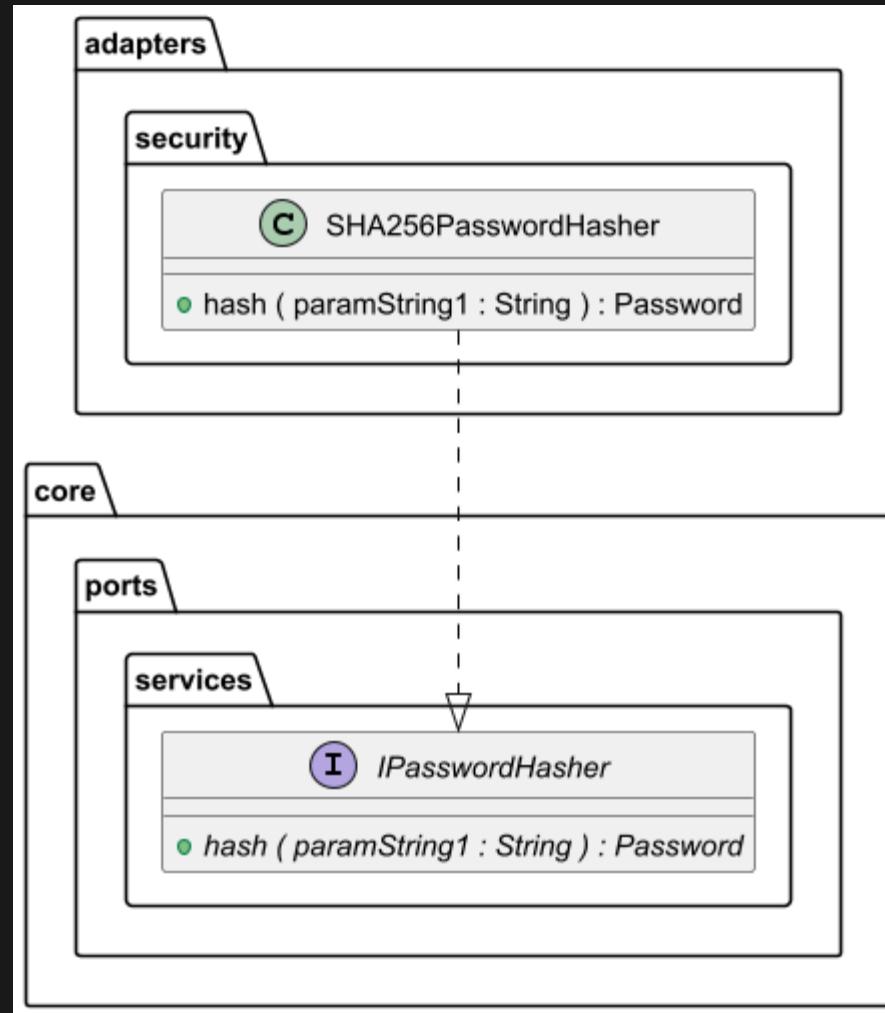
# LÖSUNG



# 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)?]

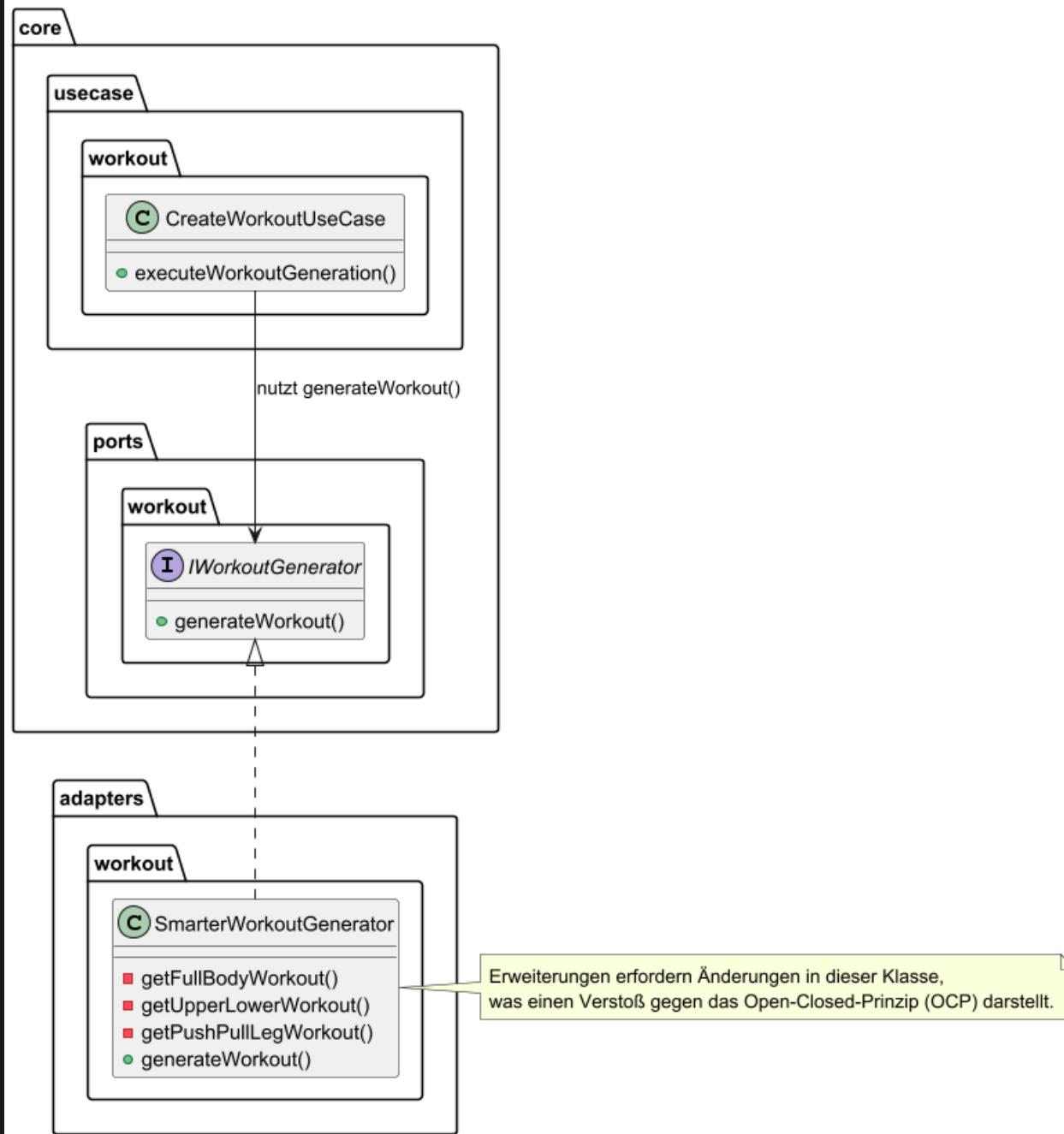
# POSITIV-BEISPIEL



Die Klasse SHA256PasswordHasher implementiert das Interface IPasswordHasher. Wird ein neuer Hashing-Algorithmus benötigt, kann eine neue Klasse implementiert werden, ohne dass bestehender Code verändert werden muss



