Programmentwurf
FitnessTracker

Name: Kölblin, Jonas
Martrikelnummer: 7150881

Abgabedatum: 31.05.2023

Allgemeine Anmerkungen:

- *Gesamt-Punktzahl: 60P (zum Bestehen mit 4,0 werden 30P benötigt)*
- *die Aufgabenbeschreibung (der blaue Text) und die mögliche Punktanzahl muss im Dokument erhalten bleiben*
- *es darf nicht auf andere Kapitel als alleiniger Leistungsnachweis verwiesen werden (z.B. in der Form "XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung")*
- *alles muss in UTF-8 codiert sein (Text und Code)*
- *das Dokument muss als PDF abgegeben werden*
- *es gibt keine mündlichen Nebenabreden / Ausnahmen – alles muss so bearbeitet werden, wie es schriftlich gefordert ist*
- *alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)*
- *die Beispiele sollten wenn möglich vom aktuellen Stand genommen werden*
  - *finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden*
  - *Ausnahme: beim Kapitel "Refactoring" darf von vorne herein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)*
- *falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden*
  - ***Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele sondern 0,5P Abzug für das fehlende Negativ-Beispiel***
  - *Beispiel*
    - *"Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt." (2P)*
      - *Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]*
      - *Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: 2P ODER falls im Code mind. eine Klasse SRP verletzt: 0,5P*
- *verlangte Positiv-Beispiele müssen gebracht werden – im Zweifel müssen sie extra für die Lösung der Aufgabe implementiert werden*
- *Code-Beispiel = Code in das Dokument kopieren (inkl. Syntax-Highlighting)*
- *falls Bezug auf den Code genommen wird: entsprechende Code-Teile in das Dokument kopieren (inkl Syntax-Highlighting)*
- *bei UML-Diagrammen immer die öffentlichen Methoden und Felder angegeben – private Methoden/Felder nur angeben, wenn sie zur Klärung beitragen*

- *bei UML-Diagrammen immer unaufgefordert die zusammenspielenden Klassen ergänzen, falls diese Teil der Aufgabe sind*
- *Klassennamen/Variablennamen/etc im Dokument so benennen, wie sie im Code benannt sind (z.B. im Dokument nicht anfangen, englische Klassennamen zu übersetzen)*
- *die Aufgaben sind von vorne herein bekannt und müssen wie gefordert gelöst werden – z.B. ist es keine Lösung zu schreiben, dass es das nicht im Code gibt*
  - *Beispiel 1*
    - *Aufgabe: Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten*
    - *Antwort: Es wurden keine Fake/Mock-Objekte gebraucht.*
    - *Punkte: 0P*
  - *Beispiel 2*
    - *Aufgabe: UML, Beschreibung und Begründung des Einsatzes eines Repositories*
    - *Antwort: Die Applikation enthält kein Repository*
    - *Punkte*
      - *falls (was quasi nie vorkommt) die Fachlichkeit tatsächlich kein Repository hergibt: volle Punktzahl*
      - *falls die Fachlichkeit in irgendeiner Form ein Repository hergibt (auch wenn es nicht implementiert wurde): 0P*
  - *Beispiel 3*
    - *Aufgabe: UML von 2 implementierte unterschiedliche Entwurfsmuster aus der Vorlesung*
    - *Antwort: es wurden keine Entwurfsmuster gebraucht/implementiert*
    - *Punkt: 0P*

Kapitel 1: Einführung (4P)

Übersicht über die Applikation (1P)

Die Fitnesstracker-Anwendung ist eine Konsolenanwendung, die es Benutzern ermöglicht, ihre Fitnessaktivitäten zu verfolgen, Trainingspläne zu erstellen und ihre Fortschritte im Laufe der Zeit zu messen. Sie dient dazu, Menschen dabei zu unterstützen, ihre Fitnessziele zu erreichen und ein gesundes Leben zu führen. Die Applikation ermöglicht Benutzern die Durchführung verschiedener Übungen und die Aufzeichnung von Wiederholungen und Gewichten. Benutzer können neue Übungen hinzufügen, Trainingspläneanlegen. Aus Benutzersicht bietet die Anwendung eine interaktive Konsolenoberfläche, über die Benutzer mit der Anwendung interagieren können. Sie können sich registrieren, anmelden, Übungen auswählen, die Anzahl der Wiederholungen und das Gewicht eingeben und die Daten speichern. Die Anwendung zeigt auch Informationen über bereits durchgeführte Übungen und deren Fortschritt an, so dass Benutzer ihren Trainingsverlauf verfolgen können. Die Fitnesstracker-Anwendung löst das Problem der manuellen Verfolgung und Aufzeichnung von Fitnessaktivitäten. Anstatt alles auf Papier oder in separaten Notizen zu notieren, können Benutzer ihre Übungen und Fortschritte bequem und zentral in der Anwendung speichern. Dies ermöglicht eine bessere Organisation, Analyse und Verfolgung der Fitnessaktivitäten im Laufe der Zeit. Die Anwendung fördert somit ein strukturiertes und effektives Training, während sie den Benutzern hilft, motiviert zu bleiben und ihre Ziele zu erreichen.

Starten der Applikation (1P)

Die Applikation kann in Windows durch Ausführen des simplen Skripts "start.bat" und unter Linux durch Ausführen von "start.sh" gestartet warden. Vorraussetzung dafür ist es, dass .Net 6 auf dem Computer installiert ist.

Anleitung:

0. *Ausgangspunkt: Repository-Ordner ("fitnesstracker")*
1. *Navigiere in den Unterordner "fitnesstracker-projekt"*
2. *Je nach Betriebssystem:*
    a. *Windows: starte das Skript "start.bat"*
    b. *Linux: starte das Skript "start.sh"*

Disclaimer: Die Architektur der Anwendung ist nahezu vollständig, aufgrund von zeitlichen Problemen wurden jedoch nicht alle geplanten UseCases vollständig implementiert. Die Anwendung ist nun somit gut um weitere UseCases erweiterbar, da sie alle nötigen Entities, Services und Repositories liefert, in ihrer tatsächlichen Funktionalität aber eingeschränkt. Dennoch wurde versucht das beste aus der Situation zu machen.

Technischer Überblick (2P)

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

Das Projekt wurde als Konsolenanwendung in C# programmiert. Es gab die zwei Sprachen C# und Java zur Auswahl und da Java im Studium bereits zur Genüge behandelt wurde, fiel die Entscheidung auf C#. Geeignet für die Implementierung sind beide Sprachen.

Die Persistierung der Daten erfolgt über Dateien im CSV-Format, ein einfaches Tabellenformat, das unter anderem auch von Excel interpretiert werden kann. Die Wahl fiel auf dieses Format, da es sich gut zur Speicherung fester Datenstrukturen eignet und im Vergleich zu zB. JSON leichter zu verarbeiten ist und weniger Speicherplatz benötigt.

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

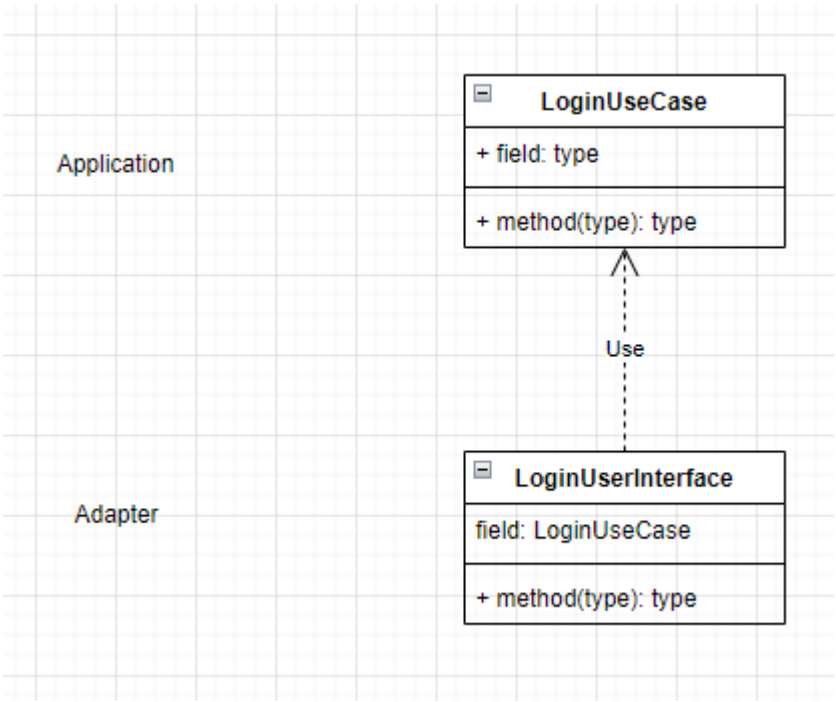
[allgemeine Beschreibung der Clean Architecture in eigenen Worten]

Die Clean Architecture bedeutet die Aufteilung des Quellcodes in die Schichten "Abstraction Code", "Domain Code", "Application Code", "Adapters" und "Plugins". Abhängigkeiten dürfen hierbei nur von außen nach innen, also von "Plugins" in Richtung "Abstraction Code", existieren (Dependency Rule). In meinem Code wurde die äußerste Schicht "Infrastructure" genannt, da sie nur eine Container-Klasse zum Transport der Repositories und Services enthält, daher erschien mir dieser ebenfalls im Bezug zur Clean Architecture im Internet auftauchende Begriff passender. Richtige Plugins gibt es nicht wirklich, da bei der Implementierung auf externe Libraries und Frameworks verzichtet werden sollte.

