Programmentwurf – 2er Gruppe
Ernährungstool – „Tracktrition“

Name: Bittner Alice
Martrikelnummer: 4744075

Name: Wöhrle Jonathan
Martrikelnummer: 4209116

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: 1P*
- *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)

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

Die Applikation dient dem Tracking der Ernährung des Users.

Sie besteht aus einer Lebensmittel-„Datenbank“ (.txt-Datei) welche von vorneherein einige Lebensmittel für den User zur Auswahl stellt. Der User kann zusätzliche Lebensmittel hinzufügen.

Der User legt sich initial ein Profil mit seinen Daten an. Sein Ernährungsbedarf wird berechnet (Energie und Makronährstoffe). Von nun an kann der User seine tägliche Ernährung tracken, indem er eingibt welche Lebensmittel er zu sich genommen hat.

Starten der Applikation (1P)

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

Requirements:

- .NET 6 Installation wird benötigt - https://learn.microsoft.com/en-us/dotnet/core/install/linux
- Prüfe Version mit
  `dotnet --version`

Starten der Anwendung:

1. Clonen des Repos:
   `git clone https://github.com/bittali/Tracktrition.git`
2. Builden der Anwendung
   `cd Tracktrition`
   `dotnet build`
3. Starten der Anwendung
   `dotnet run`

Ausführen der Unit Tests:

1. Wenn noch nicht im Tracktrition Ordner: `cd Tracktrition`
2. `dotnet test`

Technischer Überblick (2P)

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

Wir haben uns für die Verwendung von C# aus mehreren Gründen entschieden. C# bietet eine vielseitige Programmierumgebung mit einer breiten Palette von Funktionen, die sich gut für Konsolenanwendungen eignen. Es ist auch relativ leicht zu erlernen, insbesondere für diejenigen, die

bereits mit C-ähnlichen Sprachen vertraut sind. Die umfangreiche Dokumentation und die große Community sind ebenfalls von Vorteil.

Die starke Typisierung von C# trägt zu einer besseren Fehlererkennung während der Kompilierung und zur Robustheit des Codes bei. Die Plattformkompatibilität ermöglicht es uns, Konsolenanwendungen zu entwickeln, die auf verschiedenen Betriebssystemen laufen können.

Die Unterstützung der objektorientierten Programmierung (OOP) durch C# mit Funktionen wie Klassen, Vererbung und Polymorphismus ermöglicht die Erstellung modularer und wiederverwendbarer Codestrukturen.

Ein weiterer Faktor war die persönliche Entscheidung, eine neue Programmiersprache zu lernen. Die Nutzung von C# bietet uns die Möglichkeit, unsere Kenntnisse in Syntax, Funktionen und dem Ökosystem zu vertiefen und unsere Fähigkeiten als Entwickler zu verbessern.

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

[allgemeine Beschreibung der Clean Architecture in eigenen Worten]

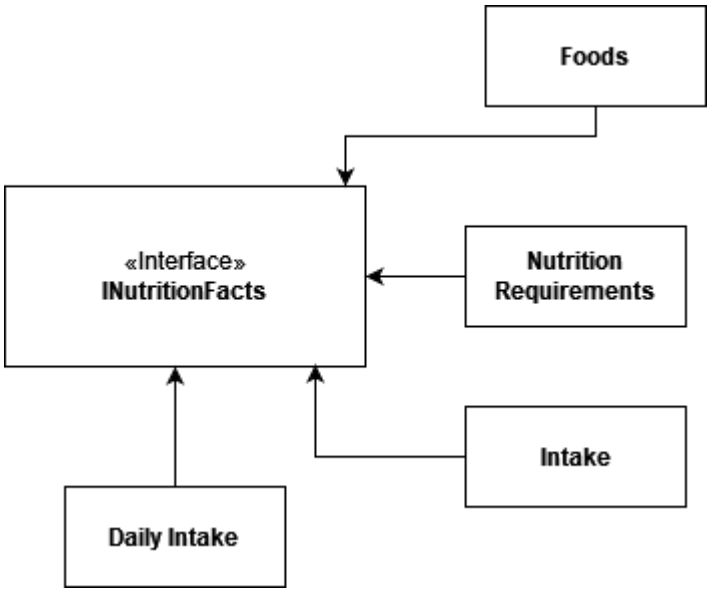
Clean Architecture ist ein Architekturprinzip für Softwareanwendungen, bei dem die Anwendung in Schichten aufgeteilt wird. Den Kern der Anwendung, also die innere Schicht, bildet die Geschäftslogik, das Fachwissen der Domäne, im Weiteren Domänenschicht genannt. Der Kern ist der langlebigste Teil der Anwendung, er umfasst beispielsweise die Entitäten und die Domänenfunktionalitäten, welche sich möglichst selten ändern. Er ist grundsätzlich unabhängig von äußeren Schichten, also von Entscheidungen über Datenbanktechnologie und Implementierungsdetails.
Weiter nach außen folgen die Schichten Application-Schicht, Adapter(-Schicht) und danach die Frameworks und Treiber. Die Application-Schicht ergibt sich aus den Use Cases der Anwendung und ist noch recht fachbezogen, er implementiert die anwendungsspezifische Geschäftslogik. Adapter sind für das Hin- und Herreichen von Daten zwischen Application-Schicht und den Frameworks und Treibern zuständig. Es geht hierbei hauptsächlich um Datenformatkonvertierung passend zu der entsprechenden Schicht. Die Frameworks und Treiber greift nur auf die Adapter zu und sie umfassen die technischen Werkzeuge. Hier werden die meisten Technologieentscheidungen eingebunden an die Anwendungen. Das umfasst die Datenbanken und die UI. Diese Schicht kann sich jederzeit ändern, was Änderungen in den Adaptern bewirkt.
Die einzelnen Schichten sind dazu da, um verschiedene Teile der Anwendung voneinander zu trennen und die Software insgesamt erweiter- und wartbarer zu machen. Durch die Trennung der Schichten können eine feste Struktur und klare Abhängigkeiten zwischen den Schichten festgelegt werden.

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]

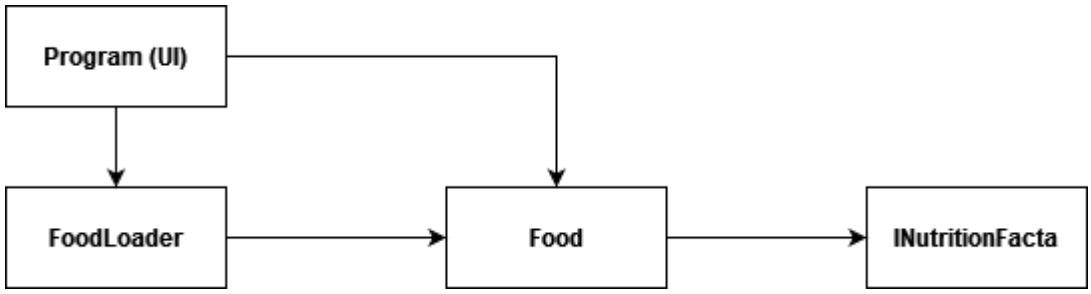
➔ Von außen nach innen: Von Application Code zu Entity

Positiv-Beispiel 1: Dependency Rule



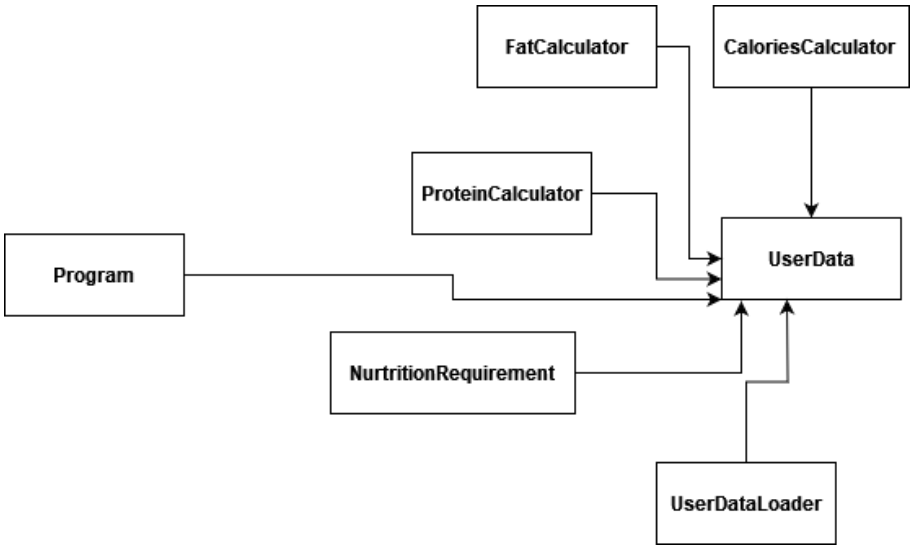
Von Enitity zum Interface besteht die Abhängigkeit schon während des Kompilierens. Das Interface ist abstrakter und liegt somit weiter innen in der Domänen-Schicht.