## Analyse OCP - Negativ-Beispiel: Aufruf des Interfaces durch CreateWorkoutUseCase



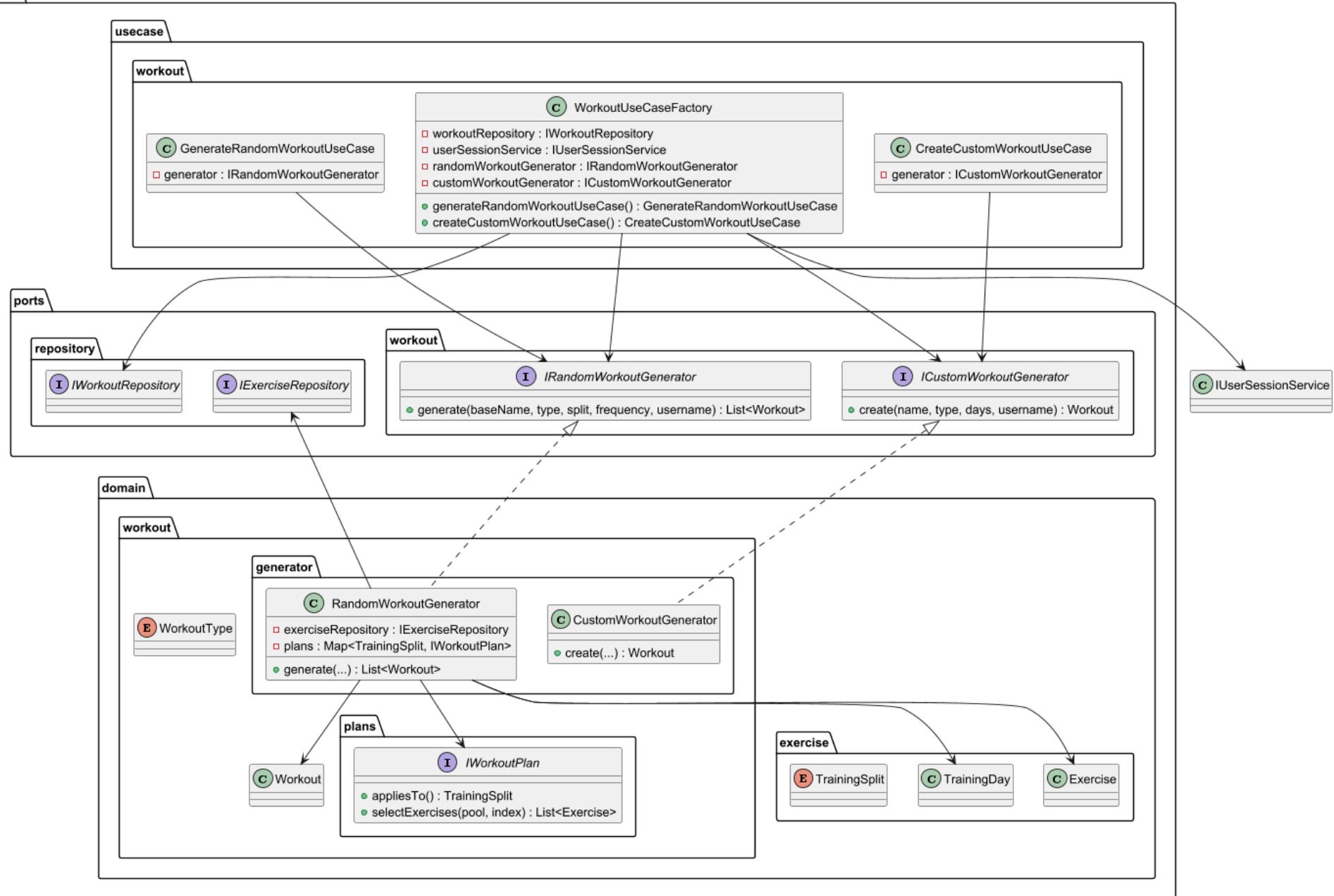
Die ursprüngliche Implementierung der Klasse `SmarterWorkoutGenerator` enthält in sich mehrere private Methoden (z. B. `getFullBodyWorkout`, `getUpperLowerWorkout`, `getPushPullLegWorkout`), die fest kodierte Logiken zur Generierung verschiedener Workout-Typen beinhalten.

**Analyse:** Möchte man einen neuen Workout-Typ hinzufügen, muss die Klasse direkt geändert werden (z. B. durch Hinzufügen neuer Methoden oder Anpassen der switch/case-Logik). Dadurch ist die Klasse nicht geschlossen für Änderungen, sondern muss modifiziert werden, was gegen das OCP verstößt.

# LÖSUNG

## OCP-konformes Design mit WorkoutUseCaseFactory

core



#### #### Warum ist das neue Design besser?

- **Open-Closed Principle (OCP):** Neue Generator-Strategien (z. B. AIWorkoutGenerator) können ergänzt werden, ohne bestehende Klassen zu ändern.
- **Use Cases sind entkoppelt:** Sie kommunizieren nur über Interfaces (IRandomWorkoutGenerator, ICustomWorkoutGenerator).
- **Zentrale Erstellung durch Factory:** Der WorkoutUseCaseFactory verwaltet alle Generatoren und übergibt sie korrekt an die UseCases.
- **Testbarkeit & Erweiterbarkeit:** Die Generatoren können separat getestet oder durch neue ersetzt werden, z. B. durch Mocks in Tests.
- **Domain Layer bleibt unabhängig:** Die Generatoren befinden sich in der Domain und sind frei von UI, DB oder Framework-Abhängigkeiten.
- Keine Änderung an bestehenden Klassen nötig – nur Registration in Factory.

# ANALYSE [LSP/ISP/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]

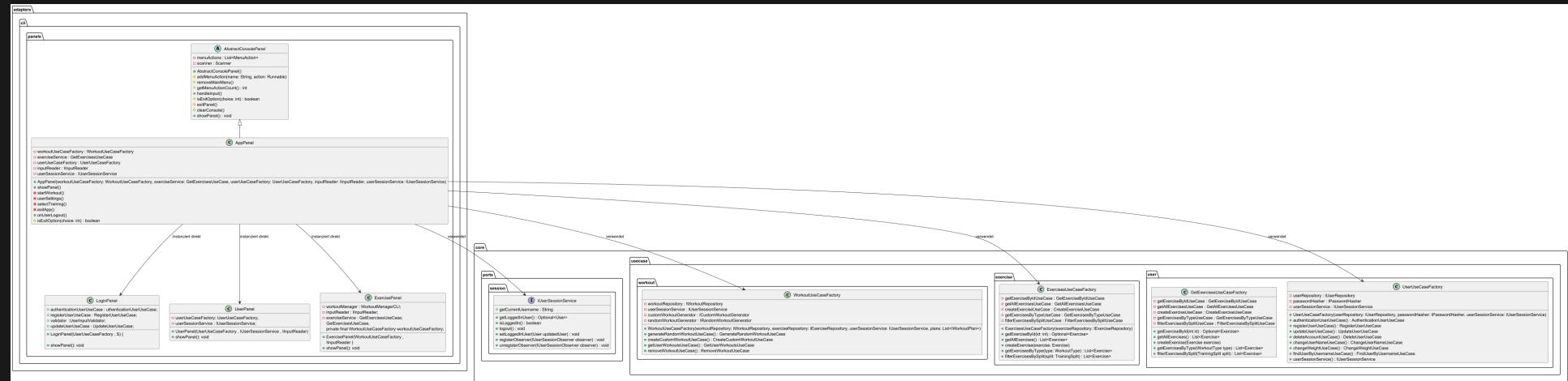
[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

# POSITIV-BEISPIEL DIP

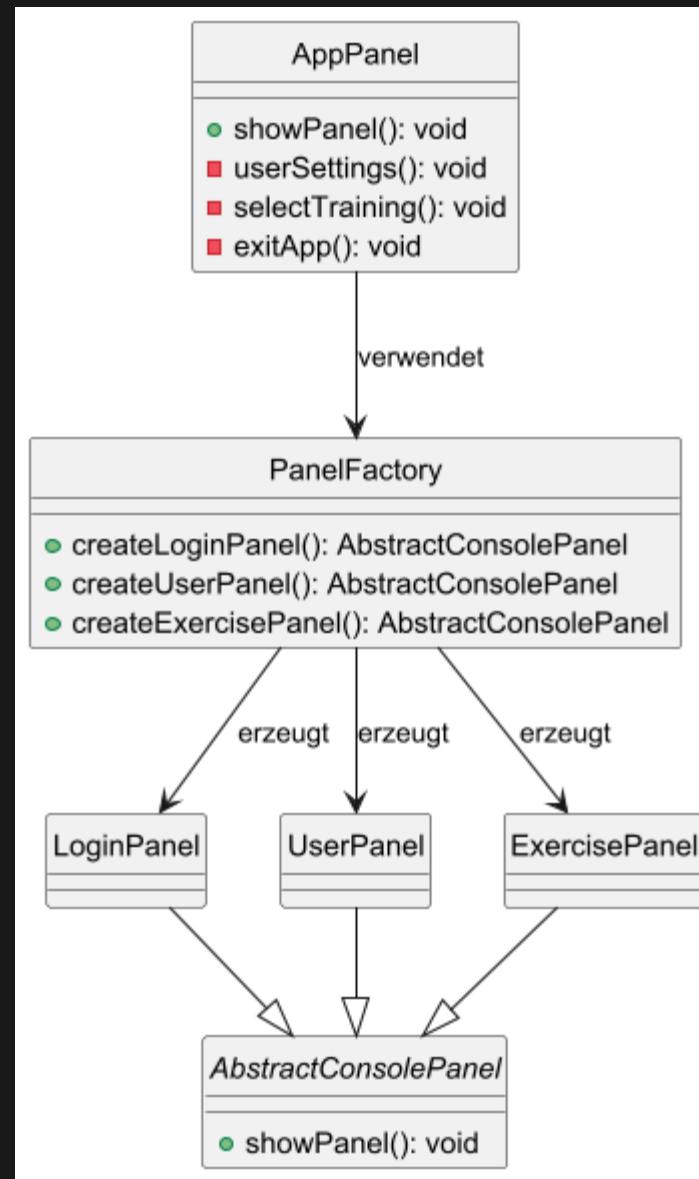
GetUserWorkoutsUseCase - Positives Beispiel für DIP mit Packages