Analyse der Dependency Rule (3P)

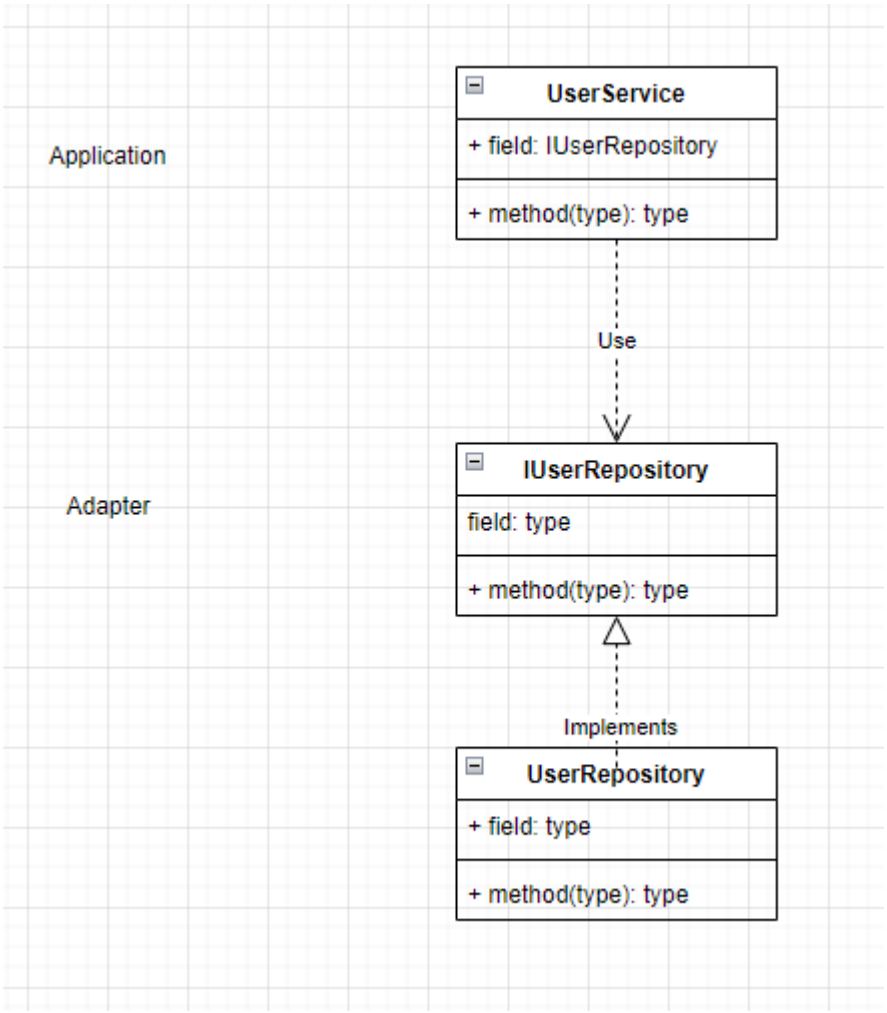
[1 Klasse, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

Positiv-Beispiel:



Die Klasse LoginUserInterface (Adapter-Schicht) hängt von der Klasse LoginUseCase (Application-Schicht) ab, aber nicht umgekehrt. Die Dependency Rule ist somit hier befolgt.

(Halb-)Negativ-Beispiel: Dependency Rule

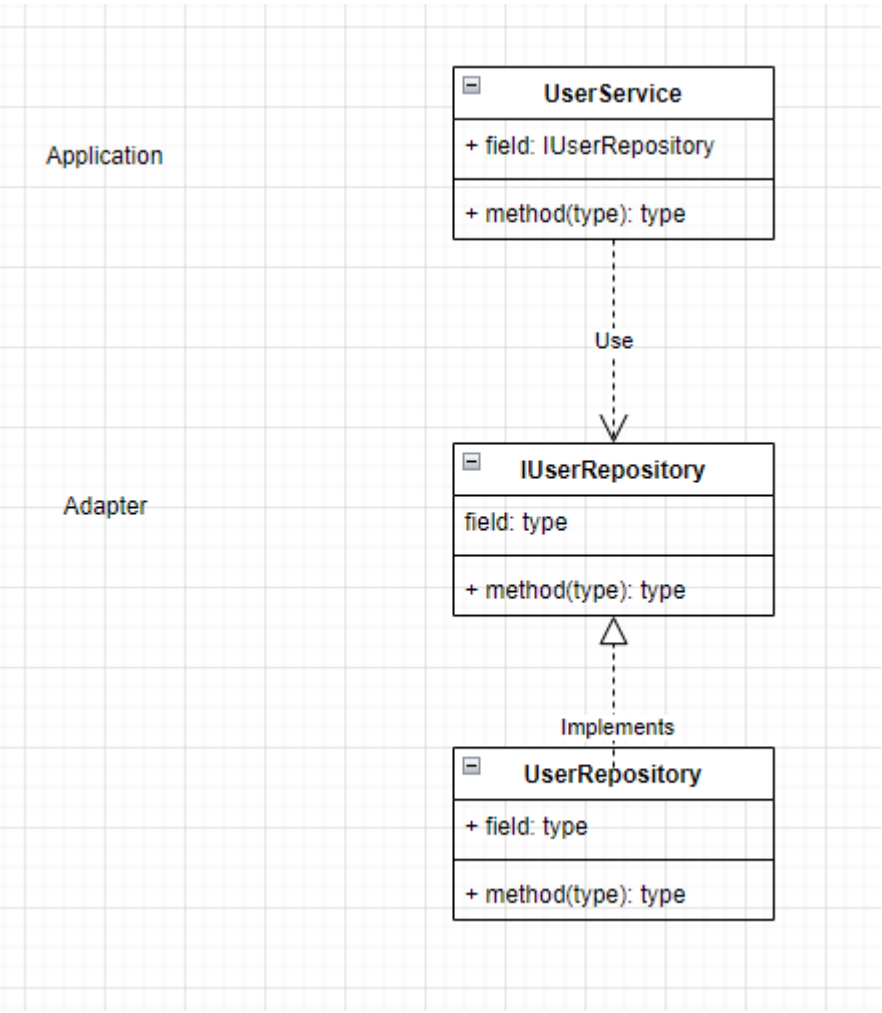


Hier verwendet die Klasse UserService (Application) indirekt die Klasse UserRepository (Adapter) und zwar über das Interface IUserRepository. Durch das Interface ist die Dependency Rule aber dennoch bestmöglich gegeben. Ich stelle hiermit die Behauptung auf, dass in meinem gesamten Code der Zugriff einer inneren Schicht auf eine äußere Schicht überall über ein Interface erfolgt. Falls ein Gegenbeispiel gefunden werden sollte, gebe ich diesen Punkt gerne ab.

Analyse der Schichten (4P)