Positiv-Beispiel 2: Dependency Rule



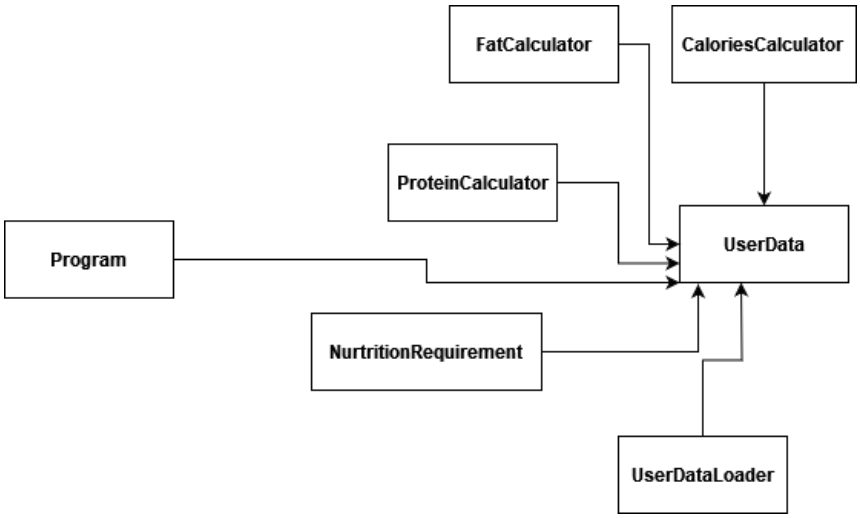
Program steuert die Benutzerschnittstelle, daher ist es weit außen, bei den Adaptern einzuordnen. FoodLoader ist ebenfalls eine Schnittstelle zur Datenbank, es befindet sich auch in der Adapterschicht. Diese greifen weiter nach innen auf die Entitäten zu. Auch hier, bei Nutrition Requirement, geht die Abhängigkeit von außen nach innen.

Positiv-Beispiel 3: Dependency Rule



Auch hier gehen die Abhängigkeiten von außen nach innen. Die Calculator sitzen im Application Code, gehören also mit zur Realisierung der Use Cases. Mit Nutrition Requirement besteht eine Abhängigkeit innerhalb derselben Schicht (Entities), was ebenfalls nicht gegen die Dependency Rule verstößt. Allerdings findet sich hier auch das Negativbeispiel mit Program (UI). Das wird nachfolgend behandelt.

Negativ-Beispiel: Dependency Rule

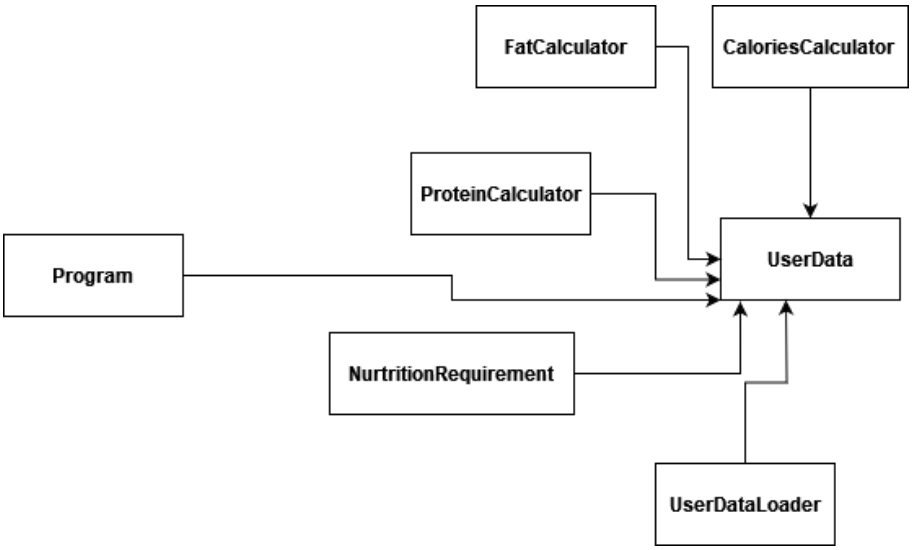


Program bzw. die Main steuert die gesamte Benutzerinteraktion. Damit liegt sie in der äußersten Schicht und sollte nur über Adapter/Controller mit den Entitäten kommunizieren, also nicht direkt davon abhängen. Dies ist hier aber nicht der Fall, da Program direkt Nutrition Requirements instanziiert.

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]

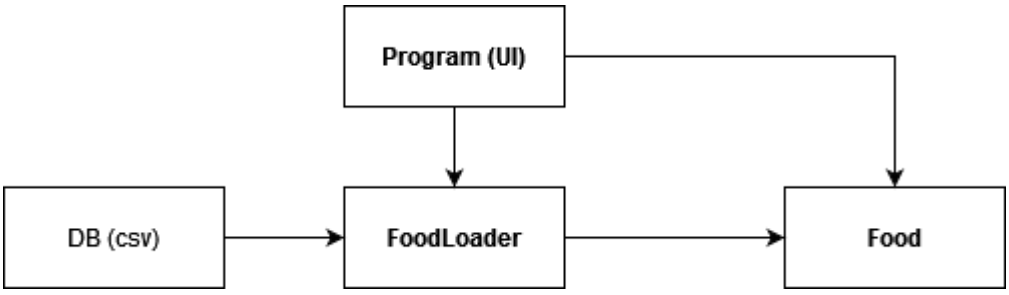
Schicht: Entities



UserData repräsentiert eine Entität im Kontext des Programms. Sie enthält Eigenschaften, die die Attribute einer Person beschreiben, wie Name, Geschlecht, Alter, Gewicht, Größe und Aktivitätsgrad. Die Klasse enthält auch Methoden, um diese Daten auszugeben.

UserData ist klar der Domänen-Schicht zuzuordnen und darin der innersten Schicht der Entitäten. Sie hat keine Abhängigkeit zu anderen Klassen und hängt von keinen Frameworks oder Bibliotheken ab. Jedoch hängen viele andere Klassen zu ihr ab.

Schicht: Adapter

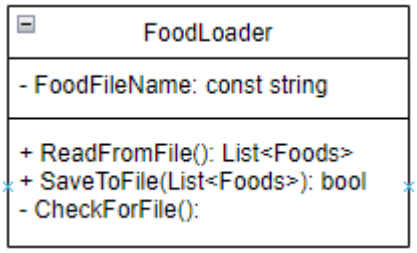


Die FoodLoader-Klasse ist eine Adapterschicht-Komponente, die sich auf das Lesen und Schreiben von Ernährungsdaten aus einer CSV-Datei konzentriert. Die Hauptaufgabe der FoodLoader-Klasse besteht darin, die Food-Entität zu laden und zu speichern. Die Klasse verwendet die StreamReader- und StreamWriter-Klassen, um die CSV-Datei zu lesen und zu schreiben. Die Datei Foods.csv wird verwendet, um Informationen über verschiedene Lebensmittel zu speichern.
Die FoodLoader-Klasse ist in dieser Schicht einzuordnen, da sie die Interaktion mit der CSV-Datei kapselt und die Datenverarbeitung von den Domain-Logikkomponenten trennt. Dies ermöglicht eine bessere Wartbarkeit und Erweiterbarkeit der Anwendung.