# NEGATIV-BEISPIEL DIP



# LOSUNG

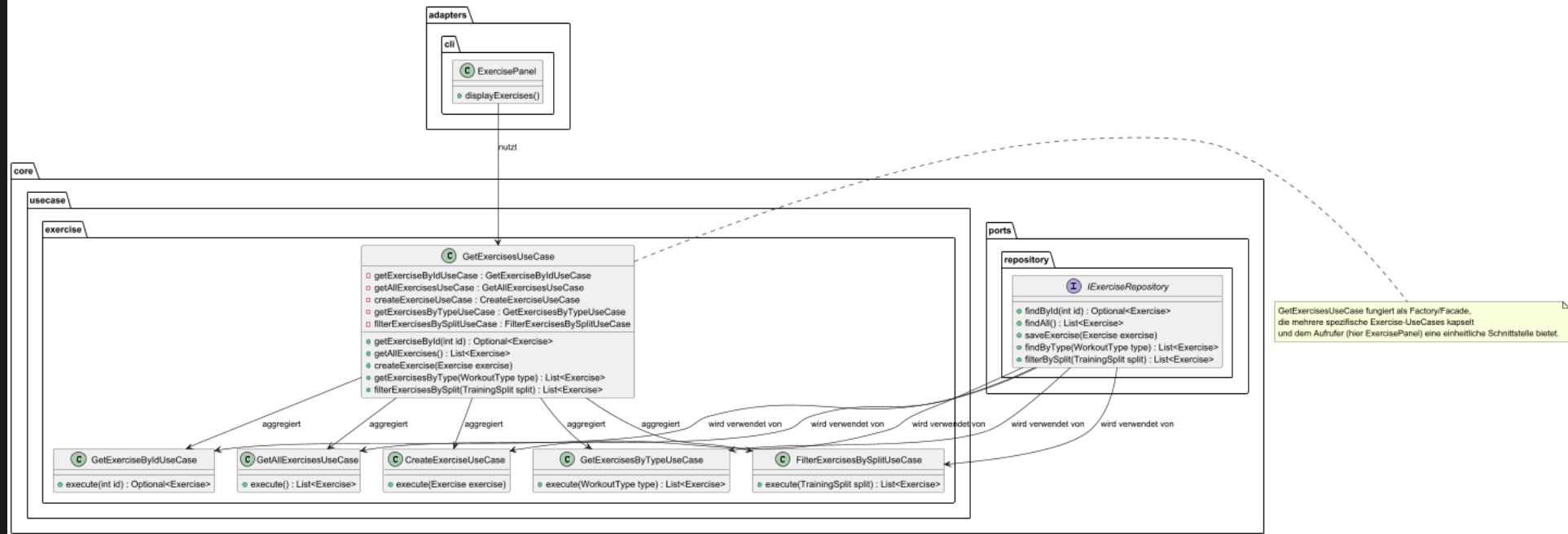


# KAPITEL 4: WEITERE PRINZIPIEN (8P)

# 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]

GRASP - GetExercisesUseCase als Creator/Facade und Verwendung



Beschreibung: Der GetExercisesUseCase übernimmt in diesem Beispiel die Rolle einer Factory bzw. Fassade, indem er in seinem Konstruktor alle spezifischen Exercise-UseCases (wie GetExerciseByIdUseCase, GetAllExercisesUseCase, etc.) initialisiert. Dadurch wird dem aufrufenden Element – hier einem beispielhaften ExercisePanel – eine einheitliche Schnittstelle bereitgestellt, um unterschiedliche Operationen (Abrufen, Erstellen, Filtern) durchzuführen.

## Begründung:

Geringe Kopplung: Das ExercisePanel muss sich nicht um die Details der einzelnen Exercise-UseCases kümmern, sondern nutzt lediglich die über den GetExercisesUseCase bereitgestellten Methoden.

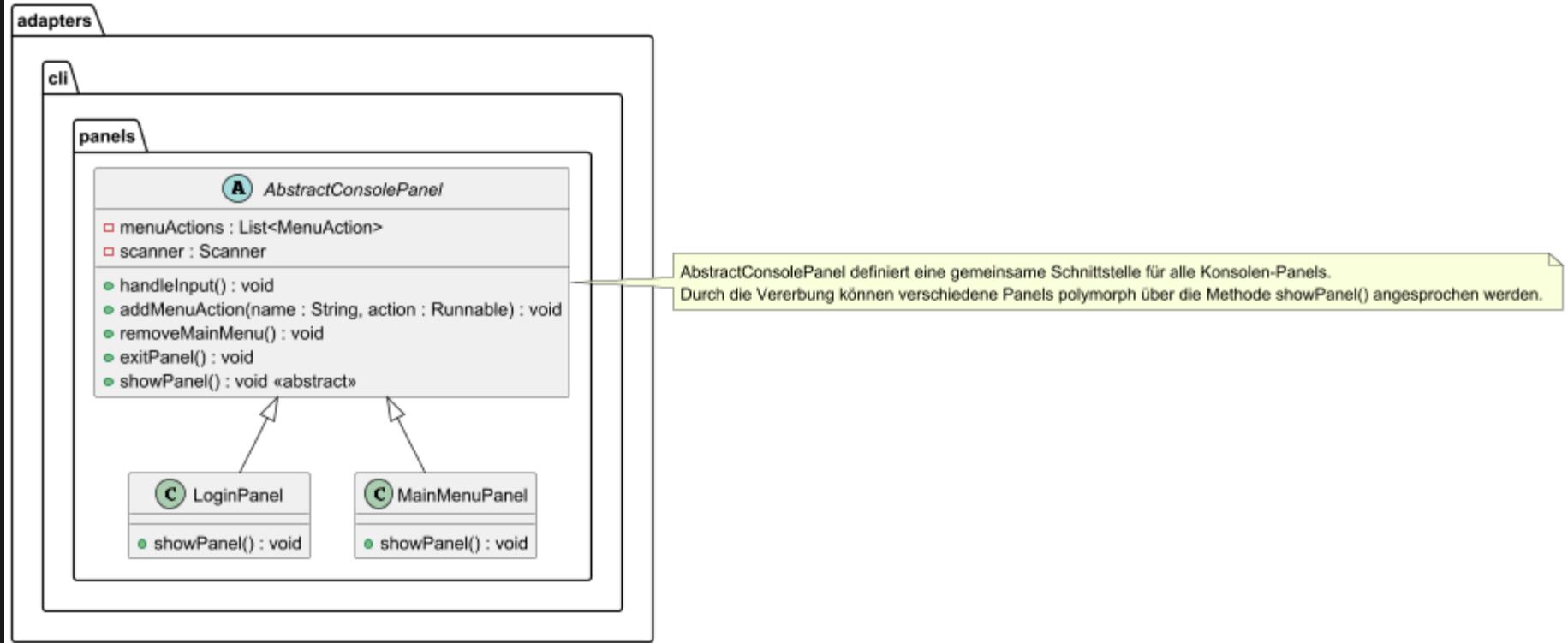
Creator/Facade-Prinzip: GetExercisesUseCase ist verantwortlich für die Erstellung und Aggregation der spezifischen Use Cases, was den Creator-Grundsatz aus GRASP widerspiegelt.

Wartbarkeit und Erweiterbarkeit: Änderungen in einem der spezifischen Use Cases wirken sich nicht direkt auf die Aufrufer aus, da der GetExercisesUseCase als zentrale Schnittstelle fungiert.

# ANALYSE GRASP: [POLYMORPHISMUS/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]

## GRASP - Polymorphismus: AbstractConsolePanel



Die abstrakte Klasse AbstractConsolePanel legt grundlegende Funktionen (z. B. handleInput, addMenuItem, removeMainMenu, exitPanel) für alle Konsolen-Panels fest. Konkrete Panels wie LoginPanel und MainMenuPanel erben von AbstractConsolePanel und implementieren die abstrakte Methode showPanel().