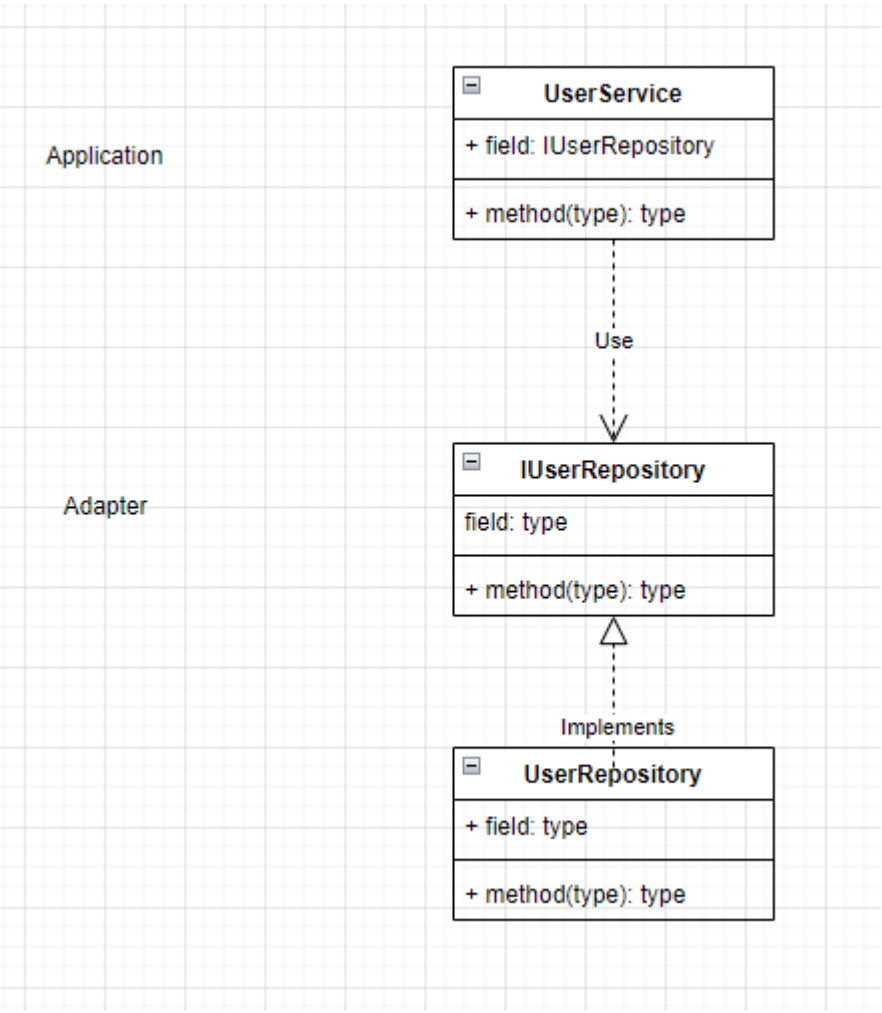
[jeweils 1 Klasse zu 2 unterschiedlichen Schichten der Clean-Architecture: jeweils UML (mind. betreffende Klasse und ggf. auch zusammenspielenden Klassen), Beschreibung der Aufgabe, Einordnung mit Begründung in die Clean-Architecture]

Schicht: Application



UserService: bietet Methoden zur Benutzerverwaltung wie zum Beispiel das Erstellen, Ändern und Finden nach ID oder Benutzername von Benutzern. Gehört in die Application-Schicht, da die Klasse Anwendungsfälle ermöglicht, ohne dabei eine direkte Schnittstelle zur Datenbank oder dem User-Interface zu liefern.

Schicht: Adapter



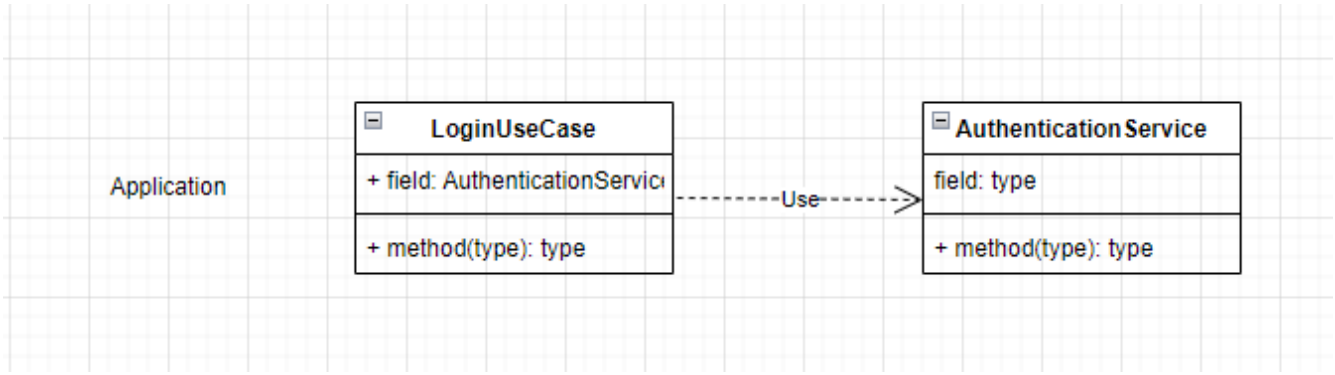
User Repository: bietet Methoden zum vereinfachten Zugriff auf Daten der Datenbank (bzw hier der CSV-Dateien) und gehört damit in die Adapter-Schicht. Ermöglicht die Methoden des UserService.

Kapitel 3: SOLID (8P)

Analyse SRP (3P)

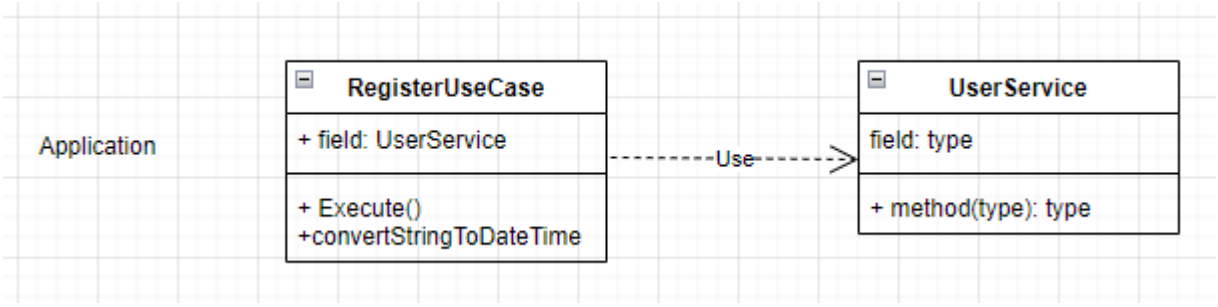
[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

Positiv-Beispiel

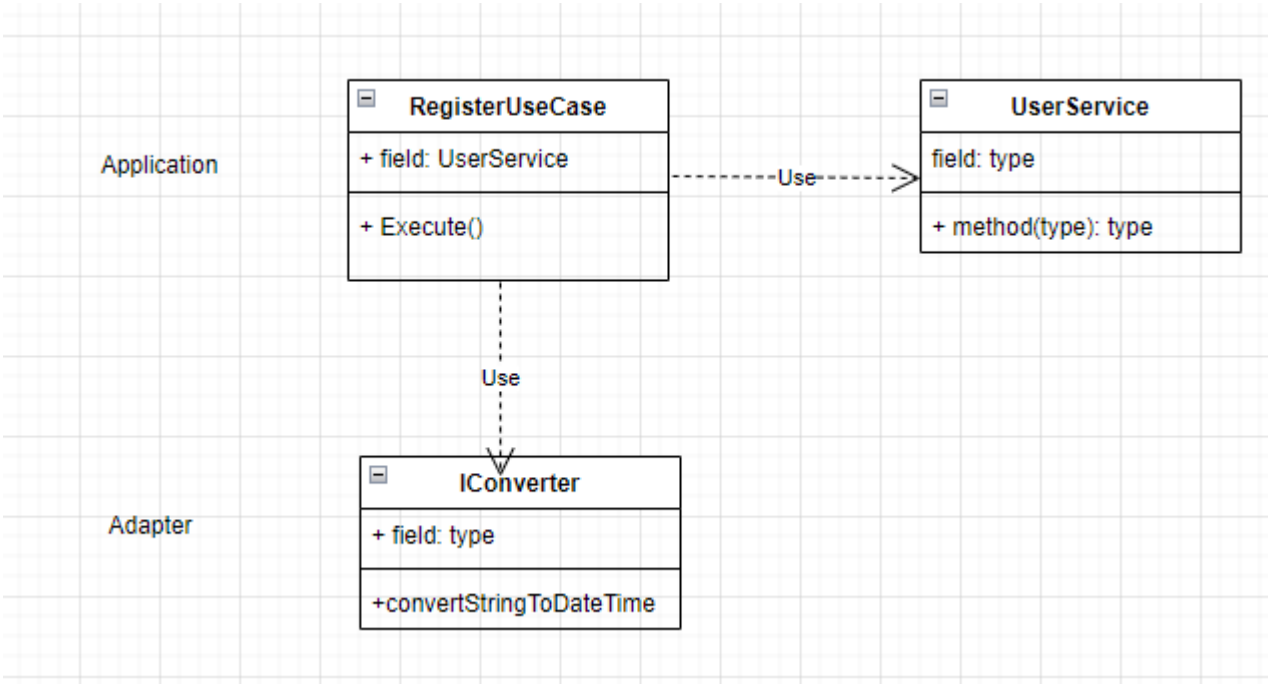


LoginUseCase: diese Klasse hat nur eine einzige Aufgabe und zwar die Ermöglichung des Login-Vorgangs. Hierfür nutzt sie die Klasse AuthenticationService.