Kapitel 3: SOLID (8P)

Analyse SRP (3P)

Positiv-Beispiel 1

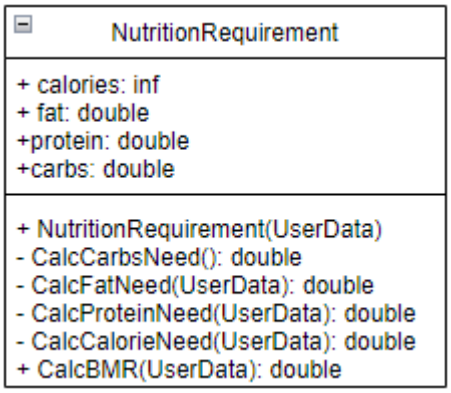


Die public Methode "ReadFoodFromFile" liest die Informationen aus der Datei "foods.csv" und gibt eine Liste von "Food"-Objekten zurück. Die Methode liest dabei die Datei zeilenweise und erstellt für jede Zeile ein neues "Food"-Objekt, das der Liste hinzugefügt wird. Davor wird jedoch die Methode ”CheckForFIle” ausgeführt.

Intern enthält die Klasse die private Methode "CheckForFile", die überprüft, ob die Datei "foods.csv" vorhanden ist. Wenn die Datei nicht existiert, wird sie erstellt.

Die Methode "SaveFoodsToFile" ermöglicht das Speichern einer Liste von "Food"-Objekten in der Datei. Dabei wird eine "StreamWriter"-Instanz verwendet, um die Lebensmittelinformationen in die Datei zu schreiben. Jedes "Food"-Objekt wird als eine Zeile mit den entsprechenden Eigenschaften gespeichert.

Positiv-Beispiel 2



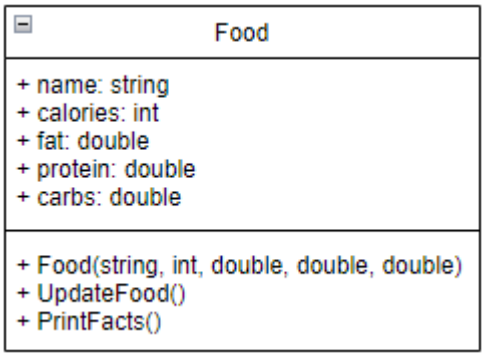
Beim Erstellen eines "NutritionRequirement"-Objekts werden die erforderlichen Werte für Kalorien, Fett, Protein und Kohlenhydrate anhand der Userdaten berechnet und den entsprechenden Eigenschaften zugewiesen.

Die Klasse beinhaltet interne Methoden zur Berechnung der spezifischen Nährstoffbedürfnisse. Die Methode "CalcCalorieNeed" ermittelt den Gesamtkalorienbedarf unter Berücksichtigung des Aktivitätsniveaus des Benutzers. "CalcCarbsNeed" ermittelt die benötigte Menge an Kohlenhydraten basierend auf dem Gesamtkalorienbedarf. "CalcFatNeed" und "CalcProteinNeed" berechnen das erforderliche Fett bzw. Protein unter Berücksichtigung des Gesamtkalorienbedarfs und der Benutzerdaten.

Des weiteren enthält die Klasse eine Methode namens "CalcBMR", um die Basal Metabolic Rate anhand der Userdaten zu berechnen. Die Basal Metabolic Rate wird zur Berechnung des Tagesbedarfs an Kalorien benötigt.

Negativ-Beispiel

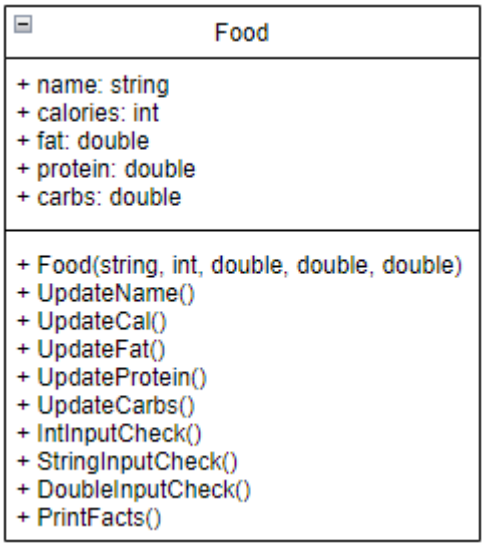
Das Negativ Beispiel besteht in Commit (https://github.com/bittali/Tracktrition/blob/00118d14924dad4ed83bc5086396b16d91033df9/Tracktrition/Data/Food.cs) und wurde im Anschluss geändert.



Der Konstruktor ermöglicht die Initialisierung der Variablen des Food-Objekts mit den übergebenen Werten.

Die Methode "updateFood" ermöglicht die Aktualisierung der Objekt Variablen über die Konsoleneingabe. Zuerst wird der User aufgefordert, einen neuen Namen für das Lebensmittel einzugeben. Dann werden die Werte für Kalorien, Fett, Protein und Kohlenhydrate ebenfalls über die Eingabe aktualisiert. Bei allen Eingaben werden die Werte auf Gültigkeit geprüft.

Die Methode "PrintFacts" gibt die Informationen des Lebensmittels in der Konsole aus, einschließlich des Namens, der Kalorien, des Fettgehalts, des Proteins und der Kohlenhydrate.



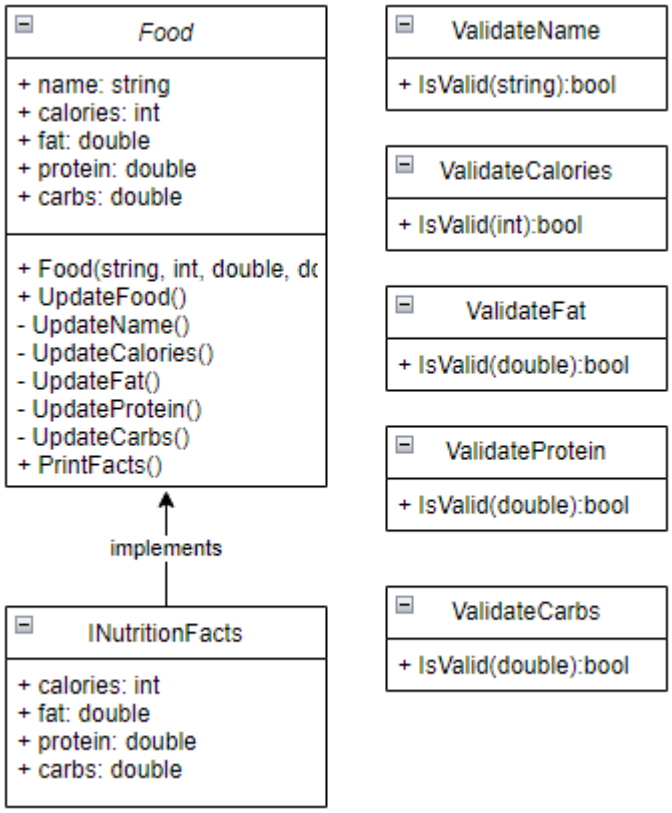
Um das Negativ Beispiel zu beseitigen, ist ein aufbrechen der “UpdateFood”-Methode notwendig. Hierzu wäre es am ratsamsten die einzelnen Werte über eigene Methoden zu realisieren und eine generische Type-Check Methode für jeden Variablentyp zu implementieren.

Analyse OCP (3P)

[zwei Klassen als positives Beispiel und eine Klasse als 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)?]