# 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]

# d76db6ff21a7d087ecac6a7be4f2a4c519c10f4b

## 1. Alte Version (Vorher)

- Direkter Zugriff auf Scanner:
  - Scanner scanner = new Scanner(System.in);
  - Scanner-Objekt wurde lokal in jeder Methode genutzt.
- Duplizierte Logik für Workout-Typ Auswahl:
  - In mehreren Methoden (z.B. individuellWorkout() und randomWorkout()) wurde der Typ jedes Mal durch eigene Konsolenabfragen ermittelt.



```
1  private void individuellWorkout()
2  {
3      System.out.print("Gib dem Workout einen Namen: ");
4      String name = scanner.nextLine();
5
6      System.out.println("Wähle Workout-Typ:");
7      System.out.println("1. Kraftsport");
8      System.out.println("2. Cardio");
9      System.out.println("3. Yoga");
10     System.out.print("Deine Wahl: ");
11     int choice = scanner.nextInt();
12     scanner.nextLine();
13
14     WorkoutType type;
15     ...
16 }
```

# LÖSUNG

```
private WorkoutType readWorkoutType()
{
    System.out.println("Wähle Workout-Typ:");
    System.out.println("1. Kraftsport");
    System.out.println("2. Cardio");
    System.out.println("3. Yoga");
    int choice = inputReader.readInt("Deine Wahl: ");
    return switch (choice)
    {
        case 1 → WorkoutType.KRAFTSPORT;
        case 2 → WorkoutType.CARDIO;
        case 3 → WorkoutType.YOGA;
        default → null;
    };
}
```

# KAPITEL 5: UNIT TESTS (8P)

# 10 UNIT TESTS (2P)

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

Testname	Beschreibung
shouldRemoveExistingWorkout	Testet, ob ein bestehendes Workout erfolgreich entfernt wird.
shouldReturnEmptyListForUserWithoutWorkouts	Überprüft, ob ein Benutzer ohne Workouts eine leere Liste erhält.
shouldReturnWorkoutsForUserWithWorkouts	Stellt sicher, dass Workouts korrekt dem eingeloggten Benutzer zugeordnet zurückgegeben werden.

**shouldRegisterNewUserSuccessfully**

Testet, ob ein neuer Benutzer erfolgreich registriert wird.

**shouldNotRegisterDuplicateUser**

Überprüft, dass die Registrierung fehlschlägt, wenn der Benutzername bereits vergeben ist.

**shouldDeleteUserAndLogoutIfLoggedIn**

Testet das Löschen eines Benutzers inklusive Logout, wenn dieser eingeloggt ist.

**shouldDeleteUserButDoNothingIfNotLoggedIn**

Überprüft das Löschen eines Benutzers, ohne Logout-Aktion wenn nicht eingeloggt.

**shouldNotThrowIfUserDoesNotExist**

Stellt sicher, dass keine Exception geworfen wird, wenn ein nicht existierender Benutzer gelöscht werden soll.

**shouldAddExerciseToRepository**

Testet, ob eine Übung korrekt ins Repository eingefügt wird.

**pushSplitReturnsCorrectCategories**

Überprüft, ob der Push-Split nur Übungen mit Brust, hinterer Schulter und Trizeps zurückgibt.

# 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

- Automatisierte Tests werden kontinuierlich beim **Build- oder Deployment-Prozess** ausgeführt.
- Einsatz eines **Testframeworks** zur Ausführung von Unit-Tests.
- Integration der Tests in ein **CI-System** (GitHub Actions ).
- Ziel: **Früherkennung** von Fehlern in zentralen Bereichen der Codebasis.
- Fokus auf **Domain-Modelle** und **Kern-UseCases** zur Absicherung der Geschäftslogik.
- Tests laufen **vollständig automatisch**, ohne manuelle Eingriffe.

# THOROUGH

33.9% generelle Coverage



Coverage on 1.3k Lines to cover



Unit Tests



Duplications on 2.8k Lines



Duplicated Blocks

- Fokussierung auf den Core 68.2% Coverage
  - UseCases
  - Domain

# PROFESSIONAL

- **Naming-Conventions:** Alle Testfälle wurden sprechend benannt nach dem Muster `shouldDoX_WhenY`.
- **Teststrukturierung:** Tests wurden nach Modulen und Schichten getrennt organisiert (Domain, UseCase, Adapter).
- Verwendung von Einheitlichen TestObjekten durch einen Builder.

# FAKES UND MOCKS (4P)

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

Bei Unit-Tests der Domänenlogik wollen wir reine Business-Regeln prüfen, ohne dass eine echte Persistenz (SQL, JPA ...) beteiligt ist.

Ein *Fake* implementiert das produktive Interface (`IWorkoutRepository`) vollständig, verhält sich funktional korrekt, speichert die Daten aber nur im Arbeitsspeicher. So bleiben Tests schnell, deterministisch und frei von I/O-Seiteneffekten



```
1 package core.ports.repository;
2
3 import core.domain.exercise.Exercise;
4 import core.domain.workout.WorkoutType;
5
6 import java.util.HashMap;
7 import java.util.List;
8 import java.util.Map;
9 import java.util.Optional;
10 import java.util.stream.Collectors;
11
12 public class InMemoryExerciseRepositoryMock implements IExerciseRepository {
13     private final Map<Integer, Exercise> storage = new HashMap<>();
14
15     @Override
16     public Optional<Exercise> findById(int id) {
17         return Optional.ofNullable(storage.get(id));
18     }
19
20     @Override
21     public List<Exercise> findAll() {
22         return List.copyOf(storage.values());
23     }
24
25     @Override
26     public void addExercise(Exercise exercise) {
27         storage.put(exercise.id(), exercise);
28     }
29
30     @Override
31     public Optional<List<Exercise>> findByType(WorkoutType type) {
32         List<Exercise> filteredExercises = storage.values().stream()
33                                         .filter(exercise -> exercise.category().matchesWorkoutType(type))
34                                         .collect(Collectors.toList());
35
36         return filteredExercises.isEmpty() ? Optional.empty() : Optional.of(filteredExercises);
37     }
38
39     @Override
40     public boolean removeExercise(int id) {
41         return storage.remove(id) != null;
42     }
43
44     public void clear() {
45         storage.clear();
46     }
47
48     public int count() {
49         return storage.size();
50     }
51 }
```

**core****ports****repository****IExerciseRepository**

- findById(int): Optional<Exercise>
- findAll(): List<Exercise>
- addExercise(Exercise)
- findByType(WorkoutType): Optional<List<Exercise>>
- removeExercise(int): boolean

**InMemoryExerciseRepository**

- findById(int): Optional<Exercise>
- findAll(): List<Exercise>
- addExercise(Exercise)
- findByType(WorkoutType): Optional<List<Exercise>>
- removeExercise(int): boolean

**InMemoryExerciseRepositoryMock**

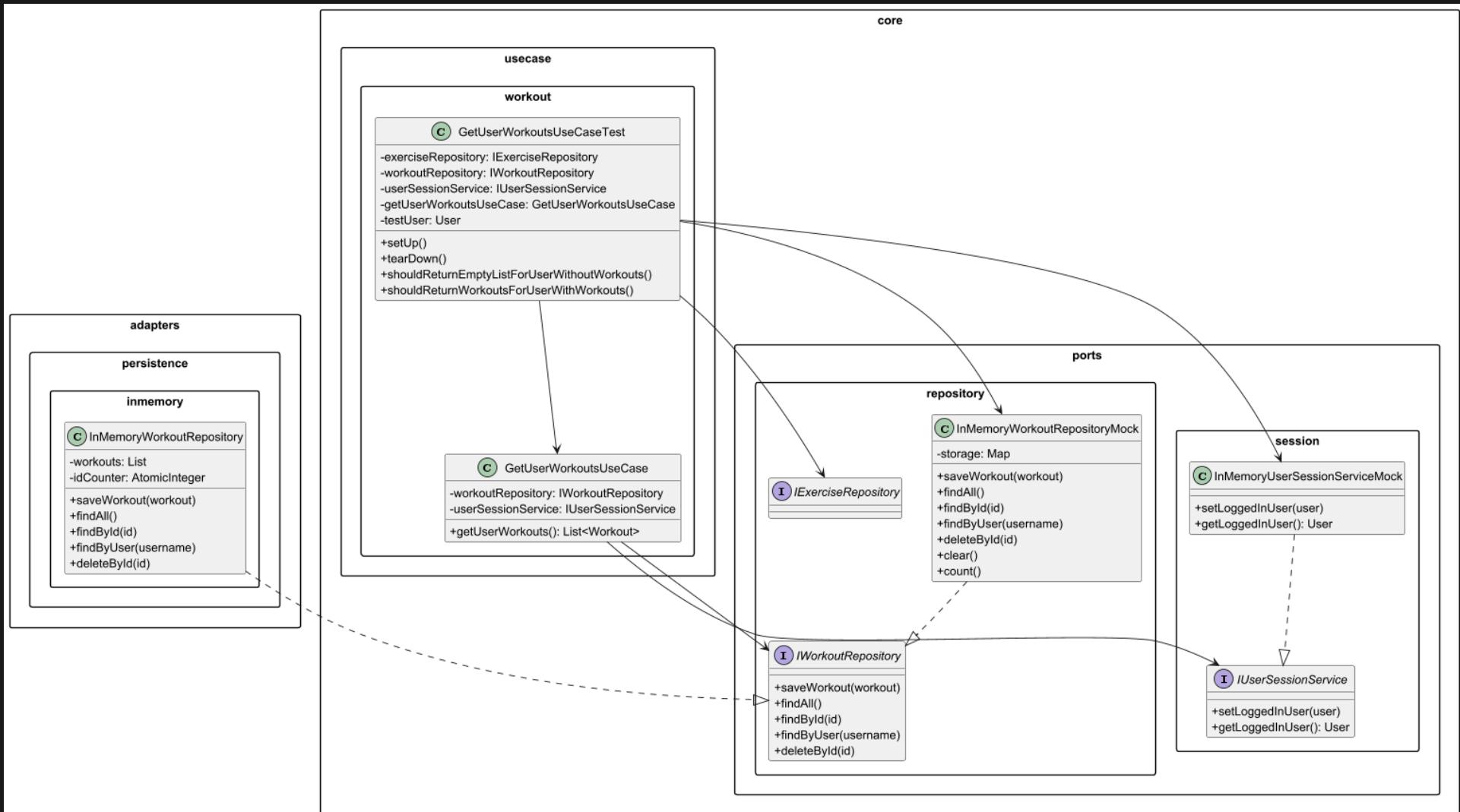
- findById(int): Optional<Exercise>
- findAll(): List<Exercise>
- addExercise(Exercise)
- findByType(WorkoutType): Optional<List<Exercise>>
- removeExercise(int): boolean
- clear(): void
- count(): int