Negativ-Beispiel



RegisterUseCase: diese Klasse ermöglicht den Registrierungsvorgang mit der Erstellung eines neuen Benutzers durch einen UserService. Allerdings wandelt sie auch die Benutzereingaben in die für die Benutzererstellung erforderlichen Formate, beispielsweise mit der Methode convertStringToDateTime. Diese Funktionalität sollte eher in die Adapter-Schicht ausgelagert werden. Lösung:

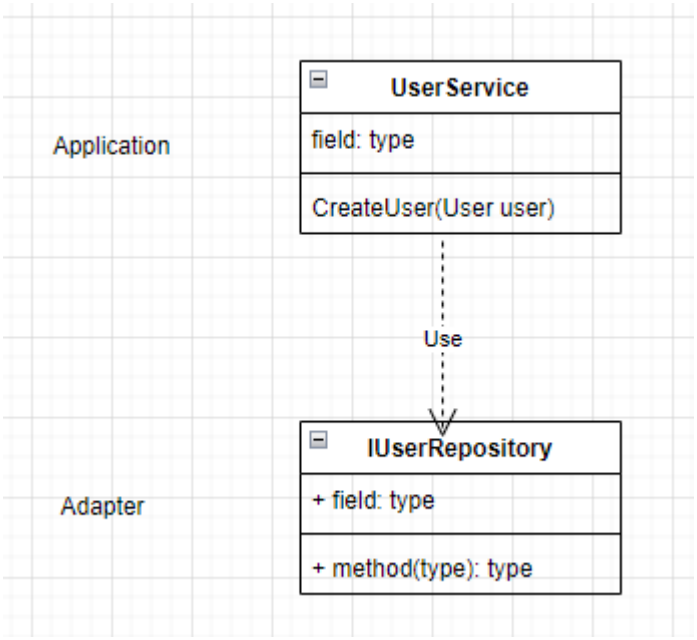


Die Funktionalität wird hier in eine Converter-Klasse ausgelagert, auf die die RegisterUseCase-Klasse über ein Interface zugreift.

Analyse OCP (3P)

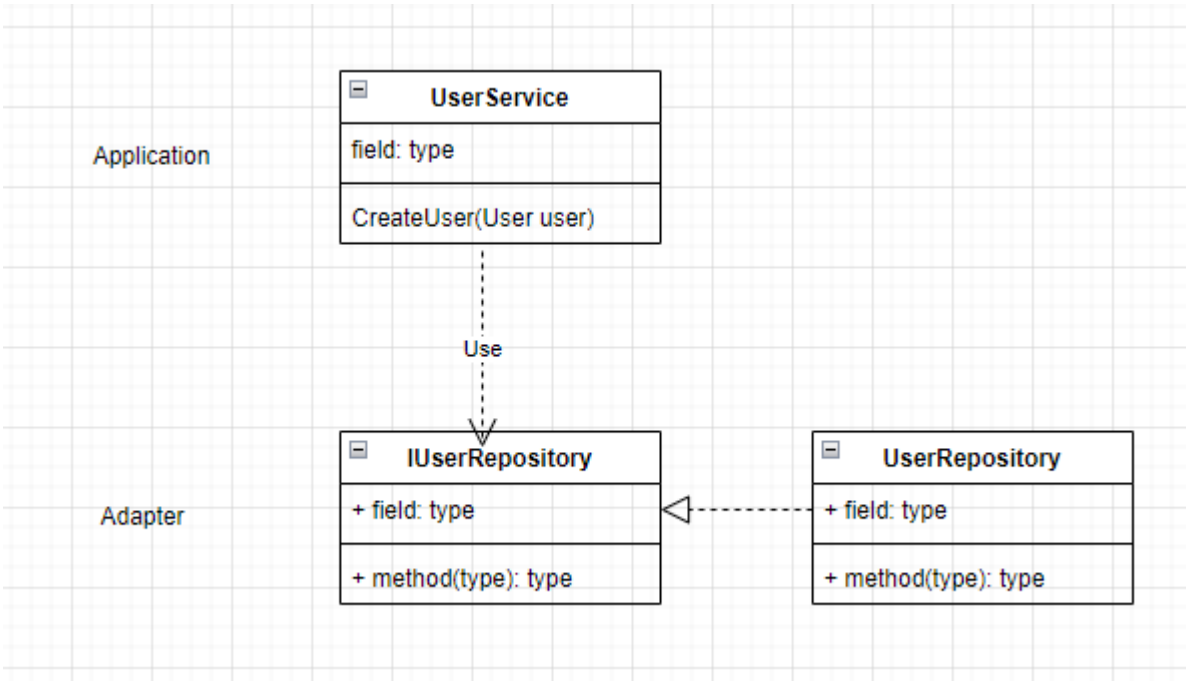
[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]

Positiv-Beispiel



Eine Klasse, die das Open-Closed-Prinzip umsetzt, ist die UserService-Klasse. Diese Klasse enthält eine Methode zum Erstellen eines Users, z. B. CreateUser(User user). Wenn neue Funktionalitäten hinzugefügt werden müssen kann dies erfolgen, indem neue Methoden hinzugefügt werden, die die vorhandene Funktionalität erweitern. Dadurch bleibt die bestehende Methode unverändert und die Klasse erfüllt das OCP. Darüber hinaus setzt die gesamte Application-Schicht das OCP um, da für neue Funktionalitäten einfach eine neue UseCase-Klasse implementiert werden kann.

Negativ-Beispiel



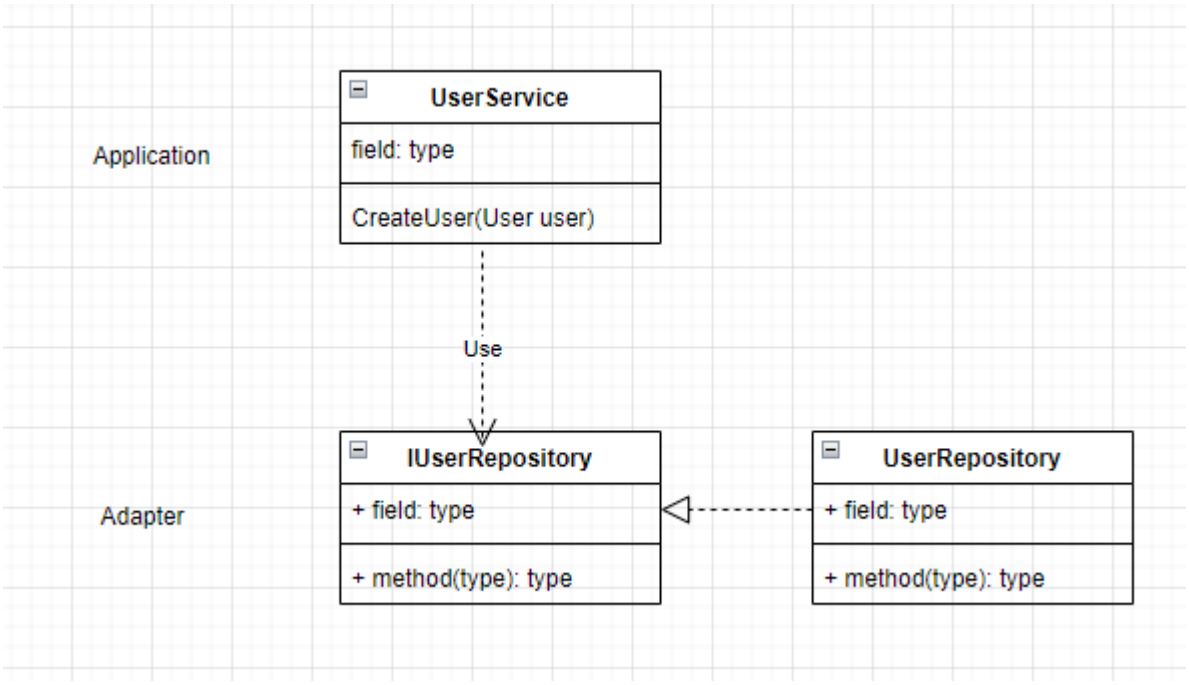
Ein Negativbeispiel könnte eine UserRepository-Klasse sein, die direkt auf eine bestimmte Datenbank zugreift, beispielsweise eine SQL-Datenbank. Wenn sich die Datenbanktechnologie ändern würde, müssten Änderungen an der UserRepository-Klasse vorgenommen werden, um die Verbindung zur neuen Datenbank herzustellen. Daher wurde hier zur Abstraktion ein Interface verwendet, so dass einfach eine andere Klasse programmiert werden kann, die das Interface implementiert ohne dass sich die Schnittstelle ändert

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]