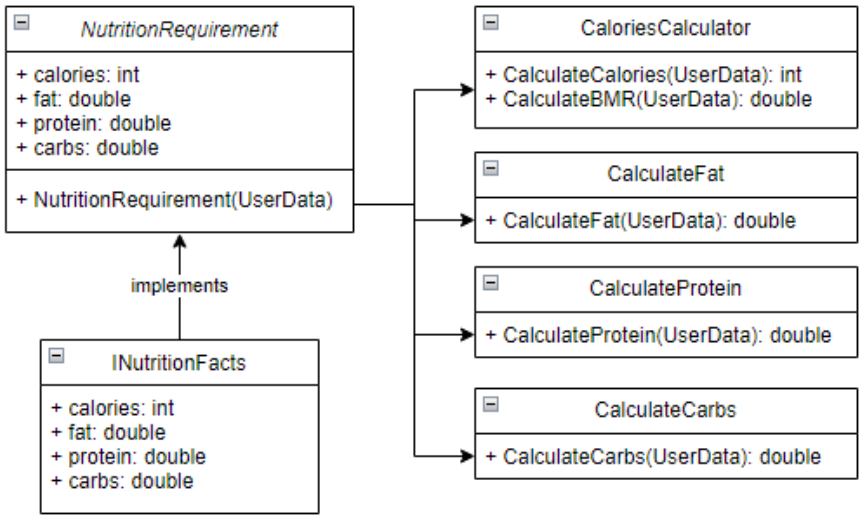
Positiv-Beispiel 1

Diese Struktur hält sich an das OCP-Prinzip, indem sie die Validierungslogik in separate Klassen aufteilt, was eine einfache Erweiterung oder Änderung des Validierungsverhaltens ohne Änderung der bestehenden Klassen ermöglicht. Neue Validierungsregeln können durch die Erstellung neuer Validierungsklassen oder die Erweiterung bestehender Klassen hinzugefügt werden. Dieser Ansatz



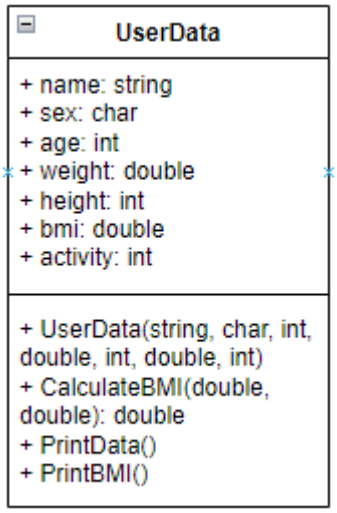
fördert die Wiederverwendbarkeit des Codes, die Wartungsfreundlichkeit und minimiert das Risiko der Einführung von Fehlern bei der Erweiterung der Validierungslogik.

Positiv-Beispiel 2

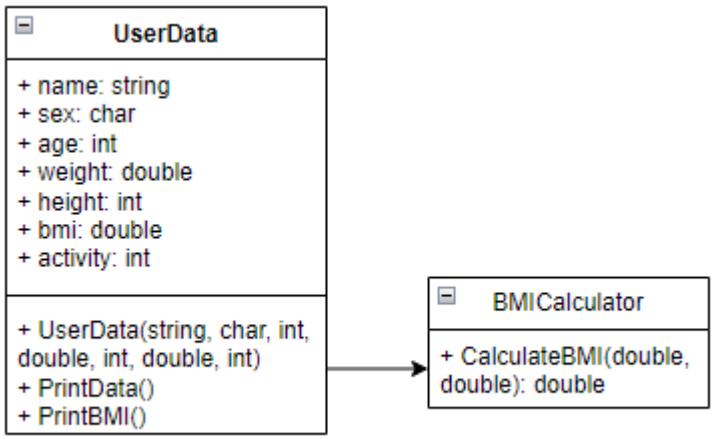


Diese Struktur hält sich ebenfalls an das OCP-Prinzip, indem sie die Berechnungen der Nährwerte in separate Klassen aufteilt, was ebenso eine einfache Erweiterung von Nährwerten oder Änderung der Berechnungen ohne Änderung der bestehenden “Calculate”-Klassen ermöglicht. Neue Berechnungen können durch die Erstellung neuer Klassen hinzugefügt werden.

Negativ-Beispiel



Um eine neue Berechnung wie z.B. Waist-to-Hip-Ratio (WtHR) einzuführen müsste die Klasse verändert bzw. eine neue Methode der Klasse hinzugefügt werden. Durch die Extraktion der ”CalculateBMI”-Methode und Restrukturierung der ”UserData”-Klasse könnten neue Berechnungen nach dem OCP-Prinzip implementiert werden. (Siehe 2. UML)

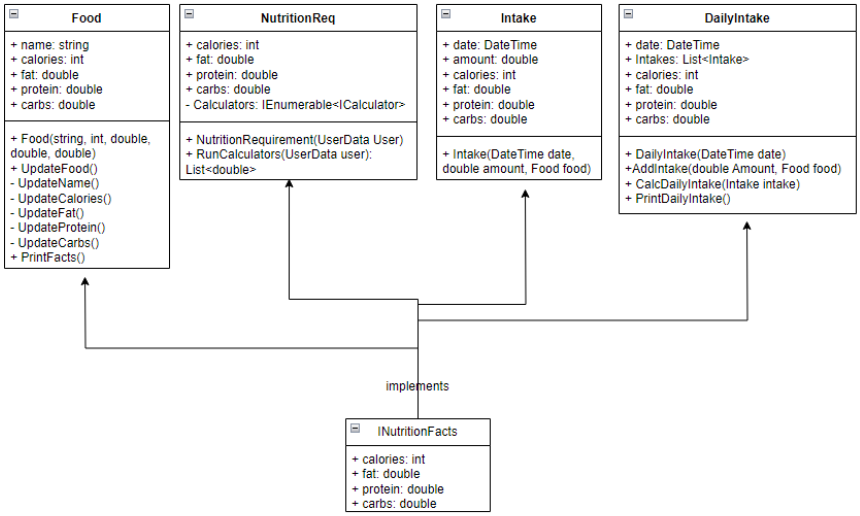


Analyse [LSP/ISP] (1P)

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

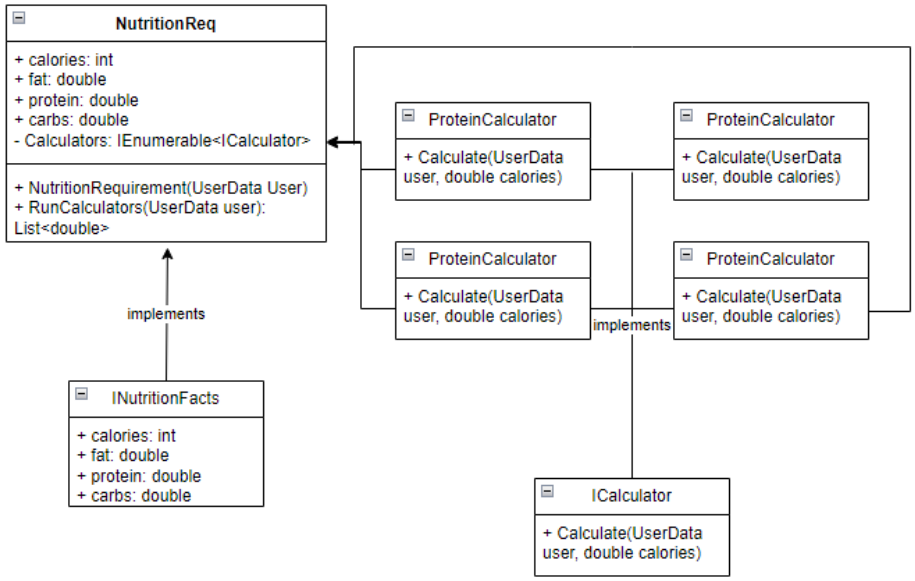
[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

Positiv-Beispiel



Durch die Interface Implementierung wird das ISP erfüllt. Durch die genaue Vorgabe des Interfaces ist die Grundstruktur jeder Klasse die das Interface implementiert vorgegeben und unveränderbar, jedoch sind die Methoden der Klassen änderbar. Jede Methode ist so für den eigenen Zweck verantwortlich und muss dabei aber auch keine unnötigen Methoden implementieren.