**util****TestObjectBuilder**

- exerciseRepo: InMemoryExerciseRepositoryMock
- withDefaultExercises(): TestObjectBuilder
- withExercise(Exercise): TestObjectBuilder
- clearAll(): void

verwendet

```
1 package core.ports.repository;
2
3 import core.domain.workout.Workout;
4
5 import java.util.HashMap;
6 import java.util.List;
7 import java.util.Map;
8 import java.util.Optional;
9
10 public class InMemoryWorkoutRepositoryMock implements IWorkoutRepository {
11     private final Map<Integer, Workout> storage = new HashMap<>();
12
13     @Override
14     public void saveWorkout(Workout workout) {
15         storage.put(workout.id(), workout);
16     }
17
18     @Override
19     public List<Workout> findAll() {
20         return List.copyOf(storage.values());
21     }
22
23     @Override
24     public Optional<Workout> findById(int id) {
25         return Optional.ofNullable(storage.get(id));
26     }
27
28     @Override
29     public List<Workout> findByUser(String username) {
30         return storage.values().stream()
31             .filter(w -> w.username().equals(username))
32             .toList();
33     }
34
35     @Override
36     public boolean deleteById(int id) {
37         return storage.remove(id) != null;
38     }
39
40     public void clear() {
41         storage.clear();
42     }
43
44     public int count() {
45         return storage.size();
46     }
47 }
48
```



# KAPITEL 6: DOMAIN DRIVEN DESIGN (8P)

# UBIQUITOUS LANGUAGE (2P)

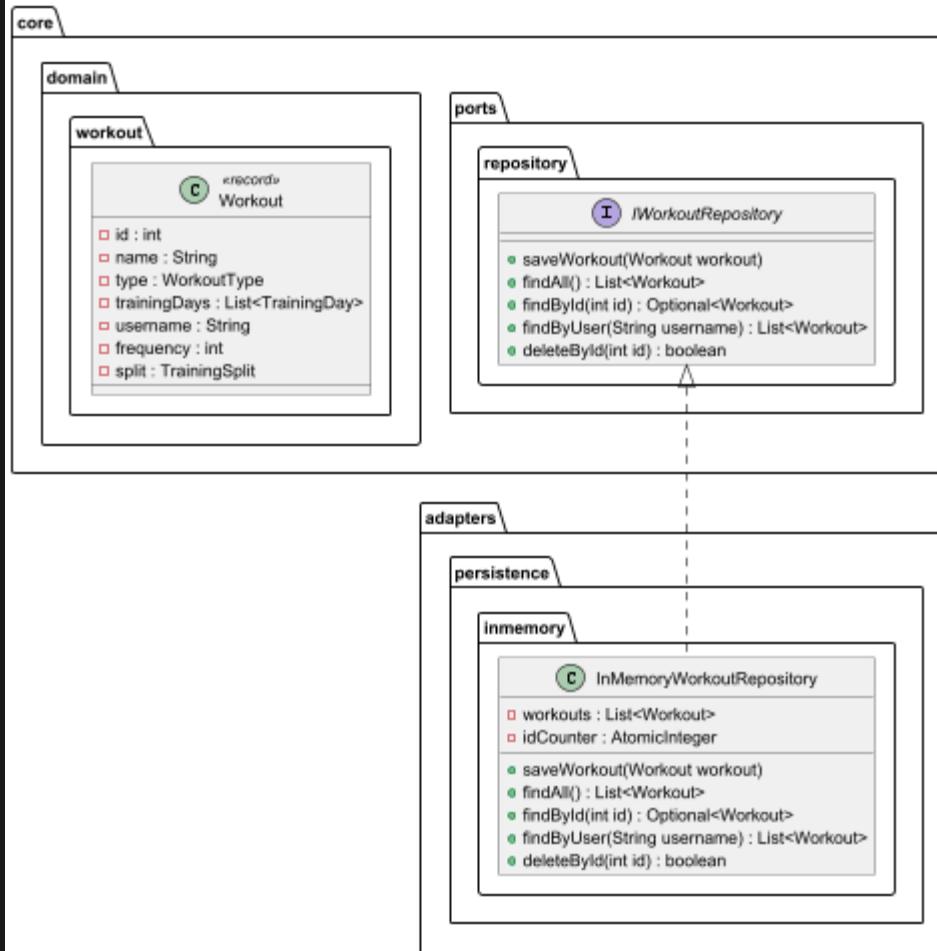
[4 Beispiele für die Ubiquitous Language; jeweils Bezeichnung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]

Bezeichnung	Bedeutung	Begründung
User	Repräsentiert einen Endnutzer der Applikation	Der Begriff „User“ wird durchgängig verwendet, um Personen zu bezeichnen, die Workouts planen und ausführen.
Workout	Eine Trainingseinheit oder -einheiten, die dem Nutzer zugeordnet sind	„Workout“ beschreibt präzise die Kerndaten und Aktionen im System, etwa zur Trainingsplanung.
Exercise	Eine einzelne Übung, die Teil eines Workouts sein kann	Der Begriff hebt die feinere Granularität innerhalb eines Workouts hervor und fördert ein gemeinsames Verständnis der Trainingsbestandteile.
TrainingSplit	Eine Aufteilung oder Strukturierung der Übungen innerhalb eines Trainingsplans	„TrainingSplit“ beschreibt, wie Übungen auf verschiedene Trainingstage verteilt werden – ein zentraler Aspekt in der Fitness-Domain.

# REPOSITORIES (1,5P)

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

### Domain Driven Design - Workout Record & InMemoryWorkoutRepository



Beschreibung: Das Repository kapselt den Datenzugriff auf Workouts. Das Interface `IWorkoutRepository` definiert die Operationen zum Speichern, Abrufen und Löschen von Workouts. `InMemoryWorkoutRepository` ist eine konkrete, in-Memory-Implementierung.

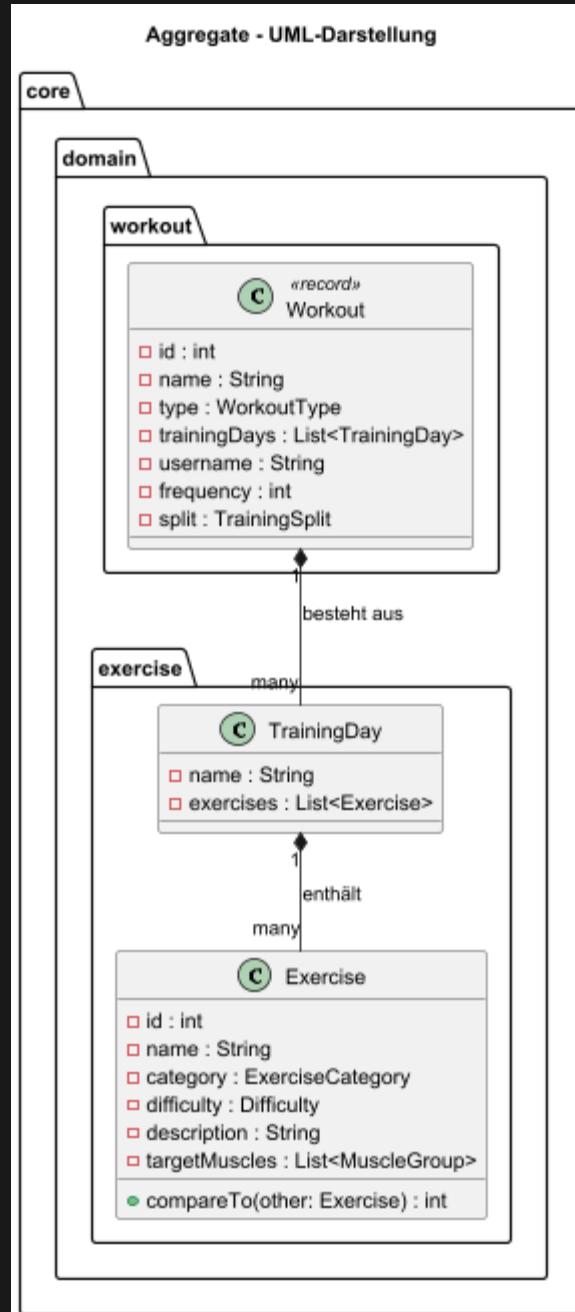
Begründung:

Entkopplung: Die Geschäftslogik muss sich nicht um die Persistenz kümmern.