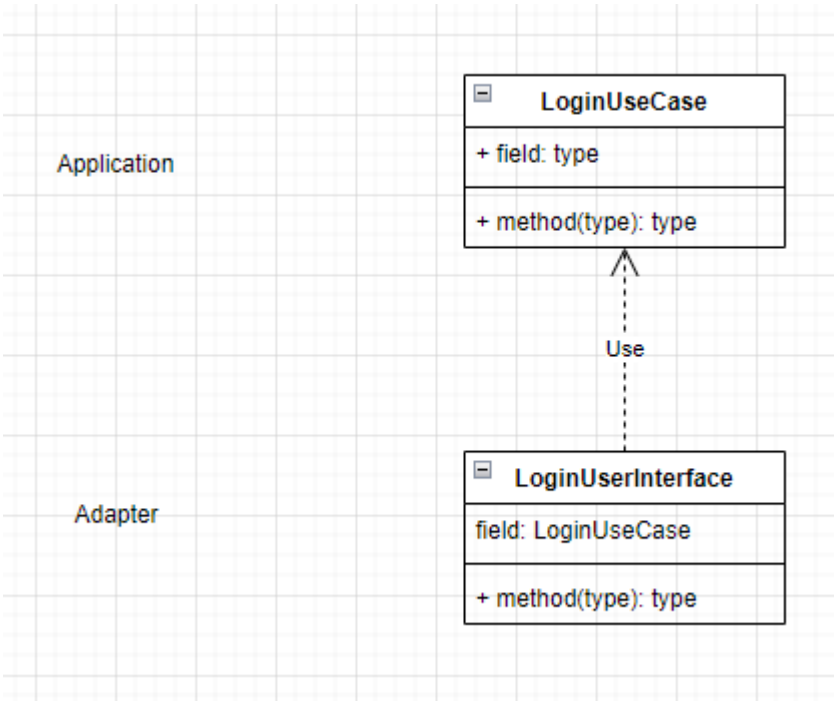
[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

Positiv-Beispiel DIP

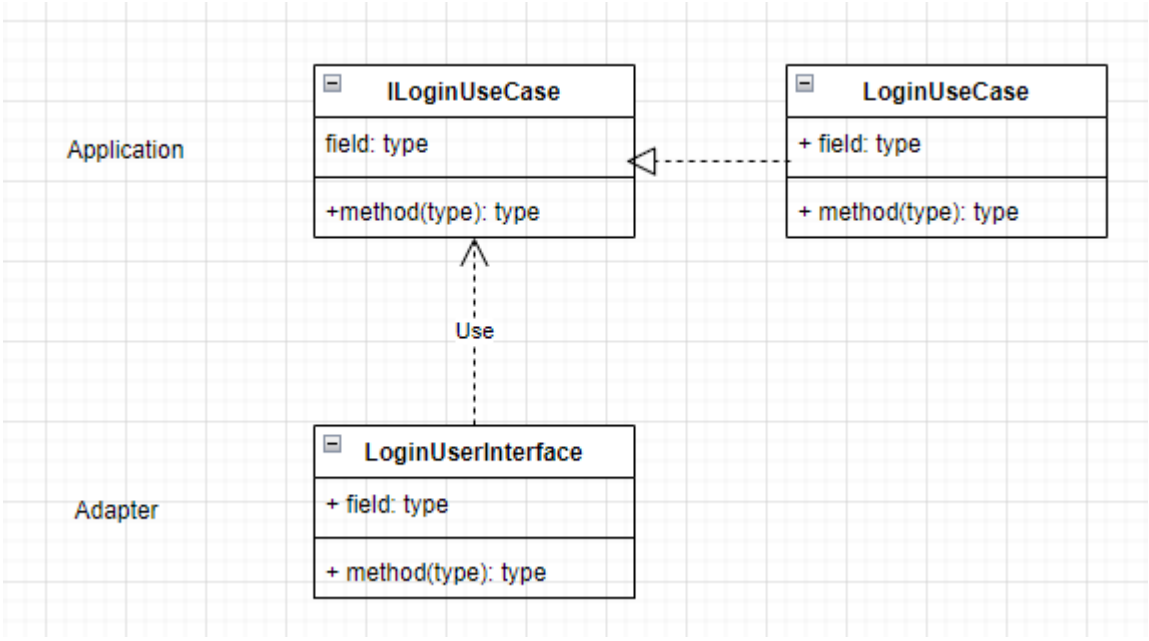


Anstelle einer direkten Abhängigkeit zu UserRepository verwendet UserService das Interface IUserRepository, welches per Dependency Injection übergeben wird, und erfüllt damit das DIP.

Negativ-Beispiel DIP



Hier existiert eine direkte Abhängigkeit anstatt dass ein Interface genutzt wird (da das in der Clean Architecture von außen nach innen erlaubt ist). Um dem DIP vollständig gerecht zu werden, könnte folgende Lösung eingesetzt werden:

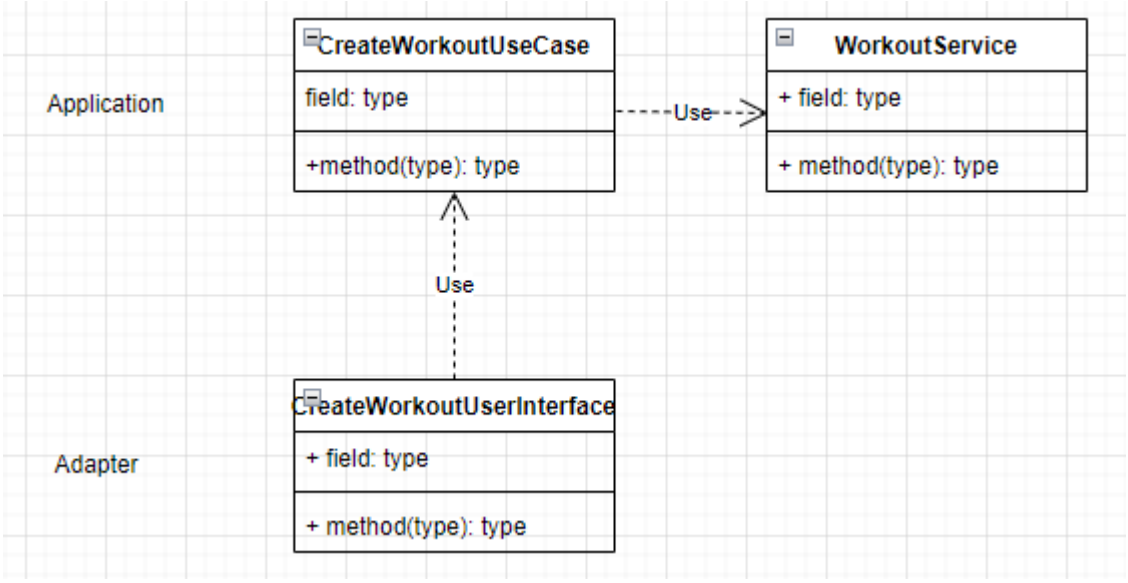


Hier wird ein Interface verwendet, um die direkte Abhängigkeit zu umgehen.

Kapitel 4: Weitere Prinzipien (8P)

Analyse GRASP: Geringe Kopplung (3P)

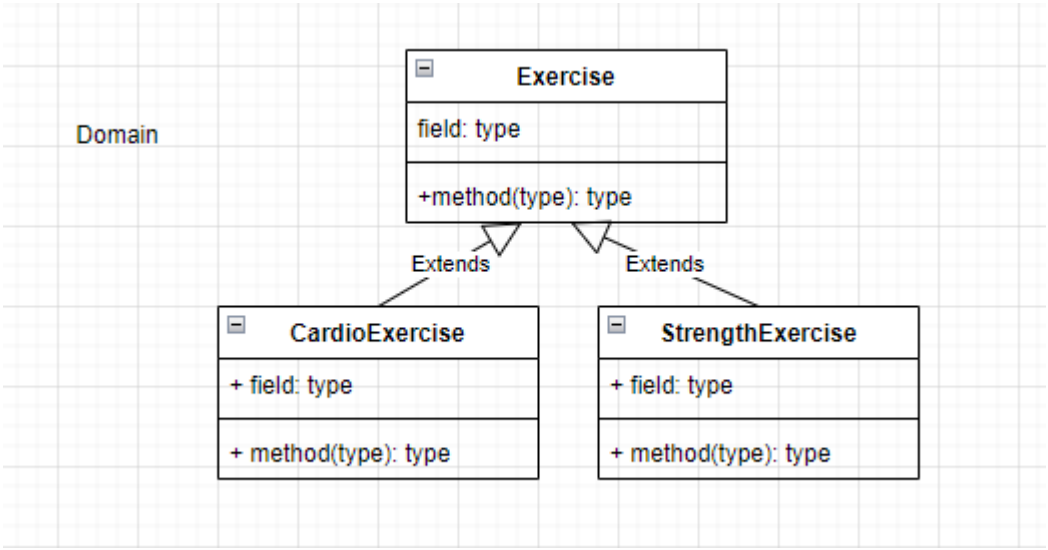
*[eine **bis jetzt noch nicht behandelte** Klasse als positives Beispiel geringer Kopplung; UML mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt]*



Die Klasse CreateWorkoutUseCase entkoppelt dein WorkoutService vom UserInterface. Sie ermöglicht den UseCase, ein neues Workout zu erstellen. Durch die UseCases beeinträchtigen Änderungen der Services das UserInterface nicht und umgekehrt.