Negativ-Beispiel

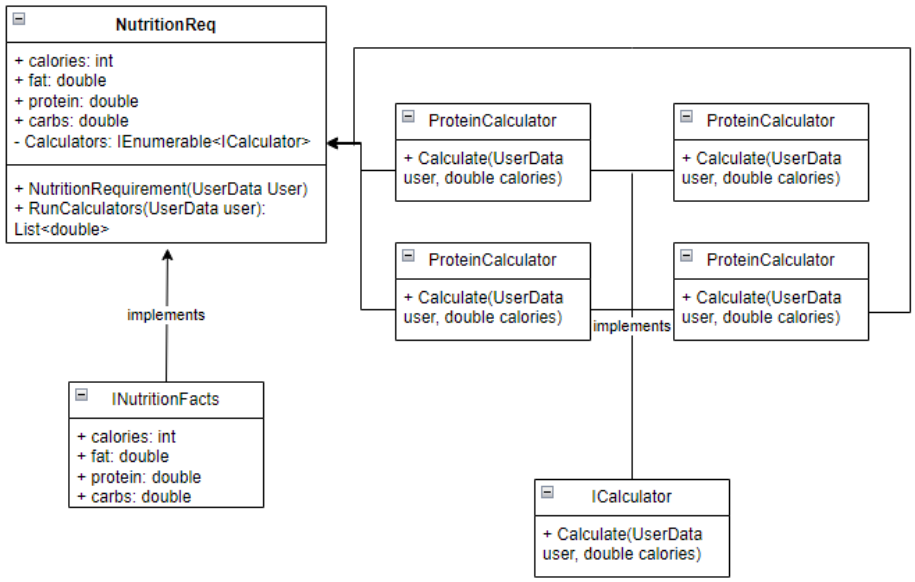


Die Klasse ICalculator erfüllt ISP nicht, da nicht für jede Calculate Methode beide Parameter benötigt werden. Die Berechnung der Calorien benötigt keine Calorien als Parameter. Ebenso benötigt die Berechnung der Carbs keine Daten über den User.

Analyse [DIP] (1P)

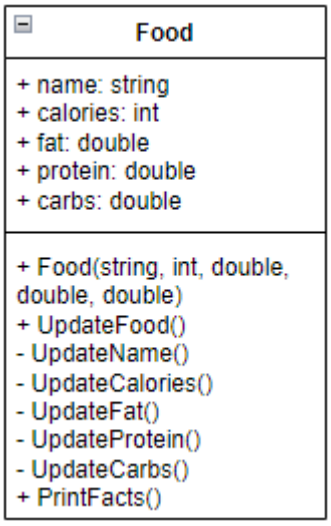
[jeweils eine Klasse als positives und negatives Beispiel für DIP; jeweils UML und Begründung, warum hier das Prinzip erfüllt/nicht erfüllt wird; beim Negativ-Beispiel UML einer möglichen Lösung hinzufügen]

Positiv-Beispiel

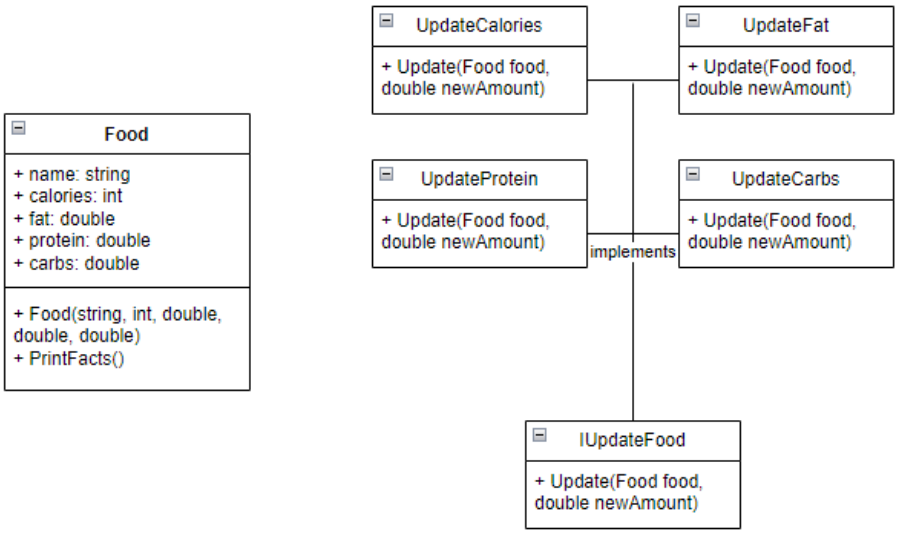


Die Calulator sind nicht von der Klasse NutritionReq abhängig, diese kann ohne Probleme geändert werden. Die Berechnungsklassen funktionieren wie gewohnt. Daher erfüllen die Calulator die DIP Bedingungen.

Negativ-Beispiel



Die ”Food”-Klasse erfüllt die DIP Bedingungen nicht, da sie nicht modular genug aufgebaut ist. Ein Lösungsweg wäre es die Update Methoden aus der Klasse auszulagen und per Interface zu implementieren.



(siehe untere Abb.)

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]*

Analyse GRASP: Polymorphismus (1,5P)

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

Zum Umsetzen von Mocks ohne Mocking-Framework wurde Polymorphismus eingesetzt. Genauer überschrieben die Mock-Klassen Methoden ihrer Base-Klassen mit Mock-Methoden (static Polymorphism).

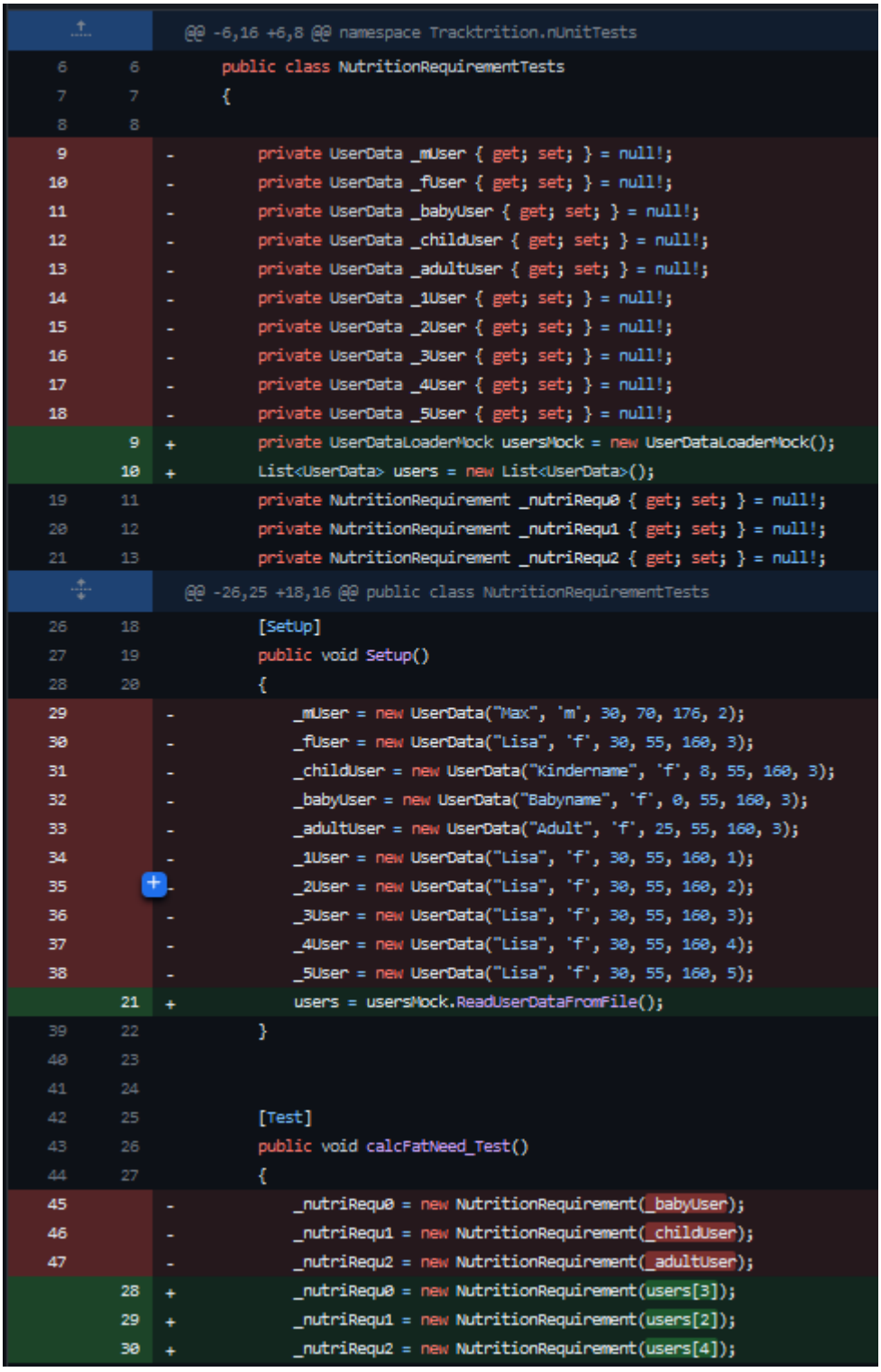
Analyse GRASP: Pure Fabrication (1,5P)