Testbarkeit: Leicht austauschbar (z. B. durch Mocks). Wartbarkeit: Änderungen im Datenzugriff erfordern nur Anpassungen im Repository.

# AGGREGATES (1,5P)

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



Beschreibung: Das Aggregate besteht im Beispiel aus dem Workout-Record, das alle zugehörigen TrainingDays und damit auch deren Exercises umfasst. Es fasst mehrere zusammengehörige Elemente zu einer konsistenten Einheit zusammen.

Begründung:

Konsistenz: Änderungen innerhalb eines Aggregates (z. B. das Hinzufügen oder Entfernen von TrainingDays) werden als zusammenhängende Einheit behandelt.

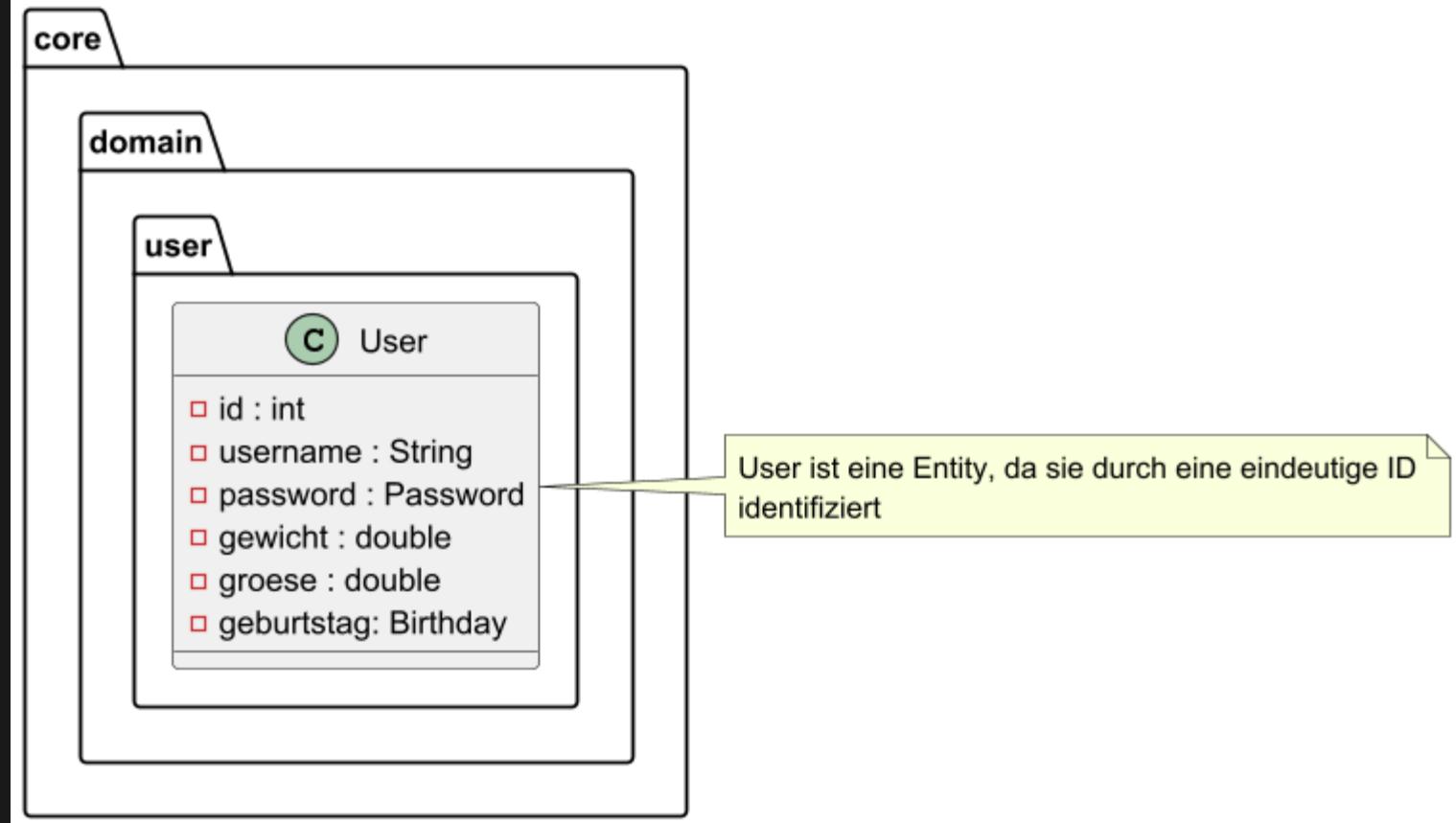
Transaktionsgrenzen: Das Aggregate definiert klare Grenzen, innerhalb derer Datenintegrität gewährleistet werden kann.

Domain Driven Design: Aggregates unterstützen die Modellierung der Domain, indem sie komplexe Zusammenhänge vereinfachen und einen zentralen Zugriffspunkt bieten.

# 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

## Domain Entity - User



Beschreibung: Die Entity User repräsentiert einen Endnutzer in der Domain. Sie besitzt eine eindeutige ID und weitere Attribute wie Username, Password und geburtstag.

Begründung:

Identität: Als Entity wird der User durch seine ID eindeutig identifiziert.

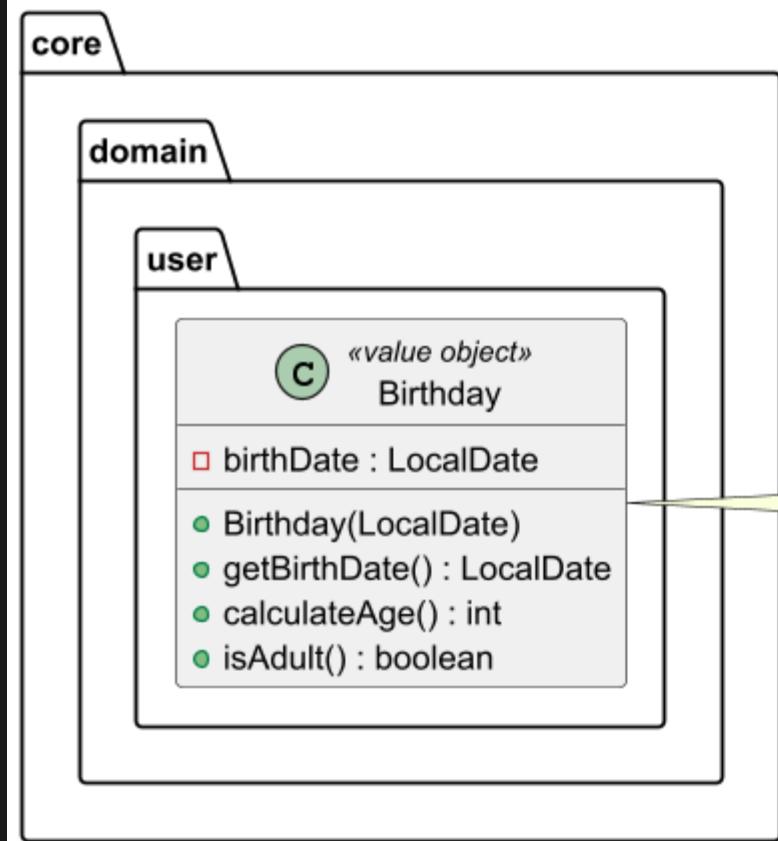
Veränderbarkeit: Eigenschaften wie das Passwort oder die Email können im Laufe der Zeit angepasst werden, ohne dass sich die Identität des Users ändert.

# 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]

## Value Object - Birthday



Birthday ist ein Value Object, da es keine eigene Identität besitzt, unveränderlich ist und ausschließlich durch seinen Wert definiert wird.

Das Value Object Birthday repräsentiert ein Geburtsdatum. Es enthält ein Datum und bietet eine Methode, um das Datum formatiert zurückzugeben.