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]

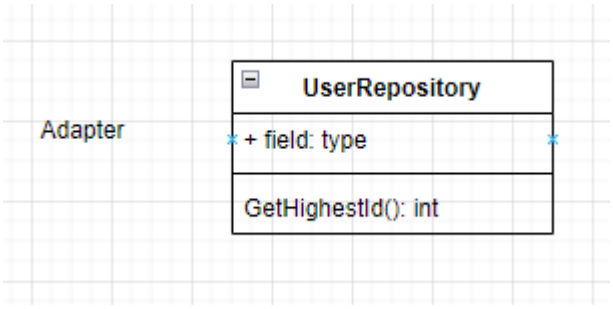


Ein positives Beispiel für Polymorphismus in der Fitnesstracker-Anwendung nach den GRASP-Prinzipien könnte die Verwendung der Klasse "Exercise" sein. Die Klasse "Exercise" könnte eine abstrakte Basisklasse sein, von der verschiedene konkrete Übungsarten (z. B. "StrengthExercise", "CardioExercise") abgeleitet werden. Jede konkrete Übungsart kann spezifische Verhaltensweisen implementieren. Cardio-Übungen gehen zum Beispiel eher auf Zeit, während bei Kraftübungen die Wiederholungen gezählt werden.

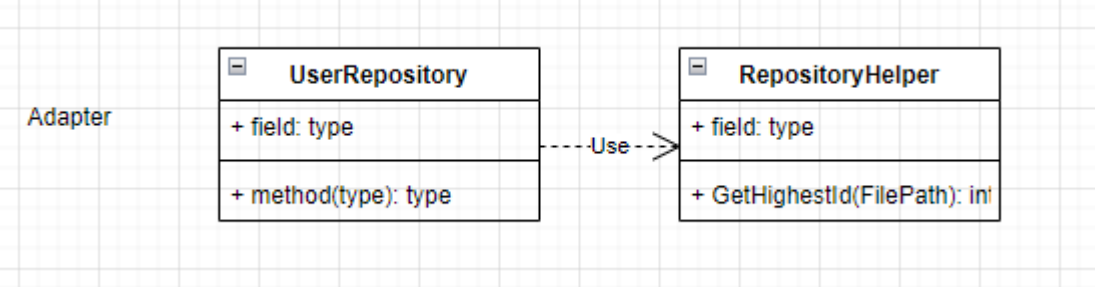
DRY (2P)

[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]

Jedes Repository enthielt die private Methode GetHighestId() um die höchste in Id in einer CSV-Datei herauszufinden:



Diese Methode wurde mit diesem Commit als statische Methode in die neue Klasse RepositoryHelper ausgelagert:



Auf diese Weise kann sehr viel doppelter Code eingespart werden.

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

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

Unit Test	Beschreibung
UserTests.GetAge_ReturnCorrectAge()	Dieser Unit-Test überprüft, ob die Methode User.GetAge() das korrekte Alter aus dem Geburtstag des Benutzers berechnet.

ATRIP: Automatic (1P)

[Begründung/Erläuterung, wie 'Automatic' realisiert wurde]

Unit Tests sollten automatisch ausgeführt werden können, idealerweise als Teil eines Build-Prozesses oder einer kontinuierlichen Integration. Dadurch wird sichergestellt, dass die Tests regelmäßig und zuverlässig durchgeführt werden.

In diesem Fall funktionieren die Tests nicht ganz automatisch, sie können jedoch mit wenigen Mausklicks in Visual Studio ausgeführt werden. Langfristig wäre jedoch eine Pipeline sinnvoll, sodass die Tests automatisiert ausgeführt werden können.

ATRIP: Thorough (1P)

[Code Coverage im Projekt analysieren und begründen]

Die Code Coverage der Tests im Projekt ist miserabel. Es gibt viel zu wenige Unit Tests. Dies beruht auf schlechtem Zeitmanagement, sowie dem immensen Aufwand des 6. Semesters, was keine Entschuldigung, aber eine Begründung darstellen soll. Stattdessen folgt eine hypothetische Code Coverage Analyse. Hier sind einige Bereiche, die normalerweise getestet werden sollten:

- *Geschäftslogik, also vor allem Methoden in der Applikations- und Domain-Schicht, in denen nicht-triviale Berechnungen stattfinden (wie zum Beispiel User.GetAge())*
- *Datenbankzugriffe, also die Methoden der Repositories in der Adapter-Schicht müssen verifiziert werden, um sicherzugehen, dass alle Lese- und Schreib-Operationen das korrekte Format verwenden und gefährliche Edge Cases korrekt behandeln*
- *Benutzereingaben, also die Methoden der UserInterfaces in der Adapter-Schicht müssen auf den Umgang mit Edge Cases bei Eingaben des Benutzers getestet werden, um die Anwendung beispielsweise gegen Injection-Angriffe zu schützen*

ATRIP: Professional (1P)

[1 positves Beispiel zu 'Professional'; Code-Beispiel, Analyse und Begründung, was professionell ist]

```
namespace FitnessTrackerTests
{
    0 references
    public class UserTests
    {
        [Theory]
        [InlineData(20010227)]
        [InlineData(00110101)]
        0 references
        public void GetAge_ReturnCorrectAge(int birthdayDate)
        {
            var birthdayYear = birthdayDate/10000;
            var birthdayMonth = birthdayDate % (birthdayYear * 10000) / 100;
            var birthdayDay = birthdayDate % (birthdayYear * 10000) % (birthdayMonth * 100);
            var testAthlete = new User("testName", "testPassword", new DateTime(birthdayYear, birthdayMonth, birthdayDay), 80.0f);
            var currentDate = DateTime.Now;
            var currentDateInt = currentDate.Year*10000 + currentDate.Month*100 + currentDate.Day;
            var expectedAge = (currentDateInt - birthdayDate) / 10000;

            int actualAge = testAthlete.GetAge();

            Assert.Equal(expectedAge, actualAge);


        }
    }
}
```

Der Test GetAge_ReturnCorrectAge(int birthdayDate) testet die Methode User.GetAge(), ob diese das korrekte Alter aus dem Geburtsdatum des Benutzers berechnet. Folgende Dinge sind professionell:

- *Die TestMethode wurde nach BestPractises in einem separaten Projekt angelegt und aussagekräftig benannt nach dem Muster "getesteteMethode_wasSollPassieren()"*
- *Die TestMethode ist zur besseren Verständlichkeit klar nach dem Arrange-Act-Assert-Muster aufgebaut. Die einzelnen Schritte sind sogar durch Absätze getrennt.*
- *Die TestMethode überprüft die zu testende Methode als datengetriebener Test unter Verwendung des [Theory]-Tags sogar mit mehreren Werten.*

Fakes und Mocks (3P)

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

Da aufgrund der schlechten Test-Situation keine Mock-Objekte eingesetzt wurden, erfolgt die Mock-Analyse leider ebenfalls hypothetisch, um wenigstens zu zeigen, dass das Problem nicht im Verständnis lag.

Bei folgenden Klassen wäre Mocking wahrscheinlich sinnvoll:

- *Mock-Objekte für Datenbankzugriffe: Um die Datenbankzugriffe zu isolieren und unabhängig von einer tatsächlichen Datenbank durchzuführen, könnten Mock-Objekte verwendet werden. Diese Mock-Objekte simulieren das Verhalten der Datenbank und ermöglichen es, bestimmte Szenarien gezielt zu testen, ohne tatsächlich auf eine Datenbank zugreifen zu müssen.*
- *Mock-Objekte für Benutzereingaben: Um Benutzereingaben zu simulieren und verschiedene Szenarien zu testen, können Mock-Objekte eingesetzt werden. Diese Mock-Objekte können vordefinierte Eingaben liefern, um sicherzustellen, dass die Anwendung korrekt auf verschiedene Benutzereingaben reagiert.*
- *Mock-Objekte für Zeit- und Datumswerte: Wenn die Fitnesstracker-Anwendung Funktionen verwendet, die auf Zeit- oder Datumswerte basieren (z. B. Zeitstempel für Workouts oder Trainingspläne), können Mock-Objekte verwendet werden, um die Zeit oder das Datum zu steuern und verschiedene Szenarien zu testen.*
- *Mock-Objekte für Service-Abhängigkeiten: Wenn die Fitnesstracker-Anwendung Service-Abhängigkeiten hat (z. B. den UserService oder den ExerciseService), können Mock-Objekte verwendet werden, um diese Abhängigkeiten zu isolieren und das Verhalten der Services zu steuern. Dies ermöglicht es, bestimmte Szenarien zu testen, ohne tatsächlich auf die realen Services zugreifen zu müssen.*

Kapitel 6: Domain Driven Design (8P)