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

DRY (2P)

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

commit 2af4c82639a11a048eeca2bf3c2aa76d37f442d5



```
@@ -6,16 +6,8 @@ namespace Tracktrition.nUnitTests
    public class NutritionRequirementTests
    {

-       private UserData _mUser { get; set; } = null!;
-       private UserData _fUser { get; set; } = null!;
-       private UserData _babyUser { get; set; } = null!;
-       private UserData _childUser { get; set; } = null!;
-       private UserData _adultUser { get; set; } = null!;
-       private UserData _1User { get; set; } = null!;
-       private UserData _2User { get; set; } = null!;
-       private UserData _3User { get; set; } = null!;
-       private UserData _4User { get; set; } = null!;
-       private UserData _5User { get; set; } = null!;
+       private UserDataLoaderMock usersMock = new UserDataLoaderMock();
+       List<UserData> users = new List<UserData>();
        private NutritionRequirement _nutriRequ0 { get; set; } = null!;
        private NutritionRequirement _nutriRequ1 { get; set; } = null!;
        private NutritionRequirement _nutriRequ2 { get; set; } = null!;
@@ -26,25 +18,16 @@ public class NutritionRequirementTests
        [SetUp]
        public void Setup()
        {
-           _mUser = new UserData("Max", 'm', 30, 70, 176, 2);
-           _fUser = new UserData("Lisa", 'f', 30, 55, 160, 3);
-           _childUser = new UserData("Kindername", 'f', 8, 55, 160, 3);
-           _babyUser = new UserData("Babyname", 'f', 0, 55, 160, 3);
-           _adultUser = new UserData("Adult", 'f', 25, 55, 160, 3);
-           _1User = new UserData("Lisa", 'f', 30, 55, 160, 1);
-           _2User = new UserData("Lisa", 'f', 30, 55, 160, 2);
-           _3User = new UserData("Lisa", 'f', 30, 55, 160, 3);
-           _4User = new UserData("Lisa", 'f', 30, 55, 160, 4);
-           _5User = new UserData("Lisa", 'f', 30, 55, 160, 5);
+           users = usersMock.ReadUserDataFromFile();
        }


        [Test]
        public void calcFatNeed_Test()
        {
-           _nutriRequ0 = new NutritionRequirement(_babyUser);
-           _nutriRequ1 = new NutritionRequirement(_childUser);
-           _nutriRequ2 = new NutritionRequirement(_adultUser);
+           _nutriRequ0 = new NutritionRequirement(users[3]);
+           _nutriRequ1 = new NutritionRequirement(users[2]);
+           _nutriRequ2 = new NutritionRequirement(users[4]);
```

Durch das Mocken von UserDataLoader konnte in den Tests das erstellen vieler neuer UserData-Objekte im SetUp durch das Nutzen des Mocks ersetzt werden. Das verhindert, dass derselbe User mehrfach angelegt werden muss.

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

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

Unit Test	Beschreibung
Klasse#Methode	
UserData#CalcBMI	Es wird getestet ob die Berechnung des BMI stimmt
DailyIntake#AddIntake	Es wird getestet ob das Hinzufügen einer Mahlzeit den Tages Intake auch erhöht
AmountCalulator#CalculatePerAmount	Es wird getestet ob die Berechnung mittels Lebensmittel und dem angegebenen Gewicht stimmt
NutritionRequirement#CalcBMR	Es wird getestet ob die Berechnung des Tagesbedarf an Kalorien stimmt
NutritionRequirement#CalcCalorieNeed	Es wird getestet, ob die Akitivität eines Users korrekt einberechnet wird
NutritionRequirement#CalcFatNeed	Es wird getestet, ob das Alter des Users entsprechend einberechnet wird in den Fettbedarf
NutritionRequirement#CalcProteinNeed	Es wird getestet, ob das Alter des Users entsprechend einberechnet wird in den Proteinbedarf

ATRIP: Automatic (1P)

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

Automatic wurde wurde mithilfe des NUnit-Frameworks realisiert. In Visual Studio können alle Test bequem über den Test Explorer durch einen Klick gestartet werden (oder durch `dotnet test` über die Kommandozeile). Die Tests wurden in der Implementierung mit Daten befüllt, sodass keine manuelle Dateneingabe notwendig ist. Durch Assert-Statements werden die Ergebnisse automatisch überprüft. Ausgegeben wird eine Übersicht mit allen bestandenen Tests und allen, die fehlschlugen (inkl. Dem erwarteten Wert und erhaltenen Wert).

ATRIP: Thorough (1P)

[Code Coverage im Projekt analysieren und begründen]

Thorough wurde im Projekt nicht erfüllt. Die Code Coverage ist sehr ausbaufähig.

ATRIP: Professional (1P)

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

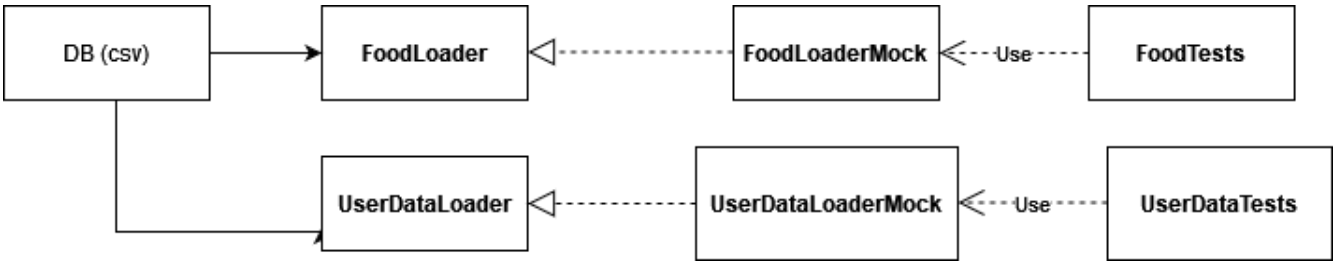
Es wurden nur Tests geschrieben, die für notwendig erachtet wurden, z.B. wurden keine Getter getestet. Die Tests dienen als Doku und zeigen was die getesteten Methoden bewirken sollen.

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]

Gemockt werden die Loader-Klassen, welche als Schnittstellen zu den Datenbanken (csv) dienen. Hier werden die Methoden gemockt, welche für das lesen der Daten aus der DB zuständig sind.

Begründung: das Auslesen von Dateien in UnitTests ist riskant/unerwünscht, da dieser Prozess leicht fehlschlagen kann, in etwa durch eine Änderung am Testsystem.