Begründung:

Unveränderlichkeit: Ein Value Object wie Birthday ändert sich nicht, sondern wird bei Bedarf komplett ersetzt.

Keine eigene Identität: Im Gegensatz zu Entitäten wird Birthday allein durch seinen Wert bestimmt. Konsistente Verwendung: Es sorgt für eine einheitliche Darstellung des Geburtsdatums in der gesamten Domain.

# KAPITEL 7: REFACTORING (8P)

# 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) ]

→ CODE SMELL 1:  
LONG METHOD

- Die Methode `runTrainingSession` übernimmt **zu viele Aufgaben auf einmal**:

- Auswahl des Modus
- Durchlaufen der Trainingsübungen
- Benutzerinteraktion
- Zeiterfassung

- Dadurch ist sie **schwer verständlich, schwer testbar und fehleranfällig**.

```
private void runTrainingSession(Workout workout, int mode)
{
    System.out.println("\n✖ Training gestartet: " + workout.name());
    LocalDateTime start = LocalDateTime.now();
    int completedExercises = 0;

    switch (mode)
    {
        case 1: // Leicht
            for (var trainingDay : workout.trainingDays())
            {
                System.out.println("\n🕒 " + trainingDay.name());
                List<Exercise> exercises = trainingDay.exercises();
                for (int i = 0; i < Math.min(3, exercises.size()); i++) // Nur 3 Übungen
                {
                    Exercise exercise = exercises.get(i);
                    System.out.println("👉 Übung: " + exercise.name() + " (" + exercise.category() + ")");
                    String confirm = inputReader.readLine("Fertig? (j/n): ");
                    if (confirm.equalsIgnoreCase("j"))
                    {
                        completedExercises++;
                    }
                }
            }
            break;

        case 2: // Intensiv
            for (var trainingDay : workout.trainingDays())
            {
                System.out.println("\n🕒 " + trainingDay.name());
                for (Exercise exercise : trainingDay.exercises())
                {
                    System.out.println("👉 Übung: " + exercise.name() + " (" + exercise.category() + ")");
                    String confirm = inputReader.readLine("Fertig? (j/n): ");
                    if (confirm.equalsIgnoreCase("j"))
                    {
                        completedExercises++;
                    }
                    System.out.println("⌚ Mach sofort weiter! Keine Pause erlaubt!");
                }
            }
            break;

        case 3: // Challenge
            for (var trainingDay : workout.trainingDays())
            {
                System.out.println("\n🕒 " + trainingDay.name());
                for (Exercise exercise : trainingDay.exercises())
                {
                    System.out.println("👉 Übung: " + exercise.name() + " (" + exercise.category() + ")");
                    String confirm = inputReader.readLine("Fertig? (j/n): ");
                    if (confirm.equalsIgnoreCase("j"))
                    {
                        completedExercises++;
                    }
                    System.out.println("🔥 Zusatzübung: 20 Burpees! Sofort machen!");
                }
            }
            break;

        default:
            System.out.println("✖ Ungültiger Modus. Abbruch.");
            return;
    }

    LocalDateTime end = LocalDateTime.now();
    System.out.println("\n✔ Training abgeschlossen!");
    System.out.println("Übungen erledigt: " + completedExercises);
    System.out.println("Gestartet um: " + start);
    System.out.println("Beendet um: " + end);
}
}
```

# LÖSUNG

```
private void runTrainingSession(Workout workout, int mode)
{
    LocalDateTime start = startTraining(workout);

    switch (mode)
    {
        case 1 → runEasyMode(workout);
        case 2 → runIntenseMode(workout);
        case 3 → runChallengeMode(workout);
        default → handleInvalidMode();
    }

    finishTraining(start);
}

private LocalDateTime startTraining(Workout workout) { /* ... */ }
private void runEasyMode(Workout workout) { /* ... */ }
private void runIntenseMode(Workout workout) { /* ... */ }
private void runChallengeMode(Workout workout) { /* ... */ }
private void finishTraining(LocalDateTime start) { /* ... */ }
```

# CODE SMELL 2: PRIMITIVE OBSESSION

Verwendung von primitiven `int` für den Trainingsmodus in `startTrainingSession()`:

```
System.out.println("\nWähle deinen Trainingsmodus:");
System.out.println("1: Leicht");
System.out.println("2: Intensiv");
System.out.println("3: Challenge");

int mode = inputReader.readInt("Modus (1/2/3): ");
runTrainingSession(selectedWorkout, mode);
```

## → BESCHREIBUNG:

- Der Trainingsmodus wird als **primitiver int-Wert** behandelt (1, 2, 3).
- **Problem:** Keine Typsicherheit, keine klare Semantik → ein Fehler passiert schnell (z.B. mode = 4 wäre möglich).



```
public enum TrainingMode
{
    LIGHT,
    INTENSE,
    CHALLENGE
}

TrainingMode mode = switch (inputReader.readInt("Modus (1/2/3): "))
{
    case 1 → TrainingMode.LIGHT;
    case 2 → TrainingMode.INTENSE;
    case 3 → TrainingMode.CHALLENGE;
    default → throw new IllegalArgumentException("Ungültiger Modus");
};
runTrainingSession(selectedWorkout, mode);
```

# 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]

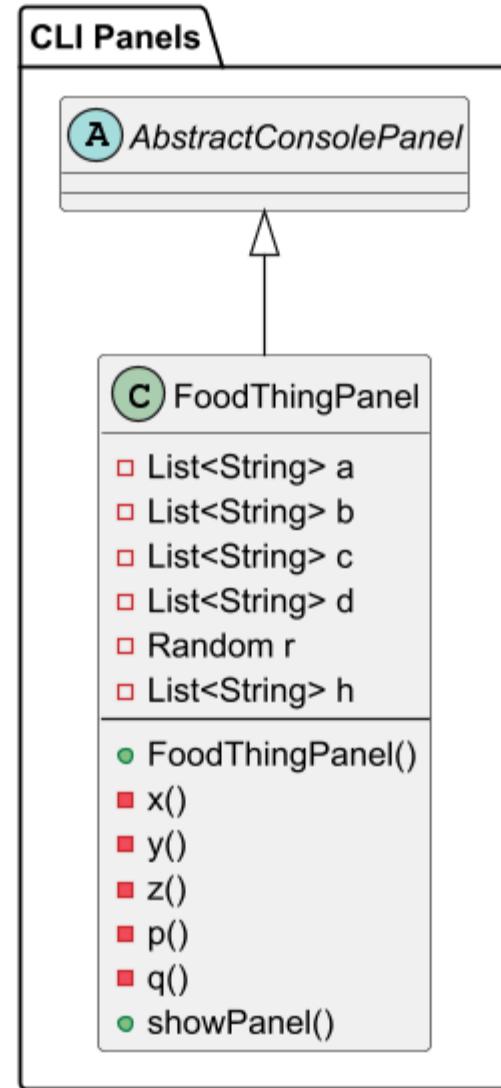
# REFACTORING – RENAME METHOD

367b369978f37a9ea8b561fb1317a69861702c80

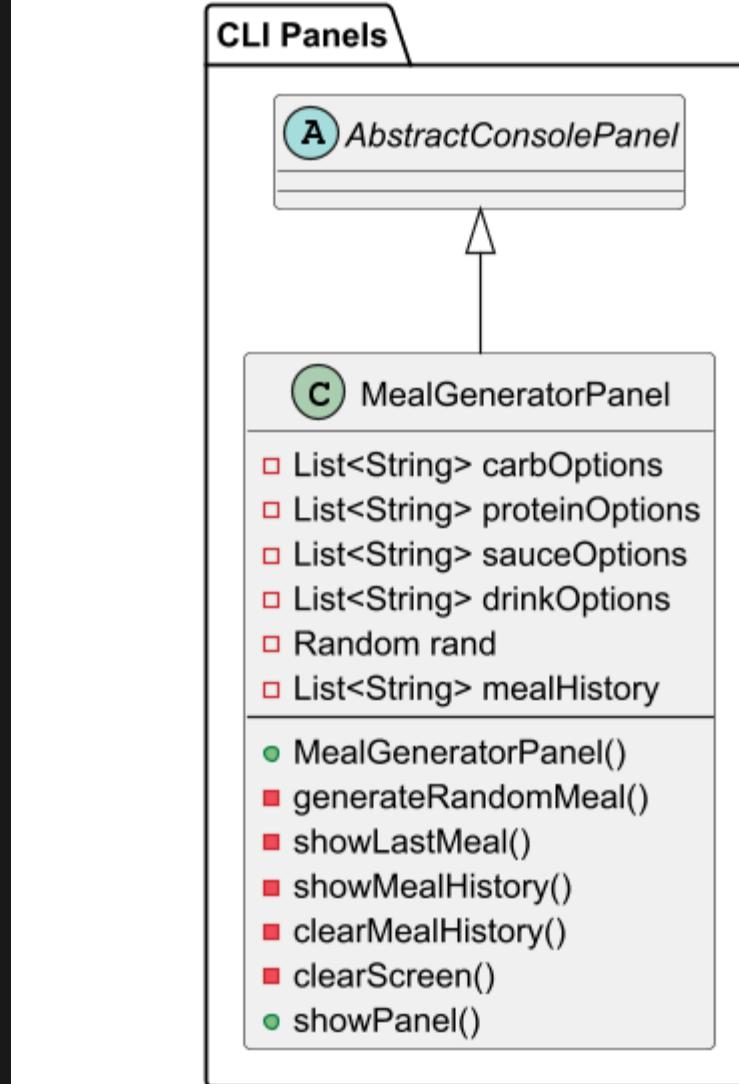
**Vorher:** Kryptische Felder (a,b,c,...), Methoden (x,y,...)  
– Funktion erkennbar nur durch Lesen des Codes.