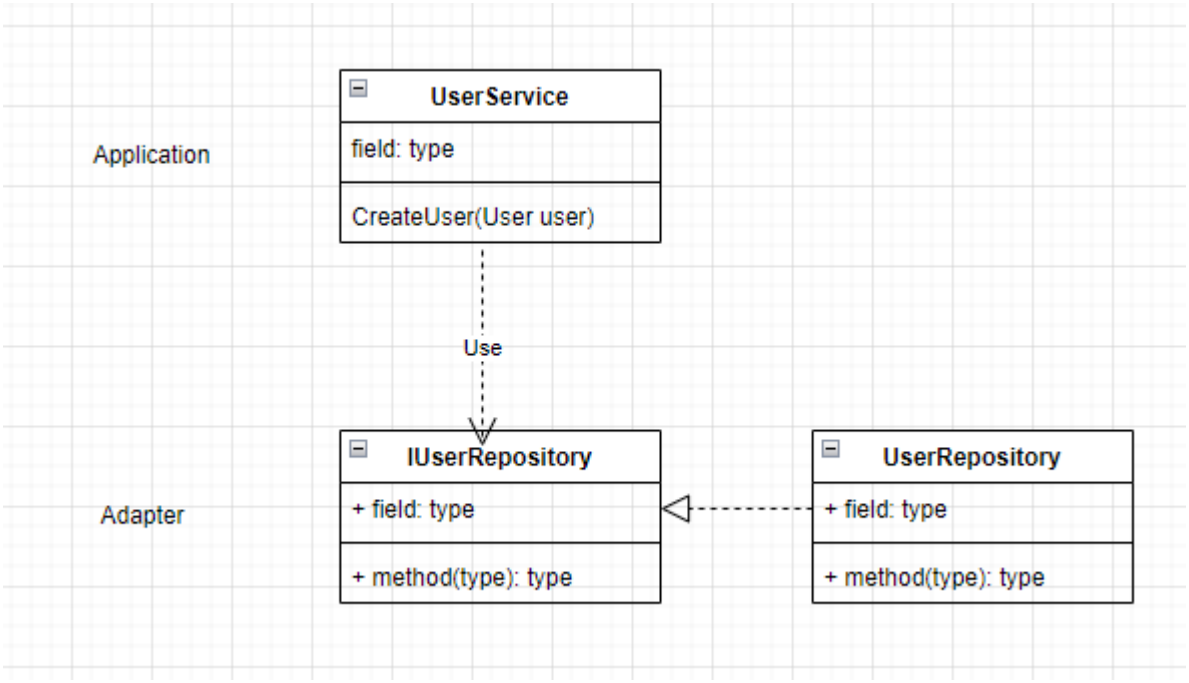
Ubiquitous Language (2P)

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

Bezeichung	**Bedeutung**	**Begründung**
Workout	Eine Trainingssitzung	Diese 4 Begriffe sind allgemein verständlich
Exercise	Eine Fitnessübung	Und können somit sowohl in der Zieldomäne
User	Ein Anwendungsnutzer	Fitness als auch in der Entwicklung
Muscle	Ein bestimmter Muskeltyp (zb. Latissimus)	Verwendet werden

Repositories (1,5P)

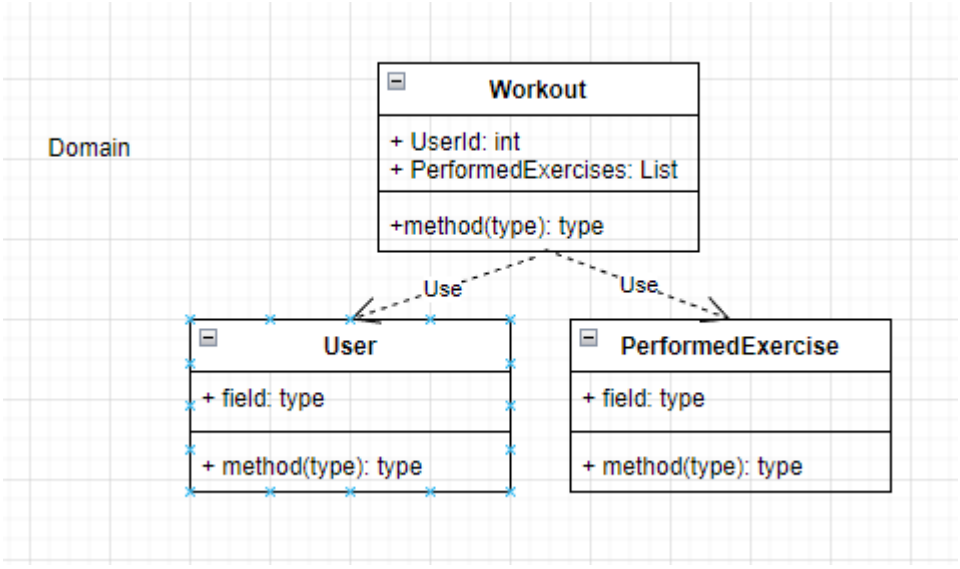
[UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde]



UserRepository: diese Klasse bietet dem UserService einen vereinfachten Zugriff auf die Benutzerpersistierung in Form der CSV-Datei. Hierfür bietet sie unter anderem Methoden zur Erstellung, Löschung und Auffindung von Benutzern. Das Repository dient als Schnittstelle und verhindert, dass der Service in der Application-Schicht Persistierungslogik enthalten muss.

Aggregates (1,5P)

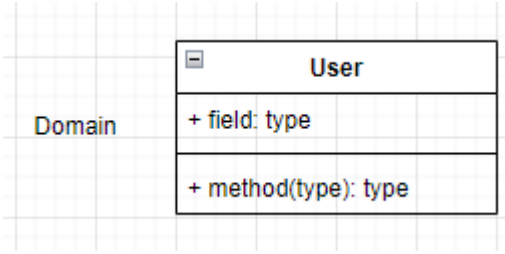
[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]



Die Klasse Workout könnte als Aggregate gesehen werden, da sie einen Zusammenhang zwischen Benutzern und Übungen herstellt. Das Workout-Aggregate sorgt dafür, dass die Zusammenhänge und die Konsistenz zwischen diesen Objekten gewahrt bleiben.

Entities (1,5P)

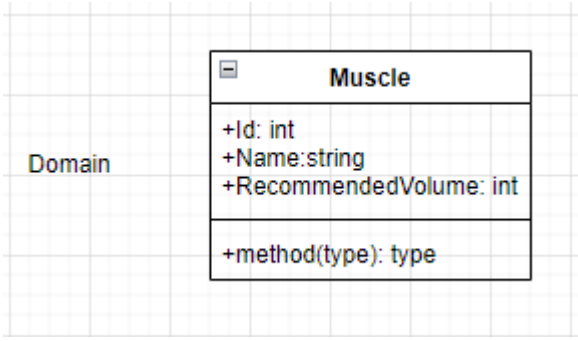
[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keine geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]



Eine Entity der Anwendung ist die Klasse User, ein *Benutzer, der sich in der Anwendung registriert und Workouts erstellt. Die User-Entität enthält Informationen wie Benutzername, Passwort, Gewicht usw. Der Einsatz ist sinnvoll, da diese Entität in der realen Domäne eine zentrale, wenn nicht die zentralste Rolle spielt.*

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 ValueObject der Anwendung könnte ein Muskel sein. Die Klasse enthält lediglich eine Id, den Namen des Muskels und eine Empfehlung zum wöchentlichen Trainingsvolumen in Sätzen. Diese Eigenschaften sollten sich nach erstmaliger Erstellung eigentlich nicht mehr ändern, es sei den durch neue wissenschaftliche Erkenntnisse. In der Anwendung wurde die Klasse jedoch nicht explizit als ValueObject implementiert (aus Gründen, die hier nicht gefragt sind).

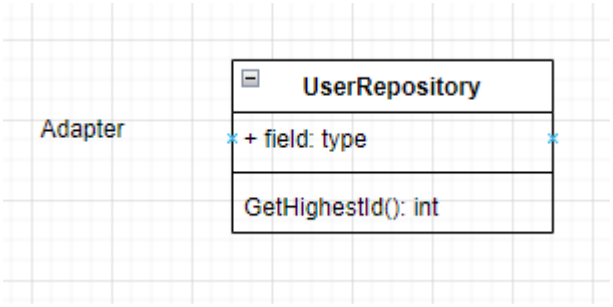
Kapitel 7: Refactoring (8P)

Code Smells (2P)

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

Doppelter Code

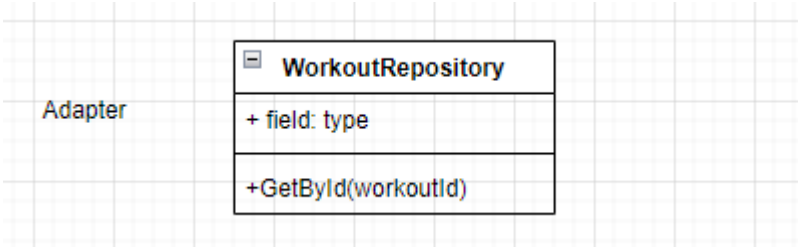
Jedes Repository enthält die private Methode GetHighestId() um die höchste in Id in einer CSV-Datei herauszufinden:



Dies bedeutet sehr viel mehrfachen Code. Eine Lösung wäre die Auslagerung der mehrfachen Methode in eine extra Klasse, auf die die Repositories zugreifen können.