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**
Intake	„Einnahme“ einer bestimmten Menge eines Lebensmittel	Dieser Begriff bezeichnet die konkrete Handlung, eine bestimmte Menge an Nahrung aufzunehmen. Sie ist allgemein verständlich.
Nutrition Facts	Nährwertangaben eines Lebensmittels oder	Dieser Begriff gehört zur Ubiquitous Language, da "Nutrition Facts" eine standardisierte Bezeichnung, für die auf US-amerikanischen Lebensmittelverpackungen angegebenen Informationen zu den Nährwerten ist. Es umfasst Angaben über die kCal, Fett, Protein und Kohlenhydrate, was hier bei allen Implementierungen des Interfaces auch benötigt wird.
BMI	Body Mass Index	Der Begriff BMI wird häufig verwendet, um den Gesundheitszustand einer Person in Bezug auf das Gewicht zu bewerten. Da der BMI ein weit verbreitetes Instrument ist, um Gewichtskategorien zu klassifizieren, gehört dieser Begriff zur Ubiquitous Language im Bereich Gesundheit und Ernährung.

BMR	Basal Metabolic Rate (Grundumsatz)	Bezeichnet (auf Englisch) den Grundumsatz des Körpers, also die Menge an Energie, die der Körper im Ruhezustand verbraucht.

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]

Aggregates (1,5P)

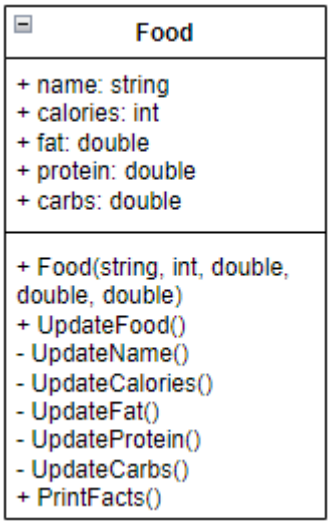
[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]

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]

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]



Die Klasse Food kann als Value Object betrachtet werden. Ein Value Object repräsentiert einen konzeptionell unveränderlichen Wert, der durch seine Attribute und nicht durch seine Identität definiert ist. Die Klasse Food erfüllt die Kriterien eines Value Objects in mehrfacher Hinsicht. Der Einsatz hier erweist sich als sinnvoll, da sich der Zweck des Programs um die Nährwerte von Nahrung bezieht. Diese können dem Objekt beim Erstellen eines neuen Objekts zu geteilt werden.

Kapitel 7: Refactoring (8P)

Code Smells (2P)

[jeweils 1 Code-Beispiel zu 3 unterschiedlichen Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]

CODE SMELL 1: Duplicated Code

```
1 reference
private void UpdateName()
{
    Console.Write("Enter the new name of the food: ");
    string? inputName = Console.ReadLine();

    while (!ValidateName.IsValid(inputName))
    {
        Console.WriteLine("Invalid input. Please enter a valid name for the food.");
        Console.Write("Enter the name of the food: ");
        inputName = Console.ReadLine();
    }

    this.name = inputName.Trim();
}

1 reference
private void UpdateCalories()
{
    bool validInput = false;

    while (!validInput)
    {
        Console.Write("Enter the new number of calories: ");
        string? input = Console.ReadLine();

        if (ValidateCalories.IsValid(input, out int caloriesValue))
        {
            this.calories = caloriesValue;
            validInput = true;
        }
    }
}
```

Codeauschnitt ist verantwortlich für das ändern der Attribute des Food Objekts. Neben anderen Code Smells, trifft hier jedoch das “Code smell – Duplicated Code” stark zu. Eine Lösung dafür wäre es eine generische Methode zu erstellen. (siehe nachfolgendem Code)

```
0 references
private void UpdateProperty<T>(string propertyName, Func<string, bool> validateInput, out T propertyValue)
{
    bool validInput = false;

    while (!validInput)
    {
        Console.Write($"Enter the new {propertyName}: ");
        string? input = Console.ReadLine();

        if (validateInput(input, out T value))
        {
            propertyValue = value;
            validInput = true;
        }
    }
}
```

CODE SMELL 2: Long Method

```
1 reference
private static UserData CreateUser()
{
    Console.WriteLine("This user does not exist");
    Console.WriteLine("Please provide following information");

    Console.Write("Enter your name: ");
    string name = ReadNonEmptyLine();

    Console.Write("Enter your sex (m/f): ");
    string? input = Console.ReadLine();

    while (string.IsNullOrWhiteSpace(input) || input.Length > 1 || (input[0] != 'm' && input[0] != 'f'))
    {
        Console.WriteLine("Invalid input. Please enter 'm' for male or 'f' for female.");
        Console.Write("Enter your sex (m/f): ");
        input = Console.ReadLine();
    }

    char sex = input[0];

    Console.Write("Enter your age: ");
    int age = ReadNonEmptyInt();

    Console.Write("Enter your weight: ");
    double weight = ReadNonEmptyDouble();

    Console.Write("Enter your height: ");
    int height = ReadNonEmptyInt();

    Console.WriteLine("Activity Levels:");
    Console.WriteLine("1: Sedentary (little or no exercise)");
    Console.WriteLine("2: Lightly active (light exercise or sports 1-3 days/week)");
    Console.WriteLine("3: Moderately active (moderate exercise 3-5 days/week)");
    Console.WriteLine("4: Very active (hard exercise 6-7 days/week)");
    Console.WriteLine("5: Super active (very hard exercise and a physical job)");

    Console.Write("Enter the activity level (1-5): ");
    int activity = ReadNonEmptyInt();

    UserData user = new(name, sex, age, weight, height, activity);

    return user;

}
```

Um diese große Methode zu verkleinern sollte sie in mehrere Methoden aufgeteilt werden.

Ausschnitt der Änderung (Weitere Funktionen im finalen Code zu sehen):

```
1 reference
private static UserData CreateUser()
{
    Console.WriteLine("This user does not exist");
    Console.WriteLine("Please provide the following information");

    string name = ReadUserName();
    char sex = ReadUserSex();
    int age = ReadUserAge();
    double weight = ReadUserWeight();
    int height = ReadUserHeight();
    int activity = ReadUserActivity();

    UserData user = new UserData(name, sex, age, weight, height, activity);

    return user;
}

1 reference
private static string ReadUserName()
{
    Console.Write("Enter your name: ");
    return ReadNonEmptyLine();
}

1 reference
private static char ReadUserSex()
{
    Console.Write("Enter your sex (m/f): ");
    string? input = Console.ReadLine();

    while (string.IsNullOrWhiteSpace(input) || input.Length > 1 || (input[0] != 'm' && input[0] != 'f'))
    {
        Console.WriteLine("Invalid input. Please enter 'm' for male or 'f' for female.");
        Console.Write("Enter your sex (m/f): ");
        input = Console.ReadLine();
    }

    return input[0];
}

1 reference
private static int ReadUserAge()
{
    Console.Write("Enter your age: ");
    return ReadNonEmptyInt();
}
```

CODE SMELL 3: Long parameter list

```
1 reference
private void calcDailyIntake(int calories, double carbs, double protein, double fat) {
    this.calories += calories;
    this.carbs += carbs;
    this.protein += protein;
    this.fat += fat;
}
```

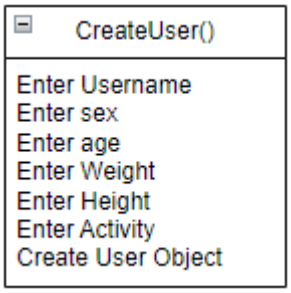
Um die lange Parameterliste zu entfernen, kann das Objekt woraus die Parameter entstehen übergegeben werden:

```
1 reference
private void calcDailyIntake(Intake intake) {
    this.calories += intake.calories;
    this.carbs += intake.carbs;
    this.protein += intake.protein;
    this.fat += intake.fat;
}
```

3 Refactorings (6P)

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

REFACTORING 1: Extract Method



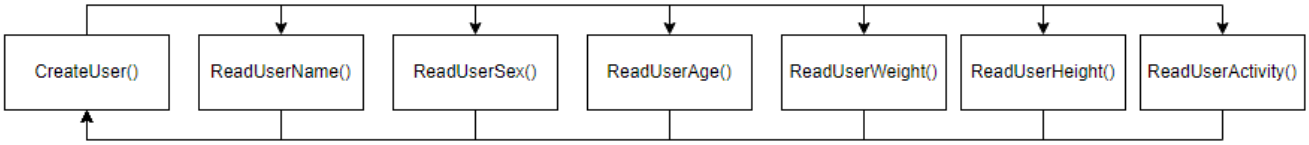
Methode vor dem Refactoring:Führt mehrere Aufgaben aus wie z.B. Enter Username oder Enter Activity und speichert diese im Anschluss in einem neu erstellten User Objekt ab.