**Nachher:** Sprechende Bezeichner (carbOptions,  
generateRandomMeal ...) bei unveränderter Logik →  
sofort verständlich

## Bevor Refactoring - FoodThingPanel (bad names)



## Nach Refactoring - MealGeneratorPanel (clear names)



# REFACTORING 1: EXTRACT METHOD + CLASS

Commit 67fb8af

Refactoring: 

- Extract Method in individuellWorkoutPanel(ExercisePanel)"
- Refactoring: Extract class WorkoutManagerCLI aus ExercisePanel"

# ANWENDUNG

In der Klasse `ExercisePanel` war die Methode `individuellWorkoutPanel()` zu lang.  
Sie wurde aufgeteilt in kleinere Methoden:

- `readWorkoutName()`
- `readWorkoutType()`
- `readFrequency()`
- `readTrainingSplit()`
- `collectTrainingDays()`

# BEGRÜNDUNG

- Bessere Lesbarkeit und Verständlichkeit
- Höhere Wartbarkeit und Testbarkeit
- Vermeidung des Code Smells Long Method
- Anwendung des **Single Responsibility Principle (SRP)**

# EXTRACT CLASS

# ANWENDUNG

In der Klasse `ExercisePanel` waren Aufgaben wie Workouts anzeigen, löschen und Details darstellen direkt enthalten.

Diese Verantwortlichkeiten wurden ausgelagert in die neue Klasse `WorkoutManagerCLI`.

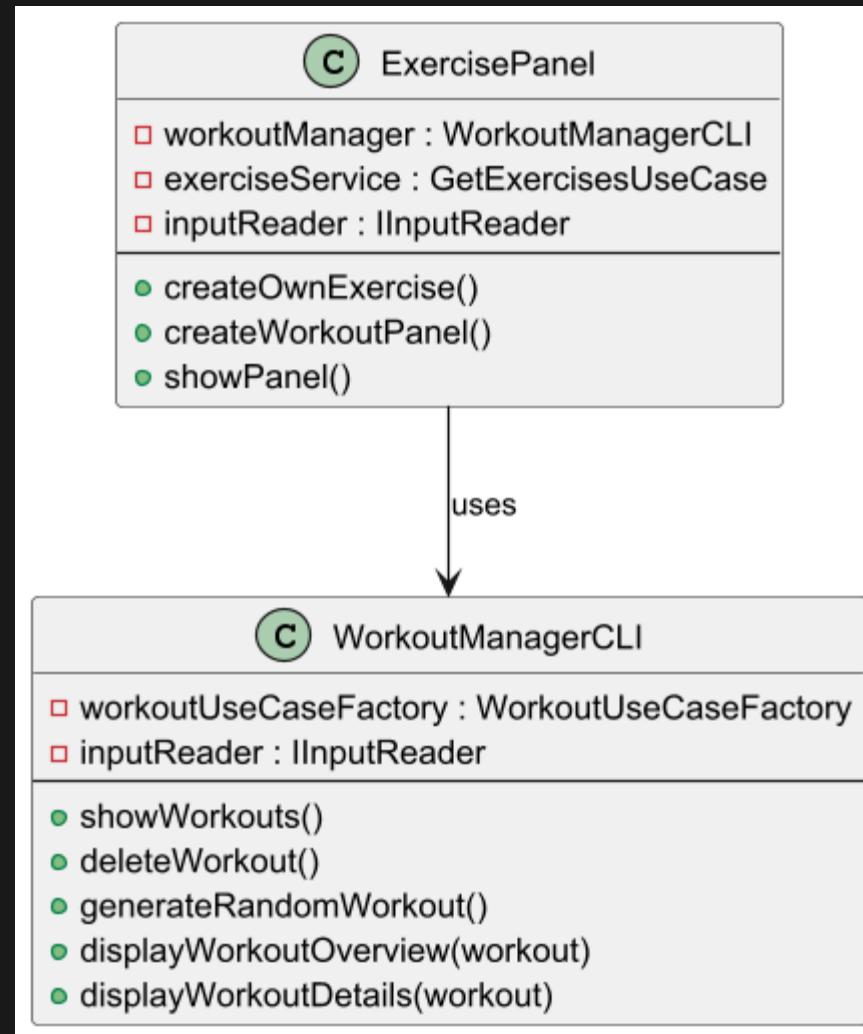
# BEGRÜNDUNG

- Reduzierung der Komplexität in ExercisePanel
- Bessere Strukturierung durch die Trennung der Verantwortlichkeiten
- Anwendung des **Single Responsibility Principle (SRP)**
- Beseitigung des Code Smells **Large Class**

**C**

## ExercisePanel

- workoutUseCaseFactory : WorkoutUseCaseFactory
- exerciseService : GetExercisesUseCase
- inputReader : IInputReader
- showWorkoutsPanel()
- deleteWorkout()
- displayWorkoutOverview()
- displayWorkoutDetails()
- individuellWorkoutPanel()
- randomWorkoutPanel()
- createOwnExercise()
- showPanel()

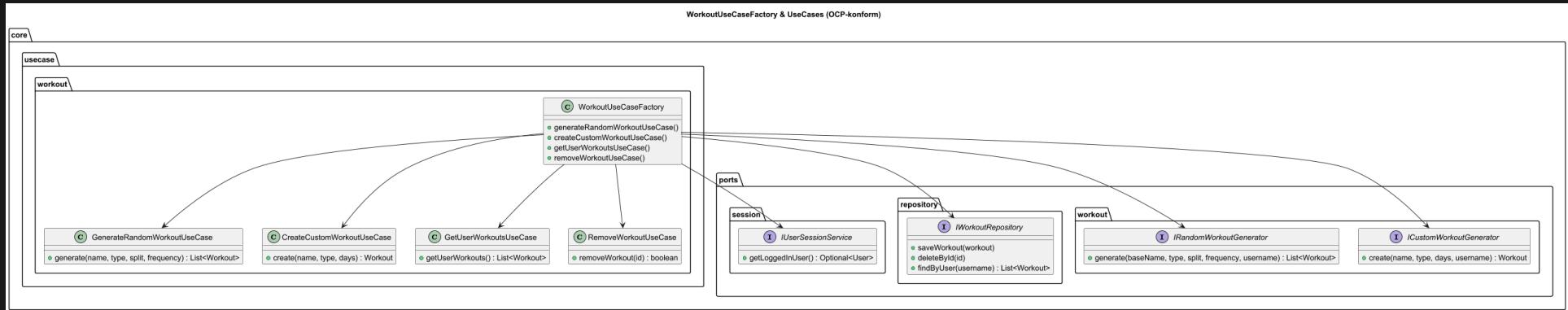


# KAPITEL 8: ENTWURFSMUSTER (8P)

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

# FACTORY

Das Factory Method Pattern wird eingesetzt, um die Erstellung verschiedener Anwendungsfälle (UseCases) zentral zu verwalten. Die `WorkoutUseCaseFactory` kapselt die Instanziierung komplexer Objekte wie `GenerateRandomWorkoutUseCase` oder `CreateCustomWorkoutUseCase`, ohne dass der aufrufende Code Details der Erstellung kennen muss. Dadurch wird der Code flexibler, wartbarer und neue UseCases können später einfach hinzugefügt werden, ohne bestehende Logik zu verändern.



# Observer

Das Observer Pattern wird genutzt, um Logout-Ereignisse zwischen Modulen zu entkoppeln. Wenn ein Benutzer über das UserPanel einen Logout oder eine Account-Lösung ausführt, ruft das UserPanel die `logout()`-Methode des LoggedInUser auf. Dieser informiert daraufhin automatisch alle registrierten Observer. Das AppPanel erhält das Logout-Ereignis und zeigt anschließend das Login-Panel neu an.