LongMethod

Die Methode GetById(int workoutId) der Klasse WorkoutRepository soll ein gespeichertes Workout anhand dessen Id aus der CSV-Datei liefern. Die Methode ist jedoch sehr lang, da alle Zeilen der CSV-Datei durchsucht werden müssen. Eine mögliche Lösung wäre die Auslagerung des Suchvorgangs in eine private Methode der Klasse.

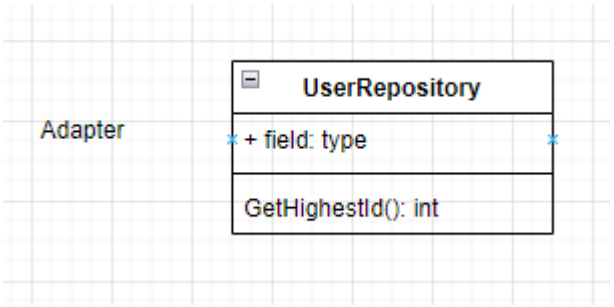


2 Refactorings (6P)

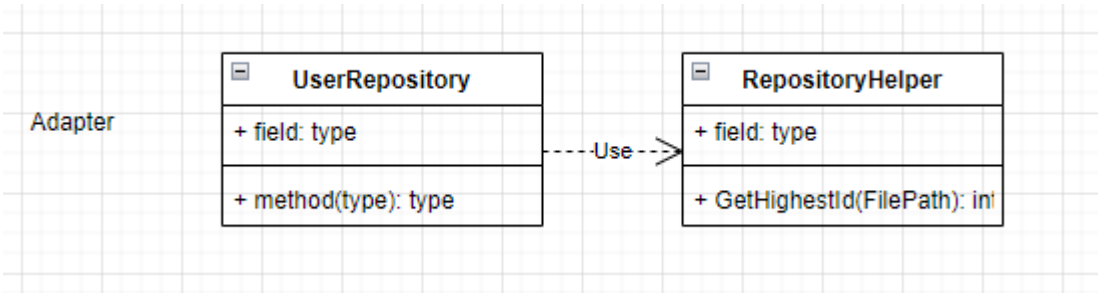
[2 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]

Doppelter Code

Jedes Repository enthielt die private Methode GetHighestId() um die höchste in Id in einer CSV-Datei herauszufinden:



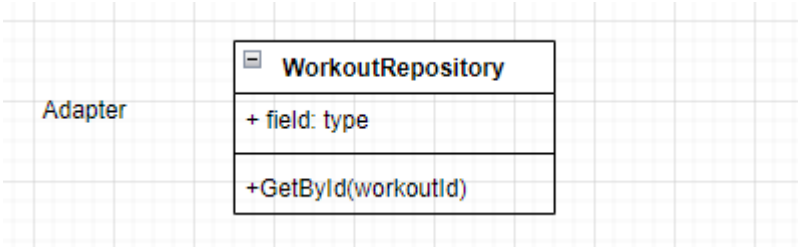
Diese Methode wurde mit diesem Commit als statische Methode in die neue Klasse RepositoryHelper ausgelagert:



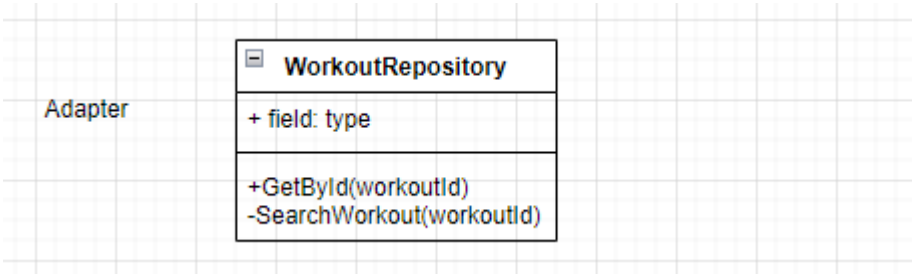
Auf diese Weise kann sehr viel doppelter Code eingespart werden.

LongMethod

Die Methode GetById(int workoutId) der Klasse WorkoutRepository soll ein gespeichertes Workout anhand dessen Id aus der CSV-Datei liefern. Die Methode war jedoch sehr lang, da alle Zeilen der CSV-Datei durchsucht werden müssen.



Um dies zu beheben wurde der Suchvorgang in der Datei in die private Methode searchWorkout(int workoutId) ausgelagert:



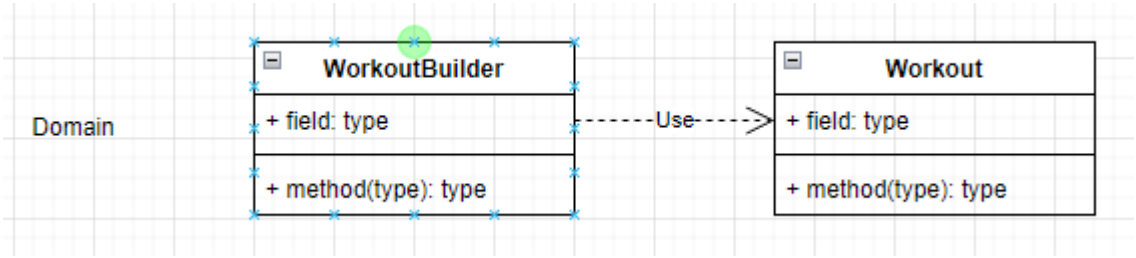
Das Ergebnis sind zwei kürzere und damit lesbare Methoden

Kapitel 8: Entwurfsmuster (8P)

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

Entwurfsmuster: Builder (4P)

Das Builder-Muster kann genutzt werden, um komplexe Objekte wie Workouts oder Trainingspläne schrittweise aufzubauen. Es ermöglicht eine flexible und schrittweise Konstruktion von Objekten, indem verschiedene Bausteine kombiniert werden. Hier wurde ein Builder für die Workout-Klasse implementiert:

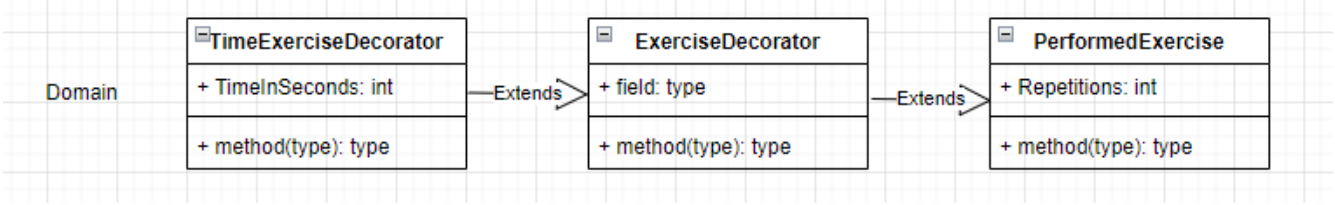


```
public class WorkoutBuilder
{
    private Workout _workout;


    public WorkoutBuilder(IWorkoutService workoutService)
    {
        _workout = new Workout(workoutService);
    }

    0 references
    public WorkoutBuilder WithExercise(PerformedExercise exercise)
    {
        _workout.AddExercise(exercise);
        return this;
    }

    0 references
    public WorkoutBuilder WithUser(User user)
    {
        _workout.UserId = user.Id;
        return this;
    }

    0 references
    public Workout Build()
    {
        return _workout;
    }
}
```

Eigentlich ist jedoch kein Klassenkonstruktur in der Anwendung so groß, dass ein Builder zwingend erforderlich ist.

Entwurfsmuster: Decorator (4P)

Hier wurde für die Klasse PerformedExercise ein Decorator implementiert, der es ermöglicht, dass anstelle der absolvierten Wiederholungen eine Zeit in Sekunden gespeichert werden kann. Dies ist beispielsweise für isometrische Übungen wie zum Beispiel der Unterarmstützposition (Plank) sinnvoll.



```
3 references
public abstract class ExerciseDecorator : PerformedExercise
{
    protected PerformedExercise _exercise;

    1 reference
    public ExerciseDecorator(PerformedExercise exercise)
        : base(exercise.ExerciseId, 0, exercise.Weight)
    {
        _exercise = exercise;
    }
}

1 reference
public class TimeExerciseDecorator : ExerciseDecorator
{
    1 reference
    public int TimeInSeconds { get; set; }

    0 references
    public TimeExerciseDecorator(PerformedExercise exercise, int timeInSeconds)
        : base(exercise)
    {
        TimeInSeconds = timeInSeconds;
    }
}
```