Commit:

https://github.com/bittali/Tracktrition/commit/4303512ed67fd58c26b05c93a5a88591e94f64e5

Problemstellung wie bei Code smells 3:

Code wird so verständlicher, Kommentare können weggelassen werden

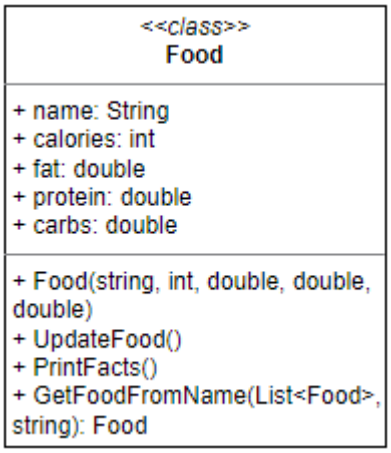


Durch das extrahieren der Aufgaben wird die überladene CreateUser-Methode vereinfacht.

REFACTORING 2: Replace Error Code with exception

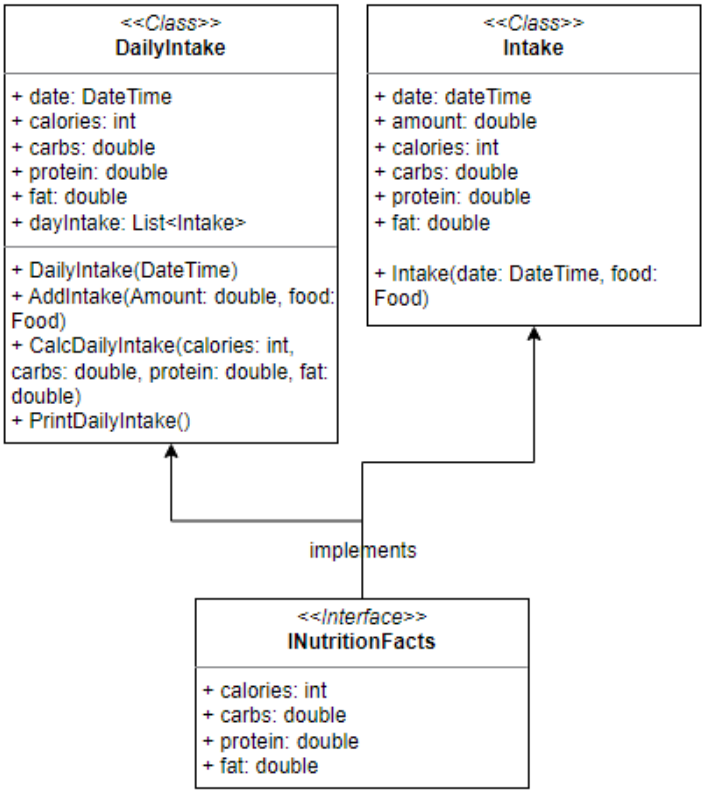
Commit:
https://github.com/bittali/Tracktrition/commit/64474b6e064e4ff148ccb5a6767e0787d92d5013

Das Uml verändert sich nicht, da sich sie Refactorisierung innerhalb der Methode abspielt.Um nicht innerhalb des Programmes mit null Werten zu arbeiten, macht es Sinn bereits vorher – beim Schritt der den null Wert erstellt – einen Fehler zu werfen.

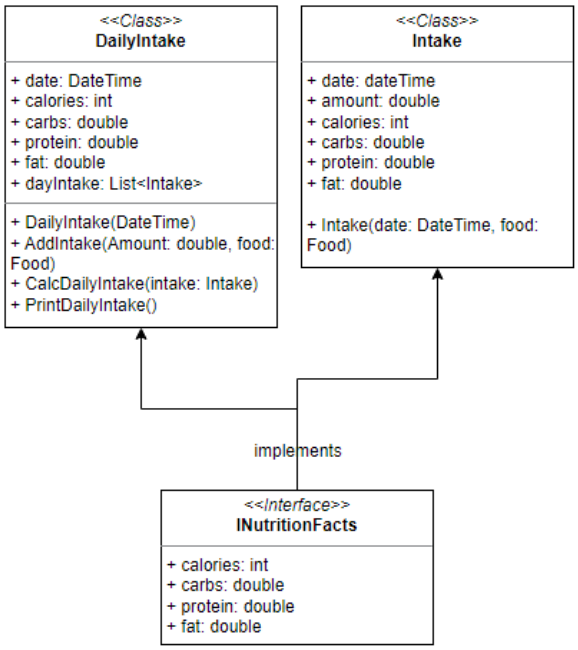


REFACTORING 3: Long parameter list

Siehe Code Smells 3



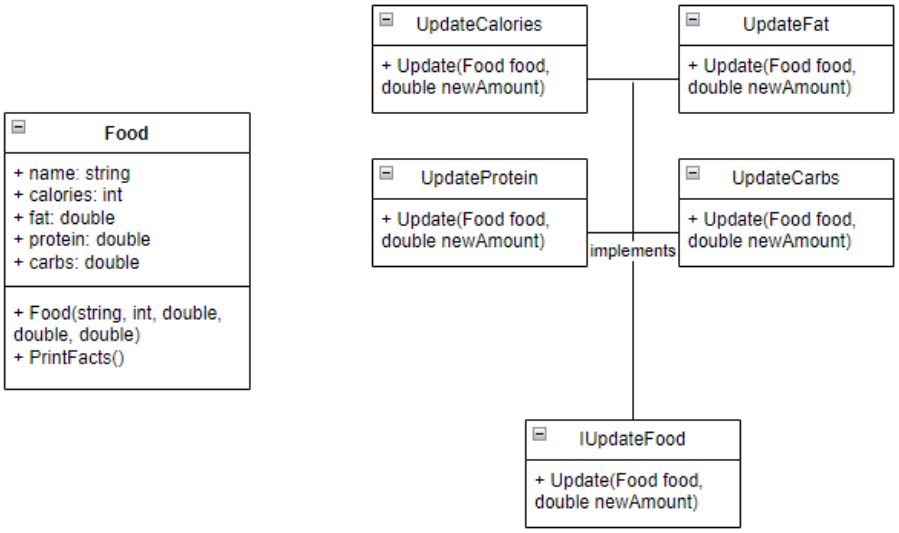
Parameter der Funktion sind zu Überladen. Alle Parameter sind jedoch Teil der Klasse Intake, wovon diese auch kommen. Hierbei ist es sinnvoll, die Parameter zu entfernen und durch das Objekt zu ersetzten. Die Variablenwerte werden dann in der Berechnung der Methode aus dem Objekt entnommen. So wird der Code übersichtlicher und die Methoden verständlicher.
Oben: Vor Refactoring
Unten: Nach Refactoring



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 Builderentwurfsmuster ist nicht implementiert, jedoch weist die Food Klasse gewisse Parallelen auf. Durch die Extraktion der Wertaktualisierungen, existiert ein Builder des Food Objektes, jedoch aufgeteilt in einzelne Klassen pro Attribut und auch nur für die Aktualisierung der Attributwerte.

Entwurfsmuster: [Name] (4P)